Category: core_algorithms

File: venv/lib/python3.12/site-packages/jwt/algorithms.py

```
def get_default_algorithms() -> dict[str, Algorithm]:
    """

class NoneAlgorithm(Algorithm):
    """
    Placeholder for use when no signing or verification
    operations are required.
    """

class HMACAlgorithm(Algorithm):
    """
    Performs signing and verification operations using HMAC
    and the specified hash function.
    """

    class RSAAlgorithm(Algorithm):
        """
        Performs signing and verification operations using
        RSASSA-PKCS-v1_5 and the specified hash function.
        """

    class ECAlgorithm(Algorithm):
        """
        Performs signing and verification operations using
        ECDSA and the specified hash function
        """

    class RSAPSSAlgorithm(RSAAlgorithm):
        """
        Performs a signature using RSASSA-PSS with MGF1
        """

    class OKPAlgorithm(Algorithm):
        """
        Performs signing and verification operations using EdDSA

        "HS256": HMACAlgorithm(HMACAlgorithm.SHA256),

        "HS384": HMACAlgorithm(HMACAlgorithm.SHA384),

        "HS512": HMACAlgorithm(HMACAlgorithm.SHA512),
    }

            "RS256": RSAAlgorithm(RSAAlgorithm.SHA256),

            "RS384": RSAAlgorithm(RSAAlgorithm.SHA384),

            "RS512": RSAAlgorithm(RSAAlgorithm.SHA512),
```

```
            "ES256": ECAlgorithm(ECAlgorithm.SHA256),

            "ES256K": ECAlgorithm(ECAlgorithm.SHA256),

            "ES384": ECAlgorithm(ECAlgorithm.SHA384),

            "ES521": ECAlgorithm(ECAlgorithm.SHA512),

            "PS256": RSAPSSAlgorithm(RSAPSSAlgorithm.SHA256),

            "PS384": RSAPSSAlgorithm(RSAPSSAlgorithm.SHA384),

            "PS512": RSAPSSAlgorithm(RSAPSSAlgorithm.SHA512),
```

```
        Compute a hash digest using the specified algorithm's hash algorithm.
```

```
    default_algorithms = {
```

```
        default_algorithms.update(
            {
```

```
    return default_algorithms

class Algorithm(ABC):
    """
```

```
            and issubclass(hash_alg, hashes.HashAlgorithm)
        ):
```

```
        SHA256: ClassVar[type[hashes.HashAlgorithm]] = hashes.SHA256

        SHA384: ClassVar[type[hashes.HashAlgorithm]] = hashes.SHA384

        SHA512: ClassVar[type[hashes.HashAlgorithm]] = hashes.SHA512

        def __init__(self, hash_alg: type[hashes.HashAlgorithm]) -> None:
            self.hash_alg = hash_alg

        SHA256: ClassVar[type[hashes.HashAlgorithm]] = hashes.SHA256

        SHA384: ClassVar[type[hashes.HashAlgorithm]] = hashes.SHA384

        SHA512: ClassVar[type[hashes.HashAlgorithm]] = hashes.SHA512

        def __init__(self, hash_alg: type[hashes.HashAlgorithm]) -> None:
            self.hash_alg = hash_alg
```

```
    # Type aliases for convenience in algorithms method signatures
    AllowedRSAKeys = RSAPrivateKey | RSAPublicKey
    AllowedECKeys = EllipticCurvePrivateKey | EllipticCurvePublicKey
    AllowedOKPKeys = (
        Ed25519PrivateKey | Ed25519PublicKey | Ed448PrivateKey | Ed448PublicKey
    )
    AllowedKeys = AllowedRSAKeys | AllowedECKeys | AllowedOKPKeys
    AllowedPrivateKeys = (
        RSAPrivateKey | EllipticCurvePrivateKey | Ed25519PrivateKey | Ed448PrivateKey
    )
    AllowedPublicKeys = (
        RSAPublicKey | EllipticCurvePublicKey | Ed25519PublicKey | Ed448PublicKey
    )

    Returns the algorithms that are implemented by the library.
    """

        "none": NoneAlgorithm(),

            "ES512": ECAlgorithm(

                ECAlgorithm.SHA512
            ),  # Backward compat for #219 fix

            "EdDSA": OKPAlgorithm(),
        }
    )

    The interface for an algorithm used to sign and verify tokens.
    """

        If there is no hash algorithm, raises a NotImplementedError.
        """

                "Expecting a EllipticCurvePrivateKey/EllipticCurvePublicKey. Wrong key provided for ECDSA algorithms"
            )

                "Expecting a EllipticCurvePrivateKey/EllipticCurvePublicKey. Wrong key provided for EdDSA algorithms"
            )

                encryption_algorithm=NoEncryption(),
            )

from typing import TYPE_CHECKING, Any, ClassVar, NoReturn, Union, cast, overload
```

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives.asymmetric.ec import (
    ECDSA,
    SECP256K1,
    SECP256R1,
    SECP384R1,
    SECP521R1,
    EllipticCurve,
    EllipticCurvePrivateKey,
    EllipticCurvePrivateNumbers,
    EllipticCurvePublicKey,
    EllipticCurvePublicNumbers,
)
from cryptography.hazmat.primitives.asymmetric.ed448 import (
    Ed448PrivateKey,
    Ed448PublicKey,
)
from cryptography.hazmat.primitives.asymmetric.ed25519 import (
    Ed25519PrivateKey,
    Ed25519PublicKey,
)
from cryptography.hazmat.primitives.asymmetric.rsa import (
    RSAPrivateKey,
    RSAPrivateNumbers,
    RSAPublicKey,
    RSAPublicNumbers,
    rsa_crt_dmp1,
    rsa_crt_dmq1,
    rsa_crt_iqmp,
    rsa_recover_prime_factors,
)
from cryptography.hazmat.primitives.serialization import (
    Encoding,
    NoEncryption,
    PrivateFormat,
    PublicFormat,
    load_pem_private_key,
    load_pem_public_key,
    load_ssh_public_key,
)

def compute_hash_digest(self, bytestr: bytes) -> bytes:
    """

    # lookup self.hash_alg if defined in a way that mypy can understand
    hash_alg = getattr(self, "hash_alg", None)
    if hash_alg is None:
        raise NotImplementedError
```

```
        digest = hashes.Hash(hash_alg(), backend=default_backend())
        digest.update(bytestr)
        return bytes(digest.finalize())
    else:
        return bytes(hash_alg(bytestr).digest())

def prepare_key(self, key: Any) -> Any:
    """
    Performs necessary validation and conversions on the key and returns
    the key value in the proper format for sign() and verify().
    """

def sign(self, msg: bytes, key: Any) -> bytes:
    """
    Returns a digital signature for the specified message
    using the specified key value.
    """

def verify(self, msg: bytes, key: Any, sig: bytes) -> bool:
    """
    Verifies that the specified digital signature is valid
    for the specified message and key values.
    """

def to_jwk(key_obj, as_dict: Literal[True]) -> JWKDict:
    ...  # pragma: no cover

def to_jwk(key_obj, as_dict: Literal[False] = False) -> str:
    ...  # pragma: no cover

def to_jwk(key_obj, as_dict: bool = False) -> Union[JWKDict, str]:
    """
    Serializes a given key into a JWK
    """

def from_jwk(jwk: str | JWKDict) -> Any:
    """
    Deserializes a given key from JWK back into a key object
    """

def prepare_key(self, key: str | None) -> None:
    if key == "":
        key = None

def sign(self, msg: bytes, key: None) -> bytes:
    return b""

def verify(self, msg: bytes, key: None, sig: bytes) -> bool:
    return False

def to_jwk(key_obj: Any, as_dict: bool = False) -> NoReturn:
    raise NotImplementedError()
```

```
def from_jwk(jwk: str | JWKDict) -> NoReturn:
    raise NotImplementedError()

SHA256: ClassVar[HashlibHash] = hashlib.sha256

SHA384: ClassVar[HashlibHash] = hashlib.sha384

SHA512: ClassVar[HashlibHash] = hashlib.sha512

def __init__(self, hash_alg: HashlibHash) -> None:
    self.hash_alg = hash_alg

def prepare_key(self, key: str | bytes) -> bytes:
    key_bytes = force_bytes(key)

def to_jwk(key_obj: str | bytes, as_dict: Literal[True]) -> JWKDict:
    ...  # pragma: no cover

def to_jwk(key_obj: str | bytes, as_dict: Literal[False] = False) -> str:
    ...  # pragma: no cover

def to_jwk(key_obj: str | bytes, as_dict: bool = False) -> Union[JWKDict, str]:
    jwk = {
        "k": base64url_encode(force_bytes(key_obj)).decode(),
        "kty": "oct",
    }

def from_jwk(jwk: str | JWKDict) -> bytes:
    try:
        if isinstance(jwk, str):
            obj: JWKDict = json.loads(jwk)
        elif isinstance(jwk, dict):
            obj = jwk
        else:
            raise ValueError
    except ValueError:
        raise InvalidKeyError("Key is not valid JSON")

def sign(self, msg: bytes, key: bytes) -> bytes:
    return hmac.new(key, msg, self.hash_alg).digest()

def verify(self, msg: bytes, key: bytes, sig: bytes) -> bool:
    return hmac.compare_digest(sig, self.sign(msg, key))

    def prepare_key(self, key: AllowedRSAKeys | str | bytes) -> AllowedRSAKeys:
        if isinstance(key, (RSAPrivateKey, RSAPublicKey)):
            return key

    def to_jwk(key_obj: AllowedRSAKeys, as_dict: Literal[True]) -> JWKDict:
        ...  # pragma: no cover

    def to_jwk(key_obj: AllowedRSAKeys, as_dict: Literal[False] = False) -> str:
        ...  # pragma: no cover
```

```
def to_jwk(
    key_obj: AllowedRSAKeys, as_dict: bool = False
) -> Union[JWKDict, str]:
    obj: dict[str, Any] | None = None

def from_jwk(jwk: str | JWKDict) -> AllowedRSAKeys:
    try:
        if isinstance(jwk, str):
            obj = json.loads(jwk)
        elif isinstance(jwk, dict):
            obj = jwk
        else:
            raise ValueError
    except ValueError:
        raise InvalidKeyError("Key is not valid JSON")

def sign(self, msg: bytes, key: RSAPrivateKey) -> bytes:
    return key.sign(msg, padding.PKCS1v15(), self.hash_alg())

def verify(self, msg: bytes, key: RSAPublicKey, sig: bytes) -> bool:
    try:
        key.verify(sig, msg, padding.PKCS1v15(), self.hash_alg())
        return True
    except InvalidSignature:
        return False

def prepare_key(self, key: AllowedECKeys | str | bytes) -> AllowedECKeys:
    if isinstance(key, (EllipticCurvePrivateKey, EllipticCurvePublicKey)):
        return key

def sign(self, msg: bytes, key: EllipticCurvePrivateKey) -> bytes:
    der_sig = key.sign(msg, ECDSA(self.hash_alg()))

def verify(self, msg: bytes, key: "AllowedECKeys", sig: bytes) -> bool:
    try:
        der_sig = raw_to_der_signature(sig, key.curve)
    except ValueError:
        return False

def to_jwk(key_obj: AllowedECKeys, as_dict: Literal[True]) -> JWKDict:
    ...  # pragma: no cover

def to_jwk(key_obj: AllowedECKeys, as_dict: Literal[False] = False) -> str:
    ...  # pragma: no cover
```

```
def to_jwk(
    key_obj: AllowedECKeys, as_dict: bool = False
) -> Union[JWKDict, str]:
    if isinstance(key_obj, EllipticCurvePrivateKey):
        public_numbers = key_obj.public_key().public_numbers()
    elif isinstance(key_obj, EllipticCurvePublicKey):
        public_numbers = key_obj.public_numbers()
    else:
        raise InvalidKeyError("Not a public or private key")

def from_jwk(jwk: str | JWKDict) -> AllowedECKeys:
    try:
        if isinstance(jwk, str):
            obj = json.loads(jwk)
        elif isinstance(jwk, dict):
            obj = jwk
        else:
            raise ValueError
    except ValueError:
        raise InvalidKeyError("Key is not valid JSON")

def sign(self, msg: bytes, key: RSAPrivateKey) -> bytes:
    return key.sign(
        msg,
        padding.PSS(
            mgf=padding.MGF1(self.hash_alg()),
            salt_length=self.hash_alg().digest_size,
        ),
        self.hash_alg(),
    )

def verify(self, msg: bytes, key: RSAPublicKey, sig: bytes) -> bool:
    try:
        key.verify(
            sig,
            msg,
            padding.PSS(
                mgf=padding.MGF1(self.hash_alg()),
                salt_length=self.hash_alg().digest_size,
            ),
            self.hash_alg(),
        )
        return True
    except InvalidSignature:
        return False

This class requires ``cryptography>=2.6`` to be installed.
"""

def __init__(self, **kwargs: Any) -> None:
    pass
```

```
def prepare_key(self, key: AllowedOKPKeys | str | bytes) -> AllowedOKPKeys:
    if isinstance(key, (bytes, str)):
        key_str = key.decode("utf-8") if isinstance(key, bytes) else key
        key_bytes = key.encode("utf-8") if isinstance(key, str) else key

def sign(
    self, msg: str | bytes, key: Ed25519PrivateKey | Ed448PrivateKey
) -> bytes:
    """
    Sign a message ``msg`` using the EdDSA private key ``key``
    :param str|bytes msg: Message to sign

    :param Ed25519PrivateKey}Ed448PrivateKey key: A :class:`.Ed25519PrivateKey`

        or :class:`.Ed448PrivateKey` isinstance
    :return bytes signature: The signature, as bytes
    """
    msg_bytes = msg.encode("utf-8") if isinstance(msg, str) else msg
    return key.sign(msg_bytes)

def verify(
    self, msg: str | bytes, key: AllowedOKPKeys, sig: str | bytes
) -> bool:
    """
    Verify a given ``msg`` against a signature ``sig`` using the EdDSA key ``key``

def to_jwk(key: AllowedOKPKeys, as_dict: Literal[True]) -> JWKDict:
    ...  # pragma: no cover

def to_jwk(key: AllowedOKPKeys, as_dict: Literal[False] = False) -> str:
    ...  # pragma: no cover

def to_jwk(key: AllowedOKPKeys, as_dict: bool = False) -> Union[JWKDict, str]:
    if isinstance(key, (Ed25519PublicKey, Ed448PublicKey)):
        x = key.public_bytes(
            encoding=Encoding.Raw,
            format=PublicFormat.Raw,
        )
        crv = "Ed25519" if isinstance(key, Ed25519PublicKey) else "Ed448"

def from_jwk(jwk: str | JWKDict) -> AllowedOKPKeys:
    try:
        if isinstance(jwk, str):
            obj = json.loads(jwk)
        elif isinstance(jwk, dict):
            obj = jwk
        else:
            raise ValueError
    except ValueError:
        raise InvalidKeyError("Key is not valid JSON")
```

Category: core_algorithms

File: venv/lib/python3.12/site-packages/django/urls/resolvers.py

```
            self._func_path = func.__class__.__module__ + "." + func.__class__.__name__
        else:
            # A function-based view
            self._func_path = func.__module__ + "." + func.__name__

    def __init__(self, prefix_default_language=True):

        self.prefix_default_language = prefix_default_language
        self.converters = {}

    def __init__(self, pattern, callback, default_args=None, name=None):
        self.pattern = pattern
        self.callback = callback  # the view

        self.default_args = default_args or {}
        self.name = name

        if inspect.isclass(view) and issubclass(view, View):
            return [
                Error(
                    "Your URL pattern %s has an invalid view, pass %s.as_view() "
                    "instead of %s."
                    % (
                        self.pattern.describe(),
                        view.__name__,
                        view.__name__,
                    ),
                    id="urls.E009",
                )
            ]
        return []

        self.default_kwargs = default_kwargs or {}
        self.namespace = namespace
        self.app_name = app_name
        self._reverse_dict = {}
        self._namespace_dict = {}
        self._app_dict = {}
```

```
                        {**defaults, **url_pattern.default_kwargs},
                        {
                            **self.pattern.converters,
                            **url_pattern.pattern.converters,
                            **converters,
                        },
                    ),
                )
            for namespace, (
                prefix,
                sub_pattern,
            ) in url_pattern.namespace_dict.items():
                current_converters = url_pattern.pattern.converters
                sub_pattern.pattern.converters.update(current_converters)
                namespaces[namespace] = (p_pattern + prefix, sub_pattern)
            for app_name, namespace_list in url_pattern.app_dict.items():

URLResolver is the main class here. Its resolve() method takes a URL (as
a string) and returns a ResolverMatch object which provides access to all
attributes of the resolved URL match.
"""
import functools
import inspect
import re
import string
from importlib import import_module
from pickle import PicklingError
from urllib.parse import quote

class ResolverMatch:

    def __init__(
        self,
        func,
        args,
        kwargs,
        url_name=None,
        app_names=None,
        namespaces=None,
        route=None,
        tried=None,
        captured_kwargs=None,
        extra_kwargs=None,
    ):
        self.func = func
        self.args = args
        self.kwargs = kwargs
        self.url_name = url_name
        self.route = route
        self.tried = tried
        self.captured_kwargs = captured_kwargs
        self.extra_kwargs = extra_kwargs
```

```
        if hasattr(func, "view_class"):

            func = func.view_class
        if not hasattr(func, "__name__"):

            # A class-based view

    def __getitem__(self, index):
        return (self.func, self.args, self.kwargs)[index]

    def __repr__(self):
        if isinstance(self.func, functools.partial):
            func = repr(self.func)
        else:
            func = self._func_path
        return (
            "ResolverMatch(func=%s, args=%r, kwargs=%r, url_name=%r, "
            "app_names=%r, namespaces=%r, route=%r%s%s)"
            % (
                func,
                self.args,
                self.kwargs,
                self.url_name,
                self.app_names,
                self.namespaces,
                self.route,
                f", captured_kwargs={self.captured_kwargs!r}"
                if self.captured_kwargs
                else "",
                f", extra_kwargs={self.extra_kwargs!r}" if self.extra_kwargs else "",
            )
        )

    def __reduce_ex__(self, protocol):

        raise PicklingError(f"Cannot pickle {self.__class__.__qualname__}.")

def get_resolver(urlconf=None):
    if urlconf is None:
        urlconf = settings.ROOT_URLCONF
    return _get_cached_resolver(urlconf)

def _get_cached_resolver(urlconf=None):
    return URLResolver(RegexPattern(r"^/"), urlconf)

def get_ns_resolver(ns_pattern, resolver, converters):
    # Build a namespaced resolver for the given parent URLconf pattern.
    # This makes it possible to have captured parameters in the parent
    # URLconf pattern.
    pattern = RegexPattern(ns_pattern)
    pattern.converters = dict(converters)
    ns_resolver = URLResolver(pattern, resolver.url_patterns)
    return URLResolver(RegexPattern(r"^/"), [ns_resolver])
```

```
class LocaleRegexDescriptor:

    def __init__(self, attr):
        self.attr = attr

    def __get__(self, instance, cls=None):
        """
        Return a compiled regular expression based on the active language.
        """
        if instance is None:
            return self
        # As a performance optimization, if the given regex string is a regular
        # string (not a lazily-translated string proxy), compile it once and
        # avoid per-language compilation.
        pattern = getattr(instance, self.attr)
        if isinstance(pattern, str):
            instance.__dict__["regex"] = instance._compile(pattern)
            return instance.__dict__["regex"]
        language_code = get_language()
        if language_code not in instance._regex_dict:
            instance._regex_dict[language_code] = instance._compile(str(pattern))
        return instance._regex_dict[language_code]

class CheckURLMixin:

    def describe(self):
        """
        Format the URL pattern for display in warning messages.
        """
        description = "'{}'".format(self)
        if self.name:
            description += " [name='{}']".format(self.name)
        return description
```

```
    def _check_pattern_startswith_slash(self):
        """
        Check that the pattern does not begin with a forward slash.
        """
        regex_pattern = self.regex.pattern
        if not settings.APPEND_SLASH:
            # Skip check as it can be useful to start a URL pattern with a slash
            # when APPEND_SLASH=False.
            return []
        if regex_pattern.startswith(("/", "^/", "^\\/")) and not regex_pattern.endswith(
            "/"
        ):
            warning = Warning(
                "Your URL pattern {} has a route beginning with a '/'. Remove this "
                "slash as it is unnecessary. If this pattern is targeted in an "
                "include(), ensure the include() pattern has a trailing '/'.".format(
                    self.describe()
                ),
                id="urls.W002",
            )
            return [warning]
        else:
            return []

class RegexPattern(CheckURLMixin):
    regex = LocaleRegexDescriptor("_regex")

    def __init__(self, regex, name=None, is_endpoint=False):
        self._regex = regex
        self._regex_dict = {}
        self._is_endpoint = is_endpoint
        self.name = name
        self.converters = {}

    def match(self, path):
        match = (
            self.regex.fullmatch(path)
            if self._is_endpoint and self.regex.pattern.endswith("$")
            else self.regex.search(path)
        )
        if match:
            # If there are any named groups, use those as kwargs, ignoring
            # non-named groups. Otherwise, pass all non-named arguments as
            # positional arguments.
            kwargs = match.groupdict()
            args = () if kwargs else match.groups()
            kwargs = {k: v for k, v in kwargs.items() if v is not None}
            return path[match.end() :], args, kwargs
        return None
```

```
def check(self):
    warnings = []
    warnings.extend(self._check_pattern_startswith_slash())
    if not self._is_endpoint:
        warnings.extend(self._check_include_trailing_dollar())
    return warnings

def _check_include_trailing_dollar(self):
    regex_pattern = self.regex.pattern
    if regex_pattern.endswith("$") and not regex_pattern.endswith(r"\$"):
        return [
            Warning(
                "Your URL pattern {} uses include with a route ending with a '$'. "
                "Remove the dollar from the route to avoid problems including "
                "URLs.".format(self.describe()),
                id="urls.W001",
            )
        ]
    else:
        return []

def _compile(self, regex):
    """Compile and return the given regular expression."""
    try:
        return re.compile(regex)
    except re.error as e:
        raise ImproperlyConfigured(
            '"%s" is not a valid regular expression: %s' % (regex, e)
        ) from e

def __str__(self):
    return str(self._regex)
```

```
def _route_to_regex(route, is_endpoint=False):
    """
    Convert a path pattern into a regular expression. Return the regular
    expression and a dictionary mapping the capture names to the converters.
    For example, 'foo/<int:pk>' returns '^foo\\/(?P<pk>[0-9]+)'
    and {'pk': <django.urls.converters.IntConverter>}.
    """
    original_route = route
    parts = ["^"]
    converters = {}
    while True:
        match = _PATH_PARAMETER_COMPONENT_RE.search(route)
        if not match:
            parts.append(re.escape(route))
            break
        elif not set(match.group()).isdisjoint(string.whitespace):
            raise ImproperlyConfigured(
                "URL route '%s' cannot contain whitespace in angle brackets "
                "<?>." % original_route
            )
        parts.append(re.escape(route[: match.start()]))
        route = route[match.end() :]
        parameter = match["parameter"]
        if not parameter.isidentifier():
            raise ImproperlyConfigured(
                "URL route '%s' uses parameter name %r which isn't a valid "
                "Python identifier." % (original_route, parameter)
            )
        raw_converter = match["converter"]
        if raw_converter is None:

            # If a converter isn't specified, the default is `str`.
            raw_converter = "str"
        try:
            converter = get_converter(raw_converter)
        except KeyError as e:
            raise ImproperlyConfigured(
                "URL route %r uses invalid converter %r."
                % (original_route, raw_converter)
            ) from e
        converters[parameter] = converter
        parts.append("(?P<" + parameter + ">" + converter.regex + ")")
    if is_endpoint:
        parts.append(r"\Z")
    return "".join(parts), converters

class RoutePattern(CheckURLMixin):
    regex = LocaleRegexDescriptor("_route")
```

```
    def __init__(self, route, name=None, is_endpoint=False):
        self._route = route
        self._regex_dict = {}
        self._is_endpoint = is_endpoint
        self.name = name
        self.converters = _route_to_regex(str(route), is_endpoint)[1]

    def match(self, path):
        match = self.regex.search(path)
        if match:
            # RoutePattern doesn't allow non-named groups so args are ignored.
            kwargs = match.groupdict()
            for key, value in kwargs.items():
                converter = self.converters[key]
                try:
                    kwargs[key] = converter.to_python(value)
                except ValueError:
                    return None
            return path[match.end() :], (), kwargs
        return None

    def check(self):
        warnings = self._check_pattern_startswith_slash()
        route = self._route
        if "(?P<" in route or route.startswith("^") or route.endswith("$"):
            warnings.append(
                Warning(
                    "Your URL pattern {} has a route that contains '(?P<', begins "
                    "with a '^', or ends with a '$'. This was likely an oversight "
                    "when migrating to django.urls.path().".format(self.describe()),
                    id="2_0.W001",
                )
            )
        return warnings

    def _compile(self, route):
        return re.compile(_route_to_regex(route, self._is_endpoint)[0])

    def __str__(self):
        return str(self._route)

class LocalePrefixPattern:

    def regex(self):
        # This is only used by reverse() and cached in _reverse_dict.
        return re.compile(re.escape(self.language_prefix))

    def language_prefix(self):
        language_code = get_language() or settings.LANGUAGE_CODE
```

```
        if language_code == settings.LANGUAGE_CODE and not self.prefix_default_language:
            return ""
        else:
            return "%s/" % language_code

    def match(self, path):
        language_prefix = self.language_prefix
        if path.startswith(language_prefix):
            return path[len(language_prefix) :], (), {}
        return None

    def check(self):
        return []

    def describe(self):
        return "'{}'".format(self)

    def __str__(self):
        return self.language_prefix

class URLPattern:

    def __repr__(self):

        return "<%s %s>" % (self.__class__.__name__, self.pattern.describe())

    def check(self):
        warnings = self._check_pattern_name()
        warnings.extend(self.pattern.check())
        warnings.extend(self._check_callback())
        return warnings

    def _check_pattern_name(self):
        """
        Check that the pattern name does not contain a colon.
        """
        if self.pattern.name is not None and ":" in self.pattern.name:
            warning = Warning(
                "Your URL pattern {} has a name including a ':'. Remove the colon, to "
                "avoid ambiguous namespace references.".format(self.pattern.describe()),
                id="urls.W003",
            )
            return [warning]
        else:
            return []

    def _check_callback(self):
        from django.views import View

    def resolve(self, path):
        match = self.pattern.match(path)
        if match:
            new_path, args, captured_kwargs = match
```

```
            # Pass any default args as **kwargs.

            kwargs = {**captured_kwargs, **self.default_args}
            return ResolverMatch(
                self.callback,
                args,
                kwargs,
                self.pattern.name,
                route=str(self.pattern),
                captured_kwargs=captured_kwargs,

                extra_kwargs=self.default_args,
            )

    def lookup_str(self):
        """
        A string that identifies the view (e.g. 'path.to.view_function' or

        'path.to.ClassBasedView').
        """
        callback = self.callback
        if isinstance(callback, functools.partial):
            callback = callback.func

        if hasattr(callback, "view_class"):

            callback = callback.view_class
        elif not hasattr(callback, "__name__"):

            return callback.__module__ + "." + callback.__class__.__name__
        return callback.__module__ + "." + callback.__qualname__

class URLResolver:

    def __init__(

        self, pattern, urlconf_name, default_kwargs=None, app_name=None, namespace=None
    ):
        self.pattern = pattern

        # urlconf_name is the dotted Python path to the module defining
        # urlpatterns. It may also be an object with an urlpatterns attribute
        # or urlpatterns itself.
        self.urlconf_name = urlconf_name
        self.callback = None

        # set of dotted paths to all functions and classes that are used in
        # urlpatterns
        self._callback_strs = set()
        self._populated = False
        self._local = Local()
```

```
def __repr__(self):
    if isinstance(self.urlconf_name, list) and self.urlconf_name:
        # Don't bother to output the whole list, it can be huge

        urlconf_repr = "<%s list>" % self.urlconf_name[0].__class__.__name__
    else:
        urlconf_repr = repr(self.urlconf_name)
    return "<%s %s (%s:%s) %s>" % (

        self.__class__.__name__,
        urlconf_repr,
        self.app_name,
        self.namespace,
        self.pattern.describe(),
    )

def check(self):
    messages = []
    for pattern in self.url_patterns:
        messages.extend(check_resolver(pattern))
    messages.extend(self._check_custom_error_handlers())
    return messages or self.pattern.check()

def _check_custom_error_handlers(self):
    messages = []
    # All handlers take (request, exception) arguments except handler500
    # which takes (request).
    for status_code, num_parameters in [(400, 2), (403, 2), (404, 2), (500, 1)]:
        try:
            handler = self.resolve_error_handler(status_code)
        except (ImportError, ViewDoesNotExist) as e:
            path = getattr(self.urlconf_module, "handler%s" % status_code)
            msg = (
                "The custom handler{status_code} view '{path}' could not be "
                "imported."
            ).format(status_code=status_code, path=path)
            messages.append(Error(msg, hint=str(e), id="urls.E008"))
            continue
        signature = inspect.signature(handler)
        args = [None] * num_parameters
        try:
            signature.bind(*args)
        except TypeError:
            msg = (
                "The custom handler{status_code} view '{path}' does not "
                "take the correct number of arguments ({args})."
            ).format(
                status_code=status_code,
                path=handler.__module__ + "." + handler.__qualname__,
                args="request, exception" if num_parameters == 2 else "request",
            )
            messages.append(Error(msg, id="urls.E007"))
    return messages
```

```
def _populate(self):
    # Short-circuit if called recursively in this thread to prevent
    # infinite recursion. Concurrent threads may call this at the same
    # time and will need to continue, so set 'populating' on a
    # thread-local variable.
    if getattr(self._local, "populating", False):
        return
    try:
        self._local.populating = True
        lookups = MultiValueDict()
        namespaces = {}
        apps = {}
        language_code = get_language()
        for url_pattern in reversed(self.url_patterns):
            p_pattern = url_pattern.pattern.regex.pattern
            if p_pattern.startswith("^"):
                p_pattern = p_pattern[1:]
            if isinstance(url_pattern, URLPattern):
                self._callback_strs.add(url_pattern.lookup_str)
                bits = normalize(url_pattern.pattern.regex.pattern)
                lookups.appendlist(
                    url_pattern.callback,
                    (
                        bits,
                        p_pattern,

                        url_pattern.default_args,
                        url_pattern.pattern.converters,
                    ),
                )
                if url_pattern.name is not None:
                    lookups.appendlist(
                        url_pattern.name,
                        (
                            bits,
                            p_pattern,

                            url_pattern.default_args,
                            url_pattern.pattern.converters,
                        ),
                    )
            else:  # url_pattern is a URLResolver.
                url_pattern._populate()
                if url_pattern.app_name:
```

```
                apps.setdefault(url_pattern.app_name, []).append(
                    url_pattern.namespace
                )
                namespaces[url_pattern.namespace] = (p_pattern, url_pattern)
            else:
                for name in url_pattern.reverse_dict:
                    for (
                        matches,
                        pat,

                        defaults,
                        converters,
                    ) in url_pattern.reverse_dict.getlist(name):
                        new_matches = normalize(p_pattern + pat)
                        lookups.appendlist(
                            name,
                            (
                                new_matches,
                                p_pattern + pat,

                        apps.setdefault(app_name, []).extend(namespace_list)
            self._callback_strs.update(url_pattern._callback_strs)
        self._namespace_dict[language_code] = namespaces
        self._app_dict[language_code] = apps
        self._reverse_dict[language_code] = lookups
        self._populated = True
    finally:
        self._local.populating = False

def reverse_dict(self):
    language_code = get_language()
    if language_code not in self._reverse_dict:
        self._populate()
    return self._reverse_dict[language_code]

def namespace_dict(self):
    language_code = get_language()
    if language_code not in self._namespace_dict:
        self._populate()
    return self._namespace_dict[language_code]

def app_dict(self):
    language_code = get_language()
    if language_code not in self._app_dict:
        self._populate()
    return self._app_dict[language_code]

def _extend_tried(tried, pattern, sub_tried=None):
    if sub_tried is None:
        tried.append([pattern])
    else:
        tried.extend([pattern, *t] for t in sub_tried)
```

```
def _join_route(route1, route2):
    """Join two routes, without the starting ^ in the second route."""
    if not route1:
        return route2
    if route2.startswith("^"):
        route2 = route2[1:]
    return route1 + route2

def _is_callback(self, name):
    if not self._populated:
        self._populate()
    return name in self._callback_strs

def resolve(self, path):
    path = str(path)  # path may be a reverse_lazy object
    tried = []
    match = self.pattern.match(path)
    if match:
        new_path, args, kwargs = match
        for pattern in self.url_patterns:
            try:
                sub_match = pattern.resolve(new_path)
            except Resolver404 as e:
                self._extend_tried(tried, pattern, e.args[0].get("tried"))
            else:
                if sub_match:
                    # Merge captured arguments in match with submatch
```

```
                sub_match_dict = {**kwargs, **self.default_kwargs}
                # Update the sub_match_dict with the kwargs from the sub_match.
                sub_match_dict.update(sub_match.kwargs)
                # If there are *any* named groups, ignore all non-named groups.
                # Otherwise, pass all non-named arguments as positional
                # arguments.
                sub_match_args = sub_match.args
                if not sub_match_dict:
                    sub_match_args = args + sub_match.args
                current_route = (
                    ""
                    if isinstance(pattern, URLPattern)
                    else str(pattern.pattern)
                )
                self._extend_tried(tried, pattern, sub_match.tried)
                return ResolverMatch(
                    sub_match.func,
                    sub_match_args,
                    sub_match_dict,
                    sub_match.url_name,
                    [self.app_name] + sub_match.app_names,
                    [self.namespace] + sub_match.namespaces,
                    self._join_route(current_route, sub_match.route),
                    tried,
                    captured_kwargs=sub_match.captured_kwargs,
                    extra_kwargs={

                        **self.default_kwargs,
                        **sub_match.extra_kwargs,
                    },
                )
            tried.append([pattern])
        raise Resolver404({"tried": tried, "path": new_path})
    raise Resolver404({"path": path})

def urlconf_module(self):
    if isinstance(self.urlconf_name, str):
        return import_module(self.urlconf_name)
    else:
        return self.urlconf_name

def url_patterns(self):
```

```
    # urlconf_module might be a valid set of patterns, so we default to it
    patterns = getattr(self.urlconf_module, "urlpatterns", self.urlconf_module)
    try:
        iter(patterns)
    except TypeError as e:
        msg = (
            "The included URLconf '{name}' does not appear to have "
            "any patterns in it. If you see the 'urlpatterns' variable "
            "with valid patterns in the file then the issue is probably "
            "caused by a circular import."
        )
        raise ImproperlyConfigured(msg.format(name=self.urlconf_name)) from e
    return patterns

def resolve_error_handler(self, view_type):
    callback = getattr(self.urlconf_module, "handler%s" % view_type, None)
    if not callback:
        # No handler specified in file; use lazy import, since
        # django.conf.urls imports this file.
        from django.conf import urls

def reverse(self, lookup_view, *args, **kwargs):
    return self._reverse_with_prefix(lookup_view, "", *args, **kwargs)

def _reverse_with_prefix(self, lookup_view, _prefix, *args, **kwargs):
    if args and kwargs:
        raise ValueError("Don't mix *args and **kwargs in call to reverse()!")

    for possibility, pattern, defaults, converters in possibilities:
        for result, params in possibility:
            if args:
                if len(args) != len(params):
                    continue
                candidate_subs = dict(zip(params, args))
            else:

                if set(kwargs).symmetric_difference(params).difference(defaults):
                    continue
                matches = True
```

```
            for k, v in defaults.items():
                if k in params:
                    continue
                if kwargs.get(k, v) != v:
                    matches = False
                    break
            if not matches:
                continue
            candidate_subs = kwargs
        # Convert the candidate subs to text using Converter.to_url().
        text_candidate_subs = {}
        match = True
        for k, v in candidate_subs.items():
            if k in converters:
                try:
                    text_candidate_subs[k] = converters[k].to_url(v)
                except ValueError:
                    match = False
                    break
            else:
                text_candidate_subs[k] = str(v)
        if not match:
            continue
        # WSGI provides decoded URLs, without %xx escapes, and the URL
        # resolver operates on such URLs. First substitute arguments
        # without quoting to build a decoded URL and look for a match.
        # Then, if we have a match, redo the substitution with quoted
        # arguments in order to return a properly encoded URL.
        candidate_pat = _prefix.replace("%", "%%") + result
        if re.search(
            "^%s%s" % (re.escape(_prefix), pattern),
            candidate_pat % text_candidate_subs,
        ):

            # safe characters from `pchar` definition of RFC 3986
            url = quote(
                candidate_pat % text_candidate_subs,
                safe=RFC3986_SUBDELIMS + "/~:@",
            )
            # Don't allow construction of scheme relative urls.
            return escape_leading_slashes(url)
# lookup_view can be URL name or callable, but callables are not
# friendly in error messages.
m = getattr(lookup_view, "__module__", None)
n = getattr(lookup_view, "__name__", None)
if m is not None and n is not None:
    lookup_view_s = "%s.%s" % (m, n)
else:
    lookup_view_s = lookup_view
```

Category: core_algorithms

File: venv/lib/python3.12/site-packages/django/template/engine.py

```
    def template_context_processors(self):
        context_processors = _builtin_context_processors
        context_processors += tuple(self.context_processors)
        return tuple(import_string(path) for path in context_processors)

    def __init__(
        self,
        dirs=None,
        app_dirs=False,
        context_processors=None,
        debug=False,
        loaders=None,
        string_if_invalid="",
        file_charset="utf-8",
        libraries=None,
        builtins=None,
        autoescape=True,
    ):
        if dirs is None:
            dirs = []
        if context_processors is None:
            context_processors = []
        if loaders is None:
            loaders = ["django.template.loaders.filesystem.Loader"]
            if app_dirs:
                loaders += ["django.template.loaders.app_directories.Loader"]
            loaders = [("django.template.loaders.cached.Loader", loaders)]
        else:
            if app_dirs:
                raise ImproperlyConfigured(

            self.__class__.__qualname__,
            "" if not self.dirs else " dirs=%s" % repr(self.dirs),
            self.app_dirs,
            ""
            if not self.context_processors
            else " context_processors=%s" % repr(self.context_processors),
            self.debug,
            repr(self.loaders),
            repr(self.string_if_invalid),
            repr(self.file_charset),
            "" if not self.libraries else " libraries=%s" % repr(self.libraries),
            "" if not self.builtins else " builtins=%s" % repr(self.builtins),
            repr(self.autoescape),
        )
```

```
    def get_default():
        """
        Return the first DjangoTemplates backend that's configured, or raise
        ImproperlyConfigured if none are configured.

class Engine:

    default_builtins = [

        "django.template.defaulttags",

        "django.template.defaultfilters",
        "django.template.loader_tags",
    ]

                    "app_dirs must not be set when loaders is defined."
                )
        if libraries is None:
            libraries = {}
        if builtins is None:
            builtins = []

        self.builtins = self.default_builtins + builtins
        self.template_builtins = self.get_template_builtins(self.builtins)

    def __repr__(self):
        return (
            "<%s:%s app_dirs=%s%s debug=%s loaders=%s string_if_invalid=%s "
            "file_charset=%s%s%s autoescape=%s>"
        ) % (

        # DjangoTemplates is a wrapper around this Engine class,
        # local imports are required to avoid import loops.
        from django.template import engines
        from django.template.backends.django import DjangoTemplates

    def get_template_builtins(self, builtins):
        return [import_library(x) for x in builtins]

    def get_template_libraries(self, libraries):
        loaded = {}
        for name, path in libraries.items():
            loaded[name] = import_library(path)
        return loaded

    def template_loaders(self):
        return self.get_template_loaders(self.loaders)
```

```
def get_template_loaders(self, template_loaders):
    loaders = []
    for template_loader in template_loaders:
        loader = self.find_template_loader(template_loader)
        if loader is not None:
            loaders.append(loader)
    return loaders

def find_template_loader(self, loader):
    if isinstance(loader, (tuple, list)):
        loader, *args = loader
    else:
        args = []

        loader_class = import_string(loader)

        return loader_class(self, *args)
    else:
        raise ImproperlyConfigured(
            "Invalid value in template loaders configuration: %r" % loader
        )

def find_template(self, name, dirs=None, skip=None):
    tried = []
    for loader in self.template_loaders:
        try:
            template = loader.get_template(name, skip=skip)
            return template, template.origin
        except TemplateDoesNotExist as e:
            tried.extend(e.tried)
    raise TemplateDoesNotExist(name, tried=tried)

def from_string(self, template_code):
    """
    Return a compiled Template object for the given template code,
    handling template inheritance recursively.
    """
    return Template(template_code, engine=self)

def get_template(self, template_name):
    """
    Return a compiled Template object for the given template name,
    handling template inheritance recursively.
    """
    template, origin = self.find_template(template_name)
    if not hasattr(template, "render"):
        # template needs to be compiled
        template = Template(template, origin, template_name, engine=self)
    return template
```

```
def render_to_string(self, template_name, context=None):
    """
    Render the template specified by template_name with the given context.
    For use in Django's test suite.
    """
    if isinstance(template_name, (list, tuple)):
        t = self.select_template(template_name)
    else:
        t = self.get_template(template_name)
    # Django < 1.8 accepted a Context in `context` even though that's
    # unintended. Preserve this ability but don't rewrap `context`.
    if isinstance(context, Context):
        return t.render(context)
    else:
        return t.render(Context(context, autoescape=self.autoescape))
```

Category: core_algorithms

File: venv/lib/python3.12/site-packages/pip/_vendor/resolvelib/resolvers.py

```
class ResolverException(Exception):

    """A base class for all exceptions raised by this module.

    Exceptions derived by this class should all be handled in this module. Any
    bubbling pass the resolver should be treated as a bug.
    """

class RequirementsConflicted(ResolverException):

    def __init__(self, criterion):
        super(RequirementsConflicted, self).__init__(criterion)
        self.criterion = criterion

    def __str__(self):
        return "Requirements conflict: {}".format(
            ", ".join(repr(r) for r in self.criterion.iter_requirement()),
        )

class InconsistentCandidate(ResolverException):

    def __init__(self, candidate, criterion):
        super(InconsistentCandidate, self).__init__(candidate, criterion)
        self.candidate = candidate
        self.criterion = criterion

    def __str__(self):
        return "Provided candidate {!r} does not satisfy {}".format(
            self.candidate,
            ", ".join(repr(r) for r in self.criterion.iter_requirement()),
        )

class Criterion(object):
    """Representation of possible resolution results of a package.

        This class is intended to be externally immutable. **Do not** mutate
        any of its attribute containers.
    """

    def __init__(self, candidates, information, incompatibilities):
        self.candidates = candidates
        self.information = information
        self.incompatibilities = incompatibilities
```

```
    def __repr__(self):
        requirements = ", ".join(
            "({!r}, via={!r})".format(req, parent)
            for req, parent in self.information
        )
        return "Criterion({})".format(requirements)

    def iter_requirement(self):
        return (i.requirement for i in self.information)

    def iter_parent(self):
        return (i.parent for i in self.information)

class ResolutionError(ResolverException):
    pass

class ResolutionImpossible(ResolutionError):

    def __init__(self, causes):
        super(ResolutionImpossible, self).__init__(causes)
        # causes is a list of RequirementInformation objects
        self.causes = causes

class ResolutionTooDeep(ResolutionError):

    def __init__(self, round_count):
        super(ResolutionTooDeep, self).__init__(round_count)
        self.round_count = round_count

class Resolution(object):
    """Stateful resolution object.

    def __init__(self, provider, reporter):
        self._p = provider
        self._r = reporter
        self._states = []

    def state(self):
        try:
            return self._states[-1]
        except IndexError:
            raise AttributeError("state")

    def _push_new_state(self):
        """Push a new state into history.

    def _add_to_criteria(self, criteria, requirement, parent):
        self._r.adding_requirement(requirement=requirement, parent=parent)

    def _remove_information_from_criteria(self, criteria, parents):
        """Remove information from parents of criteria.
```

```
def _get_preference(self, name):
    return self._p.get_preference(
        identifier=name,
        resolutions=self.state.mapping,
        candidates=IteratorMapping(
            self.state.criteria,
            operator.attrgetter("candidates"),
        ),
        information=IteratorMapping(
            self.state.criteria,
            operator.attrgetter("information"),
        ),
        backtrack_causes=self.state.backtrack_causes,
    )

def _is_current_pin_satisfying(self, name, criterion):
    try:
        current_pin = self.state.mapping[name]
    except KeyError:
        return False
    return all(
        self._p.is_satisfied_by(requirement=r, candidate=current_pin)
        for r in criterion.iter_requirement()
    )

def _get_updated_criteria(self, candidate):
    criteria = self.state.criteria.copy()
    for requirement in self._p.get_dependencies(candidate=candidate):
        self._add_to_criteria(criteria, requirement, parent=candidate)
    return criteria

def _attempt_to_pin_criterion(self, name):
    criterion = self.state.criteria[name]

def _backjump(self, causes):
    """Perform backjumping.
```

```
            def _patch_criteria():
                for k, incompatibilities in incompatibilities_from_broken:
                    if not incompatibilities:
                        continue
                    try:
                        criterion = self.state.criteria[k]
                    except KeyError:
                        continue
                    matches = self._p.find_matches(
                        identifier=k,
                        requirements=IteratorMapping(
                            self.state.criteria,
                            operator.methodcaller("iter_requirement"),
                        ),
                        incompatibilities=IteratorMapping(
                            self.state.criteria,
                            operator.attrgetter("incompatibilities"),
                            {k: incompatibilities},
                        ),
                    )
                    candidates = build_iter_view(matches)
                    if not candidates:
                        return False
                    incompatibilities.extend(criterion.incompatibilities)
                    self.state.criteria[k] = Criterion(
                        candidates=candidates,
                        information=list(criterion.information),
                        incompatibilities=incompatibilities,
                    )
                return True

        def resolve(self, requirements, max_rounds):
            if self._states:
                raise RuntimeError("already resolved")

    def _has_route_to_root(criteria, key, all_keys, connected):
        if key in connected:
            return True
        if key not in criteria:
            return False
        for p in criteria[key].iter_parent():
            try:
                pkey = all_keys[id(p)]
            except KeyError:
                continue
            if pkey in connected:
                connected.add(key)
                return True
            if _has_route_to_root(criteria, pkey, all_keys, connected):
                connected.add(key)
                return True
        return False
```

```
def _build_result(state):
    mapping = state.mapping
    all_keys = {id(v): k for k, v in mapping.items()}
    all_keys[id(None)] = None

class Resolver(AbstractResolver):
    """The thing that performs the actual resolution work."""

    def resolve(self, requirements, max_rounds=100):
        """Take a collection of constraints, spit out the resolution result.

        is a tuple subclass with three public members:
```

Category: core_algorithms

File: venv/lib/python3.12/site-packages/pip/_vendor/urllib3/contrib/appengine.py

```
            return None  # Defer to URLFetch's default.
        if isinstance(timeout, Timeout):
            if timeout._read is not None or timeout._connect is not None:
                warnings.warn(
                    "URLFetch does not support granular timeout settings, "
```

```
1. You can use :class:`AppEngineManager` with URLFetch. URLFetch is
   cost-effective in many circumstances as long as your usage is within the
   limitations.

2. You can use a normal :class:`~urllib3.PoolManager` by enabling sockets.
   Sockets also have `limitations and restrictions
   <https://cloud.google.com/appengine/docs/python/sockets/\
   #limitations-and-restrictions>`_ and have a lower free quota than URLFetch.
   To use sockets, be sure to specify the following in your ``app.yaml``::

:class:`PoolManager` without any configuration or special environment variables.
"""


class AppEnginePlatformWarning(HTTPWarning):
    pass

class AppEnginePlatformError(HTTPError):
    pass

class AppEngineManager(RequestMethods):
    """
    Connection manager for Google App Engine sandbox applications.

    Notably it will raise an :class:`AppEnginePlatformError` if:
        * URLFetch is not available.
        * If you attempt to use this on App Engine Flexible, as full socket
          support is available.
        * If a request size is more than 10 megabytes.
        * If a response size is more than 32 megabytes.
        * If you use an unsupported request method such as OPTIONS.

    def __init__(
        self,
        headers=None,
        retries=None,
        validate_certificate=True,
        urlfetch_retries=True,
    ):
        if not urlfetch:
            raise AppEnginePlatformError(
                "URLFetch is not available in this environment."
            )
```

```
    self.retries = retries or Retry.DEFAULT

def __enter__(self):
    return self

def __exit__(self, exc_type, exc_val, exc_tb):
    # Return False to re-raise any potential exceptions
    return False

def urlopen(
    self,
    method,
    url,
    body=None,
    headers=None,
    retries=None,
    redirect=True,

    timeout=Timeout.DEFAULT_TIMEOUT,
    **response_kw
):

def _urlfetch_response_to_http_response(self, urlfetch_resp, **response_kw):

        # Production GAE handles deflate encoding automatically, but does
        # not remove the encoding header.
        content_encoding = urlfetch_resp.headers.get("content-encoding")

        if content_encoding == "deflate":
            del urlfetch_resp.headers["content-encoding"]

def _get_absolute_timeout(self, timeout):

    if timeout is Timeout.DEFAULT_TIMEOUT:

                "reverting to total or default URLFetch timeout.",
                AppEnginePlatformWarning,
            )
        return timeout.total
    return timeout

def _get_retries(self, retries, redirect):
    if not isinstance(retries, Retry):

        retries = Retry.from_int(retries, redirect=redirect, default=self.retries)
```

Category: core_algorithms

File: venv/lib/python3.12/site-packages/pip/_internal/resolution/legacy/resolver.py

```
from collections import defaultdict
from itertools import chain

from typing import DefaultDict, Iterable, List, Optional, Set, Tuple

DiscoveredDependencies = DefaultDict[str, List[InstallRequirement]]

def _check_dist_requires_python(
    dist: BaseDistribution,
    version_info: Tuple[int, int, int],
    ignore_requires_python: bool = False,
) -> None:
    """
    Check whether the given Python version is compatible with a distribution's
    "Requires-Python" value.

class Resolver(BaseResolver):
    """Resolves which packages need to be installed/uninstalled to perform \
    the requested operation without breaking the requirements of any package.
    """

    def __init__(
        self,
        preparer: RequirementPreparer,
        finder: PackageFinder,
        wheel_cache: Optional[WheelCache],
        make_install_req: InstallRequirementProvider,
        use_user_site: bool,
        ignore_dependencies: bool,
        ignore_installed: bool,
        ignore_requires_python: bool,
        force_reinstall: bool,
        upgrade_strategy: str,
        py_version_info: Optional[Tuple[int, ...]] = None,
    ) -> None:
        super().__init__()
        assert upgrade_strategy in self._allowed_strategies

        self._discovered_dependencies: DiscoveredDependencies = defaultdict(list)

    def resolve(
        self, root_reqs: List[InstallRequirement], check_supported_wheels: bool
    ) -> RequirementSet:
        """Resolve what operations need to be done
```

```
def _add_requirement_to_set(
    self,
    requirement_set: RequirementSet,
    install_req: InstallRequirement,
    parent_req_name: Optional[str] = None,
    extras_requested: Optional[Iterable[str]] = None,
) -> Tuple[List[InstallRequirement], Optional[InstallRequirement]]:
    """Add install_req as a requirement to install.

def _is_upgrade_allowed(self, req: InstallRequirement) -> bool:
    if self.upgrade_strategy == "to-satisfy-only":
        return False
    elif self.upgrade_strategy == "eager":
        return True
    else:
        assert self.upgrade_strategy == "only-if-needed"
        return req.user_supplied or req.constraint

def _set_req_to_reinstall(self, req: InstallRequirement) -> None:
    """
    Set a requirement to be installed.
    """
    # Don't uninstall the conflict if doing a user install and the
    # conflict is not a user install.
    if not self.use_user_site or req.satisfied_by.in_usersite:
        req.should_reinstall = True
    req.satisfied_by = None

def _check_skip_installed(
    self, req_to_install: InstallRequirement
) -> Optional[str]:
    """Check if req_to_install should be skipped.

def _find_requirement_link(self, req: InstallRequirement) -> Optional[Link]:
    upgrade = self._is_upgrade_allowed(req)
    best_candidate = self.finder.find_requirement(req, upgrade)
    if not best_candidate:
        return None

def _populate_link(self, req: InstallRequirement) -> None:
    """Ensure that if a link can be found for this, that it is found.

def _get_dist_for(self, req: InstallRequirement) -> BaseDistribution:
    """Takes a InstallRequirement and returns a single AbstractDist \
    representing a prepared variant of the same.
    """
    if req.editable:
        return self.preparer.prepare_editable_requirement(req)
```

```
def _resolve_one(
    self,
    requirement_set: RequirementSet,
    req_to_install: InstallRequirement,
) -> List[InstallRequirement]:
    """Prepare a single requirements file.

    def add_req(subreq: Requirement, extras_requested: Iterable[str]) -> None:
        # This idiosyncratically converts the Requirement to str and let
        # make_install_req then parse it again into Requirement. But this is
        # the legacy resolver so I'm just not going to bother refactoring.
        sub_install_req = self._make_install_req(str(subreq), req_to_install)
        parent_req_name = req_to_install.name
        to_scan_again, add_to_parent = self._add_requirement_to_set(
            requirement_set,
            sub_install_req,
            parent_req_name=parent_req_name,
            extras_requested=extras_requested,
        )
        if parent_req_name and add_to_parent:
            self._discovered_dependencies[parent_req_name].append(add_to_parent)
        more_reqs.extend(to_scan_again)

def get_installation_order(
    self, req_set: RequirementSet
) -> List[InstallRequirement]:
    """Create the installation order.

    def schedule(req: InstallRequirement) -> None:
        if req.satisfied_by or req in ordered_reqs:
            return
        if req.constraint:
            return
        ordered_reqs.add(req)
        for dep in self._discovered_dependencies[req.name]:
            schedule(dep)
        order.append(req)
```

Category: core_algorithms

File: venv/lib/python3.12/site-packages/pip/_internal/resolution/resolvelib/resolver.py

```
class Resolver(BaseResolver):
    _allowed_strategies = {"eager", "only-if-needed", "to-satisfy-only"}

    def __init__(
        self,
        preparer: RequirementPreparer,
        finder: PackageFinder,
        wheel_cache: Optional[WheelCache],
        make_install_req: InstallRequirementProvider,
        use_user_site: bool,
        ignore_dependencies: bool,
        ignore_installed: bool,
        ignore_requires_python: bool,
        force_reinstall: bool,
        upgrade_strategy: str,
        py_version_info: Optional[Tuple[int, ...]] = None,
    ):
        super().__init__()
        assert upgrade_strategy in self._allowed_strategies

    def resolve(
        self, root_reqs: List[InstallRequirement], check_supported_wheels: bool
    ) -> RequirementSet:
        collected = self.factory.collect_root_requirements(root_reqs)
        provider = PipProvider(
            factory=self.factory,
            constraints=collected.constraints,
            ignore_dependencies=self.ignore_dependencies,
            upgrade_strategy=self.upgrade_strategy,
            user_requested=collected.user_requested,
        )
        if "PIP_RESOLVER_DEBUG" in os.environ:
            reporter: BaseReporter = PipDebuggingReporter()
        else:
            reporter = PipReporter()
        resolver: RLResolver[Requirement, Candidate, str] = RLResolver(
            provider,
            reporter,
        )

    def get_installation_order(
        self, req_set: RequirementSet
    ) -> List[InstallRequirement]:
        """Get order for installation of requirements in RequirementSet.

def get_topological_weights(
    graph: "DirectedGraph[Optional[str]]", requirement_keys: Set[str]
) -> Dict[Optional[str], int]:
    """Assign weights to each node based on how "deep" they are.
```

```
    def visit(node: Optional[str]) -> None:
        if node in path:
            # We hit a cycle, so we'll break it here.
            return

def _req_set_item_sorter(
    item: Tuple[str, InstallRequirement],
    weights: Dict[Optional[str], int],
) -> Tuple[int, str]:
    """Key function used to sort install requirements for installation.
```

Category: core_algorithms

File: venv/lib/python3.12/site-packages/pip/_vendor/urllib3/contrib/_appengine_environ.py

```
def is_appengine():
    return is_local_appengine() or is_prod_appengine()

def is_appengine_sandbox():
    """Reports if the app is running in the first generation sandbox.

def is_local_appengine():
    return "APPENGINE_RUNTIME" in os.environ and os.environ.get(
        "SERVER_SOFTWARE", ""
    ).startswith("Development/")

def is_prod_appengine():
    return "APPENGINE_RUNTIME" in os.environ and os.environ.get(
        "SERVER_SOFTWARE", ""
    ).startswith("Google App Engine/")
```

Category: core_models

File: venv/lib/python3.12/site-packages/django/forms/models.py

```
class ModelFormMetaclass(DeclarativeFieldsMetaclass):

class ModelForm(BaseModelForm, metaclass=ModelFormMetaclass):
    pass

def modelform_defines_fields(form_class):

        if field_classes and f.name in field_classes:

            kwargs["form_class"] = field_classes[f.name]

        self.field_classes = getattr(options, "field_classes", None)
        self.formfield_callback = getattr(options, "formfield_callback", None)

        opts = new_class._meta = ModelFormOptions(getattr(new_class, "Meta", None))

        attrs["field_classes"] = field_classes

    # Class attributes for the new form class.

    return type(form)(class_name, (form,), form_class_attrs)

    """Return a FormSet class for the given Django model class."""
    meta = getattr(form, "Meta", None)
    if (
        getattr(meta, "fields", fields) is None
        and getattr(meta, "exclude", exclude) is None
    ):
        raise ImproperlyConfigured(

        field_classes=field_classes,
    )
    FormSet = formset_factory(
        form,
        formset,
        extra=extra,
        min_num=min_num,
        max_num=max_num,
        can_order=can_order,
        can_delete=can_delete,
        validate_min=validate_min,
        validate_max=validate_max,
        absolute_max=absolute_max,
        can_delete_extra=can_delete_extra,
        renderer=renderer,
    )
    FormSet.model = model
    FormSet.edit_only = edit_only
    return FormSet
```

```
    def get_default_prefix(cls):
        return cls.fk.remote_field.get_accessor_name(model=cls.model).replace("+", "")

        "field_classes": field_classes,
        "absolute_max": absolute_max,
        "can_delete_extra": can_delete_extra,
        "renderer": renderer,
        "edit_only": edit_only,
    }
    FormSet = modelformset_factory(model, **kwargs)
    FormSet.fk = fk
    return FormSet

    # This class is a subclass of ChoiceField for purity, but it doesn't
    # actually use any of ChoiceField's implementation.

Helper functions for creating Form classes from Django models
and database field objects.
"""
from itertools import chain

from django.forms.forms import BaseForm, DeclarativeFieldsMetaclass
from django.forms.formsets import BaseFormSet, formset_factory
from django.forms.utils import ErrorList
from django.forms.widgets import (
    HiddenInput,
    MultipleHiddenInput,
    RadioSelect,
    SelectMultiple,
)
from django.utils.text import capfirst, get_text_list
from django.utils.translation import gettext
from django.utils.translation import gettext_lazy as _

def construct_instance(form, instance, fields=None, exclude=None):
    """
    Construct and return a model instance from the bound ``form``'s
    ``cleaned_data``, but do not save the returned instance to the database.
    """
    from django.db import models

        # Leave defaults for fields that aren't in POST data, except for
        # checkbox inputs because they don't appear in POST data if not checked.
        if (

            f.has_default()
            and form[f.name].field.widget.value_omitted_from_data(
                form.data, form.files, form.add_prefix(f.name)
            )
            and cleaned_data.get(f.name) in form[f.name].field.empty_values
        ):
            continue
```

```
        # Defer saving file-type fields until after the other fields, so a
        # callable upload_to can use the values from other fields.
        if isinstance(f, models.FileField):
            file_field_list.append(f)
        else:
            f.save_form_data(instance, cleaned_data[f.name])

def model_to_dict(instance, fields=None, exclude=None):
    """
    Return a dict containing the data in ``instance`` suitable for passing as
    a Form's ``initial`` keyword argument.

def apply_limit_choices_to_to_formfield(formfield):
    """Apply limit_choices_to to the formfield's queryset if needed."""
    from django.db.models import Exists, OuterRef, Q

def fields_for_model(
    model,
    fields=None,
    exclude=None,
    widgets=None,
    formfield_callback=None,
    localized_fields=None,
    labels=None,
    help_texts=None,
    error_messages=None,

    field_classes=None,
    *,
    apply_limit_choices_to=True,
):
    """
    Return a dictionary containing form fields for the given model.

    ``field_classes`` is a dictionary of model field names mapped to a form

    field class.

class ModelFormOptions:

    def __init__(self, options=None):
        self.model = getattr(options, "model", None)
        self.fields = getattr(options, "fields", None)
        self.exclude = getattr(options, "exclude", None)
        self.widgets = getattr(options, "widgets", None)
        self.localized_fields = getattr(options, "localized_fields", None)
        self.labels = getattr(options, "labels", None)
        self.help_texts = getattr(options, "help_texts", None)
        self.error_messages = getattr(options, "error_messages", None)

    def __new__(mcs, name, bases, attrs):
```

```
        new_class = super().__new__(mcs, name, bases, attrs)

            return new_class

                        "model": new_class.__name__,
                        "opt": opt,
                        "value": value,
                    }
                )
                raise TypeError(msg)

            # If a model is defined, extract form fields from it.
            if opts.fields is None and opts.exclude is None:
                raise ImproperlyConfigured(
                    "Creating a ModelForm without either the 'fields' attribute "
                    "or the 'exclude' attribute is prohibited; form %s "
                    "needs updating." % name
                )

                opts.field_classes,
                # limit_choices_to will be applied during ModelForm.__init__().
                apply_limit_choices_to=False,
            )

            missing_fields = none_model_fields.difference(new_class.declared_fields)
            if missing_fields:
                message = "Unknown field(s) (%s) specified for %s"
                message %= (', '.join(missing_fields), opts.model.__name__)
                raise FieldError(message)

            # Override default model fields with any custom declared ones
            # (plus, include all the other declared fields).

            fields.update(new_class.declared_fields)
        else:

            fields = new_class.declared_fields

        new_class.base_fields = fields

        return new_class

class BaseModelForm(BaseForm, AltersData):

    def __init__(
        self,
        data=None,
        files=None,
        auto_id="id_%s",
        prefix=None,
        initial=None,
```

```
    error_class=ErrorList,
    label_suffix=None,
    empty_permitted=False,
    instance=None,
    use_required_attribute=None,
    renderer=None,
):
    opts = self._meta
    if opts.model is None:

        raise ValueError("ModelForm has no model class specified.")
    if instance is None:
        # if we didn't get an instance, instantiate a new one
        self.instance = opts.model()
        object_data = {}
    else:
        self.instance = instance
        object_data = model_to_dict(instance, opts.fields, opts.exclude)
    # if initial was provided, it should override the values from instance
    if initial is not None:
        object_data.update(initial)
    # self._validate_unique will be set to True by BaseModelForm.clean().

    # It is False by default so overriding self.clean() and failing to call
    # super will stop validate_unique from being called.
    self._validate_unique = False
    super().__init__(
        data,
        files,
        auto_id,
        prefix,
        object_data,

        error_class,
        label_suffix,
        empty_permitted,
        use_required_attribute=use_required_attribute,
        renderer=renderer,
    )
    for formfield in self.fields.values():
        apply_limit_choices_to_to_formfield(formfield)
```

```
def _get_validation_exclusions(self):
    """
    For backwards-compatibility, exclude several types of fields from model
    validation. See tickets #12507, #12521, #12553.
    """
    exclude = set()
    # Build up a list of fields that should be excluded from model field
    # validation and unique checks.
    for f in self.instance._meta.fields:
        field = f.name
        # Exclude fields that aren't on the form. The developer may be
        # adding these values to the model after form validation.
        if field not in self.fields:
            exclude.add(f.name)

        # Don't perform model validation on fields that were defined
        # manually on the form and excluded via the ModelForm's Meta

        # class. See #12901.
        elif self._meta.fields and field not in self._meta.fields:
            exclude.add(f.name)
        elif self._meta.exclude and field in self._meta.exclude:
            exclude.add(f.name)

def clean(self):
    self._validate_unique = True
    return self.cleaned_data

def _update_errors(self, errors):

    # Override any validation error messages defined at the model level

    # with those defined at the form level.
    opts = self._meta

def _post_clean(self):
    opts = self._meta

def validate_unique(self):
    """
    Call the instance's validate_unique() method and update the form's
    validation errors if any were raised.
    """
    exclude = self._get_validation_exclusions()
    try:
        self.instance.validate_unique(exclude=exclude)
    except ValidationError as e:
        self._update_errors(e)
```

```
def _save_m2m(self):
    """
    Save the many-to-many fields and generic relations for this form.
    """
    cleaned_data = self.cleaned_data
    exclude = self._meta.exclude
    fields = self._meta.fields
    opts = self.instance._meta
    # Note that for historical reasons we want to include also
    # private_fields here. (GenericRelation was previously a fake
    # m2m field).
    for f in chain(opts.many_to_many, opts.private_fields):
        if not hasattr(f, "save_form_data"):
            continue
        if fields and f.name not in fields:
            continue
        if exclude and f.name in exclude:
            continue
        if f.name in cleaned_data:
            f.save_form_data(self.instance, cleaned_data[f.name])

def save(self, commit=True):
    """
    Save this form's self.instance object if commit=True. Otherwise, add
    a save_m2m() method to the form which can be called after the instance
    is saved manually at a later time. Return the model instance.
    """
    if self.errors:
        raise ValueError(
            "The %s could not be %s because the data didn't validate."
            % (
                self.instance._meta.object_name,
                "created" if self.instance._state.adding else "changed",
            )
        )
    if commit:
        # If committing, save the instance and the m2m data immediately.
        self.instance.save()
        self._save_m2m()
    else:

        # If not committing, add a method to the form to allow deferred
        # saving of m2m data.
        self.save_m2m = self._save_m2m
    return self.instance
```

```
def modelform_factory(
    model,
    form=ModelForm,
    fields=None,
    exclude=None,
    formfield_callback=None,
    widgets=None,
    localized_fields=None,
    labels=None,
    help_texts=None,
    error_messages=None,

    field_classes=None,
):
    """
    Return a ModelForm containing form fields for the given model. You can
    optionally pass a `form` argument to use as a starting point for
    constructing the ModelForm.

    ``field_classes`` is a dictionary of model field names mapped to a form

    field class.
    """

    # Create the inner Meta class. FIXME: ideally, we should be able to
    # construct a ModelForm without creating and passing in a temporary

    # inner class.

    if field_classes is not None:

    # If parent form class already has an inner Meta, the Meta we're
    # creating needs to inherit from the parent's inner meta.
    bases = (form.Meta,) if hasattr(form, "Meta") else ()
    Meta = type("Meta", bases, attrs)
    if formfield_callback:
        Meta.formfield_callback = staticmethod(formfield_callback)

    # Give this new form class a reasonable name.

    class_name = model.__name__ + "Form"

    form_class_attrs = {"Meta": Meta}

            "Calling modelform_factory without defining 'fields' or "
            "'exclude' explicitly is prohibited."
        )

    # Instantiate type(form) in order to use the same metaclass as form.
```

```
class BaseModelFormSet(BaseFormSet, AltersData):
    """
    A ``FormSet`` for editing a queryset and/or adding new objects to it.
    """

    def __init__(
        self,
        data=None,
        files=None,
        auto_id="id_%s",
        prefix=None,
        queryset=None,
        *,
        initial=None,
        **kwargs,
    ):
        self.queryset = queryset
        self.initial_extra = initial
        super().__init__(
            **{
                "data": data,
                "files": files,
                "auto_id": auto_id,
                "prefix": prefix,
                **kwargs,
            }
        )

    def initial_form_count(self):
        """Return the number of forms that are required in this FormSet."""
        if not self.is_bound:
            return len(self.get_queryset())
        return super().initial_form_count()

    def _existing_object(self, pk):
        if not hasattr(self, "_object_dict"):
            self._object_dict = {o.pk: o for o in self.get_queryset()}
        return self._object_dict.get(pk)

    def _get_to_python(self, field):
        """
        If the field is a related field, fetch the concrete field's (that
        is, the ultimate pointed-to field's) to_python.
        """
        while field.remote_field is not None:
            field = field.remote_field.get_related_field()
        return field.to_python
```

```
def _construct_form(self, i, **kwargs):
    pk_required = i < self.initial_form_count()
    if pk_required:
        if self.is_bound:
            pk_key = "%s-%s" % (self.add_prefix(i), self.model._meta.pk.name)
            try:
                pk = self.data[pk_key]
            except KeyError:
                # The primary key is missing. The user may have tampered
                # with POST data.
                pass
            else:
                to_python = self._get_to_python(self.model._meta.pk)
                try:
                    pk = to_python(pk)
                except ValidationError:
                    # The primary key exists but is an invalid value. The
                    # user may have tampered with POST data.
                    pass
                else:
                    kwargs["instance"] = self._existing_object(pk)
        else:
            kwargs["instance"] = self.get_queryset()[i]
    elif self.initial_extra:
        # Set initial values for extra forms
        try:
            kwargs["initial"] = self.initial_extra[i - self.initial_form_count()]
        except IndexError:
            pass
    form = super()._construct_form(i, **kwargs)
    if pk_required:
        form.fields[self.model._meta.pk.name].required = True
    return form

def get_queryset(self):
    if not hasattr(self, "_queryset"):
        if self.queryset is not None:
            qs = self.queryset
        else:

            qs = self.model._default_manager.get_queryset()

def save_new(self, form, commit=True):
    """Save and return a new model instance for the given form."""
    return form.save(commit=commit)

def save_existing(self, form, instance, commit=True):
    """Save and return an existing model instance for the given form."""
    return form.save(commit=commit)
```

```
def delete_existing(self, obj, commit=True):
    """Deletes an existing model instance."""
    if commit:
        obj.delete()

def save(self, commit=True):
    """
    Save model instances for every form, adding and changing instances
    as necessary, and return the list of instances.
    """
    if not commit:
        self.saved_forms = []

        def save_m2m():
            for form in self.saved_forms:
                form.save_m2m()

def clean(self):
    self.validate_unique()

def validate_unique(self):
    # Collect unique_checks and date_checks to run from all the forms.
    all_unique_checks = set()
    all_date_checks = set()
    forms_to_delete = self.deleted_forms
    valid_forms = [
        form
        for form in self.forms
        if form.is_valid() and form not in forms_to_delete
    ]
    for form in valid_forms:
        exclude = form._get_validation_exclusions()
        unique_checks, date_checks = form.instance._get_unique_checks(
            exclude=exclude,
            include_meta_constraints=True,
        )
        all_unique_checks.update(unique_checks)
        all_date_checks.update(date_checks)
```

```
for uclass, unique_check in all_unique_checks:
    seen_data = set()
    for form in valid_forms:
        # Get the data for the set of fields that must be unique among
        # the forms.
        row_data = (
            field if field in self.unique_fields else form.cleaned_data[field]
            for field in unique_check
            if field in form.cleaned_data
        )
        # Reduce Model instances to their primary key values
        row_data = tuple(
            d._get_pk_val() if hasattr(d, "_get_pk_val")
            # Prevent "unhashable type: list" errors later on.
            else tuple(d) if isinstance(d, list) else d
            for d in row_data
        )
        if row_data and None not in row_data:
            # if we've already seen it then we have a uniqueness failure
            if row_data in seen_data:
                # poke error messages into the right places and mark
                # the form as invalid
                errors.append(self.get_unique_error_message(unique_check))

                form._errors[NON_FIELD_ERRORS] = self.error_class(
                    [self.get_form_error()],
                    renderer=self.renderer,
                )
                # Remove the data from the cleaned_data dict since it
                # was invalid.
                for field in unique_check:
                    if field in form.cleaned_data:
                        del form.cleaned_data[field]
            # mark the data as seen
            seen_data.add(row_data)
# iterate over each of the date checks now
for date_check in all_date_checks:
    seen_data = set()
```

```
        uclass, lookup, field, unique_for = date_check
        for form in valid_forms:
            # see if we have data for both fields
            if (
                form.cleaned_data
                and form.cleaned_data[field] is not None
                and form.cleaned_data[unique_for] is not None
            ):
                # if it's a date lookup we need to get the data for all the fields
                if lookup == "date":
                    date = form.cleaned_data[unique_for]
                    date_data = (date.year, date.month, date.day)
                # otherwise it's just the attribute on the date/datetime
                # object
                else:
                    date_data = (getattr(form.cleaned_data[unique_for], lookup),)
                data = (form.cleaned_data[field],) + date_data
                # if we've already seen it then we have a uniqueness failure
                if data in seen_data:
                    # poke error messages into the right places and mark
                    # the form as invalid
                    errors.append(self.get_date_error_message(date_check))

                    form._errors[NON_FIELD_ERRORS] = self.error_class(
                        [self.get_form_error()],
                        renderer=self.renderer,
                    )
                    # Remove the data from the cleaned_data dict since it
                    # was invalid.
                    del form.cleaned_data[field]
                # mark the data as seen
                seen_data.add(data)

def get_unique_error_message(self, unique_check):
    if len(unique_check) == 1:
        return gettext("Please correct the duplicate data for %(field)s.") % {
            "field": unique_check[0],
        }
    else:
        return gettext(
            "Please correct the duplicate data for %(field)s, which must be unique."
        ) % {
            "field": get_text_list(unique_check, _("and")),
        }
```

```
def get_date_error_message(self, date_check):
    return gettext(
        "Please correct the duplicate data for %(field_name)s "
        "which must be unique for the %(lookup)s in %(date_field)s."
    ) % {
        "field_name": date_check[2],
        "date_field": date_check[3],
        "lookup": str(date_check[1]),
    }

def get_form_error(self):
    return gettext("Please correct the duplicate values below.")

def save_existing_objects(self, commit=True):
    self.changed_objects = []
    self.deleted_objects = []
    if not self.initial_forms:
        return []

def save_new_objects(self, commit=True):
    self.new_objects = []
    for form in self.extra_forms:
        if not form.has_changed():
            continue
        # If someone has marked an add form for deletion, don't save the
        # object.
        if self.can_delete and self._should_delete_form(form):
            continue
        self.new_objects.append(self.save_new(form, commit=commit))
        if not commit:
            self.saved_forms.append(form)
    return self.new_objects

def add_fields(self, form, index):
    """Add a hidden field for the object's primary key."""
    from django.db.models import AutoField, ForeignKey, OneToOneField

    def pk_is_not_editable(pk):
        return (
            (not pk.editable)
            or (pk.auto_created or isinstance(pk, AutoField))
            or (
                pk.remote_field
                and pk.remote_field.parent_link
                and pk_is_not_editable(pk.remote_field.model._meta.pk)
            )
        )
```

```
                # as it could be an auto-generated default which isn't actually
                # in the database.
                pk_value = None if form.instance._state.adding else form.instance.pk
            else:
                try:
                    if index is not None:
                        pk_value = self.get_queryset()[index].pk
                    else:
                        pk_value = None
                except IndexError:
                    pk_value = None
            if isinstance(pk, (ForeignKey, OneToOneField)):

                qs = pk.remote_field.model._default_manager.get_queryset()
            else:

                qs = self.model._default_manager.get_queryset()
            qs = qs.using(form.instance._state.db)
            if form._meta.widgets:
                widget = form._meta.widgets.get(self._pk_field.name, HiddenInput)
            else:
                widget = HiddenInput
            form.fields[self._pk_field.name] = ModelChoiceField(
                qs, initial=pk_value, required=False, widget=widget
            )
        super().add_fields(form, index)

def modelformset_factory(
    model,
    form=ModelForm,
    formfield_callback=None,
    formset=BaseModelFormSet,
    extra=1,
    can_delete=False,
    can_order=False,
    max_num=None,
    fields=None,
    exclude=None,
    widgets=None,
    validate_max=False,
    localized_fields=None,
    labels=None,
    help_texts=None,
    error_messages=None,
    min_num=None,
    validate_min=False,

    field_classes=None,
    absolute_max=None,
    can_delete_extra=True,
    renderer=None,
    edit_only=False,
):
```

```
            "Calling modelformset_factory without defining 'fields' or "
            "'exclude' explicitly is prohibited."
        )

class BaseInlineFormSet(BaseModelFormSet):
    """A formset for child objects related to a parent."""

    def __init__(
        self,
        data=None,
        files=None,
        instance=None,
        save_as_new=False,
        prefix=None,
        queryset=None,
        **kwargs,
    ):
        if instance is None:
            self.instance = self.fk.remote_field.model()
        else:
            self.instance = instance
        self.save_as_new = save_as_new
        if queryset is None:

            queryset = self.model._default_manager
        if self.instance.pk is not None:
            qs = queryset.filter(**{self.fk.name: self.instance})
        else:
            qs = queryset.none()
        self.unique_fields = {self.fk.name}
        super().__init__(data, files, prefix=prefix, queryset=qs, **kwargs)

        # Add the generated field to form._meta.fields if it's defined to make
        # sure validation isn't skipped on that field.
        if self.form._meta.fields and self.fk.name not in self.form._meta.fields:
            if isinstance(self.form._meta.fields, tuple):
                self.form._meta.fields = list(self.form._meta.fields)
            self.form._meta.fields.append(self.fk.name)

    def initial_form_count(self):
        if self.save_as_new:
            return 0
        return super().initial_form_count()
```

```
def _construct_form(self, i, **kwargs):
    form = super()._construct_form(i, **kwargs)
    if self.save_as_new:
        mutable = getattr(form.data, "_mutable", None)
        # Allow modifying an immutable QueryDict.
        if mutable is not None:
            form.data._mutable = True
        # Remove the primary key from the form's data, we are only
        # creating new instances
        form.data[form.add_prefix(self._pk_field.name)] = None
        # Remove the foreign key from the form's data
        form.data[form.add_prefix(self.fk.name)] = None
        if mutable is not None:
            form.data._mutable = mutable

@classmethod

def save_new(self, form, commit=True):
    # Ensure the latest copy of the related instance is present on each
    # form (it may have been saved after the formset was originally
    # instantiated).
    setattr(form.instance, self.fk.name, self.instance)
    return super().save_new(form, commit=commit)

def add_fields(self, form, index):
    super().add_fields(form, index)
    if self._pk_field == self.fk:
        name = self._pk_field.name
        kwargs = {"pk_field": True}
    else:
        # The foreign key field might not be on the form, so we poke at the
        # Model field to get the label, since we need that for error messages.
        name = self.fk.name
        kwargs = {
            "label": getattr(
                form.fields.get(name), "label", capfirst(self.fk.verbose_name)
            )
        }

        if to_field.has_default():
            setattr(self.instance, to_field.attname, None)

def get_unique_error_message(self, unique_check):
    unique_check = [field for field in unique_check if field != self.fk.name]
    return super().get_unique_error_message(unique_check)
```

```
def _get_foreign_key(parent_model, model, fk_name=None, can_fail=False):
    """
    Find and return the ForeignKey from model to parent if there is one
    (return None if can_fail is True and no such field exists). If fk_name is
    provided, assume it is the name of the ForeignKey field. Unless can_fail is
    True, raise an exception if there isn't a ForeignKey from model to
    parent_model.
    """
    # avoid circular import
    from django.db.models import ForeignKey

def inlineformset_factory(
    parent_model,
    model,
    form=ModelForm,
    formset=BaseInlineFormSet,
    fk_name=None,
    fields=None,
    exclude=None,
    extra=3,
    can_order=False,
    can_delete=True,
    max_num=None,
    formfield_callback=None,
    widgets=None,
    validate_max=False,
    localized_fields=None,
    labels=None,
    help_texts=None,
    error_messages=None,
    min_num=None,
    validate_min=False,

    field_classes=None,
    absolute_max=None,
    can_delete_extra=True,
    renderer=None,
    edit_only=False,
):
    """
    Return an ``InlineFormSet`` for the given kwargs.

class InlineForeignKeyField(Field):
    """
    A basic integer field that deals with validating the given value to a
    given parent instance in an inline.
    """

    default_error_messages = {
        "invalid_choice": _("The inline value did not match the parent instance."),
    }
```

```
    def __init__(self, parent_instance, *args, pk_field=False, to_field=None, **kwargs):
        self.parent_instance = parent_instance
        self.pk_field = pk_field
        self.to_field = to_field
        if self.parent_instance is not None:
            if self.to_field:
                kwargs["initial"] = getattr(self.parent_instance, self.to_field)
            else:
                kwargs["initial"] = self.parent_instance.pk
        kwargs["required"] = False
        super().__init__(*args, **kwargs)

    def clean(self, value):
        if value in self.empty_values:
            if self.pk_field:
                return None
            # if there is no value act as we did before.
            return self.parent_instance
        # ensure the we compare the values as equal types.
        if self.to_field:
            orig = getattr(self.parent_instance, self.to_field)
        else:
            orig = self.parent_instance.pk
        if str(value) != str(orig):
            raise ValidationError(
                self.error_messages["invalid_choice"], code="invalid_choice"
            )
        return self.parent_instance

    def has_changed(self, initial, data):
        return False

class ModelChoiceIteratorValue:

    def __init__(self, value, instance):
        self.value = value
        self.instance = instance

    def __str__(self):
        return str(self.value)

    def __hash__(self):
        return hash(self.value)

    def __eq__(self, other):
        if isinstance(other, ModelChoiceIteratorValue):
            other = other.value
        return self.value == other

class ModelChoiceIterator:
```

```
    def __init__(self, field):
        self.field = field
        self.queryset = field.queryset

    def __iter__(self):
        if self.field.empty_label is not None:
            yield ("", self.field.empty_label)
        queryset = self.queryset
        # Can't use iterator() when queryset uses prefetch_related()
        if not queryset._prefetch_related_lookups:
            queryset = queryset.iterator()
        for obj in queryset:
            yield self.choice(obj)

    def __len__(self):
        # count() adds a query but uses less memory since the QuerySet results
        # won't be cached. In most cases, the choices will only be iterated on,
        # and __len__() won't be called.
        return self.queryset.count() + (1 if self.field.empty_label is not None else 0)

    def __bool__(self):
        return self.field.empty_label is not None or self.queryset.exists()

    def choice(self, obj):
        return (
            ModelChoiceIteratorValue(self.field.prepare_value(obj), obj),
            self.field.label_from_instance(obj),
        )

class ModelChoiceField(ChoiceField):
    """A ChoiceField whose choices are a model QuerySet."""

    default_error_messages = {
        "invalid_choice": _(
            "Select a valid choice. That choice is not one of the available choices."
        ),
    }
    iterator = ModelChoiceIterator
```

```
def __init__(
    self,
    queryset,
    *,
    empty_label="---------",
    required=True,
    widget=None,
    label=None,
    initial=None,
    help_text="",
    to_field_name=None,
    limit_choices_to=None,
    blank=False,
    **kwargs,
):
    # Call Field instead of ChoiceField __init__() because we don't need
    # ChoiceField.__init__().
    Field.__init__(
        self,
        required=required,
        widget=widget,
        label=label,
        initial=initial,
        help_text=help_text,
        **kwargs,
    )
    if (required and initial is not None) or (
        isinstance(self.widget, RadioSelect) and not blank
    ):
        self.empty_label = None
    else:
        self.empty_label = empty_label
    self.queryset = queryset
    self.limit_choices_to = limit_choices_to  # limit the queryset later.
    self.to_field_name = to_field_name

def get_limit_choices_to(self):
    """
    Return ``limit_choices_to`` for this form field.

def __deepcopy__(self, memo):
    result = super(ChoiceField, self).__deepcopy__(memo)
    # Need to force a new ModelChoiceIterator to be created, bug #11183
    if self.queryset is not None:
        result.queryset = self.queryset.all()
    return result

def _get_queryset(self):
    return self._queryset

def _set_queryset(self, queryset):
    self._queryset = None if queryset is None else queryset.all()
    self.widget.choices = self.choices
```

```
    def label_from_instance(self, obj):
        """
        Convert objects into strings and generate the labels for the choices

        presented by this object. Subclasses can override this method to
        customize the display of the choices.
        """
        return str(obj)

    def _get_choices(self):
        # If self._choices is set, then somebody must have manually set
        # the property self.choices. In this case, just return self._choices.
        if hasattr(self, '_choices'):
            return self._choices

    def prepare_value(self, value):
        if hasattr(value, '_meta'):
            if self.to_field_name:
                return value.serializable_value(self.to_field_name)
            else:
                return value.pk
        return super().prepare_value(value)

    def to_python(self, value):
        if value in self.empty_values:
            return None
        try:
            key = self.to_field_name or 'pk'
            if isinstance(value, self.queryset.model):
                value = getattr(value, key)
            value = self.queryset.get(**{key: value})
        except (ValueError, TypeError, self.queryset.model.DoesNotExist):
            raise ValidationError(
                self.error_messages['invalid_choice'],
                code='invalid_choice',
                params={'value': value},
            )
        return value

    def validate(self, value):
        return Field.validate(self, value)

    def has_changed(self, initial, data):
        if self.disabled:
            return False
        initial_value = initial if initial is not None else ''
        data_value = data if data is not None else ''
        return str(self.prepare_value(initial_value)) != str(data_value)

class ModelMultipleChoiceField(ModelChoiceField):
    """A MultipleChoiceField whose choices are a model QuerySet."""
```

```
default_error_messages = {
    "invalid_list": _("Enter a list of values."),
    "invalid_choice": _(
        "Select a valid choice. %(value)s is not one of the available choices."
    ),
    "invalid_pk_value": _("?%(pk)s? is not a valid value."),
}

def __init__(self, queryset, **kwargs):
    super().__init__(queryset, empty_label=None, **kwargs)

def to_python(self, value):
    if not value:
        return []
    return list(self._check_values(value))

def clean(self, value):
    value = self.prepare_value(value)
    if self.required and not value:
        raise ValidationError(self.error_messages["required"], code="required")
    elif not self.required and not value:
        return self.queryset.none()
    if not isinstance(value, (list, tuple)):
        raise ValidationError(
            self.error_messages["invalid_list"],
            code="invalid_list",
        )
    qs = self._check_values(value)
    # Since this overrides the inherited ModelChoiceField.clean
    # we run custom validators here
    self.run_validators(value)
    return qs
```

```
def _check_values(self, value):
    """
    Given a list of possible PK values, return a QuerySet of the
    corresponding objects. Raise a ValidationError if a given value is
    invalid (not a valid PK, not in the queryset, etc.)
    """
    key = self.to_field_name or "pk"
    # deduplicate given values to avoid creating many querysets or
    # requiring the database backend deduplicate efficiently.
    try:
        value = frozenset(value)
    except TypeError:
        # list of lists isn't hashable, for example
        raise ValidationError(
            self.error_messages["invalid_list"],
            code="invalid_list",
        )
    for pk in value:
        try:
            self.queryset.filter(**{key: pk})
        except (ValueError, TypeError):
            raise ValidationError(
                self.error_messages["invalid_pk_value"],
                code="invalid_pk_value",
                params={"pk": pk},
            )
    qs = self.queryset.filter(**{"%s__in" % key: value})
    pks = {str(getattr(o, key)) for o in qs}
    for val in value:
        if str(val) not in pks:
            raise ValidationError(
                self.error_messages["invalid_choice"],
                code="invalid_choice",
                params={"value": val},
            )
    return qs

def prepare_value(self, value):
    if (
        hasattr(value, "__iter__")
        and not isinstance(value, str)
        and not hasattr(value, "_meta")
    ):
        prepare_value = super().prepare_value
        return [prepare_value(v) for v in value]
    return super().prepare_value(value)
```

```
def has_changed(self, initial, data):
    if self.disabled:
        return False
    if initial is None:
        initial = []
    if data is None:
        data = []
    if len(initial) != len(data):
        return True
    initial_set = {str(value) for value in self.prepare_value(initial)}
    data_set = {str(value) for value in data}
    return data_set != initial_set

return hasattr(form_class, "_meta") and (
```

Category: core_models

File: venv/lib/python3.12/site-packages/boto3/resources/model.py

```
    def __init__(self, name, definition, resource_defs):

    def __init__(self, definition, resource_defs):

    def __init__(self, name, definition, resource_defs):

    :param definition: The JSON definition

    :param resource_defs: All resources defined in the service
    """

        self._definition = definition

                definition.get('resource', {}), resource_defs
            )
        #: (``string``) The JMESPath search path or ``None``

class DefinitionWithParams:
    """
    An item which has parameters exposed via the ``params`` property.
    A request has an operation and parameters, while a waiter has
    a name, a low-level waiter name and parameters.

    :param definition: The JSON definition
    """

    def __init__(self, definition):

        self._definition = definition

class Request(DefinitionWithParams):
    """
    A service operation action request.

    :param definition: The JSON definition
    """

    def __init__(self, definition):

class Waiter(DefinitionWithParams):
    """
    An event waiter specification.

    :param definition: The JSON definition
    """

    def __init__(self, name, definition):

    :param definition: The JSON definition
```

```
    :param resource_defs: All resources defined in the service
    """

        self._definition = definition

        self._resource_defs = resource_defs

            self.type, self._resource_defs[self.type], self._resource_defs
        )

    :param definition: The JSON definition

    :param resource_defs: All resources defined in the service
    """

    :param definition: The JSON definition

    :param resource_defs: All resources defined in the service
    """

        self._definition = definition

        self._resource_defs = resource_defs
        self._renamed = {}

    def _get_has_definition(self):
        """

            for name, resource_def in self._resource_defs.items():
                # It's possible for the service to have renamed a

                        definition[has_name] = has_def
                        found = True

            definition = self._definition.get('has', {})

        for name, definition in self._get_has_definition().items():
            if subresources:
                name = self._get_name('subresource', name, snake_case=False)
            else:
                name = self._get_name('reference', name)

            action = Action(name, definition, self._resource_defs)
```

The models defined in this file represent the resource JSON description
format and provide a layer of abstraction from the raw JSON. The advantages
of this are:

classes as well as by the documentation generator.
"""

```
class Identifier:
    """
    A resource identifier, given by its name.

    def __init__(self, name, member_name=None):
        #: (``string``) The name of the identifier
        self.name = name
        self.member_name = member_name

class Action:
    """
    A service operation action.

    :type definition: dict

    :type resource_defs: dict

        #: (:py:class:`Request`) This action's request or ``None``
        self.request = None

        if 'request' in definition:

            self.request = Request(definition.get('request', {}))

        #: (:py:class:`ResponseResource`) This action's resource or ``None``
        self.resource = None

        if 'resource' in definition:
            self.resource = ResponseResource(

        self.path = definition.get('path')

    :type definition: dict

    def params(self):
        """
        Get a list of auto-filled parameters for this request.

        :type: list(:py:class:`Parameter`)
        """
        params = []

        for item in self._definition.get('params', []):
            params.append(Parameter(**item))

class Parameter:
    """
    An auto-filled parameter which has a source and target. For example,
    the ``QueueUrl`` may be auto-filled from a resource's ``url`` identifier
    when making calls to ``queue.receive_messages``.
```

```
    :param source_type: Where the source is defined.
    :type source: string
    :param source: The source name, e.g. ``Url``
    """

    def __init__(
        self, target, source, name=None, path=None, value=None, **kwargs
    ):
        #: (``string``) The destination parameter name
        self.target = target

        #: (``string``) Where the source is defined
        self.source = source
        #: (``string``) The name of the source, if given
        self.name = name
        #: (``string``) The JMESPath query of the source
        self.path = path
        #: (``string|int|float|bool``) The source constant value
        self.value = value

    :type definition: dict

        super().__init__(definition)

        self.operation = definition.get('operation')

    :type definition: dict

        super().__init__(definition)

        self.waiter_name = definition.get('waiterName')

class ResponseResource:
    """
    A resource response to create after performing an action.

    :type definition: dict

    :type resource_defs: dict

        self.type = definition.get('type')

        self.path = definition.get('path')

    def identifiers(self):
        """
        A list of resource identifiers.

        :type: list(:py:class:`Identifier`)
        """
        identifiers = []
```

```
        for item in self._definition.get('identifiers', []):
            identifiers.append(Parameter(**item))

    def model(self):
        """
        Get the resource model for the response resource.

        :type: :py:class:`ResourceModel`
        """
        return ResourceModel(

class Collection(Action):
    """

    A group of resources. See :py:class:`Action`.

    :type definition: dict

    :type resource_defs: dict

    def batch_actions(self):
        """
        Get a list of batch actions supported by the resource type
        contained in this action. This is a shortcut for accessing
        the same information through the resource model.

        :rtype: list(:py:class:`Action`)
        """
        return self.resource.model.batch_actions

class ResourceModel:
    """

    A model representing a resource, defined via a JSON description
    format. A resource has identifiers, attributes, actions,
    sub-resources, references and collections. For more information
    on resources, see :ref:`guide_resources`.

    :type definition: dict

    :type resource_defs: dict

        self.shape = definition.get('shape')

    def load_rename_map(self, shape=None):
        """
        Load a name translation map given a shape. This will set
        up renamed values for any collisions, e.g. if the shape,
        an action, and a subresource all are all named ``foo``
        then the resource will have an action ``foo``, a subresource
        named ``Foo`` and a property named ``foo_attribute``.
        This is the order of precedence, from most important to
        least important:
```

```
    if self._definition.get('load'):
        names.add('load')

    for item in self._definition.get('identifiers', []):
        self._load_name_with_category(names, item['name'], 'identifier')

    for name in self._definition.get('actions', {}):
        self._load_name_with_category(names, name, 'action')

    for name, ref in self._get_has_definition().items():
        # Subresources require no data members, just typically
        # identifiers and user input.
        data_required = False
        for identifier in ref['resource']['identifiers']:
            if identifier['source'] == 'data':
                data_required = True
                break

    for name in self._definition.get('hasMany', {}):
        self._load_name_with_category(names, name, 'collection')

    for name in self._definition.get('waiters', {}):
        self._load_name_with_category(
            names, Waiter.PREFIX + name, 'waiter'
        )

def _load_name_with_category(self, names, name, category, snake_case=True):
    """
    Load a name with a given category, possibly renaming it
    if that name is already in use. The name will be stored
    in ``names`` and possibly be set up in ``self._renamed``.

    :param snake_case: True (default) if the name should be snake cased.
    """
    if snake_case:
        name = xform_name(name)

def _get_name(self, category, name, snake_case=True):
    """
    Get a possibly renamed value given a category and name. This
    uses the rename map set up in ``load_rename_map``, so that
    method must be called once first.

    :param snake_case: True (default) if the name should be snake cased.
    :rtype: string
    :return: Either the renamed value if it is set, otherwise the
             original name.
    """
    if snake_case:
        name = xform_name(name)
```

```
def get_attributes(self, shape):
    """
    Get a dictionary of attribute names to original name and shape
    models that represent the attributes of this resource. Looks
    like the following:

def identifiers(self):
    """
    Get a list of resource identifiers.

    :type: list(:py:class:`Identifier`)
    """
    identifiers = []

    for item in self._definition.get('identifiers', []):
        name = self._get_name('identifier', item['name'])
        member_name = item.get('memberName', None)
        if member_name:
            member_name = self._get_name('attribute', member_name)
        identifiers.append(Identifier(name, member_name))

def load(self):
    """

    Get the load action for this resource, if it is defined.

    :type: :py:class:`Action` or ``None``
    """

    action = self._definition.get('load')

        action = Action('load', action, self._resource_defs)

def actions(self):
    """
    Get a list of actions for this resource.

    :type: list(:py:class:`Action`)
    """
    actions = []

    for name, item in self._definition.get('actions', {}).items():
        name = self._get_name('action', name)

        actions.append(Action(name, item, self._resource_defs))

def batch_actions(self):
    """
    Get a list of batch actions for this resource.

    :type: list(:py:class:`Action`)
    """
    actions = []
```

```
for name, item in self._definition.get('batchActions', {}).items():
    name = self._get_name('batch_action', name)

    actions.append(Action(name, item, self._resource_defs))

Get a ``has`` relationship definition from a model, where the
service resource model is treated special in that it contains

a relationship to every resource defined for the service. This
allows things like ``s3.Object('bucket-name', 'key')`` to

work even though the JSON doesn't define it explicitly.

        definitions.
"""

if self.name not in self._resource_defs:
    # This is the service resource, so let us expose all of

    # the defined resources as subresources.

    definition = {}

        # resource or to have defined multiple names that
        # point to the same resource type, so we need to
        # take that into account.
        found = False

        has_items = self._definition.get('has', {}).items()

        for has_name, has_def in has_items:

            if has_def.get('resource', {}).get('type') == name:

            # Create a relationship definition and attach it
            # to the model, such that all identifiers must be
            # supplied by the user. It will look something like:
            #
            # {
            #   'resource': {
            #     'type': 'ResourceName',
            #     'identifiers': [
            #       {'target': 'Name1', 'source': 'input'},
            #       {'target': 'Name2', 'source': 'input'},
            #       ...
            #     ]
            #   }
            # }
            #
            fake_has = {'resource': {'type': name, 'identifiers': []}}
```

```
                for identifier in resource_def.get('identifiers', []):
                    fake_has['resource']['identifiers'].append(
                        {'target': identifier['name'], 'source': 'input'}
                    )

                definition[name] = fake_has
        else:

        return definition

    def _get_related_resources(self, subresources):
        """
        Get a list of sub-resources or references.

        :rtype: list(:py:class:`Action`)
        """
        resources = []

    def subresources(self):
        """
        Get a list of sub-resources.

        :type: list(:py:class:`Action`)
        """
        return self._get_related_resources(True)

    def references(self):
        """
        Get a list of reference resources.

        :type: list(:py:class:`Action`)
        """
        return self._get_related_resources(False)

    def collections(self):
        """
        Get a list of collections for this resource.

        :type: list(:py:class:`Collection`)
        """
        collections = []

        for name, item in self._definition.get('hasMany', {}).items():
            name = self._get_name('collection', name)

            collections.append(Collection(name, item, self._resource_defs))

    def waiters(self):
        """
        Get a list of waiters for this resource.
```

```
:type: list(:py:class:`Waiter`)
"""
waiters = []

for name, item in self._definition.get('waiters', {}).items():
    name = self._get_name('waiter', Waiter.PREFIX + name)
    waiters.append(Waiter(name, item))
```

Category: core_models

File: venv/lib/python3.12/site-packages/django/db/migrations/operations/models.py

```
def _check_for_duplicates(arg_name, objs):
    used_vals = set()
    for val in objs:
        if val in used_vals:
            raise ValueError(
                "Found duplicate value %s in CreateModel %s argument." % (val, arg_name)
            )
        used_vals.add(val)

class ModelOperation(Operation):

    def __init__(self, name):
        self.name = name

    def name_lower(self):
        return self.name.lower()

    def references_model(self, name, app_label):
        return name.lower() == self.name_lower

    def reduce(self, operation, app_label):
        return super().reduce(operation, app_label) or self.can_reduce_through(
            operation, app_label
        )

    def can_reduce_through(self, operation, app_label):
        return not operation.references_model(self.name, app_label)

class CreateModel(ModelOperation):
    """Create a model's table."""
```

```
def __init__(self, name, fields, options=None, bases=None, managers=None):
    self.fields = fields
    self.options = options or {}
    self.bases = bases or (models.Model,)
    self.managers = managers or []
    super().__init__(name)
    # Sanity-check that there are no duplicated field names, bases, or
    # manager names
    _check_for_duplicates("fields", (name for name, _ in self.fields))
    _check_for_duplicates(
        "bases",
        (
            base._meta.label_lower
            if hasattr(base, "_meta")
            else base.lower()
            if isinstance(base, str)
            else base
            for base in self.bases
        ),
    )
    _check_for_duplicates("managers", (name for name, _ in self.managers))

def deconstruct(self):
    kwargs = {
        "name": self.name,
        "fields": self.fields,
    }
    if self.options:
        kwargs["options"] = self.options
    if self.bases and self.bases != (models.Model,):
        kwargs["bases"] = self.bases
    if self.managers and self.managers != [("objects", models.Manager())]:
        kwargs["managers"] = self.managers

    return (self.__class__.__qualname__, [], kwargs)

def state_forwards(self, app_label, state):
    state.add_model(
        ModelState(
            app_label,
            self.name,
            list(self.fields),
            dict(self.options),
            tuple(self.bases),
            list(self.managers),
        )
    )

def database_forwards(self, app_label, schema_editor, from_state, to_state):
    model = to_state.apps.get_model(app_label, self.name)
    if self.allow_migrate_model(schema_editor.connection.alias, model):
        schema_editor.create_model(model)
```

Category: core_models

File: venv/lib/python3.12/site-packages/botocore/model.py