Module-02

Django Templates and Models

Template-System Basics

1. **Purpose of a Django Template:**

   - **Separation of Presentation and Data:** A Django template separates the presentation (HTML, text) from the data it displays.
   - **Placeholders and Logic:** Templates contain placeholders for data and basic logic (called template tags) to manage the display of the document.

2. **Common Use:**

   - HTML Generation: Typically used to generate HTML documents, but can generate any text-based format.

3. **Simple Example Template:**

   - The example given creates an HTML page to thank a person for placing an order.

Example Template Explanation

Here is the example template with explanations:

```
<html>

<head><title>Ordering notice</title></head>

<body>

<h1>Ordering notice</h1>
```

```
<p>Dear {{ person_name }},</p>

<p>Thanks for placing an order from {{ company }}. It's scheduled to

ship on {{ ship_date|date:"F j, Y" }}.</p>

<p>Here are the items you've ordered:</p>

<ul>

{% for item in item_list %}

<li>{{ item }}</li>

{% endfor %}

</ul>

{% if ordered_warranty %}

<p>Your warranty information will be included in the packaging.</p>

{% else %}

<p>You didn't order a warranty, so you're on your own when

the products inevitably stop working.</p>

{% endif %}

<p>Sincerely,<br />{{ company }}</p>

</body>

</html>
```

Key Components Explained

HTML Structure:

- <html>, <head>, <body>: Basic HTML tags to structure the document.
- Dynamic Content with Placeholders:
- {{ person_name }}: Placeholder for the person's name.
- {{ company }}: Placeholder for the company's name.
- {{ ship_date|date:"F j, Y" }}: Placeholder for the shipping date, formatted as "Month day, Year" (e.g., June 14, 2024).
- Looping through Items:
- {% for item in item_list %}: Loop through each item in item_list.
- <li>{{ item }}</li>: Display each item in a list item (<li>).
- Conditional Logic:
- {% if ordered_warranty %}: Check if a warranty was ordered.
- If the warranty was ordered, display a message about the warranty.
- {% else %}: If no warranty was ordered, display a different message.
- Django Templates are used to generate dynamic content by separating data and presentation.
- Placeholders and Template Tags help manage and display dynamic data.
- **Example Template:** Demonstrates placeholders for personalizing messages, loops for listing items, and conditions for displaying different messages based on certain criteria (like ordering a warranty).

Using Django Template System

- Django’s template system can be used independently of the rest of Django.
- It allows you to create dynamic text-based content, typically HTML, by separating presentation from data.

Basic Steps

- Create a Template Object: Provide raw template code as a string.
- Render the Template: Use the render () method with a set of variables (context) to get the final rendered string.

```
from django import template

# Step 1: Create a Template object
t = template.Template('My name is {{ name }}.')

# Step 2: Render the template with context
c = template.Context({'name': 'Adrian'})
print(t.render(c))  # Output: My name is Adrian.

c = template.Context({'name': 'Fred'})
print(t.render(c))  # Output: My name is Fred.
```

Detailed Steps

Creating Template Objects:

- Import the Template class from django. template.
- Instantiate the Template class with raw template code.
- The template system compiles the code for efficient rendering

```
from django.template import Template

t = Template('My name is {{ name }}.')

print(t)  # Output: <django.template.Template object at 0xb7d5f24c>
```

Handling Syntax Errors:

- If there are syntax errors, a TemplateSyntaxError is raised.

```
from django.template import Template
t = Template('{% notatag %}')  # Raises TemplateSyntaxError
```

Rendering a Template:

- Create a Context object with variable values.
- Pass the context to the render() method of the template object.

```
from django.template import Context, Template
t = Template('My name is {{ name }}.')
c = Context({'name': 'Stephane'})
print(t.render(c))  # Output: u'My name is Stephane.'
```

Complex Example:

- Use multiline strings for complex templates.
- Create and render the template with context.

```
from django.template import Template, Context
import datetime

raw_template = """<p>Dear {{ person_name }},</p>
<p>Thanks for placing an order from {{ company }}. It's scheduled to
ship on {{ ship_date|date:"F j, Y" }}.</p>
{% if ordered_warranty %}
<p>Your warranty information will be included in the packaging.</p>
{% else %}
<p>You didn't order a warranty, so you're on your own when
the products inevitably stop working.</p>
{% endif %}
<p>Sincerely,<br />{{ company }}</p>"""

t = Template(raw_template)
c = Context({
    'person_name': 'John Smith',
    'company': 'Outdoor Equipment',
    'ship_date': datetime.date(2009, 4, 2),
    'ordered_warranty': False
})

print(t.render(c))
# Output:
# <p>Dear John Smith,</p>
# <p>Thanks for placing an order from Outdoor Equipment. It's scheduled to
# ship on April 2, 2009.</p>
# <p>You didn't order a warranty, so you're on your own when
# the products inevitably stop working.</p>
# <p>Sincerely,<br />Outdoor Equipment</p>
```

- **Django Templates:** Use to separate presentation from data.
- **Placeholders and Tags:** Utilize placeholders for dynamic content and tags for logic.
- **Context:** Pass variable values to templates via context.
- **Render:** Generate the final string by rendering the template with context

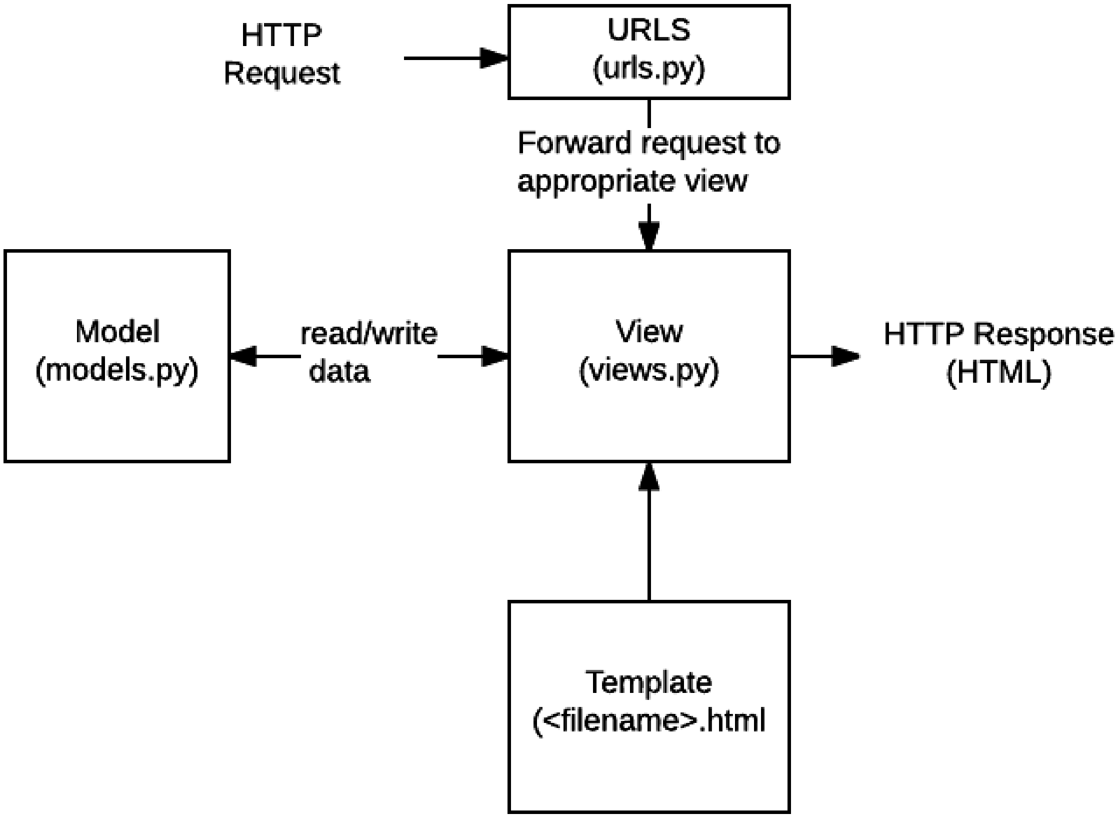


Figure: Django Request-Response Cycle

Basic Template Tags and Filters

Tags

1. **if/else Tag:**
   - Evaluates a variable to determine if it is True.
   - Displays content between {% if %} and {% endif %} if the variable is True.
   - Supports {% else %} for alternate content.
   - Allows and, or, and not for multiple conditions, but does not support combined logical operators.
   - Nested {% if %} tags can be used for complex conditions.

   Example:

```
{% if today_is_weekend %}

<p>Welcome to the weekend!</p>

{% else %}

<p>Get back to work.</p>

{% endif %}
```

2. **for Tag:**
   - Iterates over each item in a sequence.
   - Syntax: {% for X in Y %} where Y is the sequence and X is the variable for each iteration.
   - Supports reversed for reverse iteration.
   - Can nest {% for %} tags and supports {% empty %} for handling empty lists.
   - Provides a **forloop** variable with attributes like counter, counter0, revcounter, revcounter0, first, last, and parentloop.

Example:

```
<ul>
{% for athlete in athlete_list %}
<li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

3. ifequal/ifnotequal Tag:

- Compares two values and displays content if they are equal or not equal.
- Supports an optional {% else %} tag.
- Accepts hard-coded strings, integers, and decimal numbers as arguments but not complex data types.

Example:

```
{% ifequal user currentuser %}
<h1>Welcome!</h1>
{% else %}
<h1>No News Here</h1>
{% endifequal %}
```

4. Comments:

- Single-line comments use {# #}.
- Multiline comments use {% comment %} {% endcomment %}.

Example:

```
{# This is a comment #}

{% comment %}

This is a

multiline comment.

{% endcomment %}
```

Filters

- Filters modify the value of variables before they are displayed.
- Use a pipe character (|) to apply a filter.
- Filters can be chained and some accept arguments.

Examples of Important Filters:

1. **addslashes:**
   - Adds a backslash before backslashes, single quotes, and double quotes

   Example

   ```
   {{ text|addslashes }}
   ```

2. **date:**
   - Formats a date or datetime object according to a format string.

   Example:

   ```
   {{ pub_date|date:"F j, Y" }}
   ```

3. **length:**
   - Returns the length of a value (number of elements in a list or characters in a string).

   Example:

   ```
   {{ items|length }}
   ```

4. **lower:**
   - Converts a string to lowercase.

   Example:

   ```
   {{ name|lower }}
   ```

5. **truncatewords:**

   - Truncates a string to a specified number of words.

   Example:

```
{{ bio|truncatewords:"30" }}
```

MVT Development Pattern

- The Model-View-Template (MVT) is an architectural pattern that separates an application into three interconnected components: the Model, the View, and the Template.
- This pattern helps in building maintainable, scalable, and secure web applications.

Model:

- The Model is responsible for managing the data of the application.
- It handles the logic for creating, reading, updating, and deleting data from the database.
- Models are Python classes that inherit from the django.db. models. Model class.

Example:

```
# models.py

from django.db import models

class Book(models.Model):

    title = models.CharField(max_length=200)

    author = models.CharField(max_length=100)

    publication_date = models.DateField()
```

View:

- The View handles the business logic and controls the flow of the application.
- It receives requests from the user, interacts with the Model to fetch or update data, and renders the appropriate Template.
- Views are Python functions or classes that receive HTTP requests and return HTTP responses.

Example:

```
# views.py
from django.shortcuts import render
from .models import Book
def book_list(request):
    books = Book.objects.all()
    context = {'books': books}
    return render(request, 'book_list.html', context)
```

Template:

- The Template is responsible for presenting the data to the user in an HTML format.
- It defines the structure and layout of the web page.
- Templates are written using Django's template language, which provides tags and filters to control the rendering of dynamic content.

Example:

```
<!-- book_list.html -->

<!DOCTYPE html>

<html>

<head>

    <title>Book List</title>

</head>

<body>

    <h1>Book List</h1>

    <ul>

        {% for book in books %}

        <li>{{ book.title }} by {{ book.author }}</li>

        {% endfor %}

    </ul>

</body>

</html>
```

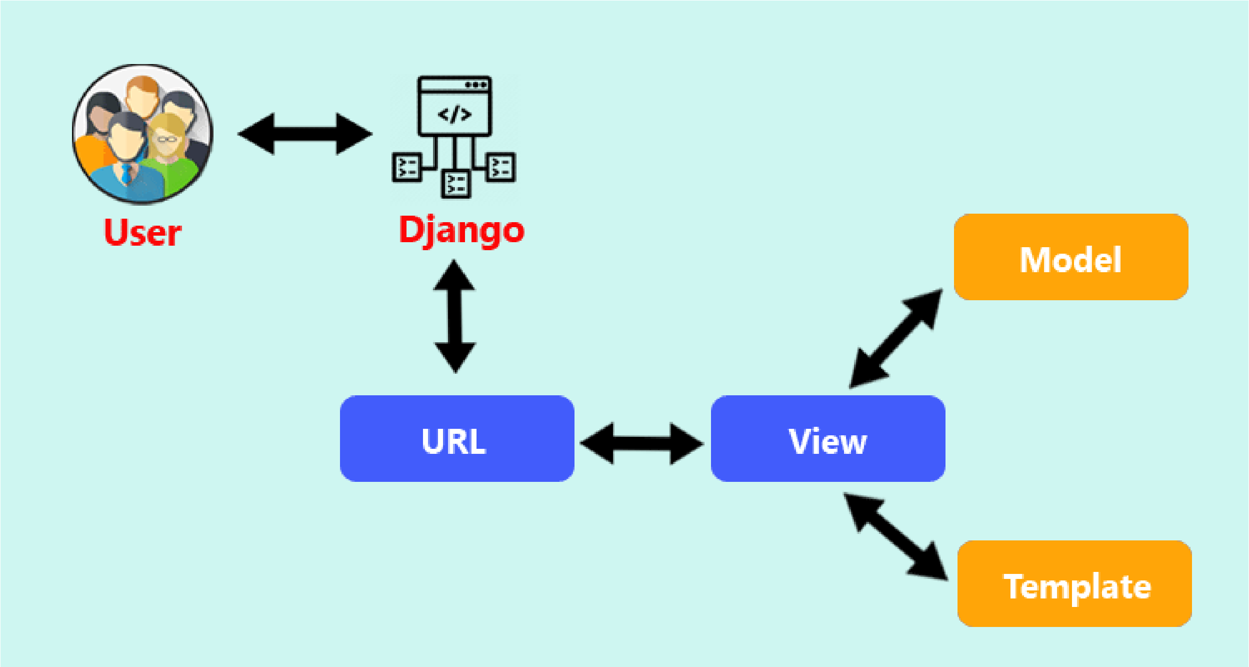


Figure: MVT Pattern

Template Loading

1. **Template Loading in Django:**
   - Django provides a powerful API for loading templates from the filesystem.
   - Templates are loaded using the get_template () function from django. template. loader.

2. **Setting Template Directories:**
   - In the settings.py file, you specify template directories using the TEMPLATE_DIRS setting.
   - Templates can be stored anywhere, but the directory must be readable by the web server user.
   - Absolute paths are recommended, but you can construct paths dynamically using Python code.

3. **Loading Templates Dynamically:**
   - Django settings files are Python code, so you can construct TEMPLATE_DIRS dynamically using Python functions.
   -

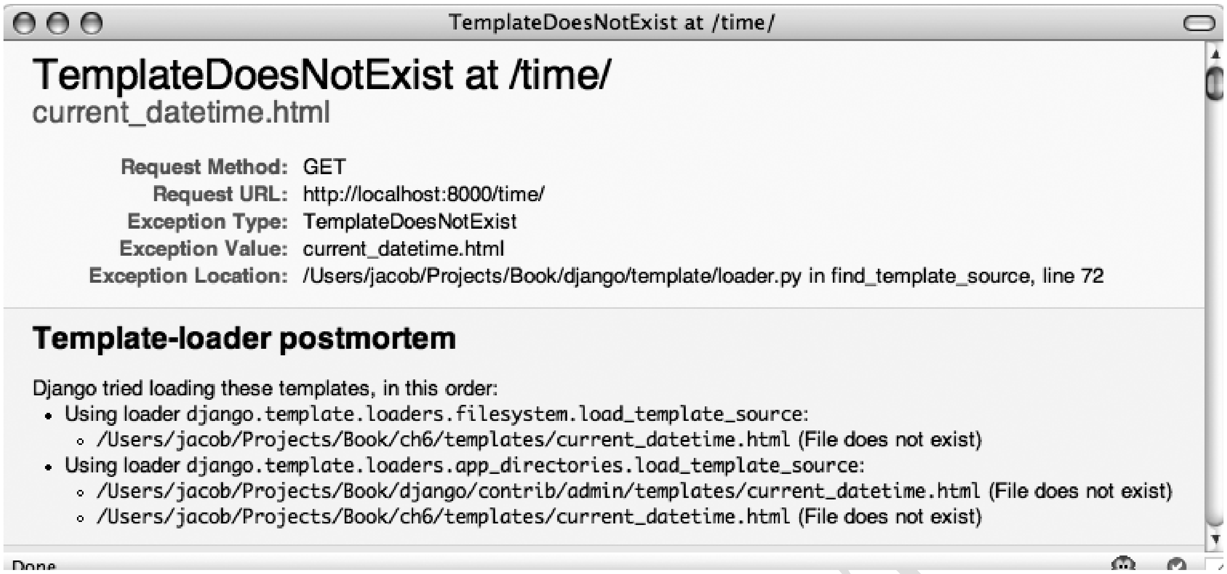
4. **Using render_to_response ():**
   - render_to_response () is a shortcut function that loads a template, renders it, and returns an HttpResponse object in one line.
   - It's commonly used instead of manually loading templates and creating context.

5. **Using locals() to Simplify Context Creation**:
   - locals() is a built-in Python function that returns a dictionary mapping all local variable names to their values.
   - It can be used to simplify passing variables to templates, reducing redundancy.

6. **Organizing Templates with Subdirectories:**
   - Templates can be organized into subdirectories within the template directory.
   - This is recommended for better organization, especially for larger projects.

7. **Using the {% include %} Template Tag:**
   - {% include %} is a built-in template tag used to include the contents of another template.
   - It's useful for reducing duplication when the same code is used in multiple templates.

8. **Behavior of {% include %} Tag:**
   - Included templates are evaluated with the context of the template that includes them.
   - If the included template isn't found, Django raises a TemplateDoesNotExist exception (in DEBUG mode) or fails silently (in production).



TemplateDoesNotExist at /time/

current_datetime.html

```
     Request Method:  GET
        Request URL:  http://localhost:8000/time/
     Exception Type:  TemplateDoesNotExist
    Exception Value:  current_datetime.html
 Exception Location:  /Users/jacob/Projects/Book/django/template/loader.py in find_template_source, line 72
```

Template-loader postmortem

Django tried loading these templates, in this order:

- Using loader `django.template.loaders.filesystem.load_template_source`:
  - `/Users/jacob/Projects/Book/ch6/templates/current_datetime.html` (File does not exist)
- Using loader `django.template.loaders.app_directories.load_template_source`:
  - `/Users/jacob/Projects/Book/django/contrib/admin/templates/current_datetime.html` (File does not exist)
  - `/Users/jacob/Projects/Book/ch6/templates/current_datetime.html` (File does not exist)

Figure: The error page shown when a template cannot be found

Template Inheritance

1. **Purpose of Template Inheritance:**
   - Template inheritance helps reduce duplication and redundancy in HTML pages by allowing common parts to be defined in a single location and reused across multiple templates.

2. **Server-Side Includes vs. Template Inheritance:**
   - While server-side includes (e.g., {% include %}) can be used to include common HTML snippets, template inheritance provides a more elegant solution by defining a base template with blocks that child templates can override.

3. **Defining Base Template:**
    - The base template contains the overall structure of the page and defines blocks using {% block %} tags.
    - Blocks represent areas of the template that can be customized or overridden by child templates.
    - Each block can have a default content, which child templates can choose to override.

Example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
<title>The current time</title>
</head>
<body>
<h1>My helpful timestamp site</h1>
<p>It is now {{ current_date }}.</p>
<hr>
<p>Thanks for visiting my site.</p>
</body>
</html>
```

4. **Using {% extends %}:**
    - Child templates use the {% extends %} tag to indicate that they inherit from a specific base template.
    - The child template overrides specific blocks defined in the base template using {% block %} tags.

Example:

```
{% extends "base.html" %}

{% block title %}The current time{% endblock %}

{% block content %}

<p>It is now {{ current_date }}.</p>

{% endblock %}
```

5. **Benefits of Template Inheritance:**
   - Reduces redundancy by allowing common elements to be defined in a single location.
   - Facilitates easier maintenance and updates, as changes made to the base template automatically reflect in all child templates.
   -

6. **Guidelines for Working with Template Inheritance:**
   - {% extends %} must be the first template tag in a child template.
   - It's better to have more {% block %} tags in base templates to provide flexibility for child templates.
   - Duplicating code across templates indicates the need for moving that code into a {% block %} in the base template.
   - Use {{ block.super }} to access and extend the content of a block from the parent template.
   - Avoid defining multiple {% block %} tags with the same name in the same template.

Example:

```
{% extends "base.html" %}

{% block title %}Future time{% endblock %}

{% block content %}

<p>In {{ hour_offset }} hour(s), it will be {{ next_time }}.</p>

{% endblock %}
```

7. **Dynamic Template Inheritance:**
    - The argument to {% extends %} can be a variable, allowing for dynamic selection of the parent template at runtime.

Overall, template inheritance in Django provides a powerful mechanism for creating maintainable and reusable HTML templates across a web application.

By defining a clear hierarchy of templates, developers can streamline the development process and ensure consistency in the site's appearance and behavior.

MVT Development Pattern

Overall Design Philosophy:

- Django encourages loose coupling and strict separation between components to facilitate easier maintenance and updates.

Model-View-Controller (MVC) Pattern:

- Django follows the MVC pattern, where "Model" represents the data access layer, "View" handles presentation logic, and "Controller" decides which view to use based on user input.

Breakdown in Django:

- Model (M): Handled by Django's database layer, including data access, validation, behaviors, and relationships.
- View (V): Handled by views and templates, responsible for selecting and displaying data on the web page or document.
- Controller (C): Managed by the framework itself, following URLconf and calling appropriate Python functions based on user input.

MTV Framework:

- Django is sometimes referred to as an MTV framework, where "M" stands for "Model," "T" for "Template," and "V" for "View."
- Model (M): Represents the data access layer, covering everything about data and its relationships.
- Template (T): Represents the presentation layer, handling how data should be displayed on a web page.
- View (V): Represents the business logic layer, acting as a bridge between models and templates.

Comparison with Other MVC Frameworks:

- In Django, views describe the data presented to the user, including which data is displayed, not just how it looks.
- Contrary to frameworks like Ruby on Rails, where controllers decide which data to present, Django views are responsible for accessing models and deferring to appropriate templates.

Underlying Concepts:

- Both interpretations of MVC (Django's and others like Ruby on Rails) have their validity, and the key is to understand the underlying concepts rather than adhering strictly to one interpretation.

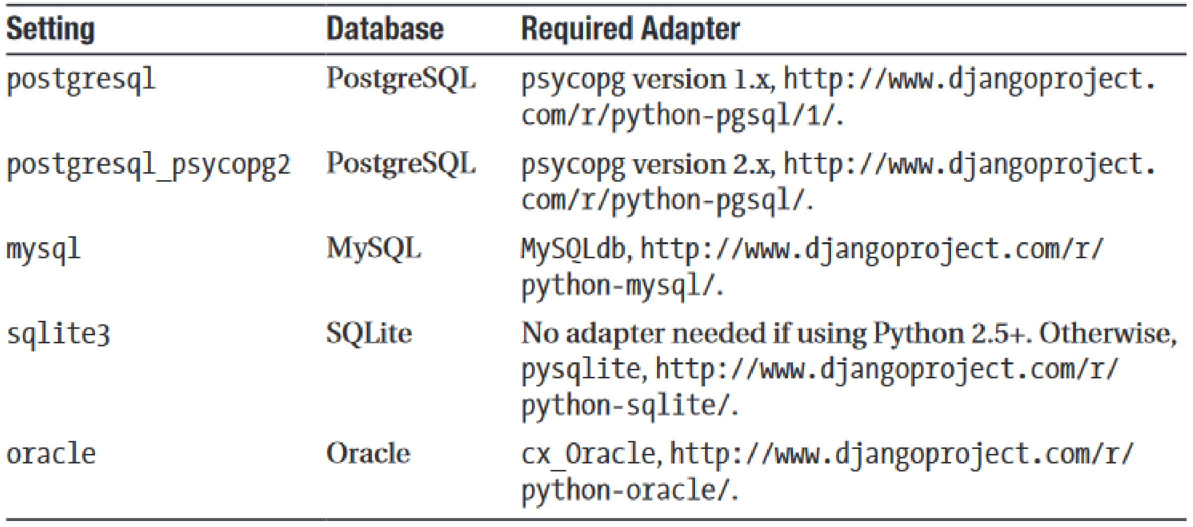
Configuring Databases

Initial Configuration:

- Django requires configuration to connect to the database server.
- Configuration settings are stored in the settings.py file.

Database Settings:

- DATABASE_ENGINE: Specifies the database engine to use. Must be set to one of the available options (e.g., PostgreSQL, MySQL, SQLite).
- DATABASE_NAME: Specifies the name of the database. For SQLite, specify the full filesystem path.
- DATABASE_USER: Specifies the username to use for database access.
- DATABASE_PASSWORD: Specifies the password for database access.
- DATABASE_HOST: Specifies the host for the database server. If using SQLite or the database is on the same computer, leave this blank.
- DATABASE_PORT: Specifies the port for the database server.



Setting	Database	Required Adapter
postgresql	PostgreSQL	psycopg version 1.x, http://www.djangoproject.com/r/python-pgsql/1/.
postgresql_psycopg2	PostgreSQL	psycopg version 2.x, http://www.djangoproject.com/r/python-pgsql/.
mysql	MySQL	MySQLdb, http://www.djangoproject.com/r/python-mysql/.
sqlite3	SQLite	No adapter needed if using Python 2.5+. Otherwise, pysqlite, http://www.djangoproject.com/r/python-sqlite/.
oracle	Oracle	cx_Oracle, http://www.djangoproject.com/r/python-oracle/.

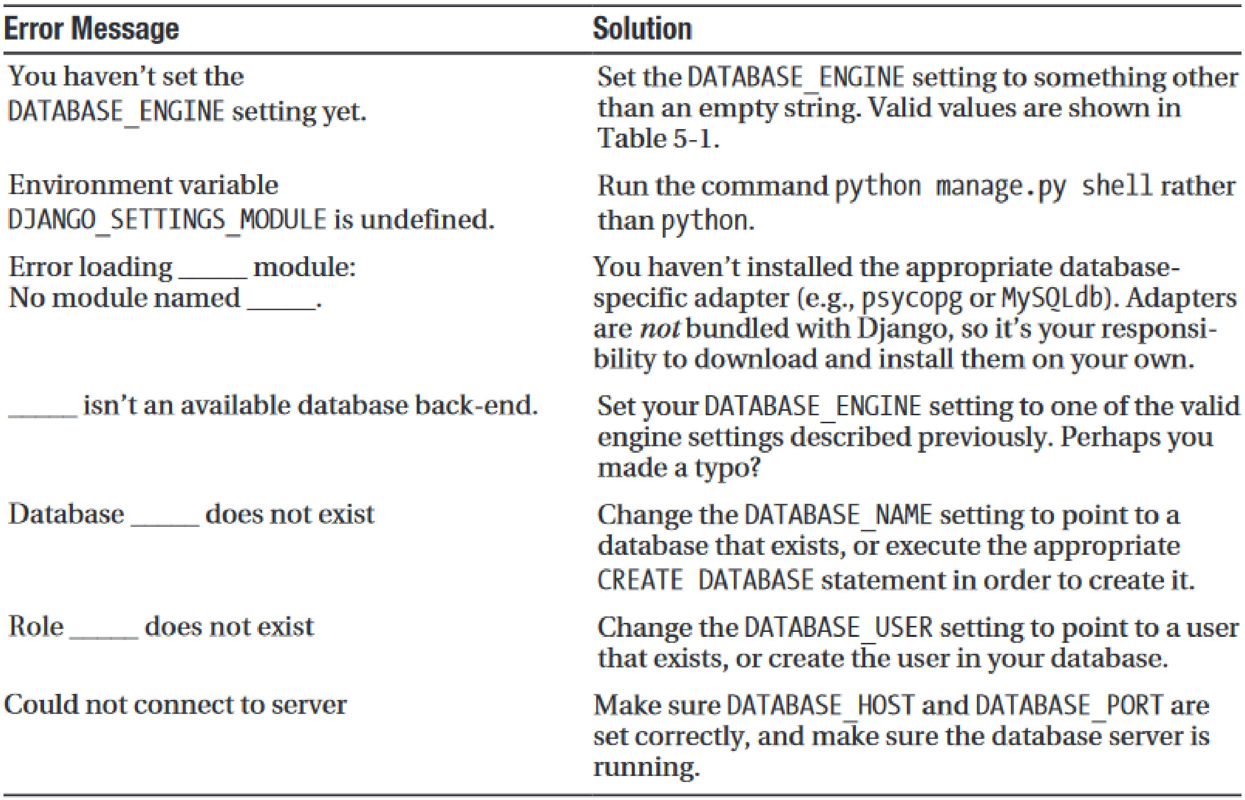
Table: - Database Engine Settings

Testing Configuration:

- After configuring the database settings, it's recommended to test the configuration.
- Use the Django shell (python manage.py shell) to test the connection.
- Import connection from django.db and create a cursor.
- If no errors occur, the database configuration is correct.

Common Errors and Solutions:

- Errors may occur if settings are incorrect or if required database adapters are missing.
- Solutions include setting correct values for settings, installing required adapters, and ensuring database existence and user permissions.



Error Message	Solution
You haven't set the DATABASE_ENGINE setting yet.	Set the DATABASE_ENGINE setting to something other than an empty string. Valid values are shown in Table 5-1.
Environment variable DJANGO_SETTINGS_MODULE is undefined.	Run the command python manage.py shell rather than python.
Error loading _____ module: No module named _____.	You haven't installed the appropriate database-specific adapter (e.g., psycopg or MySQLdb). Adapters are *not* bundled with Django, so it's your responsibility to download and install them on your own.
_____ isn't an available database back-end.	Set your DATABASE_ENGINE setting to one of the valid engine settings described previously. Perhaps you made a typo?
Database _____ does not exist	Change the DATABASE_NAME setting to point to a database that exists, or execute the appropriate CREATE DATABASE statement in order to create it.
Role _____ does not exist	Change the DATABASE_USER setting to point to a user that exists, or create the user in your database.
Could not connect to server	Make sure DATABASE_HOST and DATABASE_PORT are set correctly, and make sure the database server is running.

Table: - Database Configuration Error Messages

Defining and Implementing Models

Introduction to Django Models:

- Django models represent the "M" in the MTV (or MVC) pattern, standing for "Model."
- They describe the data in the database using Python code.

Purpose of Django Models:

- Models serve as the equivalent of SQL CREATE TABLE statements but in Python.
- Django executes SQL code behind the scenes based on these models.
- Models return Python data structures representing rows in database tables.
- They also represent higher-level concepts that SQL alone may not handle effectively.

Reasons for Using Python Models:

Introspection Overhead:

- Introspecting the database at runtime incurs overhead, especially for each request or server initialization.
- Django opts for explicit Python model definitions to reduce this overhead.

Maintaining Context:

- Writing code in Python helps maintain a consistent programming environment, reducing context switches.

Version Control and Ease of Tracking Changes:

- Storing models as Python code facilitates version control, making it easier to track changes to data layouts.

Support for Higher-Level Data Types:

- Django models offer higher-level data types (e.g., for email addresses, URLs) that SQL may lack.
- Consistency Across Database Platforms:
- Distributing a Python module describing data layouts is more pragmatic than separate sets of SQL statements for different databases.

Drawback and Handling Strategies:

- Possibility of Code-Database Sync Issues:
- Changes to Django models may require corresponding changes in the database to maintain consistency.
- Strategies for handling such issues will be discussed later in the chapter.

Utility for Generating Models:

- Django provides a utility to generate models by introspecting an existing database.
- This is particularly useful for quickly integrating legacy data into Django projects.

Your First Model

Introduction:

- This section presents an example of defining Django models for a basic book/author/publisher data layout.

Justification for Example:

- The example focuses on these entities due to their well-known conceptual relationships.
- Books, authors, and publishers form a common data layout used in introductory SQL textbooks.

Concepts, Fields, and Relationships:

- An author entity comprises fields for first name, last name, and email address.
- A publisher entity includes fields for name, street address, city, state/province, country, and website.
- A book entity contains fields for title, publication date, and relationships with authors and a single publisher.

Translation to Python Code:

- In the models.py file of the Django app (created using the startapp command), the following Python code is entered:

```
from django.db import models

class Publisher(models.Model):

    name = models.CharField(max_length=30)

    address = models.CharField(max_length=50)

    city = models.CharField(max_length=60)

    state_province = models.CharField(max_length=30)

    country = models.CharField(max_length=50)
```

```
        website = models.URLField()

    class Author(models.Model):

        first_name = models.CharField(max_length=30)

        last_name = models.CharField(max_length=40)

        email = models.EmailField()


    class Book(models.Model):

        title = models.CharField(max_length=100)

        authors = models.ManyToManyField(Author)

        publisher = models.ForeignKey(Publisher)

        publication_date = models.DateField()
```

```
CREATE TABLE "books_publisher" (

"id" serial NOT NULL PRIMARY KEY,

"name" varchar(30) NOT NULL,

"address" varchar(50) NOT NULL,

"city" varchar(60) NOT NULL,

"state_province" varchar(30) NOT NULL,

"country" varchar(50) NOT NULL,

"website" varchar(200) NOT NULL

);
```

Explanation of SQL Statement:

- The provided SQL statement creates a table named "books_publisher".
- It defines several columns including id, name, address, city, state_province, country, and website.
- The id column serves as the primary key with the serial data type, ensuring unique identifiers for each record.
- Other columns such as name, address, etc., have specified data types (varchar) and length constraints.

Django's Automatic Generation:

- Django can automatically generate the above SQL CREATE TABLE statement based on the model definitions.
- This automation simplifies the process of creating and managing database tables, reducing the manual effort required.

Many-to-Many Relationships:

- In the case of many-to-many relationships, such as the authors field in the Book model, Django creates an additional table (a "join table").
- This join table facilitates the mapping of books to authors without directly adding an authors column to the Book table.

Primary Key Handling:

- Django automatically assigns a primary key to each model if not explicitly defined.
- The default primary key is an autoincrementing integer field named id.
- Django ensures that each model has a single-column primary key, which is a requirement for data integrity.

Basic Data Access

```
# Importing the Publisher model class

from books.models import Publisher


# Creating Publisher objects and saving them to the database

p1 = Publisher(name='Apress', address='2855 Telegraph Avenue', city='Berkeley', state_province='CA', country='U.S.A.', website='http://www.apress.com/')

p1.save()


p2 = Publisher(name="O'Reilly", address='10 Fawcett St.', city='Cambridge', state_province='MA', country='U.S.A.', website='http://www.oreilly.com/')

p2.save()


# Retrieving all Publisher objects from the database

publisher_list = Publisher.objects.all()


# Printing the list of Publisher objects

print(publisher_list)
```

Explanation:

- The provided Python code demonstrates basic data access using Django's high-level Python API.
- Publisher.objects.all() fetches all the Publisher objects from the database.
- The objects are retrieved as a queryset, which is a collection of database objects of a particular model.
- The print(publisher_list) statement displays the list of Publisher objects retrieved from the database. However, the output may appear as [<Publisher: Publisher object>, <Publisher: Publisher object>], as shown in the example, because Django doesn't automatically provide a human-readable representation of objects. To display meaningful information, you can define a __str__() method in the Publisher model class.

Adding Model String Representations

1. **Purpose of Model String Representations:**
    - Model string representations are used to provide a human-readable representation of objects when they are printed or displayed.

2. **Implementing Model String Representations:**

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    def __str__(self):
        return self.name

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

    def __str__(self):
        return f"{self.first_name} {self.last_name}"

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher, on_delete=models.CASCADE)
    publication_date = models.DateField()

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

3. **Explanation of __str__() method:**
    - The __str__() method serves the same purpose as __unicode__() but is used in Python 3 and newer versions of Django.
    - It returns a string representation of the object.

4. **Usage and Effect:**
    - After implementing the __str__() method in the models, objects of these models will display meaningful information when printed or converted to strings.
    - The __str__() method for Publisher returns the name of the publisher, for Author it concatenates the first and last name, and for Book it returns the title.

5. **Updating the shell session:**
    - To see the changes take effect, exit the Python shell and start it again using python manage.py shell.
    - Now, when you retrieve objects from the database and print them, they will display the custom string representation defined by the __str__() method.

6. **Importance of __str__():**
    - Ensure that every model you define has a __str__() method to provide a meaningful string representation of objects.
    - Django uses this method's output in various situations where it needs to display objects, enhancing usability and readability.
    -

1. **Definition of Unicode Objects:**

    - Unicode objects in Python are strings that can handle a wide range of characters, including accented Latin characters, non-Latin characters, curly quotes, and obscure symbols.
    - They can represent over a million different types of characters.

2. **Encoding Comparison:**

    - Normal Python strings are encoded, meaning they use encodings like ASCII, ISO-8859-1, or UTF-8.
    - Handling fancy characters in normal strings requires tracking the encoding, or else the characters might appear incorrectly when displayed.
    - Mixing different encodings can lead to encoding problems, such as displaying "???" or other odd characters.

3. **Characteristics of Unicode Objects:**
    - Unicode objects do not have encoding; they use a consistent set of characters known as Unicode.
    - Working with Unicode objects in Python allows safe mixing and matching without worrying about encoding issues.
    - The __unicode__() method exemplifies how models in Django can have additional behavior beyond representing database tables.
    - It's a demonstration of how models can describe functionality that objects know how to perform.

Inserting/Updating data

Inserting Data:

- To insert a row into the database using Django models, create an instance of the model class with keyword arguments representing the field values.
- Instantiating the model class does not immediately affect the database.
- Data is inserted into the database by calling the save() method on the model instance.

Example:

- Example of inserting data:

```
p = Publisher(name='Apress',
        address='2855 Telegraph Ave.',
        city='Berkeley',
        state_province='CA',
        country='U.S.A.',
        website='http://www.apress.com/')
p.save()
```

Equivalent SQL:

- The SQL equivalent of inserting data

```
INSERT INTO books_publisher
(name, address, city, state_province, country, website)
VALUES
('Apress', '2855 Telegraph Ave.', 'Berkeley', 'CA', 'U.S.A.',
'http://www.apress.com/');
```

Primary Key Handling:

- When saving a new record, Django calculates and sets the primary key value for the record.
- Subsequent calls to save() update the existing record in the database instead of creating a new one.

Updating Data:

- To update data, modify the fields of the model instance and then call save() again.
- All fields are updated, not just the ones that have been changed, which may lead to race conditions in some scenarios.

Example of Update:

- Example of updating data

```
p.name = 'Apress Publishing'
p.save()
```

Equivalent SQL for Update:

- The SQL equivalent of updating data

```
UPDATE books_publisher SET
name = 'Apress Publishing',
address = '2855 Telegraph Ave.',
city = 'Berkeley',
state_province = 'CA',
country = 'U.S.A.',
website = 'http://www.apress.com'
WHERE id = 52;
```

Considerations:

- Depending on the application, updating all fields may cause race conditions.
- Strategies for updating multiple objects in one statement are available, which can be more efficient in certain scenarios.

Selecting and deleting objects

Selecting Objects:

- In web applications, querying existing database records is often more common than creating new ones.
- The process of retrieving all records for a given model in Django.

```
Publisher.objects.all ()
```

Equivalent SQL:

- The Django code above roughly translates to the following SQL query.

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher;
```

Components of Publisher.objects.all():

- Publisher: Refers to the model for which data is being looked up.
- objects: Represents a manager, which handles table-level operations on data, including data lookup. Every model automatically gets an objects manager.
- all(): A method on the objects manager that returns all rows in the associated database table. It returns a QuerySet object.

QuerySets:

- QuerySets represent a specific set of rows from the database.
- Although QuerySets resemble lists, they are objects with additional functionalities.
- QuerySets are covered in detail in Appendix C.

General Pattern:

- Any database lookup in Django follows the pattern of calling methods on the manager attached to the model being queried.

Filtering Data:

- It's common to need a subset of data from the database rather than retrieving everything at once.
- Django's API allows filtering of data using the filter () method.

Usage:

- The filter () method takes keyword arguments, which are translated into SQL WHERE clauses.

Example:

```
Publisher.objects. filter(name='Apress')
```

This translates to:

```
SELECT id, name, address, city, state_province, country, website

FROM books_publisher

WHERE name = 'Apress';
```

Using get() Method:

- The get() method retrieves a single object from the database that matches the specified criteria.

Example:

```
Publisher.objects.get(name="Apress")
```

- This returns a single object instead of a QuerySet.

Exceptions:

- If the query results in multiple objects, a MultipleObjectsReturned exception is raised.

Example:

```
Publisher.objects.get(country="U.S.A.")
```

- If no object matches the query criteria, a DoesNotExist exception is raised.

Example:

```
Publisher.objects.get(name="Penguin")
```

Exception Handling:

It's important to handle these exceptions in your code to prevent crashes.

Example:

```
try:
    p = Publisher.objects.get(name='Apress')
except Publisher.DoesNotExist:
    print("Apress isn't in the database yet.")
else:
    print("Apress is in the database.")
```

Exceptions Details:

- DoesNotExist and MultipleObjectsReturned exceptions are attributes of the model's class.
- Publisher.DoesNotExist and Publisher.MultipleObjectsReturned respectively.

Deleting Single Object:

- To delete a single object from the database, call the object's delete() method.

Example:

```
p = Publisher.objects.get(name="O'Reilly")

p.delete()
```

- After deletion, the object is removed from the database.

Bulk Deletion:

- Objects can also be deleted in bulk by calling delete() on the result of any QuerySet.

Example:

```
Publisher.objects.filter(country='USA').delete()

Publisher.objects.all().delete()
```

- The first line deletes all objects with the country set to 'USA', while the second line deletes all objects from the Publisher table.

Precautions:

- Django requires explicit use of all() if you want to delete everything in a table to prevent accidental deletion of all data.

Example:

```
Publisher.objects.all().delete()
```

- Deleting all data from a table without using all() will result in an AttributeError.

Subset Deletion:

- If you're deleting a subset of data, you don't need to include all().

Example:

```
Publisher.objects.filter(country='USA').delete()
```

- This line deletes only the objects where the country is set to 'USA'.

Schema Evolution

Understanding Database Schemas:

- Understand the concept of a database schema and how it represents the structure of a database.
- Learn about tables, columns, data types, primary keys, and foreign keys.

Django Models and Database Migrations:

- Understand how Django models map to database tables and how model fields correspond to table columns.
- Learn about Django's migration system and how it helps manage schema changes over time.

Adding New Fields:

- Learn how to add new fields to an existing model and create migrations for the same.
- Understand the concept of default values and how to handle existing data when adding new fields.

Modifying Existing Fields:

- Learn how to modify the properties of existing fields (e.g., changing field types, max lengths, etc.).
- Understand the implications of modifying fields and how to handle existing data during field modifications.

Removing Fields:

- Learn how to remove fields from a model and create migrations for the same.
- Understand the implications of removing fields and how to handle data loss or data preservation.

Renaming Fields:

- Learn the process of renaming fields in Django, as it doesn't provide a direct way to do so.
- Understand the concept of creating a new field, migrating data, and removing the old field.

Data Migrations:

- Learn about Django's RunPython operation in migrations, which allows executing custom Python code during schema changes.
- Understand how to use data migrations to handle complex data transformations or custom logic during schema evolution.

Testing and Deployment:

- Learn about testing strategies for schema changes, including creating backups and running migrations in a staging environment.
- Understand the importance of thoroughly testing migrations before applying them to a production database.

Schema Evolution Best Practices:

- Learn about best practices for schema evolution, such as incremental changes, documenting changes, and maintaining backward compatibility.
- Understand the importance of version control and collaboration when managing schema changes in a team environment.