django
el curso

Día 2

Día 2 – Contenido

- Modelos
- API para abstracción de acceso a BD
- *dispatching* de URLs
- Vistas genéricas
- *`django.contrib.admin`*

Día 2 – Contenido

- Modelos
- API para abstracción de acceso a BD
- *dispatching* de URLs
- Vistas genéricas
- *`django.contrib.admin`*

Modelos

- Fuente de definición canónica sobre información en BD
- Clases Python, POO, *django.db.models.Model*
- Módulo *<appname>.models*
- Sintaxis declarativa: atribs. de clase, métodos

```
from django.db import models

class Project(models.Model)
    name = models.CharField(max_length=100)
    created = models.DateTimeField()

    def __unicode__(self):
          return u'proyecto %s' % self.name

class Task(models.Model):
    project = models.ForeignKey(Project)
    ...
```

Modelos
Campos

- *`django.db.models.*Field`*
- Representan columnas en las tablas de la BD
- Determinan: Tipos de columnas SQL, *widgets* en formularios HTML (UI), ...
- Existe una amplia variedad: `AutoField`, `CharField`, `BooleanField`, `DateField`, `TimeField`, `DateTimeField`, `(Positive)IntegerField`, `File/ImageField`, `DecimalField`, `IPAddressField`, ...
- Pueden crearse campos personalizados (nuevos, basados en otros, múltiples campos BD...)

Modelos
Campos

Claves primarias

- Implícita si no se define una manualmente, nombre: *id*, tipo: *AutoField*
- Puede definirse manualmente usando la opción *primary_key*

```
class Project(models.Model)
    # id = models.AutoField(primary_key=True)<-Agregado por Django
    name = models.CharField(max_length=100)

class Milestone(models.Model)
    mi_pk = models.PositiveIntegerField(primary_key=True)
    completed = models.BooleanField()
```

TIP: Actualmente sólo se soportan PK simples.

TIP: propiedad *.pk* - siempre alias de la PK

Modelos
Campos

Opciones

- Hemos visto dos de ellas: *max_length* y *primary_key*
- *null (False)*: A nivel BD. Almacena campos vacíos como *NULL*. Caso especial campos texto
- *blank (False)*: A nivel UI aplicación Admin (leakage?) Permite valores en blanco en forms
- *default*: Valor por omisión

```
name = models.CharField(max_length=100, blank=True)
durac_estimada = models.TimeField(null=True, blank=True,
                                  default=8)
```

(cont)

Modelos
Campos

Opciones (cont'd)

- *unique, db_index, editable, db_column, help_text, ...*
- Nombre *verborrágico*: UI. “Inteligente” automático (+ 's') o primer argumento posicional u opción *verbose_name*

```
desc = models.CharField(max_length=30) # nombre en UI: “desc”
nick = models.CharField(unique=True, db_index=True)

deleted = models.BooleanField(default=False,
                editable=False) # No lo maneja el usuario
fld130 = models.FloatField(db_field='130_xy')
```

(cont)

Modelos
Campos

Opciones (cont'd)

➔ *choices*

```
PROVINCIA_CHOICES = (
    # Valor campo, valor UI
    ('COR', u'Córdoba'),
    ('BA', u'Buenos Aires'),
    ...
    )

class (models.Model)
    provincia = models.CharField(amx_length=3,
                    choices=PROVINCIA_CHOICES)
    ...
```

Es código Python, se lee una vez cuando arranca la aplicación.

Si notamos que debemos editar frecuentemente las opciones, deberíamos estar pensando en un modelo/tabla.

Modelos

Métodos

Permiten agregar o modificar comportamiento a nivel registro de la tabla (instancia del modelo)

- Tienen acceso a `self` y por ende a todos los atributos de la instancia

Podemos:

- Modificar métodos existentes heredados de *`django.db.models.Model`*: `.save()`, `.delete()`
- Implementar métodos usados por Django: `__unicode__()`, `.get_absolute_url()`, `__str__()`
- Implementar nuestros propios métodos, “logica del negocio” en capa M

Modelos

Relaciones

Pueden representarse las 3 relaciones típicas entre tablas del modelo relacional mediante campos en *`django.db.models`*

- Uno a muchos: `ForeignKey`
- Muchos a muchos: `ManyToManyField`
- Uno a uno: `OneToOneField`

Se declaran sólo en uno de los modelos involucrados en la relación.

Modelos

Relaciones

ForeignKey

```
class Poll(models.Model):
    # ...
    camp = models.ForeignKey('Campaign')

class Choice(models.Model):
    # ...
    poll = models.ForeignKey(Poll)

class Campaign(models.Model):
    # ...
```

- Se definen en el *extremo N* de la relación 1 a N
- Primer argumento: El nombre del modelo en el *extremo 1*
- Si el modelo relacionado todavía no ha sido definido o si es un modelo de otra app: Puede usarse un string
- Relaciones recursivas: 'self'

Modelos

Relaciones

ForeignKey

- Acceso desde instancia *extremo N* a instancia *extremo 1* (*forward*): Simplemente el nombre del campo relación
- Acceso desde instancia *extremo 1* a instancia *extremo N* (*backward*): Accesor *automático* `<lower(nombre modelo extremo N)> + "_set"` -- personalizable con opción `related_name`

```
>>> c = Choice.objects.get(pk=1)
>>> c
<Choice: Todo bien>
>>> p = c.poll
>>> p
<Poll: Cómo vá?>
>>> p.choice_set.all()
[<Choice: Todo bien>, <Choice: Bastante mal>, <Choice: Preocupado>]
```

Modelos

Relaciones

ForeignKey -- opciones

- related_name
- to_field - campo de modelo *extremo 1* con el cual se establece la relación
- verbose name - UI, visualización en formularios
- limit_choices_to - restricción adicional en Admin (leakage?)

```
class Doc(models.Model):
    # id = models.AutoField()
    isbn = models.CharField(max_length=14, unique=true)
    autor = models.ForeignKey('contacts.Author')

class Capitulo(models.Model):
    d = models.ForeignKey(Doc, verbose_name = 'Doc. maestro')

class Seccion(models.Model):
    publicacion = models.ForeignKey(Doc, to_field='isbn')
    sec_padre = models.ForeignKey('self')
```

Modelos

Relaciones

ManyToManyField

- Puede definirse en cualquiera de los dos extremos (si se va a usar Admin, se sugiere definir la relación en el modelo que se va a editar con dicha app)
- Django genera una tabla intermedia `<nombre_modelo1_nombre_modelo2>` en forma automática
- Al igual que en las foreign keys:
  - Primer argumento: El nombre del modelo relacionado
  - Si el modelo relacionado todavía no ha sido definido o si es un modelo de otra app: Puede usarse un string
  - Soporta relaciones recursivas: 'self'

Modelos

Relaciones

ManyToManyField

- Acceso desde instancia de modelo que contiene ManyToManyField (*forward*): Simplemente el nombre del campo relación
- Acceso desde instancia del modelo relacionado ('*backward*'): Accesor *automático* `<lower(nombre modelo original)> + “_set”` -- personalizable con opción `related_name`. No se crea en el caso de relaciones recursivas

Modelos

Relaciones

ManyToManyField

Si necesitamos que la relación en si misma posea datos tenemos dos opciones que involucran un 3er. modelo que debemos definir manulmente y que debe contenee FK a los modelos relacionados:

- Usar la opción `through` que nos permite indicar dicho modelo -- esta opción nos provee la posibilidad de crear relaciones y una API para usar los atributos de la relación en queries
- Método *tradicional*: Crear la relación manualmente

Modelos

Relaciones

ManyToManyField -- opciones

- related_name
- through
- verbose name – UI, visualización en formularios
- db_name - (el nombre de la tabla intermedia se genera automáticamente – puedo controlarlo con esta opción).
- symmetrical (=False) – Activar para relaciones M2M recursivas ('self') en las que quiero que se cree el acceso backward
- limit_choices_to – restricción adicional en Admin (leakage?)

Modelos

Relaciones

OneToOneField

- Al igual que en las foreign keys:
  - Primer argumento: El nombre del modelo relacionado
  - Si el modelo relacionado todavía no ha sido definido o si es un modelo de otra app: Puede usarse un string
  - Soporta relaciones recursivas: 'self'

Opciones:

- primary_key – Si deseamos que el campo sea la PK del modelo
- parent_link - asociada con herencia de modelos

Modelos

Inner class Meta

También es posible definir opciones para controlar algunos aspectos de los modelos -- “todo lo que no es un campo”:

- `db_table` (y `db_tablespace`) - Nombre de tabla (y tablespace en backend relevantes) en BD asociada al modelo
- `ordering` (colección) - Criterio de ordenamiento a usarse por omisión cuando se obtienen colecciones de instancias (QueySet's)
- `unique_together` (lista de listas) - Restricciones adicionales de unicidad que involucren valores de varios campos

Modelos

Inner class Meta (cont'd)

- `verbose_name` y `verbose_name_plural` - Para controlar representación textual del nombre del modelo
- `permissions` (lista de duplas) - Permisos adicionales a crearse en aplicación django.contrib.auth para este modelo
- `get_latest_by` (nombre de campo DateField o DateTimeField) - influye en método .latest() de API acceso a BD
- `order_with_respect_to` - Nombre de campo por que podrán ordernarse las instancias del modelo, normalmente una FK
- `abstract` - Usado en herencia de modelos

Día 2 – Contenido

- Modelos
- API para abstracción de acceso a BD
- *dispatching* de URLs
- Vistas genéricas
- *`django.contrib.admin`*

Modelos

API de abstracción de acceso a BD

El ORM de Django nos provee una API parte de la cual ya hemos examinado, veamos algunos puntos salientes

- Creacion de objetos:

```
p = Poll(question=u'Cómo vá?',
       pub_date=datetime.datetime.now())
p.save()
# ó...
p = Poll.create(question=u'Cómo vá?',
       pub_date=datetime.datetime.now())
```

- QuerySet: Representa el resultado de una consulta, es *lazy* (se posterga el acceso a la BD lo mas posible), el atributo de clase `.objects` de los modelos es una fábrica de quesysets, también lo son los propios querysets (encadenables)

Modelos

API de abstracción de acceso a BD (cont'd)

- Obteniendo todos los elementos: .all()

```
>>> Choice.objects.all()
[<Choice: Todo bien>, <Choice: Bastante mal>, <Choice:
Preocupado>]
```

- Realizando consultas y refinando las mismas: .filter(**kwargs), .exclude(**kwargs) - representan SELECT de cláusula SQL; kwargs: Parámetros de *lookup* de campos representan WHERE -- Separados por AND

```
>>> Poll.objects.filter(question__startswith=u'CONFIDENCIAL: ',
        pub_date__year=2008)
[...]
```

Modelos

API de abstracción de acceso a BD (cont'd)

- Refinamiento gradual: Encadenamos filtros aprovechando que filter() y exclude() retornan querysets

```
>>> Choice.objects.filter(...).exclude(...).filter()
[...]
```

- Rebanado de querysets: Con la sintaxis objeto[m:n] conocida de Python (también acceso a un elemento mediante un índice)

```
>>> Poll.objects.all()[3:7]
[...]
>>> Poll.objects.all()[0:2]
[...]
>>> Poll.objects.all().filter(...)[1:]
[...]
>>> Poll.objects.all().exclude(...)[1] # un elemento
<Poll: ...>
```

Modelos

API de abstracción de acceso a BD (cont'd)

Volvamos sobre los *lookups* de campos. La notación es:

```
<nombre campo>__<tipo lookup>=<argumento lookup>
```

Existen varios tipos: `(i)exact`, `(i)contains`, `in`, `gt`, `gte`, `lt`, `lte`, `(i)startswith`, `(i)endswith`, `range`, `year`, `month`, `day`, `isnull`, `search`, `(i)regex`.

En realidad `<nombre campo>` puede expandirse en una secuencia de nombres de campo de relaciones (FK, M2M, O2O) tanto directas como inversas también separados por `__` (joins SQL)

```
>>> Choice.objects.filter(poll__question__iexact=u'¿Quién será el
próximo presidente?')
[...]
>>> Choice.objects.filter(poll__region_set__postcode__exact=5000)
[...]
```

Modelos

API de abstracción de acceso a BD (cont'd)

Otros métodos de la clase QuerySet: Existen dos clases:

- Los que (como .all(), .filter() y .exclude()) retornan querysets (y por ende pueden encadenarse): .order_by(), .reverse(), .distinct, .values(), .values_list(), .dates() (retorna un DatesQuerySet), .select_related(), .extra(), .none()
- Los que no retornan querysets: .get(), .create(), .get_or_create(), .count(), .latest(), .in_bulk(), .iterator()

Por último: los querysets son evaluados cuando: se itera o se usa repr(), len() (!), list() sobre los mismos.

Día 2 – Contenido

- Modelos
- API para abstracción de acceso a BD
- *dispatching* de URLs
- Vistas genéricas
- *`django.contrib.admin`*

URL *dispatching*

- *Mapping* entre URLs usadas para acceder al recurso desde el cliente y el recurso en si mismo
- También se definen con Python, sintaxis declarativa
- Django usa el módulo determinado por el setting `ROOT_URLCONF`, `<appname>.urls`
  - Variable global `urlpatterns`
- Función `django.conf.urls.defaults.patterns()` y otras

URL *dispatching*

```
# urls.py

from django.conf.url.defaults import *

urlpattern = patterns('', # prefijo de vista, opcional
    # ER, vista, [[paramatros extra para la vista], [nombre url]]
    (r'^$', 'timetracker.views.project_list')
    )
```

- Una vista es una función Python que toma un parámetro posicional request no opcional y parámetros extra con nombre
- La vista puede representarse con su nombre debidamente importado (*callable*) o con una cadena con el Python path de la misma

URL *dispatching*

- EL URL mapper invoca la vista pasándole los parámetros de acuerdo a la captura de grupos con nombre que hace de la ER
- Pueden proporcionarse parámetros adicionales a los capturados desde la URL del request HTTP a la vista mediante el 3er. parámetro (opcional): Un diccionario con sus nombres y sus valores
- EL 4to. parámetro (opcional) permite bautizar una entrada en el URL map
- El 1er. argumento de `patterns()` (el prefjo) permite acortar las cadenas de Python paths de vistas similares proveyendo el componente de dicho path común a las mismas.

URL *dispatching*

En adición a `patterns()` existen otras funciones en `django.conf.urls.defaults` que la complementan:

- `include()` - Que reemplaza a la vista y especifica el Python path a otro módulo al cual se delegará la resolución de URLs que tengan el prefixo representado en la ER
- `urls()` - Que reemplaza el tuple completo y permite asignar un nombre a la entrada sin especificar opciones extra para la vista
- `handler404()` y `handler500()` - Nombres de vistas usadas por Django para las respectivas rtas. HTTP, deben estar definidas como símbolos globales en el módulo. Django: `from django.conf.urls.defaults import *`. Podemos proveer las propias

Día 2 – Contenido

- Modelos
- API para abstracción de acceso a BD
- *dispatching* de URLs
- Vistas genéricas
- *`django.contrib.admin`*

Vistas genéricas

Existen patrones comunes de interacción entre aplicaciones Web y sus clientes. Se abstrae esa funcionalidad en vistas reusables muy parametrizables

- Hacen uso extensivo del 3er. elemento de los tuples pasados a `patterns()` para parametrizar su comportamiento
- Debemos proveer:
  - Los datos a *visualizar* - gralmente. con un parámetro *queryset*
  - La apariencia mediante templates

Vistas genéricas

Familias

- Vistas genéricas “simples”
- Vistas genéricas basadas en fecha/hora
- Vistas genéricas de listado/detalle
- Vistas genéricas para ABM de datos (aka CRUD)

Vistas genéricas

Extensibles, cómodas

- Son funciones Python, por ende podemos llamarlas desde nuestras vistas
- Tres técnicas (combinables) para personalizar el funcionamiento de una VG:
  - Modificar el queryset con el que trabaja
  - Proporcionar información adicional al template vía la vista genérica (usando el argumento `extra_context`)
  - Realizar tareas adicionales

Pero... ojo, no volverse dependientes, sopesar cantidad de esfuerzo y código.

Día 2 – Contenido

- Modelos
- API para abstracción de acceso a BD
- *dispatching* de URLs
- Vistas genéricas
- *`django.contrib.admin`*

Aplicación Admin

`django.contrib.admin`

Aplicación opcional “lista” para ABM en BD.

- Implementada en Django
- Personalizable y extensible mediante sintaxis declarativa y código Python
- *Target audience*: Usuarios en los que se confía

Aplicación Admin

`django.contrib.admin`

Cómo activar Admin:

1. Agregar `django.contrib.admin` a setting `INSTALLED_APPS`
2. Seleccionar los modelos de nuestra aplicación que queremos administrar de este modo
3. Configurar el URL map para brindar acceso a Admin
4. Instanciar `django.contrib.admin.AdminSite` (avanzado: subclases custom del mismo y/o mas de una instancia)
5. Registrar los modelos con dicha instancia de `AdminSite` (dependiendo de si queremos personalizar la apariencia y el comportamiento de la funcionalidad de administración del modelo en cuestión podemos hacer esto vía la creación de una subclase de `django.contrib.admin.ModelAdmin`)

Aplicación Admin

`django.contrib.admin`

Normalmente los pasos 4 y 5 se implementan mediante la creación de un módulo `admin.py` en el directorio de la aplicación que contendrá todo lo relacionado a la configuración de Admin para la misma.

```
# admin.py

from django.contrib import admin
from polls.models import Choice

# django.contrib.admin.site es una instancia de AdminSite
admin.site.register(Choice)
```

```
from django.contrib import admin
from polls.models import Choice

class ChoiceAdmin(admin.ModelAdmin)
    pass # no personalizo la sub-clase de ModelAdmin
    # funcionalmente igual al ejemplo anterior

admin.site.register(Choice, ChoiceAdmin)
```

Aplicación Admin

django.contrib.admin

Cuando se usa el `AdminSite` por omisión basta con llamar a `django.contrib.admin.autodiscover()` en nuestro `urls.py` (debe ser llamada solo una vez ya que recorre `INSTALLED_APPS` y ejecuta las registraciones contenidas en los respectivos `admin.py`)

```
# urls.py

from django.contrib import admin
from django.conf.urls.defaults import *

# django.contrib.admin.site es una instancia de AdminSite
admin.autodiscover()

urlpatterns = patterns('',
    ('^admin/(.*)', admin.site.root),
    # ...
)
```

Aplicación Admin

`django.contrib.admin`

Todo lo relacionado con la apariencia de las páginas, formularios, etc. asociados con un modelo en Admin se controla en la sub-clase de ModelAdmin, con una sintaxis que no implica código, similar a la *inner* class Meta de modelos. Para una personalización mas avanzada también pueden implementarse algunos métodos pre-definidos:

- `.save_model(self, request, obj, form, change)` - Punto de convergencia - acceso a usuario actual (mediante request), al objeto que se modifica, al formulario y un Booleano que indica si se está creando o modificando el objeto.
- También existe `.save_formset(...)` pero para inlines

Aplicación Admin

`django.contrib.admin`

Algunas opciones de `ModelAdmin`:

- fields
- fieldsets
- exclude
- list_display
- list_filter
- ordering
- date_hierarchy
- form
- save_as, save_on_top
- ...

This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 2.5 Argentina License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/2.5/ar/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.