django in Depth

James Bennett • PyCon Santa Clara • March 7, 2012

Who you are

- You know Python
- You know some Django
- You want to understand how all the parts fit together

Who I am

- Working with Django for 6½ years; 5 at the Lawrence Journal-World
- Core commit bit, release manager

What we're doing here

- Django from the "bottom up"
- Deep understanding of how Django works
- Deep understanding of how each component works
- Bits and pieces of undocumented fun :)
- Django 1.4

What we're not doing here

- Beginner-level tutorial
- An API reference
- Writing code

The ORM

Look, it's 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 __unicode__(self):
        return self.title
```

A magic trick

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

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

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

AUSTRALIANS
quickmeme.com

How it works
(in excruciating detail)

Down the rabbit hole...

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

Database backends

- At least one backend module for each supported database (in `django.db.backends`)
- Goes in the `ENGINE` portion of the database settings
- Specifies extremely low-level behavior
- Provides the boundary between Django and the DB driver (psycopg, cx_Oracle, etc.)
- Made up of several classes

DatabaseWrapper

- Knows how to connect to the database and return a cursor
- Provides the transaction-management hooks
- Maps Python-level lookup types to SQL operators in the DB's dialect

DatabaseOperations

- Understands and implements a particular database's "quirks"
- Knows how to generate DB-specific SQL (typecasts, DDL, quoting, transaction begin/end/commit)
- Can specify an alternate query generator for more fine-grained SQL control

DatabaseFeatures

- What does this database support?
- Type constraints, NULL-handling quirks, etc.

DatabaseCreation

- Knows how to create your database (and test database) and tables in it
- Maps model field types to column types

DatabaseIntrospection

- Used by `inspectdb`
- Knows how to get list of tables
- Handles reverse mapping of column types to model field types
- Detects relations on DBs which allow it

Other bits

- `DatabaseClient` can fire up an interactive shell (for `dbshell`)
- `DatabaseValidation` can implement DB-specific validation of model definitions
- `DatabaseError` and `IntegrityError` provide normalized exception classes for errors from the DB

Write a new backend if…

- Django doesn't support your database or driver of choice
- You want to control the connection mechanism (you can do connection pooling at this level)
- You need really low-level overriding of operations (like field type mappings or SQL generation)

SQLCompiler

* Base classes in `django.db.models.sql.compiler`
  * Compiles a `Query` to SQL
  * Executes the SQL against the correct DB
  * Returns the results

Flavors of `SQLCompiler`

- `SQLCompiler` is the base class, and the default for most queries (i.e., `SELECT` queries)
- One subclass for each other query type: `SQLInsertCompiler`, `SQLDeleteCompiler`, etc.
- Subqueries for aggregates go through `SQLAggregateCompiler`
- `SQLDateCompiler` is a special case used for `dates()` queries

You shouldn't ever need to work with `SQLCompiler` directly...

....unless you're writing the Oracle backend....

...but just in case:

- `Query.get_compiler()` returns a `SQLCompiler` instance
- Calls `DatabaseOperations.compiler()`
- `DatabaseOperations.compiler_module` specifies the module name, `Query.compiler` specifies the class name
- Oracle backend uses this to provide a compiler which understands Oracle's SQL dialect

The Query class

- Base class in `django.db.models.sql.query`
- A big scary data structure
- Tracks all the elements of the eventual query: columns, tables, joins, orderings, etc.
- Here be dragons (`db/models/sql/query.py` is over 1900 lines of code)

What's in Query

- Attributes storing building blocks which become `SELECT`, `FROM`, `WHERE`, `HAVING`, `ORDER BY`, limit and offset, etc.
- Methods for manipulating these attributes
- Most high-level things you do in queries end up as calls to `Query.add_filter()`

Flavors of Query

- Just like `SQLCompiler`, subclasses exist for different query types: `INSERT`, `DELETE`, etc.
- Specialized subclasses for aggregates and `dates()` queries
- `RawQuery` implements `raw()`

Playing with Query

- You probably don't need to (though prior to 1.2, some DB backends did)
- Vast majority of code in `Query` just does bookkeeping
- `__str__()` is useful: returns the query as a string of SQL (generated by `SQLCompiler`)

QuerySet

- Wraps a `Query` instance
- Knows which model to query
- Implements the high-level querying methods you actually use
- Acts as a container-ish object for accessing query results

Laziness

- No results until forced to fetch them
- Results may or may not be cached depending on use

Methods which force a query

- Methods which don't return a `QuerySet`
- Methods which return the number of results or check whether there are results

Methods which force a query

- `aggregate()`
- `count()`
- `create()`
- `delete()`
- `exists()`
- `get()`
- `get_or_create()`
- `in_bulk()`
- `iterator()`
- `latest()`
- `update()`

Other ways to force a query

- Iteration (in a `for` loop or otherwise)
- Slicing (one item or multiple)
- Boolean evaluation (in an `if` statement)
- `len()`
- `repr()`
- `list()`
- Pickling
- Caching

`__repr__()` is special

- Will add `LIMIT 21` to the query
- Saves you from yourself: accidentally `repr()`-ing a `QuerySet` with a million results can suck
- Also saves you from debug pages which print string representations of variables

A `QuerySet` is a container

- `QuerySet` instances have an internal result cache
- Iterating a `QuerySet` multiple times only does the query once (subsequent iterations use the cache)

A container?

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

Indexing is special

- Trade-off: indexing/slicing imply `LIMIT/OFFSET` and generate the appropriate (new) query
- Each query returns a separate in-memory copy of the model instance(s)

Methods you've never heard of (but should use)

`update()`

- Issues a bulk update of every record in the `QuerySet`
- Executes in a single SQL `UPDATE` statement
- Doesn't execute custom `save()` methods

`delete()`

- Deletes every record in the `QuerySet`
- May or may not be a single SQL `DELETE` statement
- May or may not execute custom `delete()` methods

`exists()`

- Returns `True` if the query has results, `False` otherwise
- If the `QuerySet` already evaluated, checks the result cache
- If not, does a (fast) query to see if results would exist
- Usually better than just doing `if some_queryset`

`defer()` and `only()`

- Return "partial" model instances, with only some fields filled in
- Let you control exactly which fields are selected by the query

`values()` and `values_list()`

- Like `defer()` and `only()`, let you control which fields are selected
- Don't return model instances: `values()` returns dictionaries, `values_list()` returns lists
- `values_list(flat=True)`, with a single field name, returns a single list

Write your own!

- Writing a `QuerySet` subclass with additional query methods is easy
- Example: http://simonwillison.net/2008/May/1/orm/

Manager

- Starting point for nearly all queries
- Attached directly to a model class, accessible as `self.model` on the manager
- Exposes most of the query methods of `QuerySet`

Manager options

- Don't specify one at all, Django creates one for you and names it `objects`
- If you specify one, Django doesn't create the default `objects` manager
- One model can have multiple managers
- First manager defined is the "default" manager, and becomes the attribute `_default_manager`

How it works

- `Manager.get_query_set()` returns a `QuerySet`
- Everything else just calls methods on the returned `QuerySet`
- Except `raw()`, which directly instantiates and returns a `RawQuerySet`

Blog entries with status

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

Put it together

```
class Entry(models.Model):
    …fields and such…
    objects = EntryManager()

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

Do it in QuerySet

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


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

# Now you can do...
may_2011 = Entry.objects.filter(pub_date__year=2011,
                                pub_date__month=5)
may_2011_live = may_2011.live()
```

But be careful

- Overriding `get_query_set()` affects all queries for that manager
- Can lead to unpleasant results if you really want to fetch certain objects but your custom `QuerySet` filters them out

The power of managers

- Encapsulate common query patterns
- Got status fields on lots of models? Write a manager with query methods for it and re-use
- More exotic query types, too: want a `most_commented()` method?

Multiple databases

Database routers

- Determine which DBs get reads, writes and `syncdb`
- Can allow/disallow creation of relations
- Specified in the DATABASE_ROUTERS setting
- Default router saves objects to the DB they were queried from, uses the default DB unless otherwise specified

Manual control

- `QuerySet.using()` takes a DB connection alias and passes it down the chain
- `Manager.db_manager(name)` returns a new `Manager` instance using that connection
- `save()` and `delete()` take a `using` argument

Tracking models

- Each model instance has a new attribute: `_state`
- Instance of `django.db.models.base.ModelState`
- `ModelState.db` is the name of a DB connection
- Set automatically by `save()` and by a `QuerySet` doing retrieval

Use cases

- Sharding: write a router which determines where to read/write data
- Master/slave replication: send all writes to the master, read from the slaves
- Disparate data sets: override Manager.get_query_set() to always use a particular DB

Limitations

- Cross-database relations are not supported by most DBs
- Order of routers is significant: first router to return a DB name wins

Models

- Represent data: one model class (usually) maps to one database table
- Fields (usually) map to columns in that table
- Specify options: ordering, human-readable name, etc.
- Methods for saving/deleting
- Custom methods for business logic

Model classes

- `django.db.models.Model` uses a metaclass: `django.db.models.base.ModelBase`
- `ModelBase` sets up some attributes, delegates setup of others

`ModelBase`

- Parses `Meta` declaration, fields and inheritance, creates the `Options` instance on the model class
- Adds per-model exception classes (`DoesNotExist`, etc.)
- Sends signals for model class preparation and registers the model class

Model customization hooks

- `contribute_to_class(cls, name)` will be called by `ModelBase` while the model class is being constructed
- `django.db.models.signals.class_prepared` will be sent after `ModelBase` finishes constructing the model class

`contribute_to_class()`

- Any attribute of the model class which has this method will have it called
- Gets passed the model class and the attribute name
- Used to set up any special behavior (usually for model field types)

`class_prepared`

- Has only one argument: `sender`, which is the class just constructed
- Allows code to run any time a model class is parsed/ constructed
- Django uses this signal to ensure each model has at least one manager attached

The `Options` class

- In `django.db.models.options`
- Holds all the stuff you put in that `class Meta` declaration
- Plus other metadata about the model
- Ends up as the attribute `_meta` of the model class

Useful tricks

- `_meta.fields` is a list of the model's fields
- `_meta.many_to_many` is a list of the model's many-to-many relationships
- `_meta.get_field()` returns the actual field objects

See it in action

```
>>> from blog.models import Entry
>>> opts = Entry._meta
>>> [f.name for f in opts.fields]
['id', 'title', 'pub_date', 'body']
>>> opts.get_field(‘title’)
<django.db.models.fields.CharField object at 0x1429530>
>>> opts.get_field('title').max_length
255
```

See it in action

```
>>> from django.contrib.auth.models import User
>>> alice = User.objects.get(username='alice')
>>> bob = User.objects.get(username='bob')
>>> opts = User._meta
>>> username_field = opts.get_field('username')
>>> username_field.value_from_object(alice)
u'alice'
>>> username_field.value_from_object(bob)
u'bob'
```

Model instance lifecycle

- Instantiation calls `__init__()`
- `django.db.models.signals.pre_init` sent
- Field values initialized (if creating new object with values, or retrieving existing object from the database)
- `django.db.models.signals.post_init` sent

Model instance lifecycle

- Save data by calling `save()`...
- ...which calls `save_base()`
- `save_base()` figures out whether to do `INSERT` or `UPDATE` query and which DB to use, and saves the data
- `save()` and `save_base()` are separate for reasons of API cleanliness

Model instance lifecycle

- Delete data by calling `delete()`
- Figures out which related objects need deletion (ensures integrity even on DBs which don't support it natively)

Customization points

- Override `save()` or `delete()`
- Listen for signals sent during model life cycle

Overriding save()

```
# This is wrong
def save(self):

# This worked on 1.1
def save(self, force_insert=False, force_update=False):

# This worked on 1.2
def save(self, force_insert=False, force_update=False,
using=None):

# How you should do it on 1.3+:
def save(self, *args, **kwargs):
    # Do your custom stuff
    # Always call parent save() at some point
    super(SomeModel, self).save(*args, **kwargs)
```

Overriding `delete()`

```
# This worked in 1.1
def delete(self):

# This worked in 1.2
def delete(self, using=None):

# Once again, you should do
def delete(self, *args, **kwargs):
    # Do your custom stuff
    # Call parent delete()
    super(SomeModel, self).delete(*args, **kwargs)
```

Overriding `delete()`

- You can skip the parent class' `delete()` method if you don't want to actually delete data
- Useful for implementing "delete with undo": pair with a custom manager which hides "deleted" objects

Don't override `__init__()`.

No, really. Don't.

Useful signals

- All live in `django.db.models.signals`
- `pre_init` and `post_init` sent at beginning/end of `__init__()`
- `pre_save` and `post_save` sent at beginning/end of `save_base()`
- `pre_delete` and `post_delete` sent at beginning/end of `delete()`

The model cache

- Lives in `django.db.models.loading`, but import from `django.db.models`
- Tracks all models from all installed applications
- Prevents certain issues with duplicate copies of model classes
- Provides a generic way to get models

How models are tracked

- Combination of application “label” and model name
- `django.contrib.auth.models.User` ➟ “auth”, “user”
- These exist on the model’s Options instance as the attributes `app_label` and `module_name`

In action

```
>>> from django.db import models
>>> user_model = models.get_model(‘auth’, ‘user’)
>>> user_model
<class 'django.contrib.auth.models.User'>
>>> auth_app = models.get_app('auth')
>>> models.get_models(auth_app)
[<class 'django.contrib.auth.models.Permission'>,
<class 'django.contrib.auth.models.Group'>,
<class 'django.contrib.auth.models.User'>,
<class 'django.contrib.auth.models.Message'>]
```

Useful methods

- `get_model(app_label, model_name)` -- returns a model class
- `get_app(app_label)` -- returns that application's `models` module
- `get_models(app)` -- given `models` module, returns all model classes in it
- `get_models()` -- returns all installed models

Generic querying

```
>>> from django.db import models
>>> model_str = “some_app.some_model”
>>> mod = models.get_model(*model_str.split(‘.’))
>>> objects = mod._default_manager.all()
# etc.
```

It's everywhere

- `AUTH_PROFILE_MODULE` takes an `app_label.model_name` string
- `django.contrib.contenttypes` uses app_label/model_name pairs to track models
- Admin uses app_label/model_name pairs to identify models from URLs
- etc., etc.

Model fields

- Most live in `django.db.models`
- Some bundled apps include more field types
- Each field type represents a particular type of data and optionally constraints on values of that type

Under the hood: data types

- `get_internal_type()` can return the name of a built-in field type; DB-level data type will be same as that field type
- `db_type()` can name a custom data type for use at the DB level

Under the hood: conversion

- `to_python()` converts a value from the DB to a Python value suitable for the field type
- `get_prep_value()` converts a Python field value to (generic) DB format
- `get_db_prep_value()` is like `get_prep_value()` but applies DB-specific quirks

Under the hood: querying

- `get_prep_lookup()` converts a value to the correct (generic) format for a particular type of query (and is given the lookup type as an argument)
- `get_db_prep_lookup()` is similar, but applies DB-specific quirks

Under the hood: saving

- `pre_save()` can do generic preprocessing
- `get_db_prep_save()` allows DB-specific formatting of data to be saved

Miscellany

- `formfield()` returns a form field class suitable for this field type
- `value_to_string()` converts the field value to a string for serializers

Writing your own

- From scratch: subclass `django.db.models.Field`
- Use `django.db.models.SubfieldBase` as the metaclass
- Override any methods you need
- Full documentation: http://docs.djangoproject.com/en/dev/howto/custom-model-fields/

Writing your own

- Subclassing an existing field: just do it
- Override methods if you like
- Or don’t

A powerful use case

```
class PubDateField(models.DateField):
    pass


def get_publication_date(obj):
    opts = obj._meta
    pub_date_field = None
    for field in opts.fields:
        if isinstance(field, PubDateField):
            pub_date_field = field
            break
    return pub_date_field.value_from_object(obj)
```

Model validation

- Three flavors: field-level, instance-level, uniqueness checks
- Run the whole suite by calling a model instance's `full_clean()` method

clean_fields()

- Each model field defines a `clean()` method
- Additional validation can be added to a field in the model definition

`clean()`

- Instance-level validation
- Called after `clean_fields()`
- Can perform validation involving multiple fields simultaneously

`validate_unique()`

- Applies `unique` declarations on fields, `unique_for_*` declarations and `unique_together` in `Meta`
- Last step in the validation chain

`full_clean()`

- Raises `django.core.exceptions.ValidationError` if instance is invalid
- Must be called manually; model saving does not implicitly validate

Inheritance

- Abstract/concrete
- DB-level
- Python-level

Abstract models

- Create an abstract model by declaring `abstract = True` in `Meta`
- No database table created
- Subclasses, if not abstract, will generate table with both their own and the parent's fields
- Subclasses can subclass and override parent's `Meta`

Abstract models

- `Meta` cannot declare some attributes (`db_table`, for example)
- Special interpolation syntax for `related_name`

Use cases

- Common field set (e.g., title, publication date, author, etc.)
- Common methods
- Common `Meta` declarations
- Common custom managers

DB-level inheritance

- No special mechanism: just subclass the model you want to inherit from
- Can't directly subclass parent's `Meta`, but can override with new declarations
- Can add new managers (parent's managers are *not* inherited)

DB-level inheritance

- Always implemented as multi-table
- Subclass gets a table with implicitly-created one-to-one relation to parent (specify a `OneToOneField` with `parent_link=True` to manually control this)
- Parent/child relation can be traversed like any one-to-one relation

Use cases

- Logical “is-a” relationships: a `Restaurant` “is-a” `Place`
- And then only maybe (inheritance isn’t a great pattern)

Python-level inheritance

- Create a “proxy” subclass: `proxy = True` in `Meta`
- Will use parent’s database table
- Can add/override declarations in `Meta`
- Can add new managers (parent’s will be inherited as well)

Python-level inheritance

- Must have exactly one non-abstract parent
- Can have any number of abstract parents…
- …but abstract parents cannot define fields

Use cases

- Adding methods to existing models
- Adding managers or changing `Meta` behavior
- Far better solution than monkeypatching

General caveats

- Querying a model always returns instances of that model, never instances of parents/children
- Subclasses can *never* override parent field definitions
- Abstract classes cannot be queried
- First parent to define a name “wins”

When in doubt, don't use inheritance

Miscellaneous ORM features

Unmanaged models

- Declare `managed = False` in `Meta`
- Django doesn't attempt to create or maintain a DB table
- Django doesn't add an automatic primary-key field
- Django doesn't attempt to create many-to-many join tables
- Useful for wrapping existing tables or, more often, DB-level views

Generic relations

- Live in `django.contrib.contenttypes.generic`
- Require `django.contrib.contenttypes` installed
- Allow relation to any instance of any model

How it works

- Add a `ForeignKey` to `django.contrib.contenttypes.models.ContentType`
- Add a field which can hold a primary-key value (usually `IntegerField` or `TextField`)
- Add a `GenericForeignKey` specifying the above field names as arguments

How it works

```
class Tag(models.Model):
    content_type = models.ForeignKey(ContentType)
    object_id = models.TextField()
    object = generic.GenericForeignKey('content_type',
                                       'object_id')
    tag = models.CharField(max_length=255)
```

```
>>> jacob = User.objects.get(username='jacob')
>>> t = Tag(object=jacob, tag='bdfl')
>>> t.save()
>>> t.object
<User: jacob>
```

How it works

- The `ContentType` relation is used to determine the model class
- Then a primary-key lookup is done against that model
- Querying by generic relation always requires both values explicitly: `GenericForeignKey` fields are not legal in `filter()`, `get()`, etc.

Reverse generic relations

- Declare on the model class you'll "point" to
- Use `generic.GenericRelation`
- Sets up a reverse relationship attribute for easy queries

Q expressions

- `django.db.models.Q`
- Use normal field-lookup syntax
- Legal anywhere field-lookup syntax is (but must come as positional arguments)
- Combine with logical operators & and |
- Negate with logical ~
- Allow complex lookups, reuse of query fragments

F expressions

- `django.db.models.F`
- Takes a field name, becomes a reference to that field name
- Usable as a lookup value in queries
- Allows self-referential or field-comparing queries

extra()

- Allows a bit of raw SQL in an otherwise-normal query
- Can add extra attributes to returned model instances
- Only occasionally useful

Natural keys

- Define the method `natural_key()` on the model class
- Should return an iterable
- Example: `ContentType` returns app label/model name

Natural keys

- Serializers will use a natural key if the model defines it
- `ContentType` and `Permission` both have custom managers which do lookups by natural keys

Natural keys

```
# Normal querying
Permission.objects.get(code_name='change',
                       content_type__app_label='auth',
                       content_type__model='user')

# Using natural key
Permission.objects.get_by_natural_key('change',
                                      'auth', 'user')
```

Questions?

The forms library

Major parts

- Forms
- Fields
- Widgets
- Form-generation utilities
- Media support

Widgets

- Low-level display and parsing
- Know how to render HTML
- Know how to pull data out of a submission

Widgets

- One for each type of HTML form control: `Textarea`, `PasswordInput`, etc.
- Can also bundle arbitrary media (CSS, JavaScript) to include and use for display

Important methods

* `render(self, name, value, attrs=None)`
  * Returns HTML (as a Unicode string) to represent the widget
  * `attrs` is a dict of HTML attributes to render
  * `value` is not guaranteed to be valid!

Important methods

- `value_from_datadict(self, data, files, name)`
- Return the value input into this widget's HTML control
- `data` and `files` are the `POST` or `GET` and `FILES` dicts from an HTTP request

Important methods

- `id_for_label(self, id_)`
- Returns an HTML ID to use in tying the widget to a `label` element.

Important methods

- `build_attrs(self, extra_attrs=None, **kwargs)`
- Combines the widget instance's existing attributes with any extra attributes specified by other means
- Not useful to override, but useful in other methods (e.g., `render()`)

MultiWidget

- Special-purpose widget, provides a wrapper around multiple other widgets
- Example: combining date and time widgets for a datetime input

MultiWidget

- `value` argument to `render()` can be a list of values, one per wrapped widget
- Or a single value; `decompress()` will be called to unpack into multiple values
- Example: `SplitDateTimeWidget.decompress()` turns a `datetime.datetime` into separate date and time values

Fields

- Represent data type and validation constraints
- Have associated widgets for rendering
- Can perform validation and return values of appropriate types
- Can be arbitrarily specialized (or generic)

Field validation

- Call field's `clean()` method (three-step process)
- Convert value to appropriate Python type
- Run field's own validation
- Run attached validators
- `django.core.exceptions.ValidationError` (`django.forms.ValidationError` is an alias)

Converting field values

- `to_python(self, value)`
  - Converts `value` to the appropriate Python type for the field and returns it
  - Raises `ValidationError` if the value can't be converted

Field-level validation

- `validate(self, value)`
  - `value` has already been converted by `to_python()`
  - If `value` is valid, do nothing
  - If not, raise `ValidationError`

Validators

- Additional validation constraints, can be arbitrarily attached at any time
- All validators attached to a field will be run during validation

Validators

- A validator is a callable which takes a single argument (`value`)
- Raises `ValidationError` if invalid
- Built-in validators in `django.core.validators`

Validators

```
def require_pony(value):
    if ‘pony’ not in value:
        raise ValidationError(“A pony is required”)
```

```
# In a form:
pony = models.CharField(validators=[require_pony])
```

Validators on a field

- `run_validators(self, value)`
  - Iterates over `self.validators`, calling each
  - Traps `ValidationError` to collect error messages
  - Raises a new `ValidationError`, with all error messages, if needed

Choosing a validation approach

to_python()

- When the validation constraint is tied to the data type
- Most commonly: requiring a number

validate()

- When the constraint is intrinsic to the field type
- Choice-based fields usually need to do this

Validators

- Any other type of validation
- Most of the time you can re-use a built-in
- If not, still less code than a custom field

Error messages

- For custom validation, supply your own
- `raise ValidationError(“No you can’t have a pony”)`

Error messages

- Each field class also defines standard error messages
- These are combined with any additional messages passed when a field instance is created
- Stored in the attribute `error_messages` (a dictionary) on the instance
- Standard keys: `invalid`, `required` (fields can define others)

Error messages

```
error_messages = {
    'invalid': "No you can't have a pony",
    'required': "You must supply your own pony",
}

pony = forms.CharField(error_messages=error_messages)
```

Fields and widgets

- Each field defines two default widget classes
  - `widget` is the standard widget
  - `hidden_widget` is used for a hidden field
  - Can be overridden per-instance with the keyword argument `widget`

Fields and widgets

- `widget_attrs(self, widget)`
- Given a widget instance, return attributes which should be applied

Other important bits

- Keyword arguments when instantiating a field
  - `required` (boolean)
  - `label` (used as HTML label element)
  - `help_text` (additional longer explanation of field's requirements)

Fields for models

- `ModelChoiceField` and `ModelMultipleChoiceField`
- Correspond to `ForeignKey`/`OneToOneField` and `ManyToManyField`
- Accept a `QuerySet` from which choices are drawn
- Return the chosen object(s)

MultiValueField

- Like `MultiWidget`, "wraps" multiple fields as a single logical unit
- `MultiWidget` has a `decompress()` method, `MultiValueField` has `compress()`
- Standard example: `SplitDateTimeField`

Localization

- Added in Django 1.2: locale support includes localized data formats
  - `django.utils.formats`
  - Dates, times, number formatting, calendaring
  - Form fields automatically pull this from active locale

Forms

- Pull everything together
- Fields
- Validation
- Error handling
- Rendering

`BaseForm`

- Base class for all forms
- Default field set for each class stored in the dictionary `base_fields`
- `__init__()` sets up the dictionary `fields`, unique to each instance

`base_fields` vs. `fields`

- Altering `base_fields` changes every instance of that form class
- Altering `fields` changes only that specific instance

Building a form the hard way

```
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})

# Now you can use it...
contact_form = ContactForm()
```

Form

- Declare fields the same way as on models
- Uses a metaclass (`django.forms.DeclarativeFieldsMetaclass`) to turn this into the `base_fields` dictionary

Building a form the easy way

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

Binding data

- Instantiate the form class with some data
- `form = SomeForm(data=request.POST)`

Bound fields

- A form with data wraps its fields in `BoundField` instances
- Each `BoundField` has a field instance, a reference to the form, and the field's name within the form
- Iterating a form instance yields the `BoundField` instances

Form validation

- Call the form instance's `is_valid()` method
- Short-circuit: an unbound form is never valid

How form validation works

- Fields are validated (`_clean_fields()`)
- Form as a whole is validated (`_clean_form()`)
- If valid, the form instance gets an attribute called `cleaned_data`
- If not, an attribute called `errors`

`_clean_fields()`

- Loops over `self.fields`
- Field's `widget.value_from_datadict()` retrieves the data
- Field validated
- Custom validation on the form is called

Custom validation

- Method on the form, named `clean_<fieldname>()`
- Has access to form's `cleaned_data`
- Not called if the field's own `clean()` already raised a validation error

`_clean_form()`

- Calls form's `clean()` method
- Implement `clean()` to do form-level validation (comparing multiple fields, etc.)
- Can access `cleaned_data`
- Field values not guaranteed to be in there, though

Form-level errors

- Raised by form's `clean()`
- Special key in errors dictionary: `__all__`
- Accessible via `non_field_errors()`

Errors

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

Form display

- Default string representation of a form instance is as an HTML table (`as_table()`)
- Also available: `as_ul()`, `as_p()`
- `as_table()` and `as_ul()` do not output the containing table or list element
- None of these output wrapping `<form>` element or submit buttons

Form display

- `_html_output()`
- Arguments are strings with placeholders
- Appropriate field attributes and error messages interpolated in

Fine-tuning display

- Iterate over the form, get `BoundField` instances (in order of definition)
- Or use dictionary-style access to the form to get specific (bound) field instances

Fun with BoundField

- By default, uses the field's defined widget
- `as_text()` to get `<input type="text">`
- `as_textarea()` to get `<textarea>`
- `as_hidden()` to get `<input type="hidden">`
- `as_widget()` to get whatever widget you want

Other useful display tricks

- `is_multipart()` tells whether you need to handle file uploads
- `visible_fields()` gives all non-hidden fields
- `hidden_fields()` gives all hidden fields

Forms for models

ModelForm

- Base class is `django.forms.models.BaseModelForm`
- `django.forms.ModelForm` uses metaclass `django.forms.models.ModelFormMetaclass`

`ModelFormMetaclass`

- Parses the `Meta` declaration
- Turns it into an instance of `django.forms.models.ModelFormOptions`
- Stores it as the attribute `_meta` of the form class

ModelFormOptions

- Stores model class, fields to include and fields to exclude
- Stores dictionary of field names and widget instances to override default widgets

Mapping model fields

- Each model field class defines a method `formfield()`
- Override by defining the method `formfield_callback()` on the form class

formfield_callback()

- Receives the model field class and any defined widget from the form class
- Must return an instance of a form field class

Getting field values

- `django.forms.models.model_to_dict()`
- Uses model field's `value_from_object()` method
- Special-case handling for `ManyToManyField`

Validation

- Uses model's own validation routines
- Excludes model fields not represented in the form
- Excludes model fields which already have form-level errors

Saving

- `django.forms.models.save_instance()` and `django.forms.models.construct_instance()`
- Uses model fields' `save_form_data()` method
- Defers file-based fields until other fields have been set up (to allow dynamic `upload_to` based on other field values)

Form media

The `Media` class

- `django.forms.widgets.Media`
- Two attributes store information about media to include

CSS

- Stored in the attribute `css`, which is a dictionary
- Keys are CSS media names (`all`, `screen`, etc.)
- Values are paths to stylesheets

CSS

- Handled by `Media.add_css()`
- Uses `setdefault` and a check against existing values to avoid duplicates

JavaScript

- Stored in the attribute `js`, which is a tuple
- Items are paths to JavaScript files

JavaScript

- Handled by `Media.add_js()`
- Does checking against existing declarations to avoid duplicates

Bundling media

- Works on widget classes and form classes
- Define an inner class named `Media`, with the appropriate attributes

Example

```
class MyWidget(forms.TextInput):
    class Media:
        js = (‘foo.js’, ‘bar.js’)
```

How it works

- Widgets have a metaclass (`django.forms.widgets.MediaDefiningClass`) which parses the Media declaration
- For forms, `DeclarativeFieldsMetaclass` does the same

Media paths

- Can be full URLs, including domain
- Can be relative URLs starting with ‘/’
- Can be file names; `settings.STATIC_URL` will be prepended to generate the full URL
- `Media.absolute_path()` has the logic for this

Rendering

- String representation of a `Media` instance is the correct HTML
- Dictionary access works: `some_form.media['js']`

Rendering

- `render()` spits out all media for the instance
- `render_css()` does just the CSS
- `render_js()` does just the JavaScript
- `__unicode__()` just calls `render()`

Media and inheritance

- By default, a subclass of an existing widget or form inherits the parent's media definitions
- Specify `extend = False` in the subclass' `Media` class to change this

Combining media

- Simply add media instances
- `combined = form1.media + form2.media`
- Duplicate-checking is applied

You can use `Media` outside of forms

Make your own

```
from django.forms.widgets import MediaDefiningClass

class MyClass(object):
    __metaclass__ = MediaDefiningClass
    # Now you can define an inner ‘Media’ class on
    # subclasses

    class MyClassWithMedia(MyClass):
        class Media:
            css = {‘all’: ‘foobar.css’}
```

Questions?

The template system

Major components

- Templates
- Tag and filter libraries
- Loaders

Template loaders

- Locate templates (wherever they might be)
- Compile raw template source into a `Template` instance
- Base class:
  `django.template.loader.BaseLoader`

Anatomy of a loader

- `load_template`
- `load_template_source`
- `reset`
- Most custom loaders override `load_template_source`
- Caching/stateful loaders should also override `reset`

`load_template_source()`

- Returns a 2-tuple: (template source, origin)
- Filesystem loader searches `TEMPLATE_DIRS` and returns the file path as the origin
- App directories loader searches for `templates` directories in applications, returns the file path as origin
- Egg loader does the same, but inside eggs (and returns an egg name as the origin)

Other template languages

- A “template” is really just an object defining the method `render(self, context)`
- Write a wrapper which implements that method
- And a loader which knows how to apply it

The `Template` class

- `django.template.Template`
- Compiled from a string source
- Ultimate result is a wrapper around a list of `django.template.Node` instances

Compiling a template

- `django.template.Lexer` breaks the source string into appropriate tokens (`django.template.Token`)
- `django.template.Parser` turns the tokens into `Node` instances and returns the `NodeList`

Lexing

- Regex-based: `django.template.tag_re`
- Lexer uses defined constants to identify known syntax (tags, variables, etc.)
- Instantiate with source string and origin; `Lexer.tokenize()` returns list of `Token` instances

Tokens

- `django.template.Token`
- Stores type and contents
- Types are text, variable, comment and block
- `Token.split_contents()` breaks up contents (intelligently) into a list for further use

The parser

- `django.template.Parser`
- Instantiate with a list of tokens
- `parse()` turns the tokens into a `NodeList`

Node

- `django.template.Node`
  - Everything in the template becomes an instance of a `Node` subclass
  - `Node` must define the method `render()` which takes a `Context` instance

NodeList

- A list of `Node` instances
- Renders by iterating its nodes, calling `render()` on each and concatenating the results

Mapping tokens to nodes

- `Parser` maps plain text and variables to `TextNode` and `VariableNode`
- Comments are skipped
- All other tokens treated as tags and looked up by name

Handling tags

- Parser has a dictionary, `tags`, mapping tag names to compilation functions
- Loading new tag libraries updates this dictionary

Built-in tags and filters

- `django.template.builtins` is the list of default libraries to load
- `django.template.add_to_builtins()` can add new ones
- By default, loads `django.template.defaulttags` and `django.template.defaultfilters`

Tag loading

- Keyed by module name
- First location to have `<app>.templatetags.<name>` wins

Parser tricks

- Tag compilation functions have access to the parser
- Can parse forward, back up, look for specific tags

`parse()`

- Optional argument `parse_until` takes a list of tag names
- Continues parsing until a token of that name is reached
- `Parser.delete_first_token()` will remove that token

`next_token()`

- Returns the next token in the template
- Used by `for`/`empty`, `if`/`else`, etc.

`skip_past()`

- Takes a tag name
- Parses to just past the next tag matching that name
- Example: `endcomment`

Debugging

- `DEBUG = True` swaps out the lexer and parser
- `django.template.debug.DebugParser` and `django.template.debug.DebugLexer`

Other parsing tricks

- Node classes can set `must_be_first = True` (e.g., for `extends`)
- Overridable `enter_command` and `exit_command` (debugging parser uses these)

Variables

- `django.template.VariableNode`
- Wraps an instance of `django.template.FilterExpression`

`FilterExpression`

- Splits out variable name and any filters
- Checks that all filter names are valid
- Wraps variable in a `template.Variable` instance if possible

Variable

- Actually resolves the variable in a given `Context`
- Also understands `gettext` syntax and will apply translation when needed

Context

- Behaves *like* a dictionary
- Is actually a stack of dictionaries
- Fall-through lookup semantics: checks from top to bottom looking for variables

Context tricks

- `push()` adds a new dictionary on the top of the stack
- New variables are added to the topmost dictionary
- `pop()` removes the topmost dictionary

RenderContext

- Thread-safe storage of node rendering state
- Only the topmost dictionary is checked for variable resolution
- Each `Context` has an attached `RenderContext`
- New dictionary pushed on top at start of `render()`, popped at end

Autoescaping

- By default, all variables are escaped
- `safe` filter turns this off case-by-case
- `autoescape` tag turns it on/off for sections of a template
- `autoescape` attribute of `Context` can also control escaping

Questions?

Request/response processing

Request handlers

- Base class `django.core.handlers.base.BaseHandler`
- Implement the request/response pipeline
- One subclass for mod_python (deprecated), one for WSGI

Request handlers

- Handler's `__call__()` is the entry point for Django
- Loads middleware
- Initializes `HttpRequest` object
- Calls handler's `get_response()`

Middleware loading

- Handler's `load_middleware()` method
- Populates attributes containing request, response, view and exception middleware classes

HttpRequest

- Base class `django.http.HttpRequest`
- One subclass for mod_python, one subclass for WSGI
- mod_python: `_req` is the raw request object
- WSGI: `environ` is the original WSGI environ

`get_response()`

- Applies request middleware
- Resolves URL
- Applies view middleware
- Calls view
- Applies response middleware

URL resolution

- `django.core.urlresolvers.set_urlconf()`
  sets the (thread-local) URL configuration
- `django.core.urlresolvers.RegexURLResolver`
  is the class used to perform resolution

`RegexURLResolver`

- Instantiate with string name of a root URLconf
- Call `resolve()` with a URL path to resolve
- Returns tuple of (view, positional args, keyword args)
- Or raises `django.core.urlresolvers.Resolver404`

RegexURLPattern

- Represents a single URL pattern
- `resolve()` method takes a path
- Returns (view, args, kwargs) tuple if it matches

Resolution errors

- `Resolver404` is a subclass of `django.http.Http404`
- Handler will detect this and call resolver's `resolve404()` method
- Works with `Http404` raised from view, too

404 handlers

- Specified by root URLconf's `handler404` attribute
- Default is `django.views.defaults.page_not_found`

Views

- Three requirements to qualify as a Django view
- Callable
- Accepts an `HttpRequest` as first positional argument
- Returns an `HttpResponse` or raises an exception

Simple views

- Just Python functions
- 95% of views “in the wild”

Class-based views

- Class whose instances define `__call__()`
- Basics added in Django 1.3

Class-based views

- Base class: `django.views.generic.View`
- Implements HTTP method dispatch
- `as_view()` returns a callable object

Class-based generic views

- System of view subclasses and mixins
- Declarative attributes/methods on subclasses control behavior

Other class-based views

- One object can also be multiple views
- Can provide its own URL patterns too
- Admin does this

HttpResponse

- Lives in `django.http`
- No gateway-specific subclasses
- Handler converts `HttpResponse` to gateway-appropriate response mechanism

HttpResponse

- Subclasses for HTTP status codes
- 301, 302, 304, 400, 403, 404, 405, 410, 500
- Easy to write your own: override `status_code`

Handling errors

- Most exceptions will cause exception middleware to be applied
- `SystemExit` is not trapped
- Exceptions raised by exception middleware or 404 handler do not apply exception middleware

Handling errors

- Handler's `handle_uncaught_exception()`
- Uses root URLconf's `handler500`
- Default error view: `django.views.defaults.server_error`
- Deliberately uses empty `Context`

Middleware

- Middleware methods can modify request/response and return None
- Or return an `HttpResponse` directly (short-circuits all other request processing)
- Or raise an exception (goes straight to error handling)

Middleware calls

- `process_request()` called before URL resolution
- `process_view()` called after URL resolution
- `process_response()` called after successful `get_response()`
- Exceptions can shortcut processing

Request signals

- `django.core.signals.request_started`
- `django.core.signals.request_finished`
- `django.core.signals.got_request_exception`

Questions?

The admin

AdminSite

- Represents an admin interface
- Knows which models and actions are registered with it
- Can have multiple instances active in a single install

AdminSite

- `urls` delegates to `get_urls()`
- Auth checks (login, logout, etc.) implemented as methods

`ModelAdmin`

- Provides admin options
- And acts as a set of class-based views
- Also uses `MediaDefiningClass` metaclass

ModelAdmin

- Class attributes for easy customization
- Overridable templates
- Overridable forms
- Overridable form fields

Overriding templates

- `add_form_template`
- `change_form_template`
- `change_list_template`
- etc.

Overriding forms

- Set `form` on the `ModelAdmin`
- Or override `get_form()`

Admin forms

- `django.contrib.admin.helpers.AdminForm` implements fieldsets
- `django.contrib.admin.widgets` contains special-case widgets for certain fields

Overriding fields

- Set `fields` or `exclude`
- To control specific fields, define a custom form class
- Or define methods

Overriding fields

- `formfield_for_dbfield()`
- `formfield_for_choice_field()`
- `formfield_for_foreignkey()`
- `formfield_for_manytomany()`

Permission control

- `has_add_permission()`
- `has_change_permission()`
- `has_delete_permission()`
- `get_model_perms()`
- `queryset()`

Logging

- `log_addition()`
- `log_change()`
- `construct_change_message()`
- `log_deletion()`

Views

- add_view()/change_view()/delete_view()/history_view()
- response_add()/response_change()

ChangeList

- `django.contrib.admin.views.main.ChangeList`
- Last bit of truly “legacy” code in admin
- `ModelAdmin.get_changelist()` allows overriding

Questions?