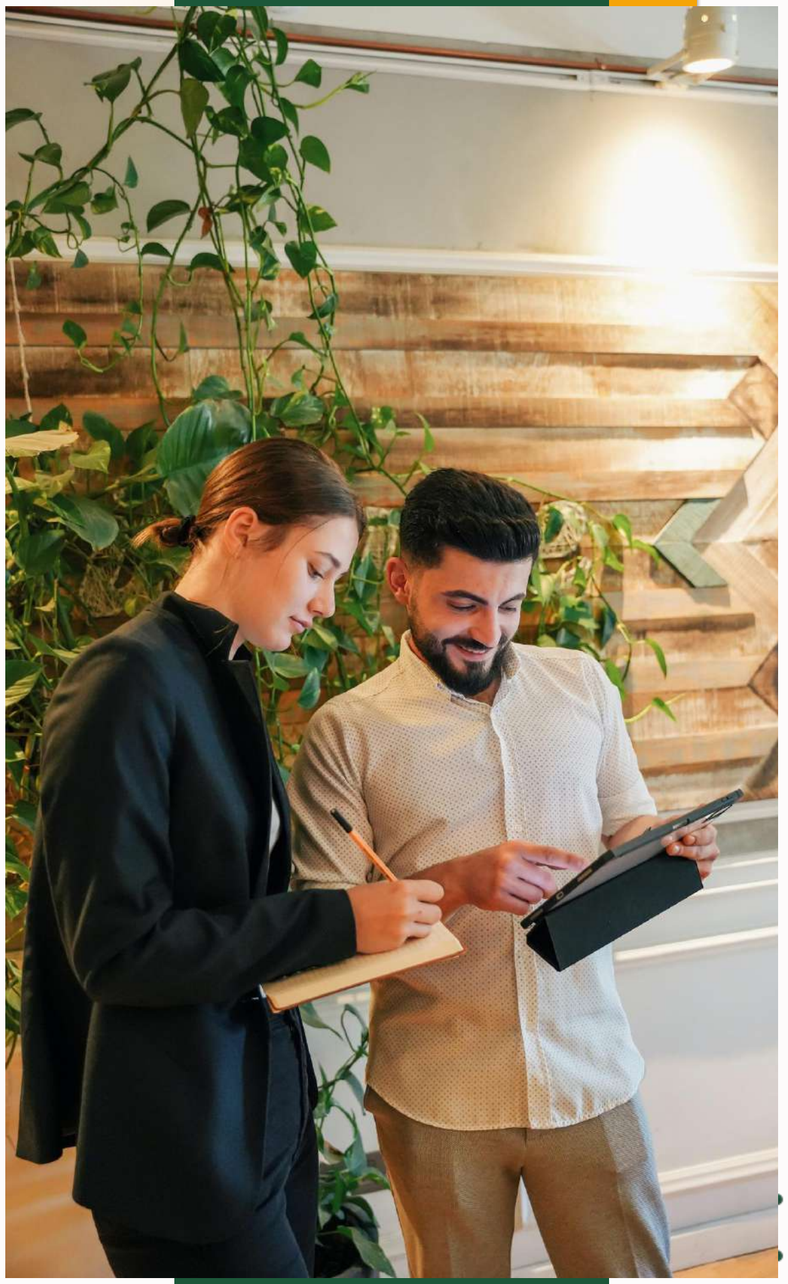
21CS62

FULLSTACK DEVELOPMENT



Course outcome (Course Skill Set) At the end of the course the student will be able to:

CO 1. Understand the working of MVT based full stack web development with Django.

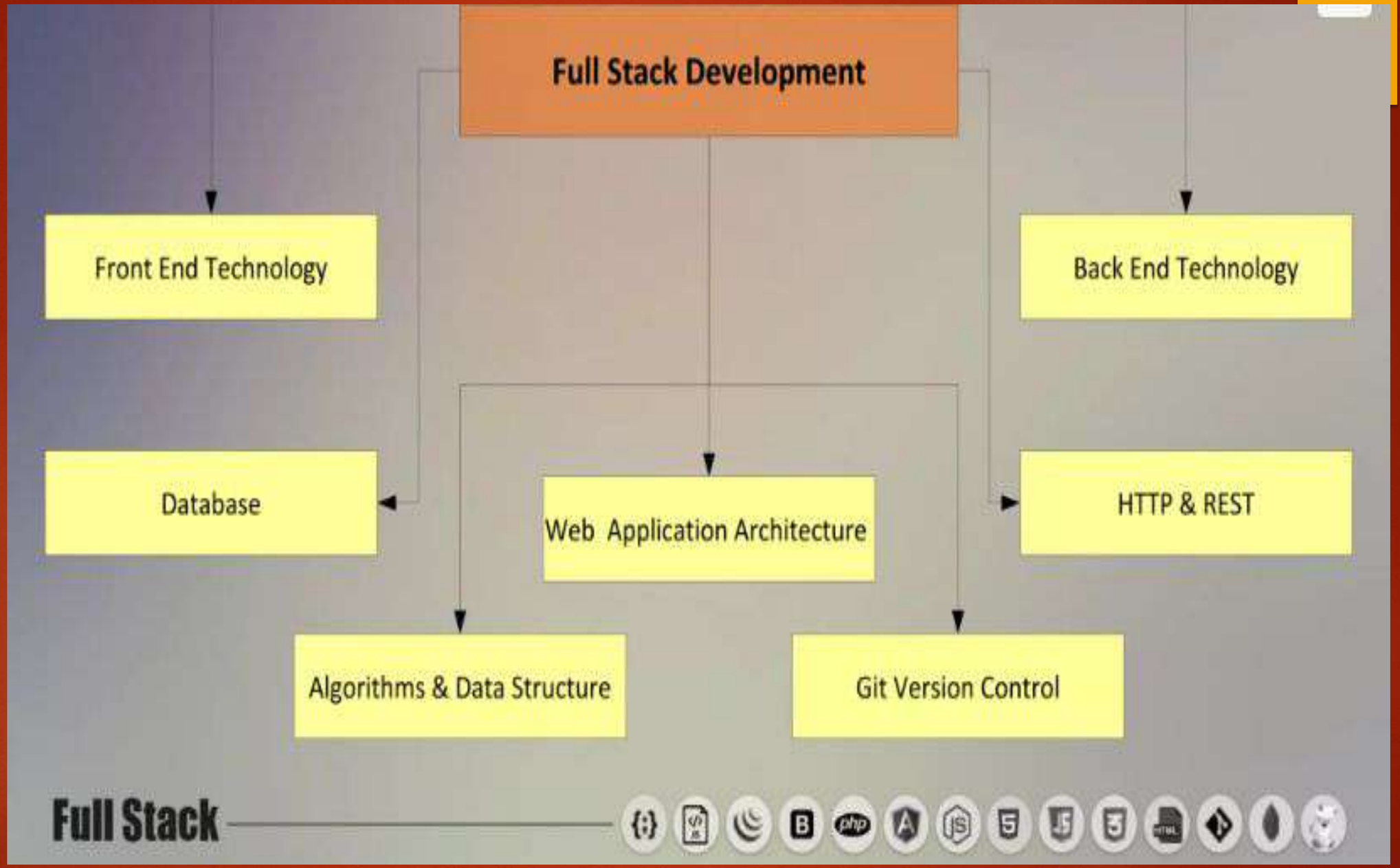
CO 2. Designing of Models and Forms for rapid development of web pages.

CO 3. Analyze the role of Template Inheritance and Generic views for developing full stack web applications.

CO 4. Apply the Django framework libraries to render nonHTML contents like CSV and PDF.

CO 5. Perform jQuery based AJAX integration to Django Apps to build responsive full stack web applications,

- **Definition**: Full-stack development involves working on both the front-end and back-end of an application.
- **Skills Required:** Proficiency in both front-end and back-end technologies.
- **Front-end development** involves creating the user interface and user experience (UI/UX) of the application, typically using technologies such as HTML, CSS, and JavaScript, along with frameworks like React, Angular, or Vue.js.
- **Back-end development** involves working with the server-side logic, databases, and APIs (Application Programming Interfaces) to ensure that the application works smoothly and can handle tasks such as data storage, retrieval, and processing. Back-end development often involves languages like Python, Java, Node.js, or PHP, and frameworks like Django, Flask, Spring, or Express.js.
- **Databases**: MySQL, MongoDB, PostgreSQL
- **Web Servers:** Apache, Nginx
- **Deployment:** AWS, Azure


Full Stack Development
Front End Technology
Back End Technology
Database
Web Application Architecture
HTTP & REST
Algorithms & Data Structure
Git Version Control
Full Stack

Brief history of full-stack development:

- **Early Web Development (1990s)**: Developers primarily focused on front-end technologies like HTML, CSS, and JavaScript.
- **Introduction of Server-Side Scripting (Late 1990s)**: As websites became more dynamic and interactive, server-side scripting languages like PHP, Perl, and ASP (Active Server Pages) were introduced. This allowed developers to generate dynamic content on the server before sending it to the client's browser.
- **Rise of Frameworks (2000s)**: The 2000s saw the rise of web development frameworks like Ruby on Rails, Django, and ASP.NET MVC. These frameworks provided developers with tools and libraries to streamline the development process and build more complex web applications.
- **Separation of Concerns (2000s)**: The concept of "Separation of Concerns" became more prevalent, advocating for the separation of presentation (front-end), application logic (back-end), and data storage (database).
- **Full-Stack Development (2010s)**: In the 2010s, as web applications became more complex and interconnected, the concept of full-stack development emerged. Full-stack developers were expected to have a broad range of skills, including proficiency in both front-end and back-end technologies.
- **Modern Full-Stack Development (Present)**: Today, full-stack development has become a standard approach to web development, with developers expected to have knowledge of front-end technologies like HTML, CSS, and JavaScript, as well as back-end technologies like Node.js, Python, or Java. Frameworks and tools like React, Angular, Vue.js, and Express.js have further facilitated full-stack development by providing developers with efficient ways to build modern web applications.

FULLSTACK DEVELOPER

FRONTEND

- HTML
- CSS
- JAVASCRIPT
- REACT
- ANGULAR

BACKEND

- PYTHON
- JAVA
- PHP
- RUBY
- DJANGO

DATABASE

- MY SQL
- MONGODB
- POSTGRE SQL
- ORACLE
- FIREBASE

TOOLS

- VS CODE
- GIT
- GITHUB

Examples of Software Developed Using Full-Stack Development

- **Netflix**: The streaming giant uses React.js for the front-end and Node.js for the back-end. This allows for a fast and seamless user experience, with Node.js handling the high traffic demands.
- **Airbnb**: Airbnb's platform is built using React.js for the front-end and Ruby on Rails for the back-end. This combination provides a user-friendly interface and efficient backend processing for booking and managing accommodations.
- **Facebook**: Facebook uses React.js for its front-end, providing a highly interactive and responsive user interface. For the back-end, Facebook uses a combination of technologies including PHP, Python, and C++.
- **LinkedIn**: LinkedIn's platform is built using a combination of front-end technologies like React.js and back-end technologies like Node.js, Java, and Scala. This allows for a seamless user experience and efficient data processing.
- **Twitter**: Twitter uses a combination of front-end technologies like React.js and back-end technologies like Node.js, Java, and Scala. This allows for real-time updates and a responsive user interface.
- **Instagram**: Instagram's platform is built using React.js for the front-end and Python for the back-end. This combination provides a visually appealing and user-friendly interface for sharing photos and videos.

Full Stack Developer

freelancermap

Responsibilities

- Software and web dev.
- Planning, and execution of test runs
- Managing complex projects

Background

- Bachelor's degree in Computer Science, IT, or Programming
- Masters is valued

Skills

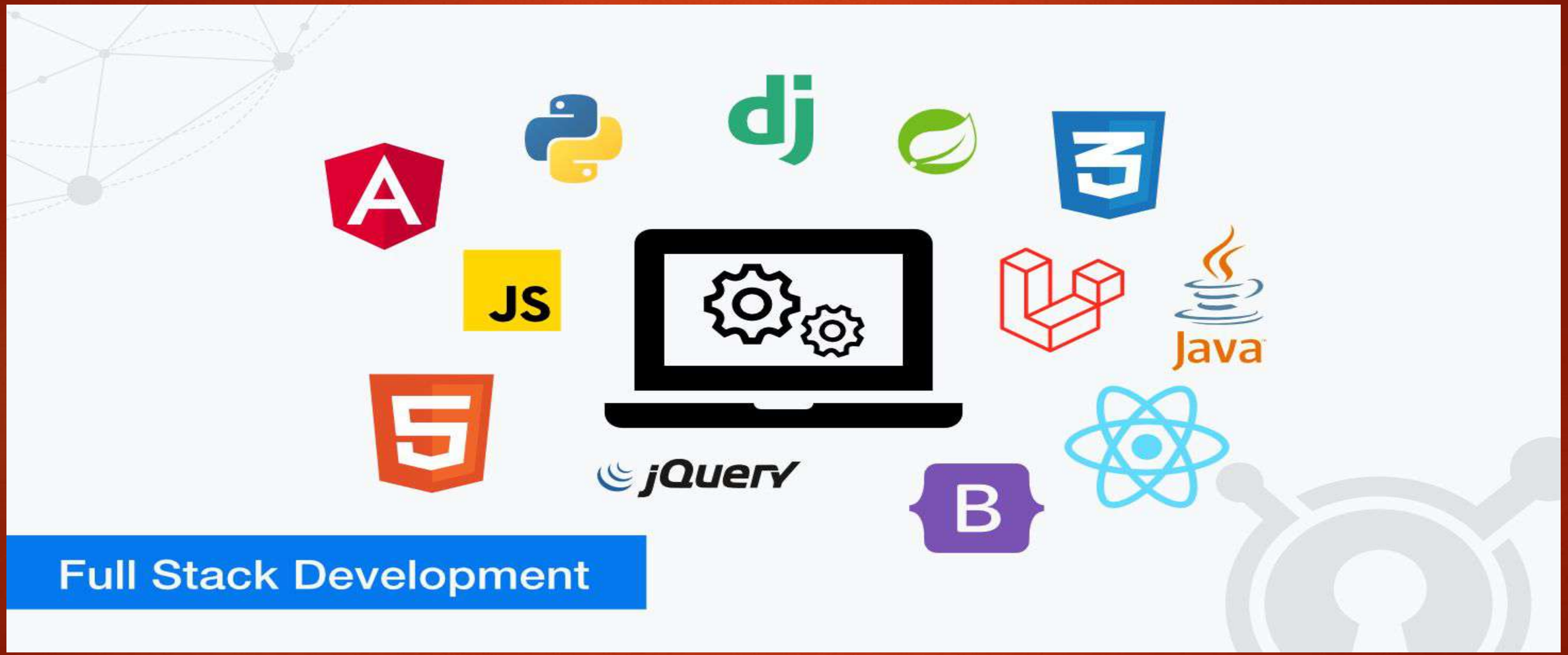
- Comprehensive knowledge in programming and web development
- Profound mathematical knowledge

Salary

Junior:	$ 50,000
Average:	$ 75,000
Senior:	$ 120,000

Career Opportunities

Software Used for Full-Stack Development



Django

Full Stack Developer

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It is well-suited for full-stack development, as it provides tools and features for both the front-end and back-end components of a web application.

Django can be used for full-stack development:

- **Back-End Development**:
  - Django provides a powerful ORM (Object-Relational Mapping) system that allows you to define your data models using Python classes. These models are then translated into database tables, making it easy to work with databases such as SQLite, PostgreSQL, MySQL, and Oracle.
  - Django includes a templating engine that allows you to create HTML templates with Python-like syntax, making it easy to generate dynamic content.

- **Front-End Development**:
  - While Django is primarily a back-end framework, it can be used to serve HTML templates and static files (such as CSS, JavaScript, and images) to the client.
  - Django integrates well with front-end frameworks like React, Angular, or Vue.js. You can use Django as a back-end API to handle data and business logic, while the front-end framework handles the presentation layer.

Full-Stack Development Workflow:

- For full-stack development with Django, you would typically start by defining your data models and setting up your database using Django's ORM.
- You would then create views to handle requests from the client and render templates or serve JSON data as needed.
- In the front-end, you can use HTML templates and static files served by Django, or you can use a front-end framework to create a single-page application (SPA) that communicates with Django's back-end API.

Module-1: MVC based Web Designing Web framework, MVC Design Pattern, Django Evolution, Views, Mapping URL to Views, Working of Django URL Confs and Loose Coupling, Errors in Django, Wild Card patterns in URLS.

Textbook 1: Chapter 1 and Chapter 3

Textbooks

1. Adrian Holovaty, Jacob Kaplan Moss, The Definitive Guide to Django: Web Development Done Right, Second Edition, Springer-Verlag Berlin and Heidelberg GmbH & Co. KG Publishers, 2009

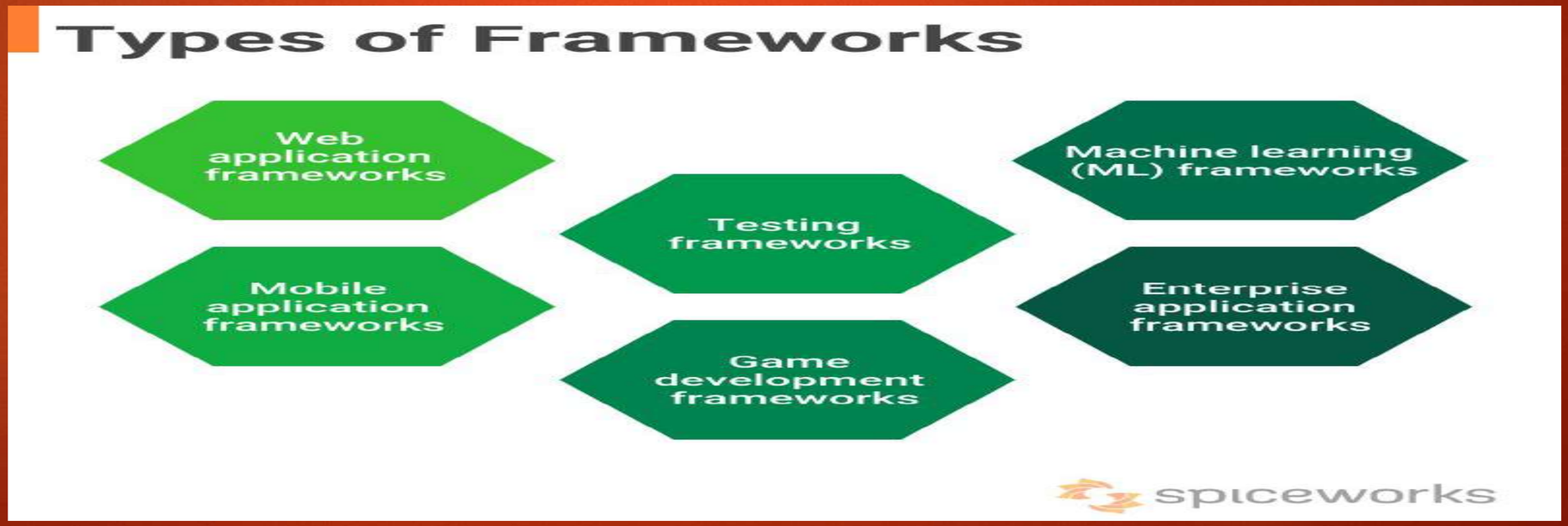
2. Jonathan Hayward, Django Java Script Integration: AJAX and jQuery, First Edition, Pack Publishing, 2011

Overview

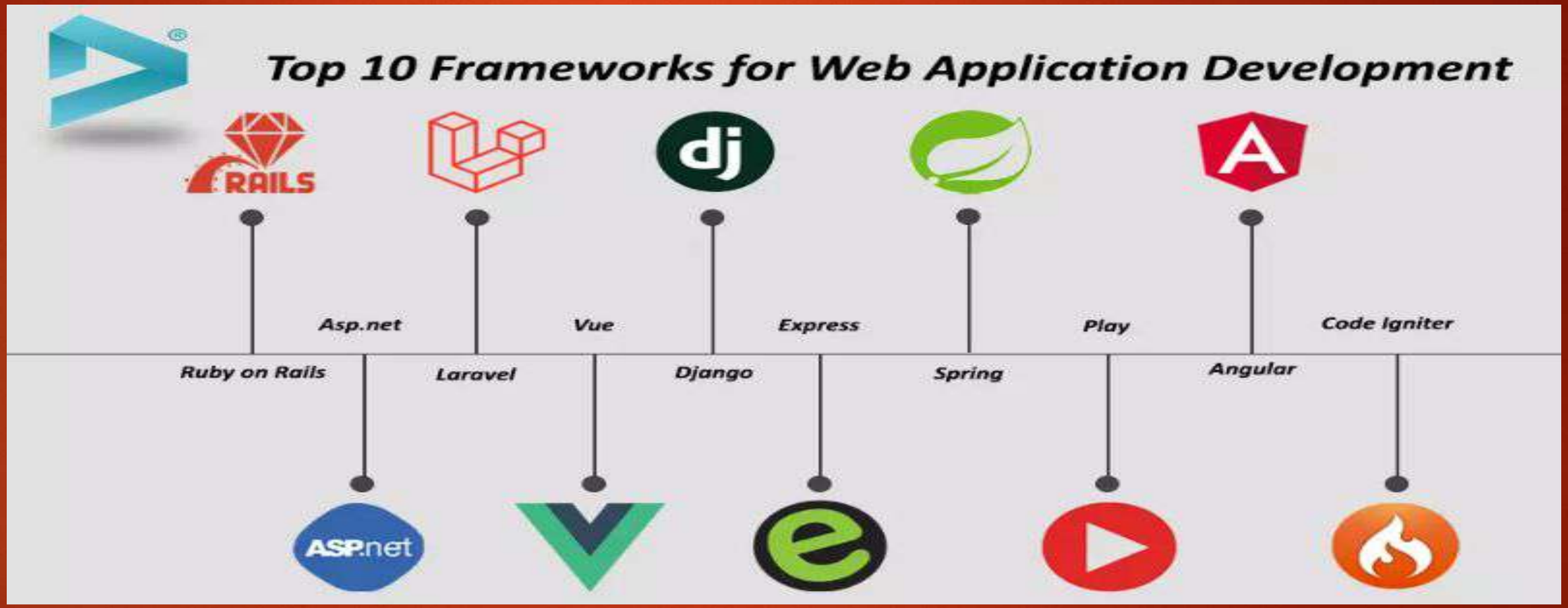
- Web framework
- MVC Design Pattern
- Django Evolution
- Views
- Mapping URL to Views
- Working of Django URL Confs and Loose Coupling
- Errors in Django
- Wildcard Patterns in URL’s

Web Framework

- Django is a free and open source web application framework , written in python programming language , which offers fast and effective dynamic website development.
- **What is framework?????????**
- A framework is a set of tools, libraries, and conventions that provide structure and guidance for developing software applications.



- A web framework is a code library that makes web development faster and easier by providing common patterns for building reliable, scalable and maintainable web applications



A simple CGI script, written in Python, that displays the ten most recently published books from a database

```
#!/usr/bin/python
import MySQLdb
print "Content-Type: text/html"
print
print "<html><head><title>Books</title></head>"
print "<body>"
print "<h1>Books</h1>"
print "<ul>"
connection = MySQLdb.connect(user='me', passwd='letmein', db='my_db')
cursor = connection.cursor( )
cursor.execute("SELECT name FROM books ORDER BY pub_date DESC LIMIT 10")
for row in cursor.fetchall( ):
print "<li>%s</li>" % row[0]
print "</ul>"
print "</body></html>"
connection.close( )
```

- What happens when multiple pages need to connect to the database?

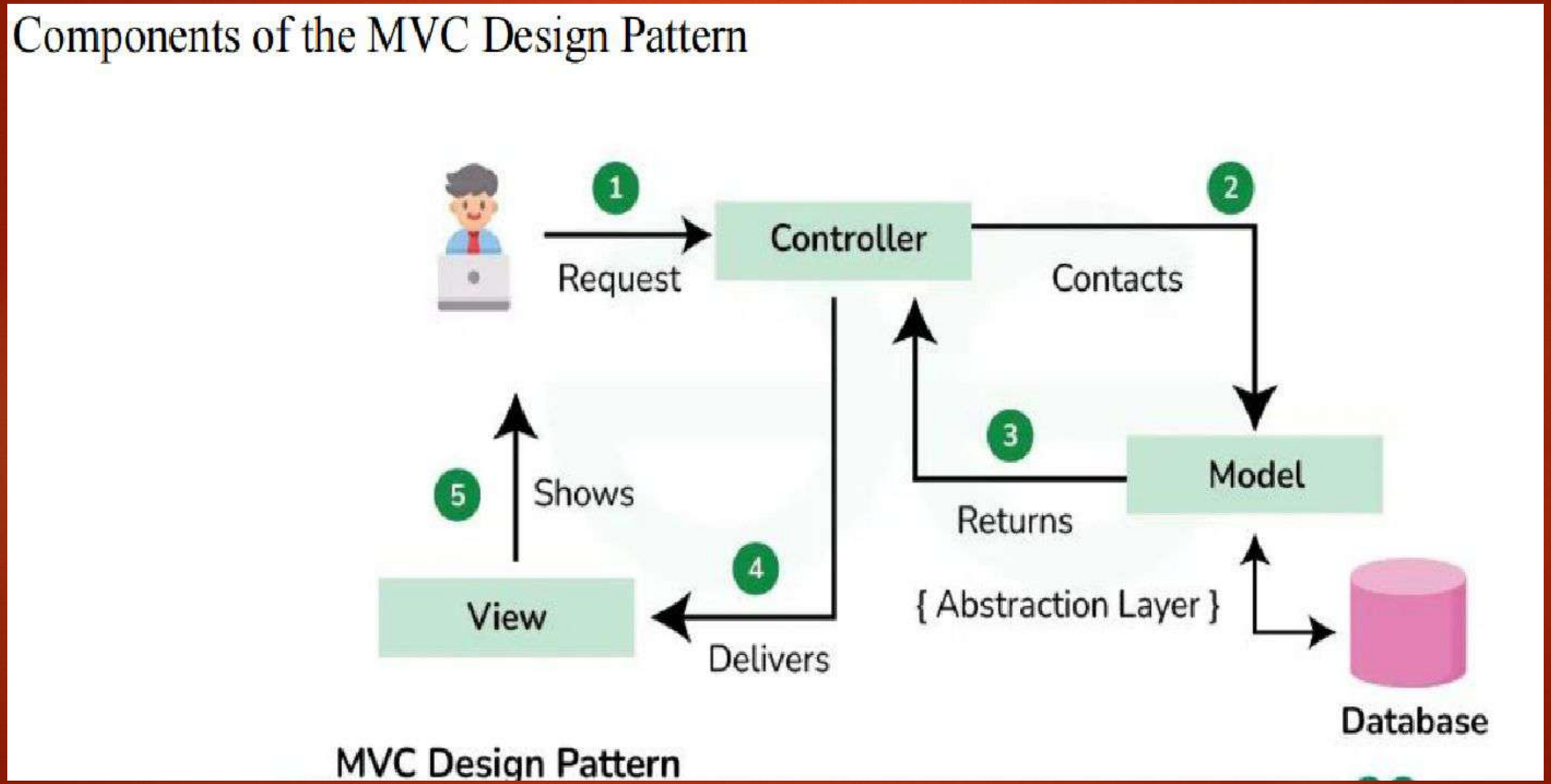
- What happens when a Web designer who has no experience coding Python wishes to redesign the page? Ideally, the logic of the page—the retrieval of books from the database— would be separate from the HTML display of the page, so that a designer could edit the latter without affecting the former.

- These problems are precisely what a Web framework intends to solve.
- A Web framework provides a programming infrastructure for your applications, so that you can focus on writing clean,maintainable code . In a nutshell, that’s what Django does.

Django advantages

- **DRY Principle (Don't Repeat Yourself):** Django emphasizes writing clean, maintainable code by reducing repetition.
- **Built-in Admin Interface:** Django provides a powerful admin interface that allows developers to quickly create, update, and delete content from the application's database. This feature is especially useful for content management systems (CMS) and internal tools.
- **ORM (Object-Relational Mapping):** Django's ORM simplifies database operations using Python classes. This eliminates the need to write SQL queries manually
- **Security Features:** Django is designed with security in mind. It provides protection against common security threats such as SQL injection etc
- **Scalability:** Django is capable of handling high traffic and scaling horizontally. It provides built-in support for caching, database sharding, and load balancing
- **Community and Ecosystem:** Django has a large and active community of developers, which means there are plenty of resources, tutorials, and third-party packages available.
- **Versatility:** Django is versatile and can be used to build a wide range of web applications, from simple websites to complex web applications.

MVC Design Pattern



1. Model

- The Model component in the MVC (Model-View-Controller) design pattern represents the data and business logic of an application.
- It is responsible for **managing the application's data, processing business rules, and responding to requests for information from other components**, such as the View and the Controller.

```
# model.py
class Post:
    def __init__(self, title, content):
        self.title = title
        self.content = content
```

```
posts = [
    Post("First Post", "This is my first blog post."),
    Post("Second Post", "This is my second blog post."),
    Post("Third Post", "This is my third blog post.")
]
```

2. View

- Displays the data from the Model to the user
- sends user inputs to the Controller.
- It is **passive** and does not directly interact with the Model.
- Instead, **it receives data from the Model and sends user inputs to the Controller for processing.**

```
<!-- templates/index.html -->
<!DOCTYPE html>
<html>
<head>
    <title>My Blog</title>
</head>
<body>
    <h1>Welcome to My Blog</h1>
    {% for post in posts %}
    <div>
        <h2>{{ post.title }}</h2>
        <p>{{ post.content }}</p>
    </div>
    {% endfor %}
</body>
</html>
```

3. Controller

- Controller acts as an intermediary between the Model and the View.
- It handles user input and updates the Model accordingly and updates the View to reflect changes in the Model.
- It contains application logic, such as input validation and data transformation.

```
# app.py
from flask import Flask, render_template
from model import posts
app = Flask(__name__)
@app.route('/')
def index():
    return render_template('index.html', posts=posts)

if __name__ == '__main__':
    app.run(debug=True)
```

Django MVT (Model-View-Template)

How does Django Work?

Django follows the MVT design pattern (Model View Template).

- Model - The data you want to present, usually data from a database.
- View - A request handler that returns the relevant template and content - based on the request from the user.
- Template - A text file (like an HTML file) containing the layout of the web page, with logic on how to display the data.

Model

- The model provides data from the database.
- In Django, the data is delivered as an **Object Relational Mapping (ORM),** which is a technique designed to make it easier to work with databases.
- Django, with ORM, makes it easier to communicate with the database, without having to write complex SQL statements.
- The models are usually located in a file called `models.py`.

View

- A view is a function or method that takes http requests as arguments,
- imports the relevant model(s), and finds out what data to send to the template, and returns the final result.
- The views are usually located in a file called `views.py`.

Template

A template is a file where you describe how the result should be represented.

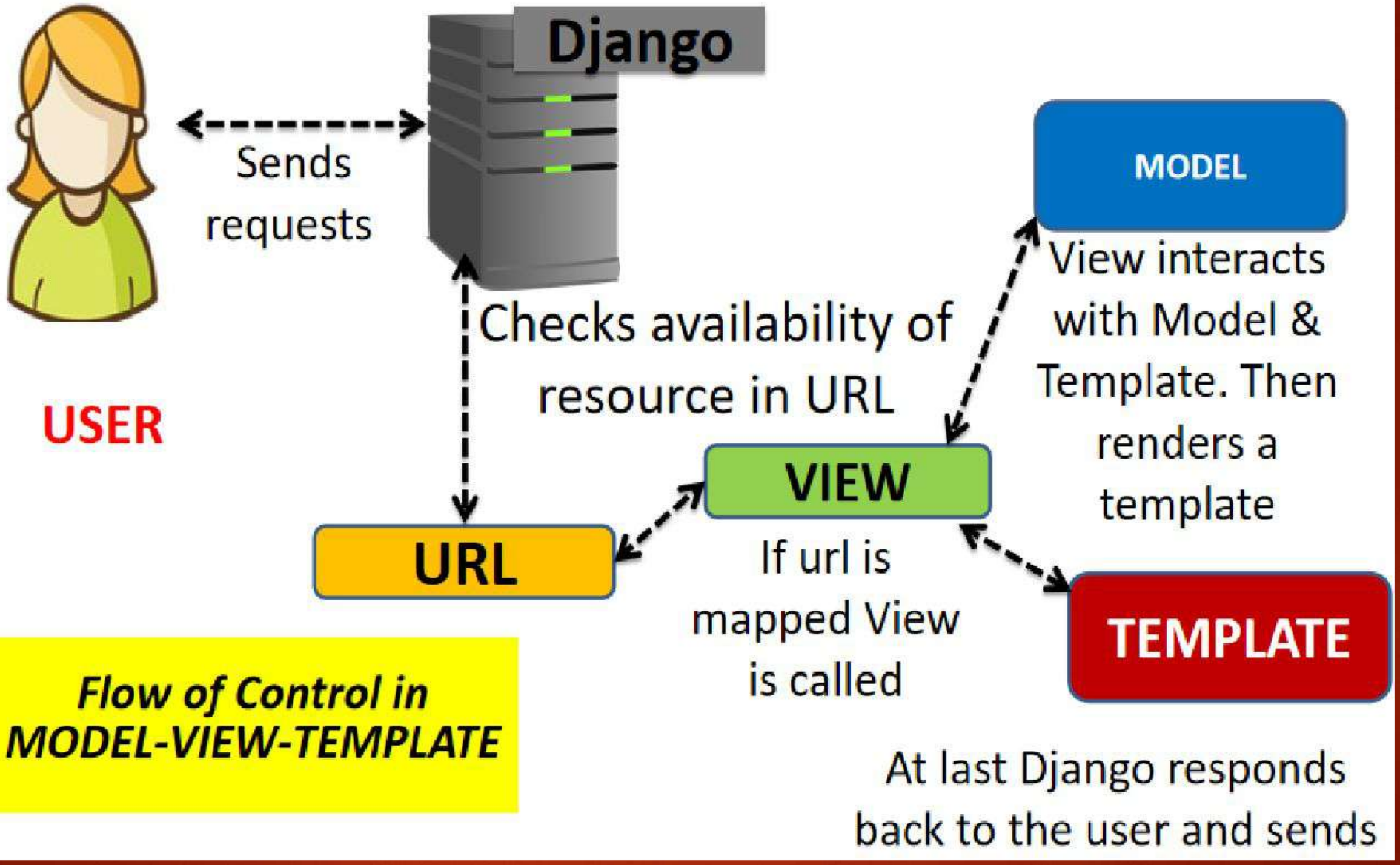
Templates are often .html files, with HTML code describing the layout of a web page

URLs

Django also provides a way to navigate around the different pages in a website.

When a user requests a URL, Django decides which *view* it will send it to.

This is done in a file called urls.py.


Django
Sends requests
USER
Checks availability of resource in URL
MODEL
View interacts with Model & Template. Then renders a template
VIEW
URL
If url is mapped View is called
TEMPLATE
Flow of Control in MODEL-VIEW-TEMPLATE
At last Django responds back to the user and sends

```python
# models.py (the database tables)

from django.db import models

class Book(models.Model):
    name = models.CharField(maxlength=50)
    pub_date = models.DateField()
```

```python
# urls.py (the URL configuration)

from django.conf.urls.defaults import *
import views

urlpatterns = patterns('',
    (r'latest/$', views.latest_books),
)
```

```python
# views.py (the business logic)

from django.shortcuts import render_to_response
from models import Book

def latest_books(request):
    book_list = Book.objects.order_by('-pub_date')[:10]
    return render_to_response('latest_books.html', {'book_list': book_list})
```

```
# latest_books.html (the template)

<html><head><title>Books</title></head>
<body>
<h1>Books</h1>
<ul>
{% for book in book_list %}
<li>{{ book.name }}</li>
{% endfor %}
</ul>
</body></html>
```

- **The models.py file** contains a description of the database table, as a Python class. This is called a model. Using this class, you can create, retrieve, update, and delete records in your database using simple Python code rather than writing repetitive SQL statements.

- **The views.py file c**ontains the business logic for the page, in the latest_books() function.This function is called a view.

- **The urls.py file** specifies which view is called for a given URL pattern. In this case, the URL /latest/ will be handled by the latest_books() function.

- latest_books.html is an **HTML template** that describes the design of the page.

1. A key advantage of such an approach is that components are **loosely coupled.**
    - That is, each distinct piece of a Django-powered Web application has a single key purpose and can be changed independently without affecting the other pieces.

b. For example, a developer can change the URL for a given part of the application without affecting the underlying implementation.

c. A designer can change a page's HTML without having to touch the Python code that renders it.

d. A database administrator can rename a database table and specify the change in a single place, rather than having to search and replace through a dozen files

Django Evolution

The classic Web developer's path goes something like this:

1. Write a Web application from scratch.

2. Write another Web application from scratch.

3. Realize the application from step 1 shares much in common with the application from step 2.

4. Refactor the code so that application 1 shares code with application 2.

This refactoring technique *focuses on breaking down large classes or methods into smaller, more manageable components*.

5. Repeat steps 2–4 several times.

6. Realize you've invented a framework. This is precisely how Django itself was created!

Developed in 2003

1

Django was initially developed by a team at the Lawrence Journal-World newspaper to manage their online news operations.

2

Open-sourced in 2005

The Django framework was released to the public, allowing the open-source community to contribute and expand its capabilities.

Continuous Updates

3

Django has undergone regular updates, with a focus on improving security, adding new features, and enhancing the framework's flexibility and performance.

Views

Our First View: Dynamic Content

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()

    html = "<html><body>It is now
    %s.</body></html>" % now

    return HttpResponse(html)
```

- **First, we import the class HttpResponse**
  - An **HTTPResponse** instance wraps the **HTTP response** from the server.
- **Then we import the datetime module from Python's standard library**
  - **The datetime module contains several functions and classes for dealing with dates and times, including a function that returns the current time.**
- **Next, we define a function called current_datetime. This is the view function.**
  - **function takes an HttpRequest object as its first parameter**

Views

Our First View: Dynamic Content

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()

    html = "<html><body>It is now
    %s.</body></html>" % now

    return HttpResponse(html)
```

- The first line of code within the function calculates the current date/time as a datetime. datetime object, and stores that as the local variable now.

- The second line of code within the function constructs an HTML response using Python's format-string capability.
  - "Replace the %s with the value of the variable now."

• Finally, the view returns an HttpResponse object that contains the generated response.

Mapping URLs to Views

- In Django, URLs are defined in the urls.py file.
- This file contains a list of URL patterns that map to views.
-
- A URL pattern is defined as a regular expression that matches a URL.
-
- When a user requests a URL, Django goes through the list of URL patterns defined in the urls.py file and finds the first pattern that matches the URL.
  - If no pattern matches, Django returns a 404 error.

Here is an example of a simple urls.py file:

```
from django.urls import path
from . import views


urlpatterns = [
path('', views.index, name='index'),

path('about/', views.about, name='about'),

path('contact/', views.contact,
name='contact'),

]
```

- In this example, we have three URL patterns. The first pattern ('') matches the home page, and maps to the index view.
-
- The second pattern ('about/') matches the about page, and maps to the about view.
-
- The third pattern ('contact/') matches the contact page, and maps to the contact view.
-

```
from django.conf.urls.defaults import *

from mysite.views import current_datetime

urlpatterns = patterns('',
(r'^time/$', current_datetime),
)
```

In the case of (r'^time/$', current_datetime), this is a regular expression-based URL pattern. It's matching the URL time/ exactly and directing it to the current_datetime view function.

- (r'^time/$'): This part defines the URL pattern.
  - ^: Matches the start of the string.
  - 'time': Matches the literal string 'time'.

  $: Matches the end of the string.
  - So, this pattern matches only when the URL is exactly time/.
- current_datetime: This is the view function to which the URL pattern is mapped. When a request matches the URL pattern, Django calls this view function to handle the request.

old:

```
urlpatterns = [
    re_path(r'^time/$', current_datetime),
]
```

new:

```
urlpatterns = [
    path('time/', current_datetime),
]
```

allow arbitrary regexes for powerful URL-matching capability, you'll probably use only a few regex patterns in practice. Here's a small selection of common patterns:

Symbol	Matches
`. (dot)`	Any character
`\d`	Any digit
`[A-Z]`	Any character, A–Z (uppercase)
`[a-z]`	Any character, a–z (lowercase)
`[A-Za-z]`	Any character, a–z (case insensitive)
`+`	One or more of the previous character (e.g., `\d+` matches one or more digit)
`[^/]+`	All characters until a forward slash (excluding the slash itself)
`?`	Zero or more of the previous character (e.g., `\d*` matches zero or more digits)
`{1,3}`	Between one and three (inclusive) of the previous expression

How Django Processes a Request

when you run the Django development server and make requests to Web pages:

1. **The command python manage.py runserver imports a file called settings.py from the same directory.**

   This file contains all sorts of optional configuration for this particular Django instance, but one of the most important settings is ROOT_URLCONF. The ROOT_URLCONF setting tells Django which Python module should be used as the URLconf for this Web site.

```
ROOT_URLCONF = 'my_tennis_club.urls'


TEMPLATES = [
    {
```

2. When a request comes in—say, a request to the URL http://127.0.0.1:8000/faq/ http://127.0.0.1:8000/contact/

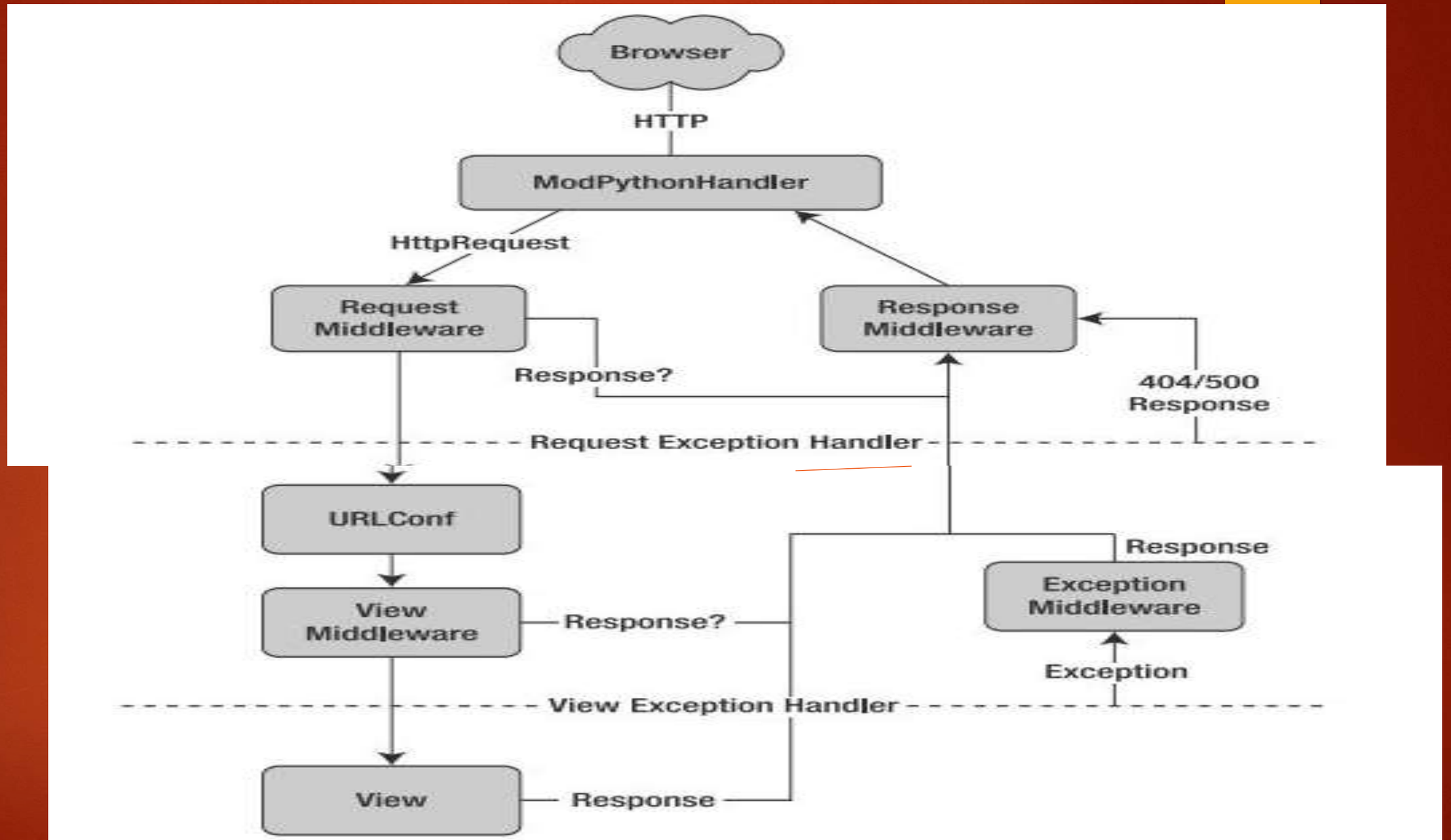
—Django loads the URLconf

Then it checks each of the URLpatterns in that
URLconf in order, comparing the requested URL with the patterns one at a time,

When it finds one that matches, it calls the view function asso-
ciated with that pattern

3. The view function is responsible for returning an HttpResponse object.

How Django Processes a Request: Complete Details



Understanding the flow of a Django request and response involves a series of steps that happen behind the scenes when a user interacts with a Django web application.

1. **Request Initiation**:
   - The process begins when a user makes a request to a Django application by entering a URL in the browser or clicking a link. This sends an HTTP request to the web server hosting the Django application.

2. **Web Server**:
   - The web server (e.g., Nginx, Apache) receives the HTTP request and forwards it to the Django application via a WSGI (Web Server Gateway Interface) server such as Gunicorn or uWSGI.

3. **WSGI Server**:
   - The WSGI server acts as a bridge between the web server and the Django application. It receives the request from the web server and passes it to the Django application.

4. **URL Routing**:
   - Django uses the `urls.py` file to determine which view should handle the request. The URL dispatcher maps the URL pattern to the appropriate view function or class-based view.

5. **View Processing**:
   - The matched view function or class-based view processes the request. The view may interact with the database, render templates, or perform other logic to generate a response.
   - If the view needs to access the database, it uses Django's ORM (Object-Relational Mapping) to query the database models.

6. **Template Rendering**:
   - If the view returns an HTML response, it typically uses a template. The view will pass data to the template, which renders it into a complete HTML page.
   - Templates are stored in the `templates` directory and use the Django template language to dynamically generate HTML.

7. **Middleware**:
   - Before the response is sent back to the client, it passes through a series of middleware components.
Middleware can perform various functions, such as modifying the request or response, handling sessions, or performing authentication checks.
   - Middleware components are defined in the `MIDDLEWARE` setting in the `settings.py` file and are executed in the order they are listed.

8. **Response Object**:
   - The view function or class-based view returns a response object. This can be an `HttpResponse` (for HTML responses), `JsonResponse` (for JSON responses), or other types of responses.
   - The response object contains the content to be sent to the client, along with HTTP headers and status codes.

9. **WSGI Server Response**:
   - The response object is sent back to the WSGI server, which forwards it to the web server.

   - The web server receives the response from the WSGI server and sends it back to the client's browser.

10 **Client Browser**:
   - The client's browser receives the response and renders the content for the user to see. This can be an HTML page, a JSON response, or other types of content.

Here is a simplified flow diagram:

1. **Client** makes an HTTP request →
2. **Web Server** (e.g., Nginx) →
3. **WSGI Server** (e.g., Gunicorn) →
4. **Django URL Dispatcher** (urls.py) →
5. **Django View** (view function or class) →
   - **Database Access** (via ORM, if needed) →
   - **Template Rendering** (if HTML response) →
6. **Django Middleware** →
7. **Response Object** (HttpResponse, JsonResponse, etc.) →
8. **WSGI Server** →
9. **Web Server** →
10. **Client** (Browser renders response)

URLconfs and Loose Coupling

In Django, URLconfs (URL configurations) are used to map URL patterns to views. This mapping is typically done in the `urls.py` file of your Django app.

URLconfs provide a way to loosely couple URLs to views, which means that the URL structure of your application can be changed without affecting the underlying view logic.

This separation allows you to change the URL structure of your application without having to modify the views themselves, promoting better code maintainability and reusability.

Working of Django URL Confs and Loose Coupling,

central mechanism for routing incoming HTTP requests to the appropriate view functions or class-based views. Here's how they work:

1. URL Patterns: URLconfs consist of a collection of URL patterns, each of which associates a URL pattern (expressed as a regular expression or a simple string) with a view function or class.
2. Regular Expression Matchers:
3. Modular Structure: URLconfs can be organized hierarchically, allowing you to include other URLconfs using the include() function.
4. Named URL Patterns: URL patterns can be given names, making it easier to reference them in templates or view functions
5. Namespacing: URLconfs support namespacing, which allows you to differentiate between URLs with the same name in different parts of your project. This is particularly useful in large projects with multiple apps.

```
from django.urls import path
from . import views


urlpatterns = [
    path('home/', views.home_view, name='home'),
    path('about/', views.about_view, name='about'),
    # Other URL patterns
]
```

```
from django.urls import path, include


urlpatterns = [
 path('myapp/', include('myapp.urls')),
 # Other URL patterns
]
```

In this example, the `include('myapp.urls')` function includes the URLconf defined in `myapp.urls` into the main URLconf.

404 Errors

This can happen for a few reasons:

1. URL configuration: Check your urls.py file to ensure that the URL pattern you're trying to access is correctly defined.
2. View function: If the URL pattern is correct, ensure that the corresponding view function exists and is correctly imported.
3. Template existence: If your view is supposed to render a template, make sure the template exists in the correct location.
4. Database query: If your view relies on database queries, ensure that the data it's trying to access exists in the database.
5. Static files: If the 404 error is related to static files (like CSS, JavaScript, or images), ensure that the static files are located in the correct directory and are properly linked in your templates.
6. Permissions: Check if the user has permission to access the URL. Django's PermissionDenied exception can also result in a 404 error.

Your Second View: Dynamic URLs

a separate view function for each hour offset, which might result in a URLconf like this:

```
urlpatterns = patterns('',
(r'^time/$', current_datetime),
(r'^time/plus/1/$', one_hour_ahead),
(r'^time/plus/2/$', two_hours_ahead),
(r'^time/plus/3/$', three_hours_ahead),
(r'^time/plus/4//$', four_hours_ahead),
```

Wildcard URLpatterns

```
from django.conf.urls.defaults import *
from mysite.views import current_datetime, hours_ahead
urlpatterns = patterns('',
(r'^time/$', current_datetime),
(r'^time/plus/\d+/$', hours_ahead),
)
```

This URLpattern will match any URL such as /time/plus/2/, /time/plus/25/, or even /time/plus/1000000000000/.

```
(r'^time/plus/\d{1,2}/$', hours_ahead),
```

views.py:

```
def current_datetime(request):
now = datetime.datetime.now()
html = "<html><body>It is now %s.</body></html>" % now
return HttpResponse(html)
def hours_ahead(request, offset):
offset = int(offset)
dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
return HttpResponse(html)
```

Errors in Django, Wild Card patterns in URLS.