System check framework

The system check framework is a set of static checks for validating Django projects. It detects common problems and provides hints for how to fix them. The framework is extensible so you can easily add your own checks.

Checks can be triggered explicitly via the `check` command. Checks are triggered implicitly before most commands, including `runserver` and `migrate`. For performance reasons, checks are not run as part of the WSGI stack that is used in deployment. If you need to run system checks on your deployment server, trigger them explicitly using `check`.

Serious errors will prevent Django commands (such as `runserver`) from running at all. Minor problems are reported to the console. If you have inspected the cause of a warning and are happy to ignore it, you can hide specific warnings using the `SILENCED_SYSTEM_CHECKS` setting in your project settings file.

A full list of all checks that can be raised by Django can be found in the System check reference.

Writing your own checks

The framework is flexible and allows you to write functions that perform any other kind of check you may require. The following is an example stub check function:

```
from django.core.checks import Error, register


@register()
def example_check(app_configs, **kwargs):
    errors = []
    # ... your check logic here
    if check_failed:
        errors.append(
            Error(
                "an error",
                hint="A hint.",
                obj=checked_object,
                id="myapp.E001",
            )
        )
    return errors
```

The check function *must* accept an `app_configs` argument; this argument is the list of applications that should be inspected. If `None`, the check must be run on *all* installed apps in the project.

The check will receive a `databases` keyword argument. This is a list of database aliases whose connections may be used to inspect database level configuration. If `databases` is `None`, the check must not use any database connections.

The `**kwargs` argument is required for future expansion.

Messages

The function must return a list of messages. If no problems are found as a result of the check, the check function must return an empty list.

The warnings and errors raised by the check method must be instances of `CheckMessage`. An instance of `CheckMessage` encapsulates a single reportable error or warning. It also provides context and hints applicable to the message, and a unique identifier that is used for filtering purposes.

The concept is very similar to messages from the message framework or the logging framework. Messages are tagged with a `level` indicating the severity of the message.

There are also shortcuts to make creating messages with common levels easier. When using these classes you can omit the `level` argument because it is implied by the class name.

- `Debug`
- `Info`
- `Warning`
- `Error`
- `Critical`

Registering and labeling checks

Lastly, your check function must be registered explicitly with system check registry. Checks should be registered in a file that's loaded when your application is loaded; for example, in the `AppConfig.ready()` method.

```
register(*tags)(function)
```

You can pass as many tags to `register` as you want in order to label your check. Tagging checks is useful since it allows you to run only a certain group of checks. For example, to register a compatibility check, you would make the following call:

```
from django.core.checks import register, Tags


@register(Tags.compatibility)
def my_check(app_configs, **kwargs):
    # ... perform compatibility checks and collect errors
    return errors
```

You can register "deployment checks" that are only relevant to a production settings file like this:

```
@register(Tags.security, deploy=True)
def my_check(app_configs, **kwargs): ...
```

These checks will only be run if the `check --deploy` option is used.

You can also use `register` as a function rather than a decorator by passing a callable object (usually a function) as the first argument to `register` .

The code below is equivalent to the code above:

```
def my_check(app_configs, **kwargs): ...


register(my_check, Tags.security, deploy=True)
```

Field, model, manager, template engine, and database checks

In some cases, you won't need to register your check function – you can piggyback on an existing registration.

Fields, models, model managers, template engines, and database backends all implement a `check()` method that is already registered with the check framework. If you want to add extra checks, you can extend the implementation on the base class, perform any extra checks you need, and append any messages to those generated by the base class. It's recommended that you delegate each check to separate methods.

Consider an example where you are implementing a custom field named `RangedIntegerField` . This field adds `min` and `max` arguments to the constructor of `IntegerField` . You may want to add a check to ensure that users provide a min value that is less than or equal to the max value. The following code snippet shows how you can implement this check:

```
from django.core import checks
from django.db import models


class RangedIntegerField(models.IntegerField):
    def __init__(self, min=None, max=None, **kwargs):
        super().__init__(**kwargs)
        self.min = min
        self.max = max

    def check(self, **kwargs):
        # Call the superclass
        errors = super().check(**kwargs)

        # Do some custom checks and add messages to `errors`:
        errors.extend(self._check_min_max_values(**kwargs))

        # Return all errors and warnings
        return errors

    def _check_min_max_values(self, **kwargs):
        if self.min is not None and self.max is not None and self.min > self.max:
            return [
                checks.Error(
                    "min greater than max.",
                    hint="Decrease min or increase max.",
                    obj=self,
                    id="myapp.E001",
                )
            ]
        # When no error, return an empty list
        return []
```

If you wanted to add checks to a model manager, you would take the same approach on your subclass of `Manager`.

If you want to add a check to a model class, the approach is *almost* the same: the only difference is that the check is a classmethod, not an instance method:

```
class MyModel(models.Model):
    @classmethod
    def check(cls, **kwargs):
        errors = super().check(**kwargs)
        # ... your own checks ...
        return errors
```

Changed in Django 5.1:
In older versions, template engines didn't implement a `check()` method.

Writing tests

Messages are comparable. That allows you to easily write tests:

```
from django.core.checks import Error

errors = checked_object.check()
expected_errors = [
    Error(
        "an error",
        hint="A hint.",
        obj=checked_object,
        id="myapp.E001",
    )
]
self.assertEqual(errors, expected_errors)
```

Writing integration tests

Given the need to register certain checks when the application loads, it can be useful to test their integration within the system checks framework. This can be accomplished by using the `call_command()` function.

For example, this test demonstrates that the `SITE_ID` setting must be an integer, a built-in check from the sites framework:

```
from django.core.management import call_command
from django.core.management.base import SystemCheckError
from django.test import SimpleTestCase, modify_settings, override_settings


class SystemCheckIntegrationTest(SimpleTestCase):
    @override_settings(SITE_ID="non_integer")
    @modify_settings(INSTALLED_APPS={"prepend": "django.contrib.sites"})
    def test_non_integer_site_id(self):
        message = "(sites.E101) The SITE_ID setting must be an integer."
        with self.assertRaisesMessage(SystemCheckError, message):
            call_command("check")
```

Consider the following check which issues a warning on deployment if a custom setting named `ENABLE_ANALYTICS` is not set to `True`:

```
from django.conf import settings
from django.core.checks import Warning, register


@register("myapp", deploy=True)
def check_enable_analytics_is_true_on_deploy(app_configs, **kwargs):
    errors = []
    if getattr(settings, "ENABLE_ANALYTICS", None) is not True:
        errors.append(
            Warning(
                "The ENABLE_ANALYTICS setting should be set to True in deployment.",
                id="myapp.W001",
            )
        )
    return errors
```

Given that this check will not raise a `SystemCheckError`, the presence of the warning message in the `stderr` output can be asserted like so:

```python
from io import StringIO

from django.core.management import call_command
from django.test import SimpleTestCase, override_settings


class EnableAnalyticsDeploymentCheckTest(SimpleTestCase):
    @override_settings(ENABLE_ANALYTICS=None)
    def test_when_set_to_none(self):
        stderr = StringIO()
        call_command("check", "-t", "myapp", "--deploy", stderr=stderr)
        message = (
            "(myapp.W001) The ENABLE_ANALYTICS setting should be set "
            "to True in deployment."
        )
        self.assertIn(message, stderr.getvalue())
```