ResumenesTeoriaDAI.pdf

martasw99

Apuntes Variados

4º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

MARTA SOTO WERNER

ÍNDICE

TEMA 1: CODIFICACIÓN DE CARACTERES UNICODE

En Europa Occidental usamos 4 codificaciones distintas para los signos del alfabeto latino:

- **ASCII**
- **ISO-8859-1** "Latin1" → (ANSI en Windows)
- **UTF-8**
- **UTF-16** (**UNICODE** en Windows)

1. ASCII (American Standard Code for Information Interchange)

Es un código de **7 bits** creado en 1963, solo para los signos que se usan en **inglés**.

NOTA: El **fin de línea** depende del sistema operativo:

- `\n` **Unix**
- `\r\n` **Windows**
- `\r` **MAC**

2. CÓDIGOS ISO-8859

Son **16 códigos distintos de 8 bits**, ampliaciones del ASCII, que permiten usar signos alfabéticos usados en otras lenguas distintas del inglés.

LATIN-1: En Europa Occidental usamos el conjunto de carácteres Latin-1 en alguna de sus variantes:

→ **ISO-8859-1**

→ **ISO-8859-15** (Revisión para añadir el signo del euro €)

→ **Windows-1252 (ANSI)**

3. UNICODE

Es una **codificación universal** con un repertorio de más de 120000 signos de cualquier alfabeto, cada uno tiene asignado un `code point.` Los intérpretes de Unicode manejan automáticamente la dirección de escritura y la composición de signos.

FORMATOS UNICODE: Unicode se puede codificar de 3 maneras

→ **UTF-8** Es el más generalizado, sobretodo en Internet y en UNIX

→ **UTF-16** Se usa en Windows

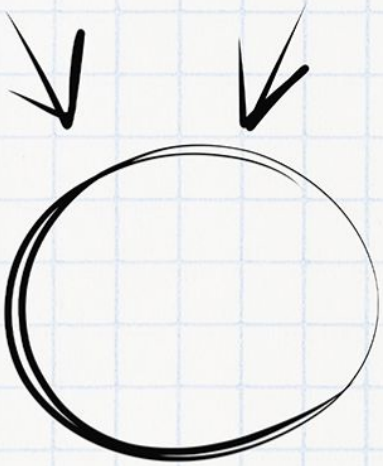
→ **UTF-32**

Imagínate aprobando el examen

Necesitas tiempo y concentración

Planes	PLAN TURBO	PLAN PRO	PLAN PRO+
Descargas sin publi al mes	10	40	80
Elimina el video entre descargas	✓	✓	✓
Descarga carpetas	✗	✓	✓
Descarga archivos grandes	✗	✓	✓
Visualiza apuntes online sin publi	✗	✓	✓
Elimina toda la publi web	✗	✗	✓
Precios Anual	0,99 € / mes	3,99 € / mes	7,99 € / mes

Ahora que puedes conseguirlo, ¿Qué nota vas a sacar?



WUOLAH

Apuntes Variados

Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

3.1. UTF-8

Es una **codificación variable de 8 bits** compatible con ASCII (1 byte, 2, 3 o 4). Es la **codificación en protocolos de Internet** y se presupone a menos que se indique otra cosa.

```
# En HTML5, después de la declaración
<meta charset="UTF-8">
# En HTML4, y HTML5 en el <head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
# En XML, utf-8 por defecto, si no:
<?xml version="1.0" encoding="ISO-8859-1">
```

4. SOPORTE EN C++

En Unicode **no todos los carácteres tienen 1 byte** → En estos casos en C++ usaremos el **tipo** `wchar_t` en lugar de `char` y las **funciones** `wcs`

5. UNICODE EN PYTHON

Desde Python3 el tipo **str** contiene carácteres unicode, sin embargo en Python2 existe el tipo unicode distinto del str.

CODIFICACIÓN-DESCODIFICACIÓN Python3

En Python3 existen los tipos `str` **(unicode) y** `binary`**.** Para convertir pasar desde /hasta las distintas codificaciones están las funciones `byte.decode()` y `str.encode()`

```
a = 'cañón'
print (a.encode('utf8'))
b'ca\xc3\xb1\xc3\xb3n'

print (a.encode('latin1'))
b'ca\xf1\xf3n'
```

ENTRADA / SALIDA en Python3

Se trabaja en Unicode internamente y **sólamente se debe convertir** a o desde una codificación concreta **la entrada / salida.**

`write` y `open` codifican y descodifican automáticamente (UTF-8 por defecto)

```
f = open ('texto.txt', 'w', encoding='utf-8')
f.write(a)
f.close()

with open('texto.txt', encoding='utf-8') as f:
   print (f.readline())
```

MARTA SOTO WERNER

TEMA 2: HTTP

1. CLIENTE - SERVIDOR

Las aplicaciones web están **distribuidas** en dos partes:

CLIENTE:

- **Proactivo**
- Inicia la comunicación
- Hace requerimientos a **un solo servidor**
- Se encarga de la interfaz con el usuario

SERVIDOR:

- **Reactivo**
- Espera requerimientos para contestarlos
- Responde a **varios clientes**
- Se encarga de la lógica del programa y de la comunicación con la base de datos.

La comunicación entre el cliente y el servidor puede ser:

1. Síncrona: El cliente hace la petición y espera hasta que el servidor le contesta
2. Asíncrona: El cliente hace la petición, sigue trabajando y en algún momento recibirá la respuesta del servidor.

VENTAJAS:

- El cliente se ejecuta en un navegador, por lo que es **independiente de la plataforma** y vale para cualquier SO incluyendo móviles.
- Se **adapta fácilmente** a los cambios en los dispositivos de acceso a la red.
- Es **fácil hacer actualizaciones**
- Se puede usar desde cualquier sitio **remotamente**
- Se puede poner como **Software as a Service** (Con subscripciones)

2. HTTP (Hiper Text Transfer Protocol)

Para comunicar clientes y servidores se utiliza HTTP que es un **protocolo sin estados**, es decir, sin memoria. Esto quiere decir que no necesita que el servidor guarde información o estados sobre cada usuarios durante las múltiples solicitudes.

HTTP es un **protocolo ASCII** con peticiones (**request**) y respuestas (**responses**).

Petición HTTP: Request

Diferenciamos los siguientes métodos:

- **GET** (Leer): Solicita una representación de un recurso específico. Se usa para recuperar datos. Los parámetros de consulta van en la URL URLificados
- **PUT** (Actualizar): Reemplaza todas las prestaciones actuales del recurso de destino con la carga útil de la petición.
- **POST** (Crear): Para enviar una entidad a un recurso específico, causando cambios en el servidor. Las variables del formulario van como contenido en el requerimiento después de todas las cabeceras.
- **DELETE** (Borrar): Borra un recurso específico

HTTP protocolo sin estados

Como hemos comentado antes, HTTP es un protocolo sin estados, pero hay veces que algunas webs necesitan rastrear el progreso del usuario de una página a otra, por lo que necesita **almacenar algún tipo de información para el usuario.** Como solución a este problema surge el uso de las Cookies, sesiones del servidor, variables ocultas (cuando la página contiene un formulario), reescritura de URL utilizando parámetros codificados...

> **COOKIES:** Se **almacenan en el navegador** como un formato de archivo de texto. Se almacena una cantidad límite de datos (sólo se permite 4kb). La variable `$_COOKIE` no contendrá varias cookies con el mismo nombre.
>
> Podemos acceder fácilmente a los valores de las cookies, por lo que es **menos seguro**. La función `setcookie()` debe aparecer ANTES de la etiqueta `<html>`.
>
> **SESIONES:** Se **almacenan en el lado del servidor.** Se almacena una cantidad limitada de datos. En una sesión, se mantiene información de la conexión, asociando cookies con un token aleatorio, con un registro en el servidor, por lo tanto no podemos acceder fácilmente y es **más seguro.**

TEMA 3: Micro Frameworks Web-Flask

1. CARACTERÍSTICAS DE FLASK

Flask es un software para apoyar el desarrollo de aplicaciones web, concretamente un **Micro Framework.** Sus principales características son:

- Servidor de desarrollo y depurador incorporado.
- Soporte integrado para pruebas unitarias.
- Plantillas Jinja2
- Soporte para Cookies seguras (sesiones en el lado del cliente)
- Compatible con Unicode
- Profusamente documentado

Manejo de URLs con Flask

Utilizamos el decorador **`route()`**

```
@app.route('/')
def index():
    return 'Página principal'
```

```
@app.route('/hola')
def hola():
    return 'Hola'
```

Para usar **variables** en la url lo hacemos de la siguiente forma:

```
@app.route('/user/<username>') # Captura una parte del URL
def mostrar_nombre_usuario(username): # y la pasa como parámetro
    return f" Usuario : {username}"
```

La **url** para entrar en la página web con mi usuario sería: `/user/marta`

Entrada de variables GET [parámetros]

Es muy importante importar la clase request: **`$from flask import request`**

```
@app.route('/prog')
def prog():   # los parámetros están en el request
    parametro1 = request.args.get('par1')   # hola
    parametro2 = request.args.get('par2')   # pepe
    return parametro1 + ' ' + parametro2
```

La **url** sería: `/prog?par1=hola&par2=pepe`

Cabeceras HTTP en Flask

Las añadiremos utilizando `response.headers.add()`

```
@app.route('/png')
def png():
    response = Response()
    response.headers.add('Content-Type', 'image/png')
    response.set_data(imagen_png)   # Datos binarios
    return response
```

Métodos Http (GET, POST, PUT, ...)

```
@app.route('/login', methods=['GET', 'POST'])

def login():
    if request.method == 'POST':
        hacer_login()
    else:
        enviar_formulario_para_login()
```

Redirecciones y errores

Debemos importar de flask las siguientes clases:

```
$from flask import abort, redirect, url_for
```

```
@app.route('/')
def index():
    return redirect(url_for('login'))

@app.route('/login')
def login():
    abort(401)
    estoNoSeEjecuta()

@app.errorhandler(404)
def page_not_found(error):
    return "Página no encontrada", 404
```

Cookies en Flask

Las cookies pertenecen a la clase **`request`** de Flask y debemos diferenciar entre:

1. **Leer Cookie:**

```
@app.route('/')
def index():            # las cookies están en el request
    username = request.cookies.get('username')

      # use cookies.get(key) instead of cookies[key] to not get a
      # KeyError if the cookie is missing.
```

2. **Escribir Cookie:**

```
@app.route('/')
def index():
    response = make_response("logeado como the_username")
    response.set_cookie('username', 'the_username')
    return response
```

Sesiones en Flask

Se manda un cookie de sesión con un token "aleatorio" y se asocia con un registro persistente en el servidor. Debemos añadir las siguientes clases:

```
$ from flask import Flask, session, redirect, url_for, escape, request
```

Como hemos dicho antes, lo primero que debemos hacer es definir el **"token aleatorio"** :

```
app = Flask(__name__)

# Set the secret key to some random bytes.
app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'
```

A continuación ya podemos declarar y usar las sesiones para almacenar toda la información que necesitemos:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        session['username'] = request.form['username']
        return redirect(url_for('index'))
    return '''
        <form method="post">
            <p><input type=text name=username>
           <p><input type=submit value=Login>
        </form>
  '''
```

```
@app.route('/')
def index():
if 'username' in session:
  return f'Logged in as {escape(session['username']}
return 'You are not logged in'
```

```
@app.route('/logout')
def logout():
    # remove the username from the session
    session.pop('username', None)
    return redirect(url_for('index'))
```

Logs en Flask

Podemos tener un registro de eventos usando el módulo **logging** de python:

```
@app.route('/')
def principal():
  app.logger.debug('Arranque de la aplicacion')
  return 'Ejemplo para logs'
```

MARTA SOTO WERNER

TEMA 4: Modelo Vista Controlador (MVC)

Consiste en separar en archivos y carpetas distintas (módulos, packages) el código relativo a:

- Base de Datos → **Model**
- Interfaz del usuario→ **View**
- Comunicaciones y lógica → **Controller**

1. CONCEPTO

Modelo
- → Se encarga de la **interacción con la BD** (`select`, `insert`, `update`, `delete`)
- → Lógica relacionada con los datos (restricciones, campos derivados ...)
- → Se **comunica con el Controlador**

Vista
- → **Representación visual de los datos** (Interfaz gráfica)
- → **Interacción** con el **usuario**
- → Usualmente HTML, CSS y JavaScript
- → Se **comunica con el Controlador** a través de plantillas (templates)

Controller
- → **Recibe órdenes del usuario y se encarga de solicitar los datos al modelo y de comunicárselos a la vista.**
- → Recibe la entrada de las vistas o urls (routing)
- → Procesa los request (`GET`, `POST`, `PUT`, `DELETE`)
- → Rellena y envía las plantillas

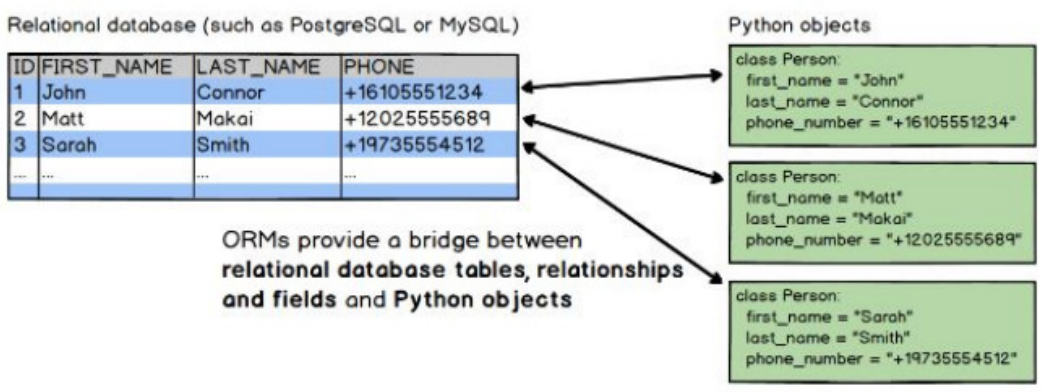
En **python** se organizan las carpetas de la siguiente manera:

```
mi_app/
  ├── run.py
  ├── mi_app/              # código
  │    ├── model.py
  │    └── controler.py
  ├── templates/           # plantillas html (vistas)
  │    ├── base.html
  │    ├── entrada.html
  │    └── ...
  └── static/              # archivos fijos (assets)
       ├──css/
       ├──imagenes/
       └──js/
```

2. ORMs

El **Mapeo de Objeto-Relacional**, es un modelo de programación que consiste en la transformación de las tablas de una base de datos en una serie de entidades que **simplifique las tareas básicas de acceso a los datos para el programador.**

Django ORM: Muy completo y permite evitar tener que escribir SQL a mano, añadiendo la posibilidad de cambiar de motor de base de datos sin que sea traumático.



- **model.py con ORM:**

```python
from django.db import models

class Musician(models.Model):
    first_name = models.CharField(max_length=50, unique=True)
    last_name = models.CharField(max_length=50, required=False)
    instrument = models.CharField(max_length=100)

class Album(models.Model):
    artist = models.ForeignKey(Musician, on_delete=models.CASCADE)
    name = models.CharField(max_length=100)
    release_date = models.DateField(default=datetime.date.today)
    num_stars = models.IntegerField()
```

- **controller.py con ORM:**

```
# Consultas

todos       = Musician.objects.all()

uno         = Musician.objects.get(first_name="Pepe")

pianistas   = Musician.objects.filter(instrument='piano').order_by('first_name'

garcia      = Musician.objects.fiter(last_name__contains='Garcia')

albumes_de_pepe = Album.objects.filter(artist=uno)
```

- **view.html con plantillas y html:**

```
<p> My string: {{my_string}} </p>
<p> Value from the list: {{Users[2]}} </p>
<p> Loop through the list: </p>
<ul>
  {% for user in Users %}
      <li> {{user.name}} </li>
  {% endfor %}
</ul>
```

MARTA SOTO WERNER

TEMA 5: PLANTILLAS (Templates)

Podemos tener la potencia del sistema de procesamiento de los templates de Django corriendo en aplicaciones escritas en Python, por ejemplo Flask, bottle...

1. FLASK: Templates con Jinja

Flask usa **Jinja** como motor de plantillas, destacamos:

- HTML escaping
- Filtros
- Herencia de plantillas
- Compilado a python nativo eficiente
- Sintaxis configurable (para acomodarse mejor a otros formatos de salida como LaTex o JavaScript)

holaMundo.py:

```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def hello_world():
    return render_template('hola.html')
```

hola.html:

```html
<html>
  <head>
    <title>¡Hola Mundo! con Plantilla</title>
  </head>
  <body>
    <p>¡Hola Mundo! (con plantilla)</p>
  </body>
</html>
```

1.1 Paso de variables

archivo.py →

```python
return render_template('hola.html', usuario=user)
```

archivo.html →

```
{% if usuario %}
    <h1>Bienvenido {{ usuario }}!</h1>
{% else %}
    <h1>Bienvenido desconocido!</h1>
{% endif %}
```

1.2 Filtros

Efectúan modificaciones en el html:

```
{{ 42.55|round }}
    -> 43.0
{{ 42.55|round(1, 'floor') }}
    -> 42.5
{{ 42.55|round|int }}
        -> 43

{{ 43221|filesizeformat }}
-> 43 KB
```

1.3 Acceso a diccionarios

```
def hello_world():
    tabla = [{'a': 1, 'b':2}, {'a': 7, 'b':8}]
    return render_template('hola.html', rows=tabla)


<ul>
{% for row in rows %}
    <li>{{ row.a }}, {{ row.b }} </li>
{% endfor %}
</ul>
```

2. SINTAXIS MOUSTACHE

- **{% ... %}** Sentencias
- **{{ ... }}** Expresiones que serán impresas en la salida de la plantilla
- **{# ... #}** Comentarios (No aparecerán en la salida)
- **# ...** Sentencias de una línea

3.HTML SEGURO

Por defecto **Jinja2** "escapea" el html que le enviamos <,>,'. Se puede desactivar:

- Usando la clase **Markup** en el código de python → Es lo más recomendado
- Usar el filtro "**Isafe**" dentro de la plantilla → `{{variable|safe}}`
- **Desactivar** el sistema de "**escaping**" temporalmente → `{% autoescape false %}`

4.HERENCIA DE PLANTILLAS

Permite crear una base que luego será personalizada en otras plantillas:

base.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %} Mi sitio  web {% endblock %}
</head>
<body>
    <div id="content">
            {% block content %}
            {% endblock %}
        </div>
```

hijo.html:

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

{% block title %} Otra página {% endblock %}

{% block content %}
    <h1>Título de la página</h1>
    <p class="important">
      ¡Bienvenido piltrafilla!
    </p>
{% endblock %}
```

MARTA SOTO WERNER

TEMA 6: HTML5, CSS y Frameworks

1. HTML5

Es la versión actual de Html (2014), sus principales **novedades** son:

- **Nuevo parsing**, se dispara con → `<!DOCTYPE html>`
- **Elementos multimedia** → `<audio>, <video> y <canvas>`
- **Elementos semánticos** para sustituir a y → `<section> , <article>, <header> , <footer>, <nav>` …
- **Nuevos elementos para forms**: `<datalist> , tipos de entrada para campos: text, date, url, email, tel, number, color, etc` con validación del navegador.
- **Inclusión de SVG y MATHML**
- **Soporte de almacenamiento local:** `Web storage`

2. CSS FRAMEWORKS

A la hora del diseño de una página web **ayudarán a las plantillas** para:

- **Elementos complejos** que podamos poner (menús, enlaces, forms...)
- **Situar** los elementos en la página, que queden legibles e iguales en todos los navegadores.
- **Que cambie** adecuadamente con el tamaño del dispositivo: móvil, tablet o PC.

3. BOOTSTRAP ¿QUÉ ES Y CÓMO FUNCIONA?

Es un **framework CSS y JavaScript** diseñado para la creación de interfaces limpias con un diseño responsive, es decir, es una colección de plantillas de css y js originalmente usada en Twitter. Las **funciones** más usadas son:

- **Responsive design** (adaptación automática a móviles y tablets)
- **Grid System** (Posicionar los elementos en la página)
- **Interface UI** (forms, botones, menús, etc.)

Bootstrap en líneas generales, se basa en una estructura dividida de 12 columnas que los desarrolladores pueden gestionar en función de sus necesidades y preferencias, en función de cuatro tamaños de dispositivos.

4. ALTERNATIVA: MaterializeCSS

Es un **framework CSS** que permite crear sitios y apps web con los principios de Material Design (creado por Google). Puede ser usado en dos formas, **Materialize** y **Sass**, dependiendo de las preferencias y la experiencia se pueden seleccionar cualquiera de las dos versiones.

Incluye:

- **Grids →** 3 tamaños de plantilla
- **Blockquotes →** (Posicionar los elementos en la página)
- flow text (responsive text), botones, checkboxes, barras de navegación, estilos para tablas y cards.

TEMA 7: XML A FONDO

1. XML (eXtensible Markup Language)

Es un **metalenguaje** de propósito general para **el intercambio y almacenamiento de información**. Es un subconjunto simplificado del SGML, diseñado para que sea legible también por las personas.

2. ¿PARA QUÉ SIRVE XML?

Principalmente sirve para **representar información estructurada en la web** (todos documentos) de modo que esta información pueda ser almacenada, transmitida, procesada, visualizada e impresa por diversos tipos de aplicaciones y dispositivos.

Hay **dos niveles de restricción** en los archivos XML:

- **Válidos →** Cumplen con la sintaxis del XML
- **Bien formados →** Que además tengan una especificación propia para sus elementos (que se puedan validar con un *DTD o un XML Schema*)

3. XML BIEN FORMADOS

En un archivo XML bien formado nos podemos encontrar:

- **Prólogo →** Etiqueta inicial y quizás con una declaración de un DTD o esquema

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE xxx  PUBLIC "http://...">
```

- **Elementos →** Delimitados por etiquetas de principio y fin. Se estructuran en árbol (siempre hay un elemento raíz) Los **elementos vacíos** van con marca de autocierre `<x/>`

```
<raiz>
   <elemento_1> texto </elemento_1>
   <elemento_2> texto </elemento_2>
   <elemento-vacio/>
</raiz>
```

- **Atributos →** Van en la etiqueta de apertura y es obligatorio tener el valor entre comillas

```
<root>
   <elemento_1 atributo="valor"> texto </elemento_1>
   <elemento_2> texto </elemento_2>
   <elemento-vacio/>
</root>
```

- **Referencias a caracteres →** Se puede poner cualquier signo en la codificación que se use, normalmente UTF-8 o con su código numérico UNICODE o con referencias a **entidades predefinidas**

```
&#NNNN;   Notación decimal

&#xNNNN;  Notación hexadecimal

&referencia_a_entidad_caracter;
```

- **Carácteres reservados**

`&` → `&amp` `<` → `&lt` `>` → `&gt`

`"` → `&quot` `'` → `&apos`

- **Comentarios** `<!-- Comentario -->`

- **Secciones CDATA (Character data) →** Sirven para incluir texto que el parser no va a procesar

```
<![CDATA[
 ...
]]>
```

- **Instrucciones de procesamiento →** Se utilizan para poner indicaciones para las aplicaciones que procesan archivos XML.

  La **sintaxis** es : `<? target instruction ?>`

  Ejemplo: Se pueden aplicar hojas de estilo a documentos xml

```
<?xml-stylesheet href="es.css" type="text/css">
```

4. DTDs Y XML SCHEMMAS

Un **DTD** (Document Type Definition) o un **XML Schema** son **declaraciones de la sintaxis** para un lenguaje XML particular, con las definiciones de los elementos y los atributos.

Para que un documento XML sea **válido**, tiene que tener una referencia a un DTD o a un Schemma.

5. PROCESADO DE ARCHIVOS XML

Hay 3 maneras (como mínimo) de procesar archivos XML (o HTML):

- Usando **expresiones regulares**
- Con parsers **SAX**
- Con parsers **DOM**

Las **librerías para SAX y DOM** son estándar y existen con las mismas funciones para C, C++, PHP, Java, Perl, etc.

BIBLIOTECA DE EVENTOS SAX (Simple Api para Xml)

SAX es una librería **basada en eventos:** Se va leyendo el archivo y se generan llamadas a funciones conforme se leen las etiquetas.

MANEJO DINÁMICO DEL ÁRBOL DOM (Document Object Model)

DOM es una API para manejar la estructura de árbol de un documento XML: Se lee todo el archivo en memoria y se accede a cada nodo del árbol para leerlo, modificarlo, ampliarlo...

TEMA 8: PERSISTENCIA - BASES DE DATOS

Veamos una a una todas las opciones con las que contamos para la persistencia de datos.

1. SERIALIZACIÓN DE DATOS

- Con el **módulo pickle** guardamos un objeto en el disco

```
import pickle
D = {'a':1, 'b':2}  # o cualquier estructura de datos

with open('d.dat', 'wb') as f:
        pickle.dump(D, f) # serializa (convierte a un flujo de byt

with open('d.dat', 'rb') as f
        D = pickle.load(f) # deserializa

print(D['a'])
```

- **Archivos indexados tipo dbm**: Con dbm se accede a bases de datos tipo `Berkely` y DB como si estuvieran en memoria

```
import dbm
db = dbm.open('datos.dat', 'c')

# Insert (en disco)
db['237483J'] = 'Pepito Pérez|c/ Pedro Antonio|10|7º B'

# Select (desde el disco)
datos = db['237483J'].split('|')  # una lista

db.close()
```

dbm — Interfaces to Unix "database

2. PERSISTENCIA DE OBJETOS

Shelve añade una capa de software a dbm para serializar los objetos

```
import shelve

db = shelve.open('datos.dat')

# Insert (en disco)
db['237483J'] = {'nombre': 'Pepito Pérez',
                 'dir':'c/ Pedro Antonio, 10, 7º B'}

# Select (desde el disco)
datos = db['237483J']

db.close() # flush
```

shelve - Python object

Las aplicaciones web se suelen ejecutar en paralelo en varias hebras, **pickleshare** añade concurrencia a **shelve**:

```
from pickleshare import PickleShareDB
db = PickleShareDB('~/datos.dat')

# Insert (en disco)
db['237483J'] = {'nombre': 'Pepito Pérez',
                 'dir':'c/ Pedro Antonio, 10, 7º B'}

# Select (desde el disco)
datos = db['237483J']
```

3. BASES DE DATOS

A diferencia de los archivos indexados, las Bases de Datos:

- Están en un proceso **servidor** separado (excepto SQLite)
- Permiten **búsquedas por varios índices**
- Tienen un manejo estandarizado

Usar bases de datos con una **librería cliente:**

```
# Conectar con el servidor
def create_connection(host_name, user_name, user_password):
  connection = None
  try:
    connection = mysql.connector.connect(
        host=host_name, user=user_name, passwd=user_password)
    print("Connection to MySQL DB successful")
  except Error as e:
    print(f"The error '{e}' occurred")

  return connection
```

Dos tipos de librerías:

```
# Con clientes de BD (distintos para cada DB)
result = connection.execute("select username from users")
for row in result:
    print("username: %s" % row['username'])


# Con ORMs (código igual para cualquier BD)
result = connection.query(users).all()
for user in result:
   print("username: %s" % user.username)
```

Librerías Object Relational Mapping

Una capa de software para aislar la BD SQL del resto del código.

4. BASES DE DATOS NoSQL

Son bases de datos que surgieron para tratar con **Big Data,** tipos de datos complejos, más índices , distintas maneras de consulta, etc.

5. MONGODB

Es una base de datos, orientada a **documentos estilo JSON,** (BSON) que incluye información del tipo de dato.

En MongoDB una base de datos está formada por **colecciones** (equivalente a las **tablas** en SQL) que son un conjunto de **documentos** (equivalente a las **filas**)

```
from pymongo import MongoClient
client = MongoClient('mongodb://localhost:27017/')

# base de datos
db = client['test-database']
```

6. MONGOENGINE

Es un **ODM** que nos permite **mapear la estructura de nuestra base de datos a objetos** en nuestras aplicaciones python que se conectan a instancias como MongoDB. El propósito es poder usar información de nuestra base de datos como si se tratasen de objetos. Ofrece numerosas ventajas, como la manera CRUD de escribir nuestra aplicación y mejorar nuestro código.

TEMA 9: API RESTfull

1. SERVICIOS WEB: REST

Son un conjunto de protocolos y estándares **para comunicar aplicaciones e intercambiar datos en internet.** Se usa *http* y el *puerto 80* para evitar problemas con los firewall.

La interfaz (API) puede ser **tipo**:

- **REST →** Más simple
- **SOAP, WSDL →** De nivel más alto, cada vez menos usados

Este término se usa ahora para referirse a interfaces simples, que solo usan HTTP y JSON o XML en la respuesta. Se basan en una sintaxis universal para acceder a los recursos en internet: las U**niform Resource Identifier (URI)**, donde se código tanto por ciento. **REST** es un **protocolo sin estado.**

REST-FULL

En su versión más extendida, se usan los **verbos de HTTP para las llamadas del cliente al servidor,** asociandolos a operaciones CRUD.

Un **servicio RESTfull debe seguir 4 principios:**

- Usar los **verbos de HTTP** explícitamente
- Ser **completamente sin estado:** Para que el resultado de la llamada no dependa de las anteriores
- Usar **URIs estilo path:** que se puedan leer por humanos (SEO) y que muestren una jerarquía sustituyendo espacios en blanco por guiones o subrayados.
- **Devolver XML o JSON**

Usar los verbos de HTTP explícitamente con llamadas AJAX:

- **GET →** para **requerir** un recurso (Read / SELECT)
- **POST →** para **crear** un recurso (Write / INSERT)
- **PUT →** para **cambiar** el estado de un recurso (Write / UPDATE)
- **DELETE →** para **borrar** un recurso (DELETE)

MARTA SOTO WERNER

TEMA 10: FRAMEWORK DJANGO

1. FRAMEWORK DJANGO

Sus principales características son:

- Es el **framework más usado para python**
- Empezó a desarrollarse como herramienta para sitios de prensa
- Está enfocado a **sitios basados en bases de datos** (Ejemplo: Ebay)
- **Software libre** y primera versión en 2008
- Django contiene gran cantidad de **módulos** (Autentificación, Autorización, Sesiones, Seguridad, Signals, Templates, Formularios, Messages, E-mail ...)
- Tiene una **arquitectura MVT**
  - **Modelo**: Interfaz con la BD mediante ORM
  - **Vista**: Programa equivalente a controler en MVC
  - **Template**

2. FLASK VS DJANGO

FLASK	DJANGO
- Microframework (Librería) 2MB - Se hace menos código para empezar - Respuesta más rápida - Prototipos, sitios sin usuarios - Optimización de rendimiento Pirates use Flask, the Navy uses Django	- Framework (Librería + scripts) 7MB - Más código hecho (usuarios, interface de administración, ORM, etc) - Modelo de usuarios, autenticación, roles, middleware, etc

En cuanto a **código**:

Flask

```
from flask import render_template

@app.route('/hello/<username>')
def hello():
        return render_template('hello.html', name=username)
```

Django

```
from django.shortcuts import render

def hello(request, username):
        context = {
                'name': username,
        }
        return render(request, 'hello.html', context)
```

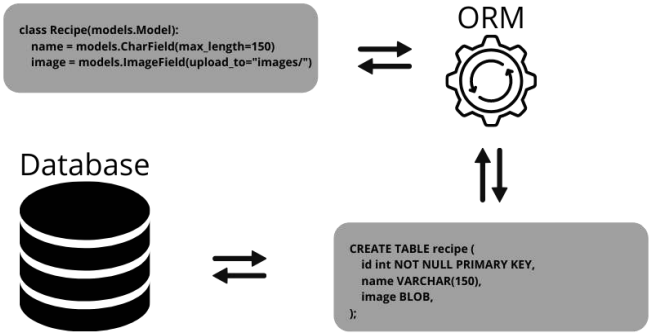
3. PASOS PARA CREAR UNA APLICACIÓN EN DJANGO

1. **Crear** un **proyecto**
2. Crear una **aplicación dentro del proyecto**
3. Poner la **base de datos** (SQL) y demás en **settings.py**
4. **Crear** la **BD**
5. Definir el **modelo** en **app/models.py**
6. Añadir los **módulos** y **middleware** en **settings.py**
7. Crear los **mappings** para los **urls** en **urls.py**
8. Definir las **vistas** en **app/views.py** y los **templates**
9. Aplicar **test**
10. **Desplegar la aplicación** en el servidor web de producción

TEMA 11: MODELOS EN DJANGO

1. MODEL

Django **utiliza un ORM** para Bases de Datos SQL



2. CLASES EN LUGAR DE TABLAS

```python
from django.db import models

class Músico(models.Model):
    nombre = models.CharField(max_length=50)
    género_musical  = models.CharField(max_length=50)
    instrument = models.CharField(max_length=100)

class Album(models.Model):
    músico            = models.ForeignKey(Músico, on_delete=models.CASCADE)
    título            = models.CharField(max_length=100)
    fecha_lanzamiento = models.DateField()
    likes             = models.IntegerField()
```

3. CARACTERÍSTICAS DJANGO MODEL

Una **clase Model** tiene:

- Declaración de campos
- Métodos
- Metadatos

Por defecto, Django asigna **una llave primaria automática** a no ser que se asigne un campo como `'primary_key'` o como `'unique'`

```python
id = models.AutoField(primary_key=True)
```

4. RELACIONES

Muchos a uno

```python
from django.db import models

   class Manufacturer(models.Model):
      # ...

   class Car(models.Model):
      manufacturer = models.ForeignKey(Manufacturer,
                                       on_delete=models.CASCADE)
      # ...
```

Muchos a muchos

```python
from django.db import models

   class Author(models.Model):
      # ...

   class Books(models.Model):
      # ...
      author = models.ManyToManyField(Author)
```

Uno a uno

```python
from django.db import models
from django.contrib.auth.models import User

class Profile(models.Model):

        user = models.OneToOneField(User, on_delete=models.CASCADE)
        bio = models.TextField(max_length=500, blank=True)
```

5. ADMIN.PY

Para que aparezcan en la app de admin de la BD:

```python
# app/admin.py

from django.contrib import admin
from .models import Musician, Album

admin.site.register('Musician')
admin.site.register('Album')
```

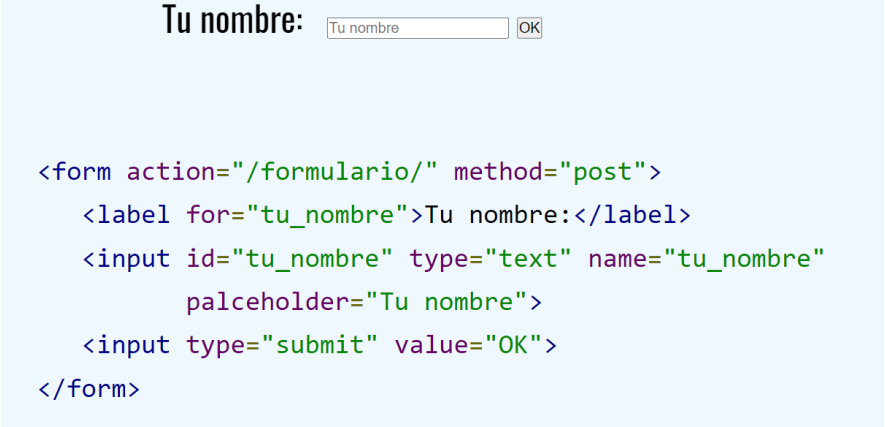
TEMA 12: FORMULARIOS

1. CLASE FORMS

Django incluye la clase forms para facilitar la entrada de datos desde **formularios html.**

La clase **incluye:**

- **Generación de html en las plantillas**
- **Validación** en el servidor
- **Tratamiento** de los **errores**
- **Seguridad CSRF**




```
<form action="/formulario/" method="post">
   <label for="tu_nombre">Tu nombre:</label>
   <input id="tu_nombre" type="text" name="tu_nombre"
          palceholder="Tu nombre">
   <input type="submit" value="OK">
</form>
```

2. SERIALIZADORES

Desde **APIs** se usan serializadores para:

- ➔ **Transformar desde los objetos de la BD a JSON (o XML)**
- ➔ **Desde JSON a la BD**, comprobando validadores y devolviendo error en su casa.

TEMA 13: AUTENTIFICACIÓN Y AUTORIZACIÓN EN DJANGO

Django tiene un **sistema para la autentificación y la autorización de usuarios** basados en un modelo para usuarios que incluye:

- **Autenticación**: Gestión de Contraseñas
- **Roles**: grupos de usuarios
- **Autorización**: Permisos sobre las vistas y el modelo
- **Forms** y view tools

Django **utiliza sesiones y middleware** para gestionar automáticamente la autenticación de usuarios, con funciones para comprobar el usuario y la contraseña en la base de datos de usuarios.

Contiene además un **decorador** para ponerlo en las vistas que requieran usuarios autentificados.

AUTENTICACIÓN Y REGISTRO

Django ya proporciona plantillas, formularios y vistas para la autentificación y creación de usuarios: **Authentication views,** incluyendo el cambio de contraseña, la confirmación, etc.

Pero es más cómodo utilizar el **plugin django-allauth** que además tienen prevista la activación de cuentas por email o la autenticación delegada en redes sociales.

MARTA SOTO WERNER

TEMA 14: JAVASCRIPT, ES6 Y JQUERY

1. ¿QUÉ ES JQUERY?

JQuery es considerado un **framework de JavaScript o ambiente de desarrollo,** es decir, no es más que un conjunto de utilidades las cuales no necesitan ser programadas ya que ya fueron programadas, probadas y podemos utilizarlas de manera muy simplificada.

¿QUÉ PODEMOS HACER CON jQUERY?

Permite agregar efectos y funcionalidades complejas a nuestro sitio web, como por ejemplo: galerías de fotos dinámicas y elegantes, validación de formularios, calendarios, aparecer y desaparecer elementos en nuestra página, etc.

Ejemplo de uso (evento de click)

```
$( document ).ready(function() {

    $( "a" ).click(function( event ) {

        alert( "Thanks for visiting!" );

    });

});
```

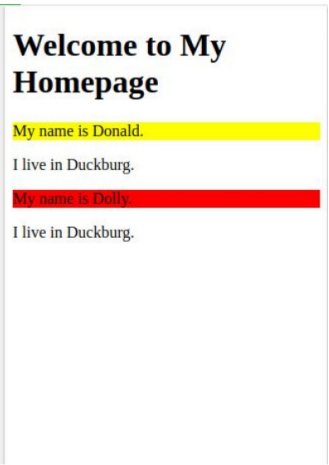
Ejemplo de uso para modificar class="a" y div id="2"

```
src="https://ajax.googleapis.com
/ajax/libs/jquery/3.3.1/jquery.m
in.js"></script>
<script>
$(document).ready(function(){
  $(".a").css("background-
color","yellow");
});
$(document).ready(function(){
  $("#2").css("background-
color","red");
});
</script>
</head>
<body>

<h1>Welcome to My Homepage</h1>

<p class="a">My name is Donald.
</p>
<p>I live in Duckburg.</p>

<div id="2">My name is Dolly.
</div>
<p>I live in Duckburg.</p>
```



2. AJAX (Asynchronous JavaScript And XML)

AJAX por definición se usa en aplicaciones que siendo ejecutadas en el cliente (en el navegador) se comunican con el servidor de manera asíncrona para obtener datos y solo recargar parte de la página necesaria.

TEMA 15: INTERNACIONALIZACIÓN DEL SOFTWARE

1. L10N (Localization)

Adaptar el software a otro país con otro:

- Alfabeto
- Idioma de la interfaz
- Formato de fechas
- Moneda
- Formato de las cantidades

2. I18N (Internationalization)

Preparar los programas para poder localizarlos fácilmente y para que cambien de entorno dinámicamente según la elección del usuario

3. LOCALIZAR

En general, **localizar** consiste en **traducir**:

- **La interfaz:** menús, cuadros de diálogo, plantillas, etc. Están en archivo .resx (windows) o .po (linux)
- **La ayuda**
- El resto de la **documentación**
- **Adaptación** cultural de la **interfaz**

Todo esto es **tarea de los traductores.**

4. VENTAJAS

- El ejecutable es único para todas las lenguas
- Los traductores están en archivos independientes, hechos por especialistas
- Se puede cambiar de idioma en tiempo de ejecución, dependiendo de la instalación del SO (escritorio) o del navegador (WEB)

5. GETTEXT

Gettext es una **librería tradicional de UNIX para I18N,** la que se usa en linux para los programadores de escritorio, php, python, etc.

UNIX: gettext y archivos.mo

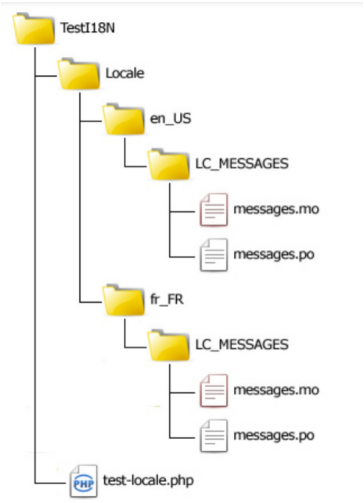
Los textos para cada lengua van en archivos **.mo (machine object)**, compilados a partir de los textos en archivos **.po (portable object)**

Están en un directorio a parte para cada **locale (Lengua)** por defecto en `/usr/share/locale.`

Para cada entorno se utiliza un **"locale"** distinto, cada uno con su nombre:

```
lengua[_pais][.codificacion][@modificacion]
```

- **es →** Español
- **es_AR →** Español para Argentina
- **es_AR.UTF-8 →** en codificación utf-8
- **es_ES@euro →** español para España modificación euro



Comandos:

- `locale` : para ver el locale actual
- `locale -a` : para ver todos los locales instalados

LOCALE: Cambio del comportamiento de alguna de las funciones de la librería estándar del C, según el valor de las variables de entorno asignadas a cada categoría.

Para **cambiar los mensajes ponemos:**

> En lugar de `$ printf"Hola mundo" → $ print gettext("Hola mundo\n")`

> También se puede **redefiniendo la función gettext:** `$ printf_("Hola mundo\n")`

Gettext busca el texto correspondiente al locale actual en el tiempo de ejecución en un archivo distinto.

ARCHIVOS .po (Portable Object)

Los **archivos .mo para cada lengua se generan a partir de archivos de texto ".po".**

Por ejemplo para mi traducción al inglés haría un archivo "mi:traduccion_EN.po" al directorio en_US que contendría:

```
#Traducciones al inglés
msgid "Hola mundo\n"
msgstr "Hello world\n"
…
```

Se puede usar el **programa xgettext** para sacar automáticamente una plantillas con todos los mensajes bajo gettext.

Una vez traducidos los **archivos .po se compilan con `msgfmt` a archivos.mo** y se ponen en los directorios correspondientes a los locales.

Proceso de traducción

1. Poner **gettext()** donde vaya a traducir el código fuente
2. Crear la **plantilla .pot** pasando la utilidad xgettext al código fuente
3. **Traducir rellenando una plantilla** para cada lengua, poniéndolas en un **archivo .po**
4. **Compilar** las traducciones **con** la utilidad **msgfmt** a un **archivo .mo**
5. **Poner las traducciones** en sus **locales correspondientes** al instalar el programa

Formas del plural

Para estos casos se utiliza la **función `ngettext`:**

```
$ printf(ngetttext("Un archivo", %d archivos", n),n);
```

- → El primer argumento es el singular → se usa como índice en la búsqueda
- → El segundo es el plural
- → El tercer es el número

TEMA 16: DESPLIEGUE

Para **poner en producción una aplicación web** necesitamos hacer algún cambio:

- ➔ Deshabilitar el ambiente de depuración
- ➔ Cambiar el servidor web, por el definitivo de producción

1. DESPLIEGUE DE DJANGO

Para deshabilitar el ambiente de desarrollo debemos ir al a**rchivo `settings.py` y cambiar:**

```
# settings.py

# DEBUG = True, pasa a:

DEBUG = False  # y

# ALLOWED_HOSTS = [] pasa a:

ALLOWED_HOSTS = ['*'] # o el que sea en producción
```

2. WSGI

Hay que **conectar la aplicación con un servidor web de producción** en un lugar del runserver. Django, como el resto de los frameworks de python, necesita un servidor web con interface.

WSGI tiene previsto que haya middleware entre el servidor web y la aplicación, a ambos lados del API.

2. GUINCORN

Guincorn es un **servidor web wsgi** muy sencillo de instalar que sustituye al runserver de desarrollo:

```
# para instalar
sudo pip3 install gunicorn # o en requirements.txt

# para ejecutar el puerto 8000
/usr/local/bin/gunicorn mi_app.wsgi --bind: 0.0.0.0:8000
```

Django ya tiene el **archivo .wsgi en el mismo directorio de settings.py.**

3. SUPERVISIÓN

Para arrancar y mantener arrancado el servidor gnunicorn podemos usar **`supervisord`**, que vigilará que el proceso siempre esté ejecutándose (en el caso de no usar contenedores)

```
# antes pip install supervisor, o en requirements.txt
# en /etc/supervisor/conf.d/ # copiado desde Dockerfile
[program:gunicorn]
command=/usr/local/bin/gunicorn sitio_web.wsgi  --bind 0.0.0.0:8000
directory=/path/donde/este/manage.py
user=elquesea
autostart=true
autorestart=true
redirect_stderr=true
```

También podríamos usar **systemd**.

4. NGINX

Además necesitaremos **un segundo servidor web** para servir los archivos estáticos, balance de carga, cifrado, etc.

Nginx sirve para **servir los archivos, balance de carga, restricciones de acceso, cifrado, etc.**

5. STATIC

Django dejará de servir los archivos en `/static y / media` cuando esté `DEBUG = False`, por tanto haremos un **script para copiar los archivos al directorio donde los sirva nginx**

```
#/bin/bash
cp -r static/ /var/www/static
cp -r media/ /var/www/media
```

6. DOCKER

De esta manera **se independiza la instalación del servidor de producción**, el ejecutable va a ser siempre el mismo.

Además **la instalación va a simplificarse** al utilizarse scripts de aprovisionamiento.

Dockerfile en un solo contenedor o docker-compose.