django IN DEPTH

JAMES BENNETT · PYCON MONTRÉAL
9TH APRIL 2015

ABOUT YOU

- You know Python
- You know some Django
- You want to understand how it *really* works

ABOUT ME

- Working with Django 8½ years, 5 at Lawrence Journal-World
- Commit bit since 2007
- Obsessive documenter

WHAT THIS IS

- Django “from the bottom up”
- Deep understanding of the components and how they fit together
- Things the documentation doesn’t cover
- Django 1.8

WHAT THIS ISN’T

- Beginner tutorial
- API reference
- Writing applications (or any code)

THE ORM

A BLOG

```
from django.db import models

class Entry(models.Model):
    title = models.CharField(max_length=255)
    pub_date = models.DateTimeField()
    body = models.TextField()

    def __str__(self):
        return self.title
```

A MAGIC TRICK

```
>>> from blog.models import Entry
>>> Entry.objects.all()
```

↓

```
SELECT blog_entry.id,
       blog_entry.title,
       blog_entry.body
FROM blog_entry;
```

↓

```
[<Entry: My first entry>, <Entry: Another entry>]
```

HOW DID THAT HAPPEN?

DOWN THE RABBIT HOLE…

- Model
- Manager
- QuerySet
- Query
- SQLCompiler
- Database backend

DATABASE BACKENDS

- Base implementation, plus one per supported database (built-in ones in django/db/models/backends/)
- Goes in the `ENGINE` part of database settings
- Specifies extremely low-level behavior
- Is the boundary between Django and the DB driver modules (psycopg, cx_Oracle, etc.)

`DatabaseWrapper`

- Subclasses `BaseDatabaseWrapper` from `django.db.backends.base`
- Does what it says on the tin
- Carries basic information about the SQL dialect it speaks (column types, lookup and pattern operators, etc.)
- Also knows how to control transaction behavior

DatabaseOperations

- What the name implies
- Knows how to do common casts and value extractions
- Knows how to do flushes and sequence resets

`DatabaseFeatures`

- What does this database support?
- Whether casts are needed, whether some types exist, `SELECT FOR UPDATE` support, how `NULL` ordering works, etc.

DatabaseCreation

- Quirks of creating the database/tables
- Mostly now deals with the variations in index creation/removal

`DatabaseIntrospection`

- Used by the `inspectdb` management command
- Knows how to get lists of tables, columns, relationships
- Knows how to map DB’s internal column types back to Django field types

`DatabaseSchemaEditor`

- Used in migrations
- Knows DB-specific things about modifying the schema

`DatabaseClient`

- Used by the `dbshell` management command
- Knows how to open interactive console session to the DB

SHOULD YOU WRITE ONE?

- Probably not.
- Writing a DB backend is really only a good idea if your DB or driver of choice is currently unsupported

`SQLCompiler`

- Turns a Django `Query` instance into SQL for your database
- Subclasses for non-`SELECT` queries: `SQLInsertCompiler` for `INSERT`, `SQLDeleteCompiler` for `DELETE`, etc.

THE BRIDGE

- `Query.get_compiler()` returns a `SQLCompiler` instance for that `Query`
- Which calls the `compiler()` method of `DatabaseOperations`, passing the name of the desired compiler
- Which in turn looks at `DatabaseOperations.compiler_module` to find location of `SQLCompiler` classes for current backend

HERE BE DRAGONS

- `SQLCompiler` is scary, complex code
- The `as_sql()` method is where the ugly magic happens
- Pokes into attributes of the `Query` to find the building blocks, and `DatabaseOperations` to help translate to appropriate SQL

DON'T TRY THIS AT HOME

- MySQL and Oracle backends provide custom compilers
- MySQL needs it because subqueries
- Oracle needs it because `LIMIT` and `OFFSET` are weird
- You probably don't ever want to write your own

`Query`

- Data structure and methods representing a database query
- Lives in django/db/models/sql/
- Two flavors: `Query` (normal ORM operations) and `RawQuery` (for `raw()`)

BASIC STRUCTURE

- Attributes representing query parts
- select, tables, where, group_by, order_by, aggregates, etc.
- The `as_sql()` method pulls it all together and makes nice (we hope) SQL out of it

In simple terms, a Query is a tree-like data structure, and turning it into SQL involves going through its various nodes, calling the as_sql() method of each one to get its output.

This is a pattern we'll see again.

HERE BE MORE DRAGONS

THE SPRAWL

- django/db/models/sql/query.py is over 2,000 lines of code
- And still doesn’t do everything: the rest of django/db/models/sql is support code for it

WHY?

- Django’s high-level ORM constructs are infinitely and arbitrarily chain-able
- Every time you add something new, it has to be merged into the existing query structure
- This is a very hard problem.

Most of the complexity of the `Query` comes from this merging process. *Every* legal combination of `QuerySet` methods and options has to be supported here, plus hooks for custom extensions (such as end-user-supplied or third-party lookup options).

As a result, you almost never want to write a custom subclass of `Query`; instead, subclass `QuerySet` and work at a higher level.

Except…

QUERY EXPRESSIONS AND FUNCTIONS

QUERY EXPRESSIONS

- Allow more complex logic and references
- Allow access to more advanced SQL constructs

BUILT-IN EXAMPLES

- `When` – implements `WHEN … THEN` in SQL
- `Case` – implements `CASE` statements in SQL
- `F` – implements references to columns or annotations
- `Func` – implements calls to SQL functions

DATABASE FUNCTIONS

- Implemented using the `Func` query expression
- Allows queries to call out to SQL functions
- Built-in examples: support for `UPPER`, `SUBSTRING` and `COALESCE`

CUSTOM LOOKUPS

- Added in Django 1.7
- Just subclass `django.db.models.Lookup`, implement `as_sql()` and give it a `lookup_name`
- Register it for all field types, or for specific field types, as you like
- Can also transform results by subclassing `Transform` instead of `Lookup`

WE'VE SEEN THIS BEFORE

- Custom lookups and transforms are given access to the `SQLCompiler` instance, and the left- and right-hand sides of the expression
- And the `as_sql()` method is how we output the correct SQL to use

`QuerySet`

- django/db/models/query.py
- Wraps a `Query` instance in a nice API
- Actually implements the high-level query methods you use
- Acts as a container-ish object for the results

LAZINESS

- Most of the time, does nothing except twiddle its (and the wrapped `Query`'s) attributes in response to method calls, returning the modified `QuerySet` object
- Until you force it to execute the query

HOW TO FORCE A QUERY

- Call a method that doesn't return a `QuerySet`
- Call a method which requires actually querying the database to populate results, check existence of results, check the number of results, etc.
- Including special Python methods

CUSTOM CLASS BEHAVIOR

- `QuerySet` gets a lot of its behavior from Python class customization
- Implements `__iter__()` to be iterable (this forces a query), implements `__len__()`, `__bool__()` and `__nonzero__()` to check number/existence of results, `__getitem__` (for slicing/indexing), etc.

ONE SPECIAL NOTE

- `__repr__()` will perform a query, and will add a `LIMIT 21` to it
- Saves you from yourself: accidentally `repr()`-ing a `QuerySet` with a million results in it is bad for your computer/server

IT'S A CONTAINER

- Each instance of `QuerySet` has an internal results cache
- Iterating will pull from the results cache if populated rather than re-executing the query

IT'S A CONTAINER?

```
>>> my_queryset = SomeModel.objects.all()
>>> my_queryset[5].some_attribute
3
>>> my_queryset[5].some_attribute = 2
>>> my_queryset[5].save()
>>> my_queryset[5].some_attribute
3
>>> # WAT
```

WHAT HAPPENED?

- A performance trade-off
- Calling a `QuerySet` method will usually clone the pre-existing `QuerySet`, apply the change, and return the new instance (doing a new query)
- Except for iteration, length/existence checks, which can re-use the existing `QuerySet` instance's results cache without doing a new query

USEFUL METHODS

`update()`

- Tries to make the requested change in a single SQL `UPDATE` query instead of updating each row individually
- *Doesn't* execute custom `save()` methods on the model, and *doesn't* send `pre_save` or `post_save` signals

`delete()`

- Like update(), but for deleting; tries to do a single SQL `DELETE` query
- *Doesn't* execute custom `delete()` methods on the model, but *does* send `pre_delete` and `post_delete` signals (including for things deleted by cascade)

`exists()`

- Returns `True` or `False` depending on whether the `QuerySet` has results
- If the `QuerySet` is already evaluated, uses its results cache
- If not, does a (fast!) query to check for existence of results
- Usually faster than doing boolean evaluation of the `QuerySet`

`defer() / only()`

- Return partial model instances with only some fields queried/filled
- Give you fine-grained control over exactly which columns are queried in the DB
- Access to un-queried fields will result in a new query to fetch the data

`values() / values_list()`

- Also let you control which fields are retrieved
- Don't return model instances at all; `values()` returns dictionaries, `values_list()` returns lists
- `values_list(flat=True)`, with a single field name, returns a single flattened list

SOLVING THE N+1 PROBLEM

- `select_related()` – get results plus foreign-key related objects in a single query (joining in SQL)
- `prefetch_related()` – can fetch many-to-many and generic relations with one query per relation (joining in Python)

BUT WAIT, THERE'S MORE

- `update_or_create()` – like `get_or_create()` but looks for an existing instance to update
- `select_for_update()` – locks the selected rows until end of transaction
- `extra()` – close to `raw()`, but lets you add custom clauses to a regular `QuerySet`
- And many more: https://docs.djangoproject.com/en/1.8/ref/models/querysets/

WRITE YOUR OWN

- Advantage: a whole lot simpler than writing a custom `Query` subclass
- Advantage: becomes chain-able since your custom methods will (presumably) return an instance of your custom `QuerySet`
- Disadvantage: you usually also need a custom manager to go with it

MANAGERS

- The high-level interface
- Attach directly to a model class (and know which model they're attached to, via `self.model`)
- Create and return a `QuerySet` for you
- Expose most of the methods of `QuerySet`

OPTIONS

- Don't specify one at all – Django will create a default one and name it `objects`
- If you specify one, Django doesn't create the default `objects` manager
- One model class can have multiple managers
- First one defined becomes the default (accessible via `_default_manager`)

HOW IT WORKS

- Manager’s `get_queryset()` method returns a QuerySet for the model
- Almost everything else just proxies through to that
- Except `raw()`, which instantiates a `RawQuerySet`

```
from django.db import models

class Entry(models.Model):
    LIVE_STATUS = 1
    DRAFT_STATUS = 2
    STATUS_CHOICES = (
        (LIVE_STATUS, ‘Live’),
        (DRAFT_STATUS, ‘Draft’),
    )
    status = models.IntegerField(
        choices=STATUS_CHOICES
        default=LIVE_STATUS
    )
```

CUSTOM MANAGER

```
class LiveEntryManager(models.Manager):
    def live(self):
        return self.filter(
            status=self.model.LIVE_STATUS
        )

class Entry(models.Model):
    …
    objects = LiveEntryManager()

live_entries = Entry.objects.live()
```

A BETTER WAY

```
class EntryQuerySet(models.QuerYSet):
    def live(self):
        return self.filter(
            status=self.model.LIVE_STATUS
        )

class EntryManager(models.Manager):
    def get_queryset(self):
        return EntryQuerySet(self.model)

live_in_april = Entry.objects.filter(
    pub_date__year=2015,
    pub_date__month=4).live()
```

CAUTION

- Overriding `get_queryset()` will affect *all* queries made through that manager
- Usually best to keep a vanilla manager around so you can access everything normally

MODELS

- Finally!
- The actual representation of the data and associated logic
- One model class = one table; one model field = one column

THE MODEL METACLASS

- `django.db.models.base.ModelBase`
- Does the basic setup of the model class
- Handles `Meta` declaration, sets up default manager if needed, adds in model-specific exception subclasses, etc.

Most of the actual heavy lifting is *not* done in the metaclass itself; instead, anything which requires special behavior — like many model fields — should define a method named `contribute_to_class()`, which will be called and passed the model class and the name it will be bound to in the class.

`ModelBase` then loops through the attribute dictionary of the new class, and calls `contribute_to_class()` on anything which defines it.

MODEL FIELDS

- Subclasses of `django.db.models.Field`
- Fields do a lot of work

DATA TYPES

- `get_internal_type()` can return a built-in field type if similar enough to that type
- Or `db_type()` can return a custom database-level type for the field

VALUE CONVERSION

- `to_python()` converts from DB-level type to correct Python type
- `value_to_string()` converts to string for serialization purposes
- Multiple methods for preparing values for various kinds of DB-level use (querying, storage, etc.)

MISCELLANY

- `formfield()` returns a default form field to use for this field
- `value_from_object()` takes an instance of the model, returns the value for this field on that instance
- `deconstruct()` is used for data migrations (i.e., what to pass to __init__() to reconstruct this value later)

FLAVORS OF INHERITANCE

- Abstract parents
- Multi-table
- Proxy models

ABSTRACT MODELS

- Indicated by `abstract = True` in the `Meta` declaration
- Don’t create a database table
- Subclasses, if not abstract, generate a table with both their own fields and the abstract parent’s
- Subclasses can subclass and/or override abstract parent’s `Meta`

MULTI-TABLE INHERITANCE

- No special syntax: just subclass the other model
- Can’t directly subclass parent `Meta`, but can override by re-declaring options
- Can add new fields, managers, etc.
- Subclass has implicit `OneToOneField` to parent

PROXY MODELS

- Subclass parent, declare `proxy = True` in `Meta`
- Will reuse parent's table and fields; only allowed changes are at the Python level
- Proxy subclass can define additional methods, manager, etc.
- Queries return instances of the model queried; query a proxy, get instances of the proxy

UNMANAGED MODELS

- Set `managed = False` in `Meta`
- Django will not do *any* DB management for this model: won't create tables, won't track migrations, won't auto-add a primary key if missing, etc.
- Can declare fields, define `Meta` options, define methods normally aside from that

Unmanaged models are mostly useful for wrapping a pre-existing DB table, or DB view, that you don't want Django to try to mess with.

While they *can* be used to emulate some aspects of model inheritance, declaring a proxy subclass of a model is almost always a better way of doing that.

QUESTIONS?

THE FORMS LIBRARY

MAJOR COMPONENTS

- Forms
- Fields
- Widgets
- Model ⇔ form conversion
- Media support

WIDGETS

- One for each type of HTML form control
- Low-level operations
- Know how to render appropriate HTML
- Know how to pull their data out of a submission
- Can bundle arbitrary media (CSS, JavaScript) to include and use when rendering

IN AND OUT

- When data is submitted, a widget's `value_from_datadict()` method pulls out that widget's value
- When displaying a form, a widget's `render()` method is responsible for generating that widget's HTML

`MultiWidget`

- Special class that wraps multiple other widgets
- Useful for things like split date/time
- `decompress()` method unpacks a single value into multiple

FIELDS

- Represent data type and validation constraints
- Have widgets associated with them for rendering

VALIDATING DATA

- Call field's `clean()` method
- This calls field's `to_python()` method to convert value to correct Python type
- Then calls field's `validate()` method to run field's own built-in validation
- Then calls field's `run_validators()` method to run any additional validators supplied by end user
- `clean()` either returns a valid value, or raises `ValidationError`

VALIDATORS

```
def require_pony(value):
    if ‘pony’ not in value:
        raise forms.ValidationError(
            ‘Value must contain a pony’
        )


# In some form class:
pony = forms.CharField(validators=[
    require_pony,
])
```

CHOOSING A VALIDATION SCHEME

- `to_python()` in a field: when validation constraint is tied tightly to the data type
- `validate()` in a field: when validation constraint is intrinsic to the field
- `validators` argument to a field: when the basic field does *almost* all the validation you need, and it's simpler to write a validator than a whole new field

ERROR MESSAGES

- Django has a set of standard ones
- Built-in fields supplement these
- Stored in a dictionary `error_messages` on the field instance
- Standard keys: `required`, `invalid`

FIELDS AND WIDGETS

- Each field technically has *two* widgets: second one is used when it’s a hidden field
- `widget_attrs()` method gets passed the widget instance, and can return a dictionary of HTML attributes to pass to it

FORMS

- Tie it all together in a neat package
- Provide the high-level interface you’ll actually use

FORMS HAVE A METACLASS

- You can use it directly if you want: `django.forms.BaseForm`
- But probably not a good idea unless you're constructing form classes dynamically at runtime

The main thing to know about is that a form class ends up with two field-related attributes: `base_fields` and `fields`.

`base_fields` is the default set of fields for *all* instances of that form class. `fields` is the set of fields for a specific instance of the form class.

BUILD A FORM DYNAMICALLY

```
base_fields = {
    ‘name’: forms.CharField(max_length=255),
    ‘email’: forms.EmailField(),
    ‘message’: forms.CharField(
        widget=forms.Textarea
    ),
}

ContactForm = type(‘ContactForm’,
                   (forms.BaseForm,),
                   {‘base_fields’: base_fields})
```

IS EQUIVALENT TO

```
class ContactForm(forms.Form):
    name = forms.CharField(max_length=255)
    email = forms.EmailField()
    message = forms.CharField(
        widget=forms.Textarea
    )
```

WORKING WITH DATA

- Instantiating a form with data will cause it to wrap its fields in instances of `django.forms.forms.BoundField`
- Each `BoundField` has a field instance, a reference to the form it's in, and the field's name within that form
- `BoundField` instances are what you get when you iterate over the form

VALIDATION

- Call the form's `is_valid()` method
- That applies field-level validation
- Then form-level validation
- Then sets up either `cleaned_data` or `errors` attribute

FIELD VALIDATION

- For each field, call its widget's `value_from_datadict()` to get the value for the field
- Call field's `clean()` method
- Call any `clean_<fieldname>()` method on the form (if field `clean()` ran without errors)
- Errors go into form's `errors` dict, keyed by field name

FORM VALIDATION

- Form's `clean()` method
- Happens after field validation, so `cleaned_data` may or may not be populated depending on prior errors
- Errors raised here end up in error dict under the key `__all__`, or by calling `non_field_errors()`

ERROR MESSAGES

- Stored in an instance of `django.forms.utils.ErrorDict`
- Values in it are instances of `django.forms.utils.ErrorList`
- Both of these know how to print themselves as HTML

DISPLAYING A FORM

- Default representation of a form is as an HTML table
- Also built in: unordered list, or paragraphs
- None of these output the containing `<form></form>` or the submit elements
- `_html_output()` method can be useful for building your own custom form HTML output

`ModelForm`

- Introspects a model class and makes a form out of it
- Basic structure (declarative class plus inner options class) is similar to models
- Uses model meta API to get lists of fields, etc.
- Override/configure things by setting options in `Meta`

GETTING FORM FIELDS AND VALUES

- Calls the `formfield()` method of each field in the model
- Can be overridden by defining the method `formfield_callback()` on the form
- Each field's `value_from_object()` method called to get the value of that field on that model instance

SAVING

- `django.forms.models.construct_instance()` builds an instance of the model with form data filled in
- `django.forms.models.save_instance()` actually (if `commit=True`) saves the instance to the DB
- Saving of many-to-many relations is deferred until after the instance itself has been saved; with `commit=False`, you have to manually do it

FORM MEDIA

- Forms and widgets support inner class named `Media`
- Which in turn supports values `css` and `js`, listing CSS and JavaScript assets to include when rendering the form/widget
- `css` is a dict where keys are CSS media types; `js` is just a list
- Media objects are combinable, and results only include one of each asset

Actually, *any* class can support media definitions; widgets and forms are just the ones that support them by default.

To add media to another class, have that class use the metaclass `django.forms.widgets.MediaDefiningClass`. It will parse an inner `Media` declaration and turn it into a media property with the same behaviors (i.e., combinable, prints as HTML, etc.) as on forms.

QUESTIONS?

THE TEMPLATE SYSTEM

MAJOR COMPONENTS

- Engine
- Loader
- Template
- Tag and filter libraries
- Context

TEMPLATE ENGINES

- New in Django 1.8
- Wrap a template system in a consistent API
- Built-in engines for Django’s template language and for Jinja2

TEMPLATE ENGINES

- Subclass `django.template.backends.base.BaseEngine`
- Must implement `get_template()`, can optionally implement other methods
- Typically will define a wrapper object around the template for consistent API

TEMPLATE LOADERS

- Do the hard work of actually finding the template you asked for
- Required method: `load_template_source()`, taking template name and optional directories to look in
- Should return a 2-tuple of template contents and template location

TEMPLATE OBJECTS

- Instantiate with template source
- Implement `render()` method receiving optional context (dictionary) and optional request (`HttpRequest object`), returning string

Of course, this glosses over a lot of details of how a template language works…

THE DJANGO TEMPLATE LANGUAGE

KEY CLASSES

- All in `django/template/base.py`
- Template is the high-level representation
- `Lexer` and `Parser` actually turn the template source into something usable
- `Token` represents bits of the template during parsing
- `Node` and `NodeList` are the structures which make up a compiled `Template`

TEMPLATE LEXING

- Instantiate with template source, then call `tokenize()`
- Splits the template source using a regex which recognizes start/end syntax of tags and variables
- Creates and returns a list of tokens based on the resulting substrings

TOKENS

- Come in four flavors: `Text`, `Var`, `Block`, `Comment`
- Gets the text of some piece of template syntax, minus the start/end constructs like `{% … %}` or `{{ … }}`

TEMPLATE PARSING

- Instantiate `Parser` with list of tokens from `Lexer`, then call `parse()`
- Each tag and filter (built-in or custom) gets registered with `Parser`
- Each tag provides a compilation function, which `Parser` will call to produce a `Node`
- Variables get automatically translated into `VariableNode`, plain text into `TextNode`

TAG COMPILATION

- Each tag must provide this
- It gets access to the token representing the tag, *and* to the parser
- Can just be simple and instantiate/return a `Node` subclass
- Or can use the parser to do more complex things like parsing ahead to an end tag, doing things with all the nodes in between, etc.

NODES

- Can be an individual `Node`, or a `NodeList`
- Defining characteristic is the `render()` method which takes a `Context` as argument and returns a string

The full process transforms the string source of the template into a `NodeList`, with one Node for each tag, each variable and each bit of plain text.

Then, rendering the template simply involves having that `NodeList` iterate over and `render()` its constituent nodes, concatenating the results into a single string which is the output.

VARIABLES

- Basic representation is a `VariableNode`
- Filters are represented by `FilterExpression` which parses out filter names, looks up the correct functions and applies them
- `render()` consists of resolving the variable in the context, applying the filters, returning the result

TEMPLATE CONTEXT

- Lives in `django/template/context.py`
- Behaves like a dictionary
- Is actually a *stack* of dictionaries, supporting push/ pop and fall-through lookups
- First dictionary in the stack to return a value wins

The stack implementation of Context is crucial, since it allows tags to easily set groups of variables by pushing a new dictionary on top, then clean up after themselves by popping that dictionary back off the stack.

Several of the more interesting built-in tags use this pattern.

However, be aware of the performance implications: just as nested namespaces in Python code will slow down name lookups, so too lots of nested dictionaries on the context stack will slow down variable resolution.

`RenderContext`

- Attached to every `Context`: is a thread-safety tool for simultaneous renderings of the same template instance
- Each call to `render()` pushes its own dictionary on top, pops when done rendering, and *only* the topmost dictionary is searched during name resolution in `RenderContext`
- Tags with thread-safety issues can store their state in the `RenderContext` and know they'll get the right state

QUESTIONS?

REQUEST/RESPONSE PROCESSING

THE ENTRY POINT

- Request handler, in `django/core/handlers`
- `WSGIHandler` is the only one supported now
- Implements a WSGI application

HANDLER LIFECYCLE

- Sets up middleware
- Sends `request_started` signal
- Initializes `HttpRequest` object
- Handler calls its own `get_response()` method
- Transforms `HttpResponse` into appropriate outgoing format, and returns that

`get_response()`

- Apply request middleware
- Resolve URL
- Apply view middleware
- Call view
- Apply response middleware
- Return response

The bulk of `get_response()` is actually error handling. It wraps everything in `try/except`, implements the exception handling for failed URL resolution and errors in middleware or views, and also wraps the view in a transaction which can be rolled back in case of error.

URL RESOLUTION

- High-level: `django.core.urls.RegexURLResolver`
- Takes URLconf import path, and optional prefix (for `include()`)
- Will import the URLconf and get its list of patterns
- Implements the `resolve()` method which returns the actual view and arguments to use

URL PATTERNS

- Instances of `RegexURLPattern`
- Stores regex, callback (the view), and optional name and other arguments
- Implements a `resolve()` method to indicate whether a given URL matches it

MATCHES

- Returned in the form of a `ResolverMatch` object
- Unpacks into view, positional arguments, keyword arguments
- Has other information attached, including optional namespace, app name, etc.

URL RESOLUTION

- `RegexURLResolver` iterates over the supplied patterns, popping prefixes as necessary (for `include()`)
- Keeps a list of patterns it tried and didn't get a match on
- Returns the first time it gets a `ResolverMatch` from calling `resolve()` on a `RegexURLPattern`
- Or raises `Resolver404` if no match, includes list of failed patterns for debugging

This process is potentially nested many times; each URLconf you `include()` from your root URLconf will have its own associated `RegexURLResolver` (prefixed).

There is short-circuit logic to check that the prefix matches before proceeding through the patterns, to avoid pointlessly doing match attempts that are guaranteed to fail.

Under the hood, the RegexURLResolver of the root URLconf is given a prefix of “^/“

ERROR HANDLING

- Handler's `get_exception_response()` method is invoked
- First tries to look up a handler for the error type in the root URLconf (handler404, handler500, etc.) and call and return that response
- If all else fails, handler's `handle_uncaught_exception()` is called

The error handling is robust in the sense that it can keep information from leaking, but not in the sense of always returning something friendly to the end user.

If the initial handling of an error fails, Django promotes it to an error 500 and tries to invoke the `handler500` view. If attempts to handle the exception keep raising new ones, Django gives up and lets the exception propagate out.

For this reason, it's important to have your `handler500` be as bulletproof as possible, or just use Django's default implementation.

Django never attempts to catch `SystemExit`.

REQUEST AND RESPONSE

- `HttpRequest` is technically handler-specific, but there's only one handler now
- `HttpResponse` comes in multiple flavors depending on status code
- Built-in: 301, 302, 400, 403, 404, 405, 410, 500
- Plus `JsonResponse` for JSON output

VIEWS

- Must meet three requirements to be a Django view
- Is callable
- Accepts an `HttpRequest` as first positional argument
- Returns an `HttpResponse` or raises an exception

FUNCTIONS OR CLASSES

- Most end-user-written views are likely to be functions, at least at first
- Class-based views are more useful when reusing/customizing behavior

CLASS-BASED GENERIC VIEWS

- Inheritance diagram is complex
- Actual use is not
- Most of the base classes and mixins exist to let functionality be composed à la carte in the classes end users will actually be working with

THE BASICS

- `self.request` is the current `HttpRequest`
- Dispatch is done by `dispatch()` method, based on request method, to methods of the correct name: `get()`, `post()`, etc.
- You probably want `TemplateResponseMixin` somewhere in your inheritance chain if it's not already, for easy template loading/rendering
- Call `as_view()` when putting it in a URLconf, which ensures the `dispatch()` method will be what's called

The great advantage of class-based views is in the composability/reusability of functionality, and the ease with which they avoid the large argument lists functions would require to support the same customization.

The primary disadvantage is the proliferation of mixins and base classes needed to provide all the combinations of behaviors Django’s generic views support.

QUESTIONS?