



El futuro digital  
es de todos

MinTIC



# Ciclo 3: Desarrollo de Software

## 05 Back-End

```
element* item = el->FirstChildElement(); item != 0; item = item->NextChildElement()
{
    boost::stringref el_name = item->Attribute( "name" );
    boost::stringref type = item->Attribute( "type" );
    ...
    std::string str;
    float x = boost::lexical_cast<float>( item->Attribute( "x" ) );
    float y = boost::lexical_cast<float>( item->Attribute( "y" ) );
    float offset = boost::lexical_cast<float>( item->Attribute( "offset" ) );
    ...
    spriteDescList::iterator sp = sprite_descs.begin();
    while( sp != sprite_descs.end() && sp->name_ != spritename )
        ++sp;
}
```



El futuro digital  
es de todos

MinTIC

# Objetivo de Aprendizaje

**Identificar** los principales elementos asociados a la construcción de un **componente lógico** (back-end), usando el lenguaje de programación **Python** y el framework **Django REST**.



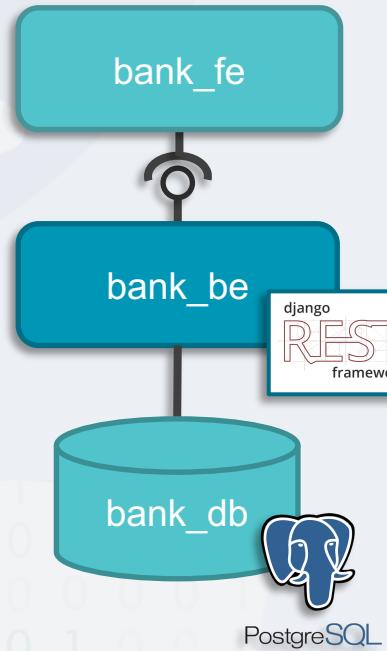
El futuro digital  
es de todos

MinTIC

# Parte 1



# Django: Caso de estudio



En sesiones anteriores se desarrolló la **capa de datos** del caso de estudio, por medio del despliegue de la base de datos **bank\_db**. En esta y en las próximas sesiones se realizará el desarrollo y el despliegue de la **capa lógica**, es decir, del componente de **back-end** del caso de estudio: **bank\_be**. En este se implementará un sistema de registro y autenticación con tokens, utilizando para su construcción **Django REST Framework**.



# Framework: Django

Un framework es un **conjunto** de conceptos, prácticas y criterios **estandarizados** que proporcionan una **estructura completa** que **facilita el desarrollo de software**.



Para el desarrollo del componente back-end se utilizará **Django**, un **framework** de **desarrollo web** escrito en **Python** que sigue el patrón arquitectónico **Modelo-Vista-Controlador (MVC)**. En general, Django fomenta la **reusabilidad**, la **conectividad**, el **desarrollo rápido**, y el principio 'Don't repeat yourself' (**DRY**).



# MVC: Definición

MVC o Modelo-Vista-Controlador es un **patrón arquitectónico** ampliamente utilizado en el **desarrollo de aplicaciones**, que indica la **separación** de la aplicación en tres componentes:

1. El **Modelo**: encargado del manejo de los **datos**.
2. La **Vista**: encargada de proveer una **interfaz** al usuario.
3. El **Controlador**: encargado de gestionar la **lógica** de la aplicación.

Este patrón es utilizado en múltiples frameworks, sin embargo, **cada uno lo implementa a su manera**.





El futuro digital  
es de todos

MinTIC

# Paquete: Django REST

Un paquete es un **conjunto de funcionalidades** específicas encapsuladas que **simplifican** tareas complejas y que se pueden **utilizar en múltiples proyectos**.



Django REST Framework es un paquete que se utilizará en el caso de estudio, pues dispone de un **conjunto de funcionalidades** que permiten la construcción fácil y rápida de **APIs** de tipo **Web**, que siguen el estilo arquitectónico **REST**.

[Imagen] Creacion y consumo de APIs con Django REST Framework. (s. f.). [PNG]. Azul School. <https://www.azulschool.net/wp-content/uploads/2021/04/Creacion-y-consumo-de-APIs-con-Django-REST-Framework.png>



# Django: Proyecto y Aplicaciones

En Django se manejan 2 **niveles** de jerarquía **diferentes**, pero **igualmente importantes**: los **proyectos** y las **aplicaciones**.

El proyecto es donde se guardan todas las **configuraciones** del servidor del componente, y donde se **administran** las **aplicaciones** que este utiliza.

Las aplicaciones por otra parte, son los **módulos** que contienen las **funcionalidades** del componente.





El futuro digital  
es de todos

MinTIC

# Parte 2



# Caso de Estudio



En esta sesión se va a realizar la **configuración preliminar** del componente de la capa lógica, la cual permitirá la implementación en sesiones futuras del **sistema de autenticación** con tokens, y del **almacenamiento de datos** de los usuarios en el **componente lógico**. Sin embargo, para realizar dicha configuración es necesario **aclarar** primero algunos **conceptos** que se utilizarán.



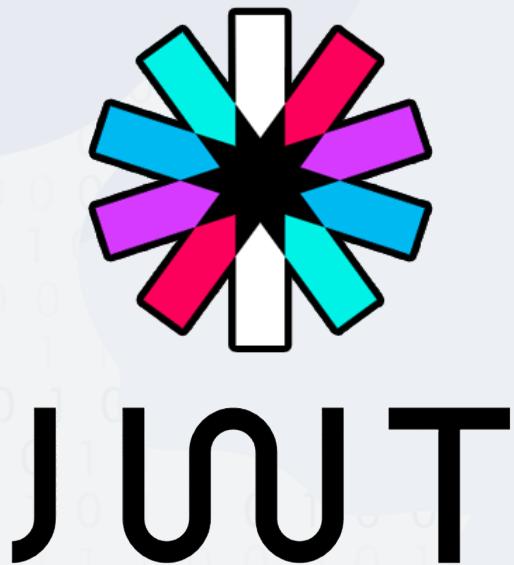
# JSON Web Token: Definición

Abreviados **JWT**, los **JSON Web Token** son un **mecanismo** estandarizado que permite el **intercambio seguro** de datos entre dos partes. Este mecanismo es muy utilizado en los procesos de **autenticación** ya que **protege** los datos de autenticación y **no utiliza memoria**, pues no requiere almacenar las sesiones en la base de datos del sistema. Esto último, debido a que un **token** es una **cadena de texto** que contiene **información codificada**. Por lo tanto, toda información necesaria para verificar la identidad de un usuario y conocer el estado de la sesión, se encuentra **encriptada en el token**.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpvag4gRG9lIiwiawF0IjoxNTE2MjM5MDIyfQ.ikFGEvw-Du0f30vBaA742D_wqPA5BBHXgUY6wwqab1w
```



# JSON Web Token: Uso



Los JWT se utilizan en un proceso de autenticación de la siguiente forma :

1. El usuario se autentica en el sistema **enviando** al componente lógico el **usuario** y la **contraseña** de su cuenta.
2. El componente recibe las credenciales y verifica que el **usuario existe** y que la **contraseña** ingresada es **correcta**.
3. Si las credenciales son **correctas**, el componente responde al usuario con un **token asociado** a la cuenta.
4. Cuando el usuario necesita realizar una **petición** que requiera de **autenticación**, envía la petición al componente lógico junto con el **token asociado** a su cuenta. De esta forma, el componente **verifica** el token y **ejecuta** la petición si el token es válido.



# JSON Web Token : Tipos

Siguiendo este mismo flujo, en el desarrollo del componente lógico se utilizarán 2 tipos de tokens: los **access token** y los **refresh token**.

Un **access token** es aquel que tiene una validez por un **corto periodo de tiempo**, por ejemplo, 5 minutos.

Por otro lado, un **refresh token** es aquel que es válido durante **24 horas**, y se utiliza para obtener un **nuevo access token** cuando el anterior pierde su validez.





# Simple JWT: Definición

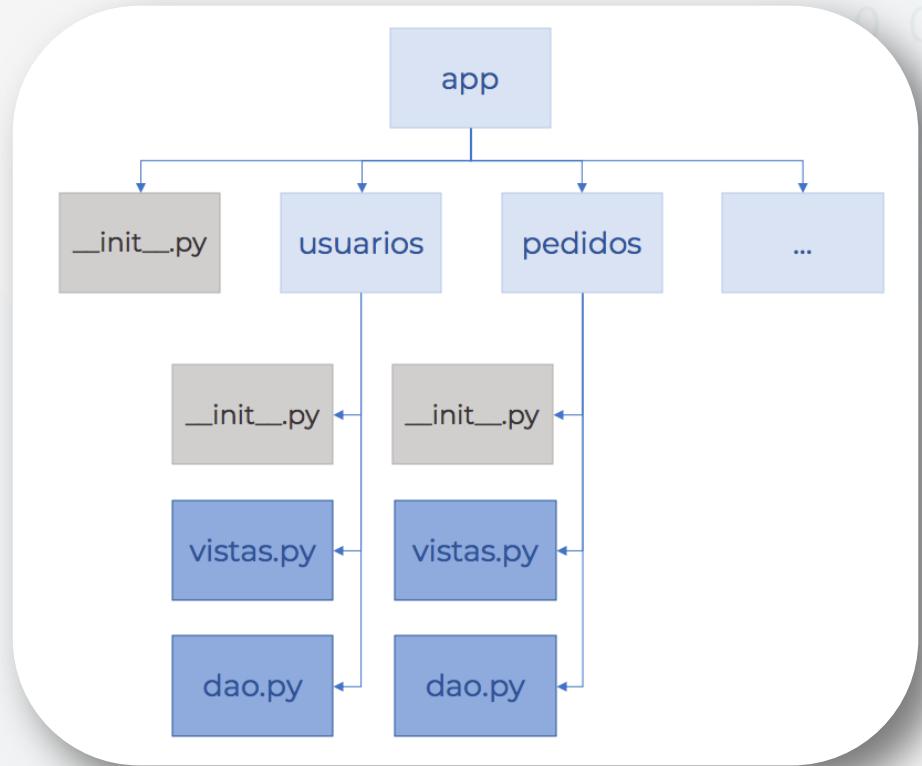
Por defecto, los proyectos de **Django REST Framework** implementan un **sistema de registro y autenticación** de usuarios, lo cual agiliza el desarrollo de APIs que requieran esta funcionalidad. En el desarrollo del componente lógico se utilizará **dicho sistema** junto al paquete **Simple JWT**. Este último se encargará de **integrar** todo el **sistema de tokens** dentro del sistema de autenticación provisto por Django REST. De esta forma, se ahorrará bastante trabajo y se obtendrá un **sistema de autenticación que implementa los JWT**.





# Crear paquetes de Python

Como se ha visto, los **paquetes** son muy útiles para **reutilizar código** y funcionalidades, sin embargo, este no es su único uso, pues también se utilizan para **agrupar código y ordenar** la estructura del proyecto. Para esto, se crean **carpetas** que contienen los **archivos de Python** (Archivos .py) y se les añade el archivo **`__init__.py`**, el cual se encarga de **exportar las funcionalidades** de los archivos Python para que puedan ser **accedidos y utilizados** por el resto de la aplicación.





El futuro digital  
es de todos

MinTIC

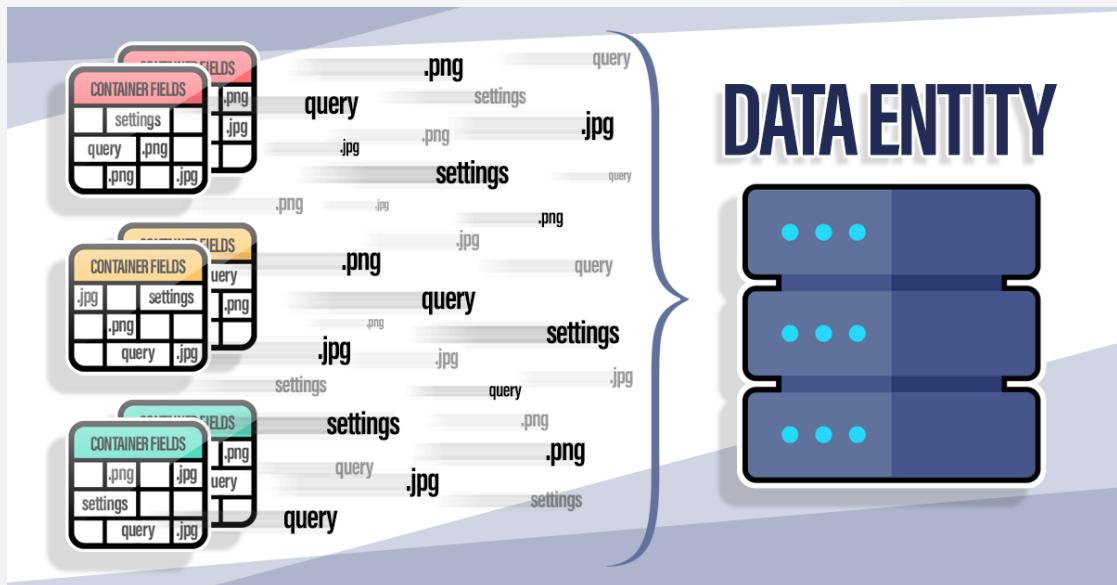
# Parte 3



# Entidades de Datos: Definición

Una **entidad de datos** es un **abstracción** que representa un **objeto** de la vida **real**. Una entidad permite agrupar la **información** relevante del **objeto** de acuerdo al **contexto** del sistema, dicha información se estructura en una serie de **atributos**, los cuales tienen un **tipo**, un **nombre**, y ciertas **restricciones**.

En la **práctica**, una **entidad de datos** es un **modelo, tabla o clase** sobre el cual se definen los correspondientes atributos.

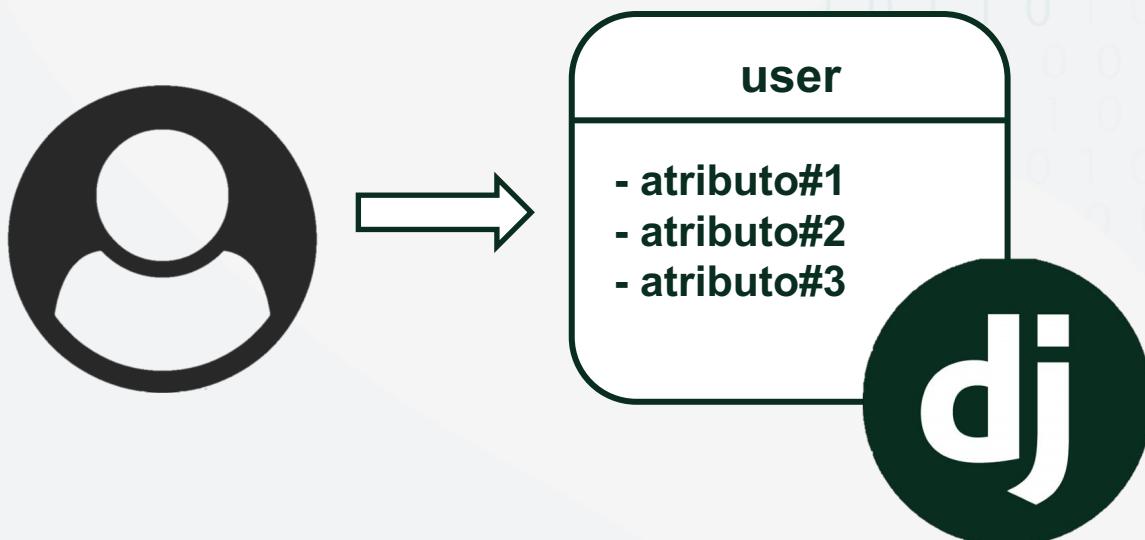




# Django: Modelos

En **Django** las **entidades de datos** se representan a través de **clases** llamadas **Modelos**, estos modelos no se definen desde cero sino que **parten** de clases **bases** las cuales le **agregan funcionalidades** adicionales, esto **facilita** el proceso de **desarrollo**.

**Django** también provee una serie de **clases y objetos** que permiten definir los **nombres, tipos y restricciones** de los **atributos** de manera **fácil y sencilla**.

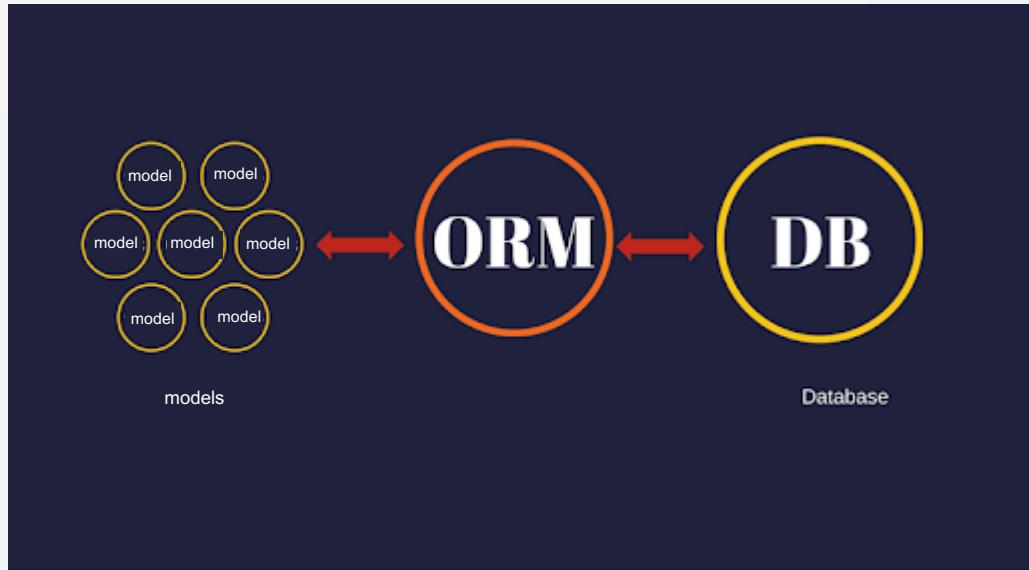




# Django: ORM

Un **ORM** es una **herramienta** que a través de los **modelos** permite **definir** la **estructura** con la que se le dará **persistencia** a los **datos**.

Además de la estructura, el **ORM** también se encarga por completo de la **comunicación** con la **base de datos**, esto **elimina** la necesidad de hacer consultas con el lenguaje **SQL**, ya que el **ORM** define una serie de **funciones** que **cumplen** el mismo **objetivo**.

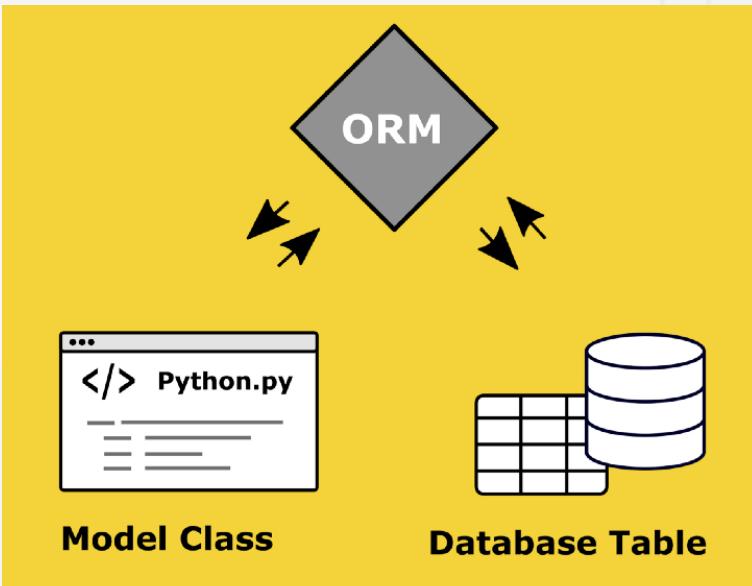




# Django: Migraciones

Para asegurar la **persistencia** de datos en la **capa lógica** es necesario tener un **esquema compatible** en la **base de datos**. Para lograr esto, el **ORM** ofrece el mecanismo de **migraciones**.

Una **migración** permite **crear** un **esquema** en una **base de datos** a **partir** de los **modelos** definidos en la **capa lógica**, este **esquema** es **equivalente** y **compatible** con la estructura de la capa lógica.





El futuro digital  
es de todos

MinTIC

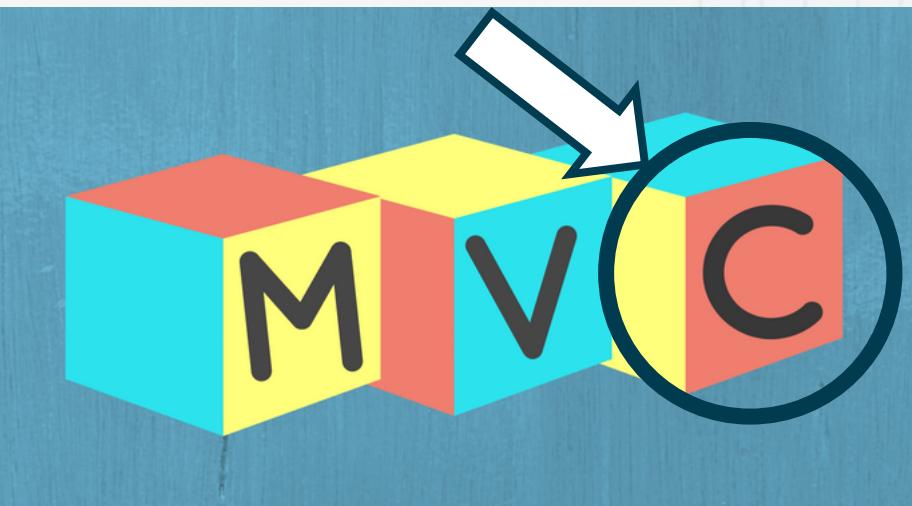
# Parte 4



# MVC: Controller

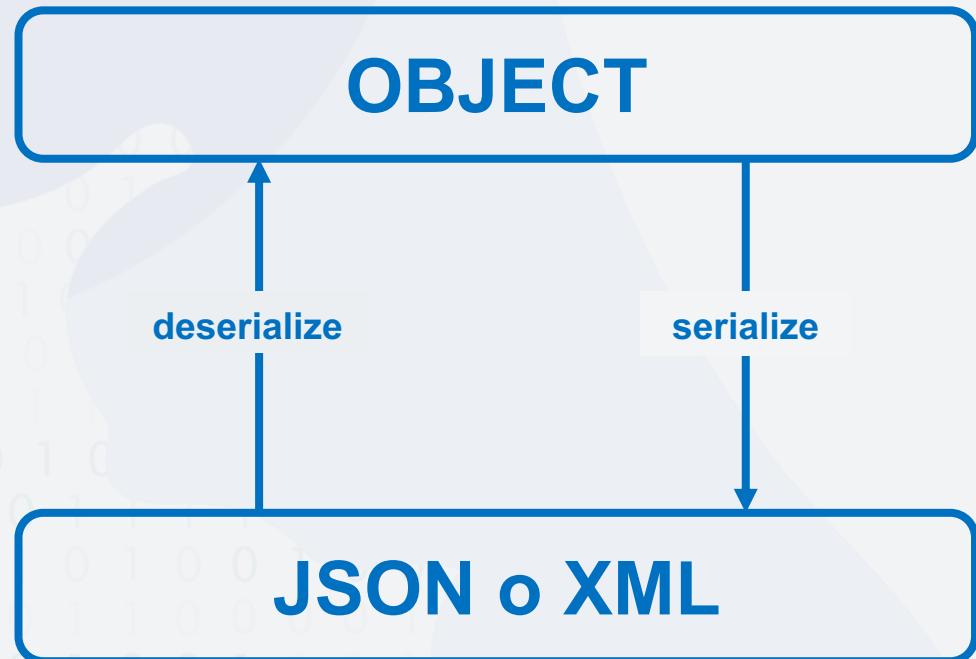
Dentro del **patrón arquitectónico MVC** el **Controller** o **Controlador** se encarga de **mantener** una **comunicación** activa y **manejar** las **interacciones** entre el **usuario** a través de las **vistas** y la **base de datos**, a través del **modelo**.

En la mayoría de **casos**, el **procesamiento** realizado en los componentes de **back-end** se suelen realizar en el **Controller**.





# Django: Serializers



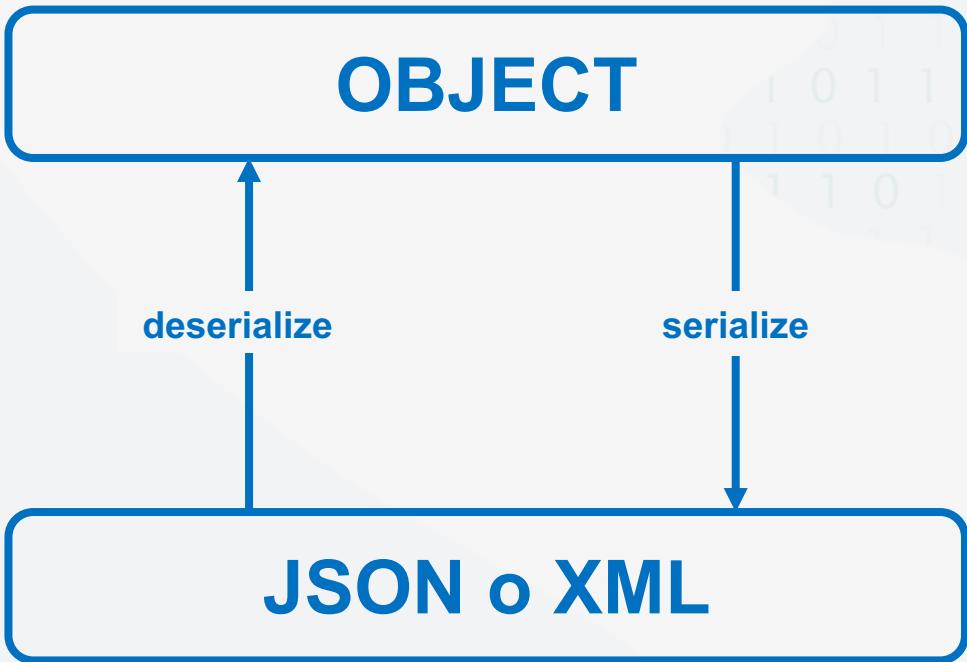
Un **serializer** es un tipo de **controlador** cuya principal funcionalidad es **transformar información** de un **formato X** a un **formato Y**, y **viceversa**, en Django es útil ya que permite transformar los **objetos creados** a partir de la **información almacenada** en la **base de datos** a un **formato JSON o XML**, el cual es comprensible para el usuario final.



# Django: Serializers

Un **serializer** esta compuesto por **dos principales procesos**, en primer lugar esta el proceso de **serialización** el cual consiste en transformar un **objeto** en un formato **JSON** o **XML**, entendible por el usuario, y en segundo lugar el proceso de **deserialización** el cual consiste en transformar un **JSON** o **XML** en un **objeto**.

A pesar de que el nombre puede generar confusión, un **serializer** debe tener un **proceso de serialización** y un **proceso de deserialización**.





El futuro digital  
es de todos

MinTIC

# Parte 5



El futuro digital  
es de todos

MinTIC

# Caso de estudio: Vistas



En sesiones anteriores se explicaron y desarrollaron los **modelos** y **controladores** del **componente lógico**. Para finalizar el desarrollo del componente, en esta sesión se desarrollará el nivel faltante del patrón arquitectónico MVC: **las vistas**.

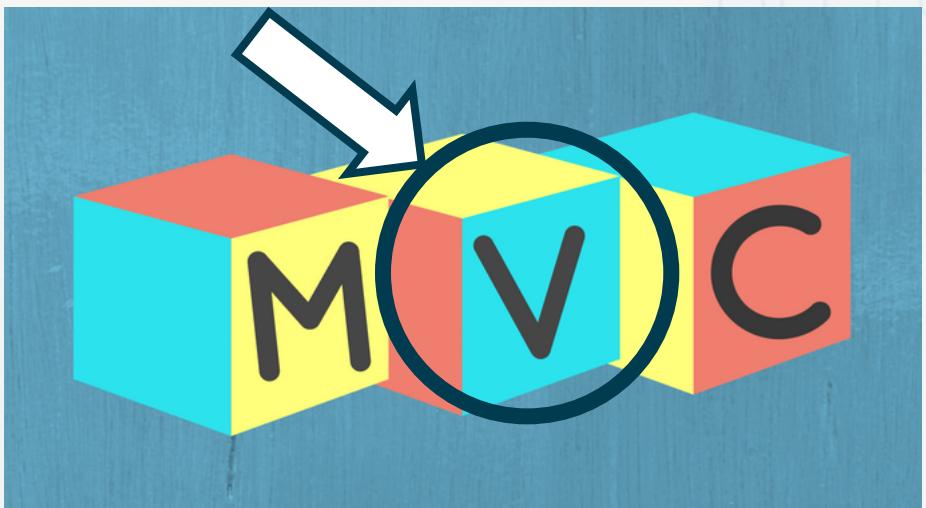
Para ello, a continuación se explicarán **algunos conceptos importantes** para su desarrollo.



# MVC: Vistas

Como ya se ha mencionado, las **vistas** son el nivel que provee una **interfaz** al usuario final, esto quiere decir que son el nivel del componente al cual el usuario **envía peticiones** y del cual **recibe una respuesta**.

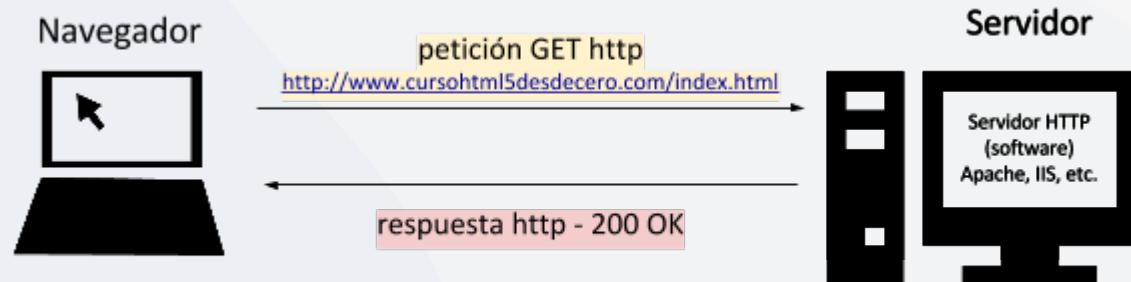
Dado que el componente lógico es una **API** de tipo **REST**, las **funcionalidades** se expondrán a los usuarios por medio de **peticiones HTTP**, y en este caso, se enviarán las **respuestas** al usuario en **formato JSON**.





# Peticiones HTTP: Definición

Las peticiones HTTP son **mensajes** enviados por un **cliente** (web, móvil, etc.), para **iniciar una acción** en un **servidor**, independientemente de si dicha acción retorna una respuesta o no. Estas contienen **bastante información**, que permite, entre otras cosas, establecer la conexión con el servidor. Sin embargo, para el desarrollo del componente lógico solo se debe saber que una **petición HTTP** se realiza **por medio de una URL**, que tiene un **Body** donde se ingresan los **datos necesarios** para ejecutar la acción en el servidor, y que **siempre** incluye un **método HTTP: GET, POST, PUT o DELETE**.





# JSON: Definición

Existen múltiples **formatos** que se pueden seguir para **enviar** información de **respuesta** al usuario que realiza una **petición HTTP**, sin embargo, uno de los más utilizados es el formato **JSON** (JavaScript Object Notation). Este se utiliza para representar **datos estructurados** con la sintaxis de un objeto de JavaScript. Como se puede apreciar en la imagen, dicha **sintaxis** es muy **similar** a la sintaxis de un **diccionario en Python**.

```
data.json
1  {
2   "clients": [
3     {
4       "first_name": "Sigrid",
5       "last_name": "Mannock",
6       "age": 27,
7       "amount": 7.17
8     },
9     {
10      "first_name": "Joe",
11      "last_name": "Hinners",
12      "age": 31,
13      "amount": [
14        1.9,
15        5.5
16      ]
17    },
18    {
19      "first_name": "Theodoric",
20      "last_name": "Rivers",
21      "age": 36,
22      "amount": 1.11
23    }
24  ]
25 }
```

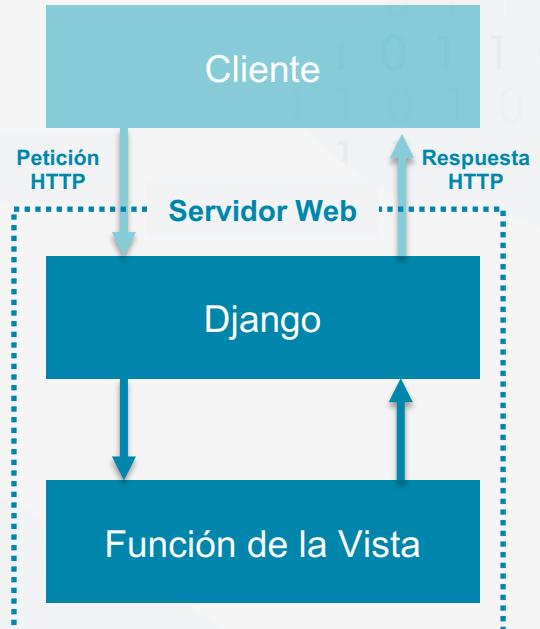
Lín. 1, Col. 1 Espacios: 4 UTF-8 LF JSON



# Django: Vistas

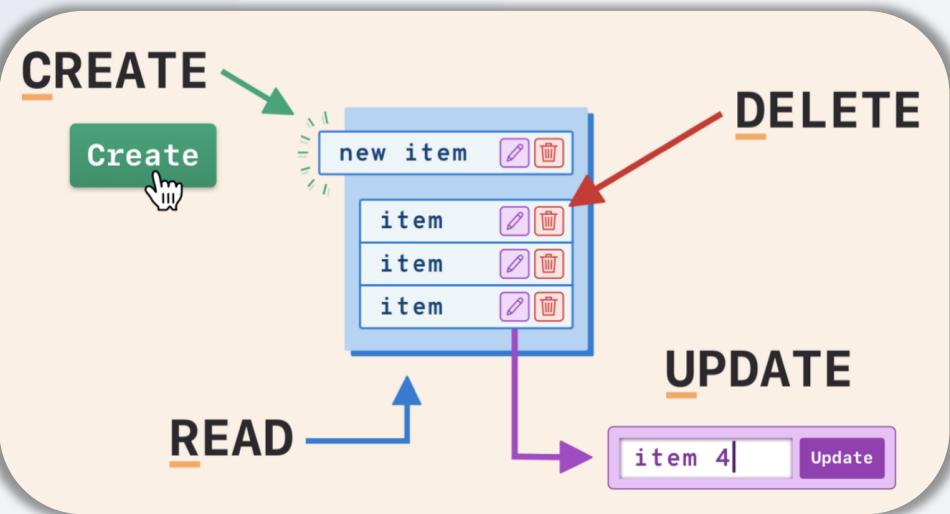
Para **exponer** las funcionalidades al usuario, en Django se **asocia** cada **vista** con una **URL** a la que el usuario tendrá acceso. La **implementación** de estas **vistas** en Django se realiza por medio de **clases** que tienen **funciones asociadas** a al menos uno de los **métodos HTTP**: GET, POST, PUT y/o DELETE.

De esta forma, cuando una petición **llega** al servidor por medio de una URL, esta se **remite** a la **vista** correspondiente. Así, la vista se encarga de **ejecutar la acción solicitada** y de **retornar una respuesta** al usuario de acuerdo con la implementación (solo si el método HTTP es uno de los métodos implementados en la vista).





# Django REST: Vistas



En el **desarrollo de las vistas** se utilizarán las **implementaciones** dispuestas por **Django REST**. En estas se encuentran desde las vistas más **completas** que implementan todo el **CRUD** para un modelo, hasta la vista más **básica** que se utiliza para crear una vista **desde cero**.

En la guía de esta sesión se mostrará el **funcionamiento** de dos de estas implementaciones. Sin embargo, Django REST ofrece **muchas más** implementaciones para **adaptarse** a las **necesidades** de cualquier proyecto.



El futuro digital  
es de todos

MinTIC

# Parte 6



# Despliegue: Definición

Desplegar es una de las **tareas** que se debe realizar cuando se desea llevar los **cambios** de la **aplicación** desde el **entorno** de desarrollo **local** al entorno de **producción**, de tal forma que el usuario final pueda **acceder** a la **aplicación** desarrollada, a través de **internet**. Para ello, se ejecuta la aplicación en un **servidor** y se permite el **acceso** desde cualquier dispositivo conectado a internet.

Existen varios tipos de despliegue, entre los cuales se encuentran el **local** y el **remoto**.





El futuro digital  
es de todos

MinTIC

# Despliegue Local: Definición



Un despliegue local es aquel que se realiza en un **computador** o **servidor propio**. Normalmente se realiza para **probar** que una aplicación funciona **correctamente**, sin embargo, si se desea **exponer** la aplicación en **internet** es necesario realizar una **inversión de tiempo** y **dinero** considerables, para **configurar** y **adecuar** los equipos utilizados.

[Imagen] Internet. (s. f.). [JPG]. CDN. <https://cdn.mos.cms.futurecdn.net/79d2Ms7twJB6KbrRXV8dmG-1200-80.jpg>



# Despliegue Remoto: Definición

Un despliegue remoto es aquel que se realiza en un **computador o servidor en la nube**, y al que por ende, solo se tiene **acceso** a través de **internet**. Dos de las formas más comunes para realizar un despliegue remoto son: mediante el uso de una **máquina virtual** y mediante el uso de una **plataforma como servicio** destinada para ello.

Las plataformas como servicio ofrecen mayores facilidades, pues basta con **subir el código** de la aplicación, y estas se encargan de realizar el **despliegue** y de **mantener** la aplicación en funcionamiento.





El futuro digital  
es de todos

MinTIC

# Despliegue: Heroku

Heroku es una plataforma en la **nube** dispuesta por Salesforce.com que permite el **despliegue** remoto de **aplicaciones** escritas en distintos lenguajes de programación, entre los que se encuentran **Java**, **Python** y **JavaScript**.

Esta permite a los desarrolladores **construir**, **ejecutar** y **escalar** aplicaciones **fácilmente**, independientemente del lenguaje de programación en que fue desarrollada.



[Imagen] Salesforce Logo. (s. f.). [PNG]. Wikimedia. [https://upload.wikimedia.org/wikipedia/commons/thumb/f/f9/Salesforce.com\\_logo.svg/1200px-Salesforce.com\\_logo.svg.png](https://upload.wikimedia.org/wikipedia/commons/thumb/f/f9/Salesforce.com_logo.svg/1200px-Salesforce.com_logo.svg.png)

[Imagen] Heroku Logo. (s. f.-b). [PNG]. Wikimedia. [https://upload.wikimedia.org/wikipedia/commons/thumb/e/ec/Heroku\\_logo.svg/2560px-Heroku\\_logo.svg.png](https://upload.wikimedia.org/wikipedia/commons/thumb/e/ec/Heroku_logo.svg/2560px-Heroku_logo.svg.png)



El futuro digital  
es de todos

MinTIC

# Pruebas: Postman

Luego de desplegar el componente lógico, se deberán realizar las **pruebas** necesarias para comprobar que el **componente lógico** está funcionando **correctamente**. Para ello, se utilizará **Postman**, una herramienta que ofrece la capacidad de realizar **peticiones HTTP** a cualquier **API** en la web.



POSTMAN

[Imagen] Postman Logo. (s. f.). [PNG]. Medium. [https://miro.medium.com/max/1200/0\\*li4wyJ4yMq\\_0Vm\\_U.png](https://miro.medium.com/max/1200/0*li4wyJ4yMq_0Vm_U.png)

UNIVERSIDAD NACIONAL  
DE COLOMBIA

Mision  
TIC2022



# Peticiones HTTP

Existen 3 formas de **enviar información** al servidor por medio de una **petición HTTP**:

1. Por medio de la **URL**, donde se incluyen **parámetros** sencillos como una cadena corta, o un número.
2. Por medio de los **Headers** de la petición, donde se incluye **información corta** necesaria, como por ejemplo el token de autorización.
3. Por medio del **Body** de la petición, donde se incluye cualquier tipo de **información estructurada**, generalmente con formato **JSON**.

