



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 del framework Django REST.



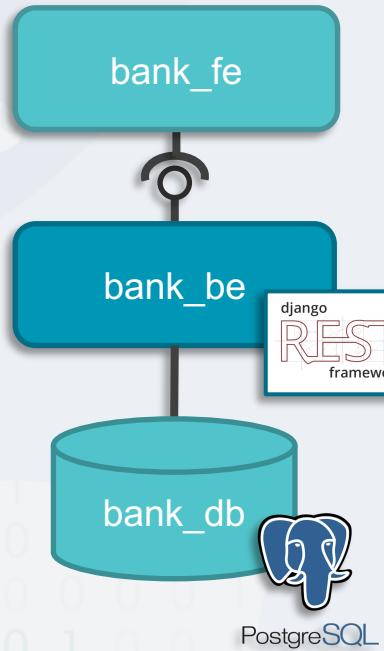
El futuro digital  
es de todos

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.



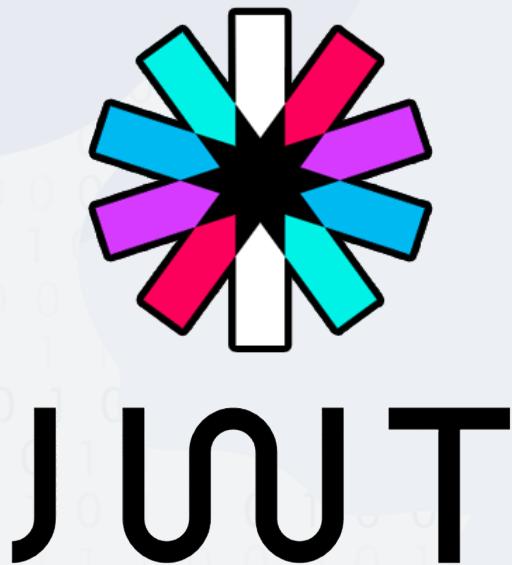
# JSON Web Token: Definición

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

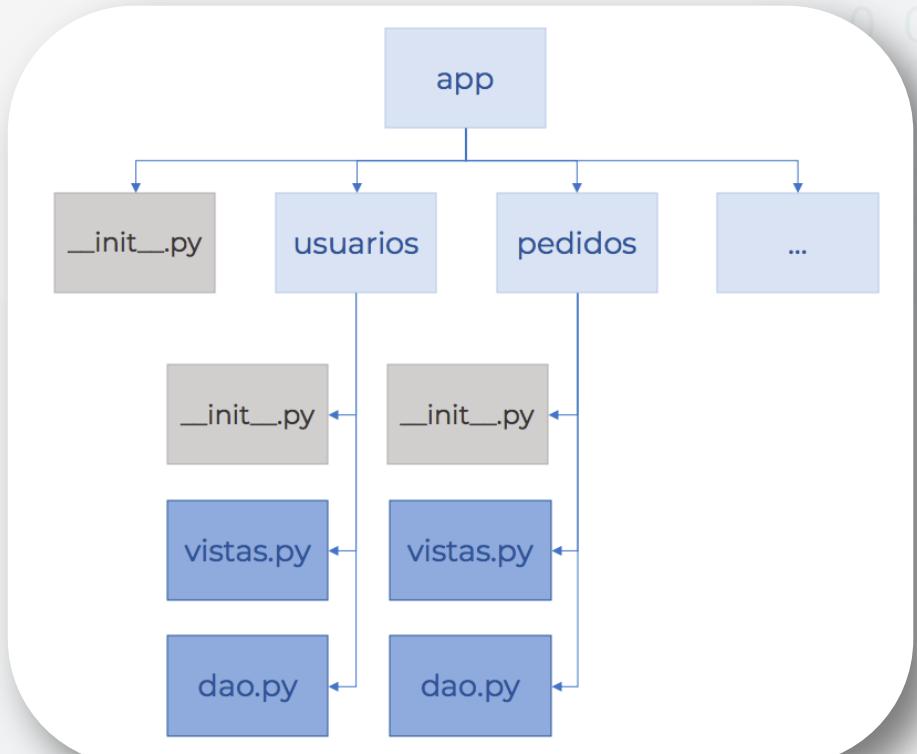
Por defecto, los proyectos de **Django REST Framework** implementan un **sistema de registro y autenticación** de usuarios, lo cual agiliza el desarrollo de APIs que requieran esta funcionalidad. En el desarrollo del componente lógico se utilizará **dicho sistema** junto al paquete **Simple JWT**. Este 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

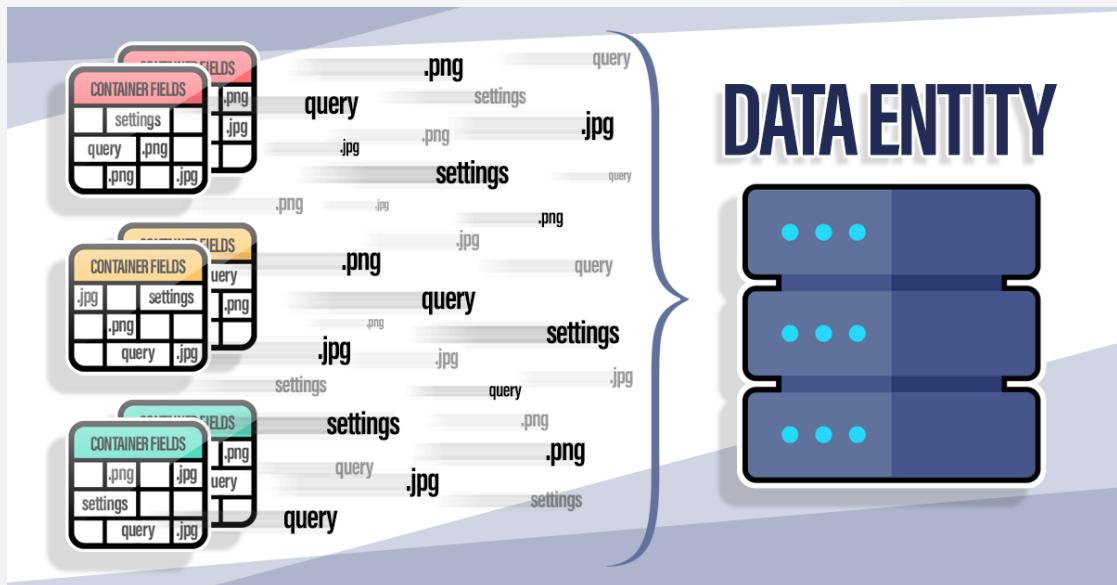
# Parte 3



# 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.

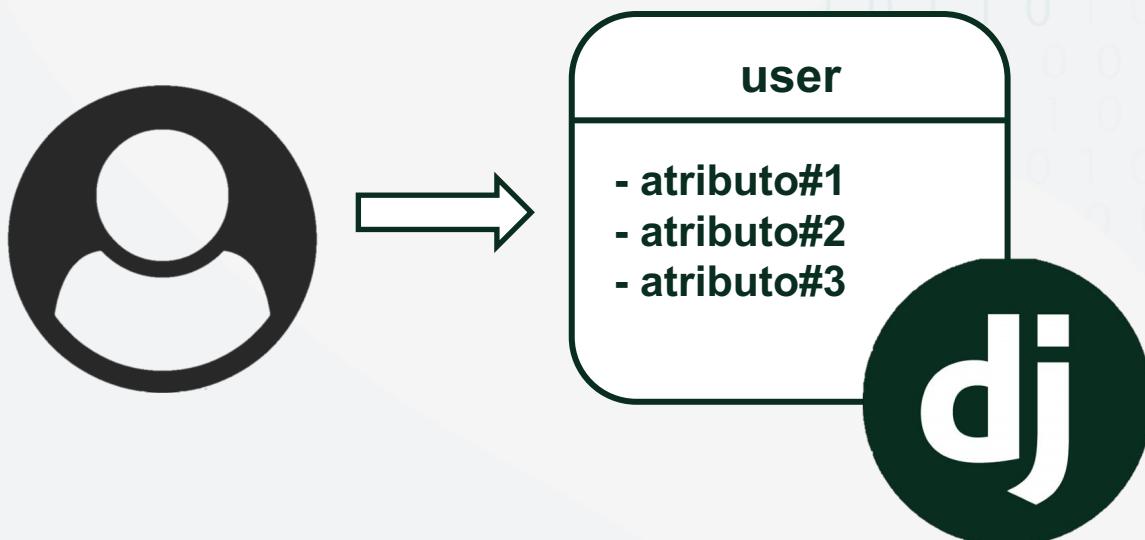




# Django: Modelos

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**.

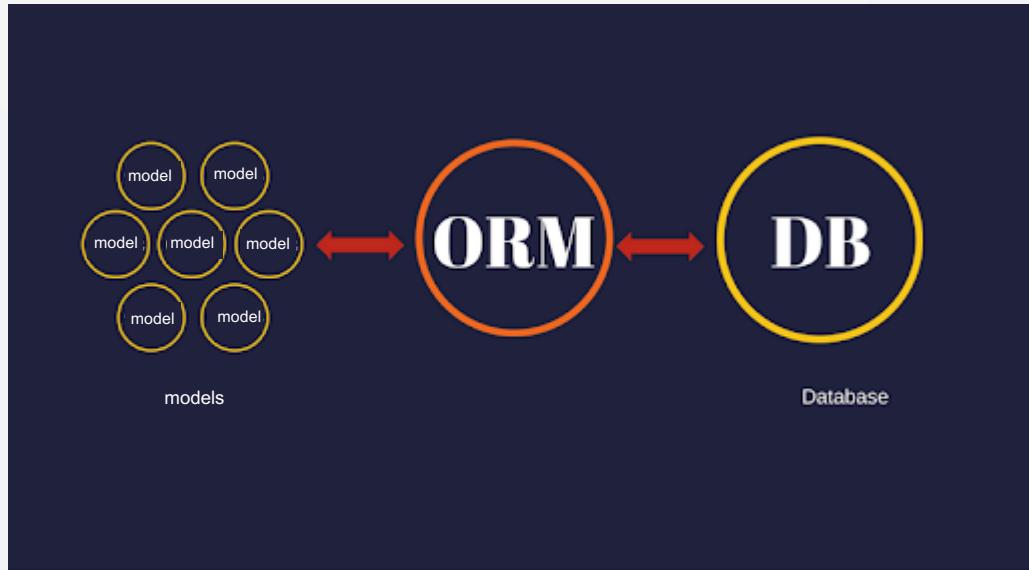




# Django: ORM

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**.

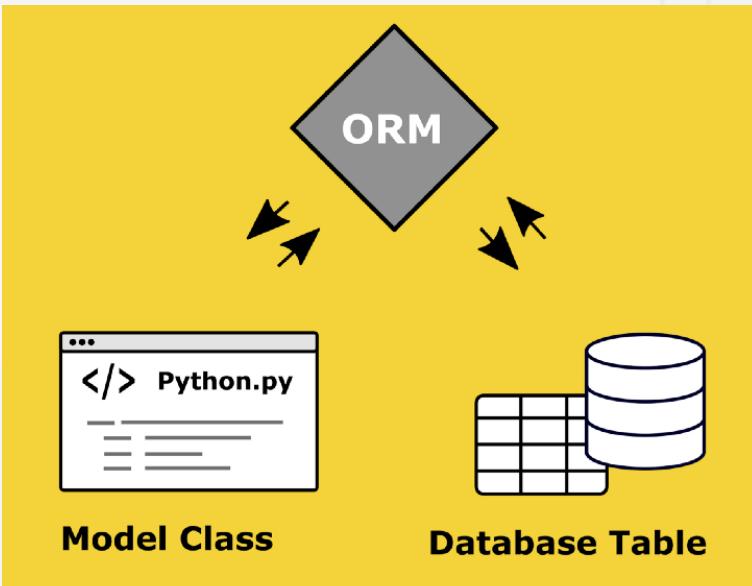




# Django: Migraciones

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

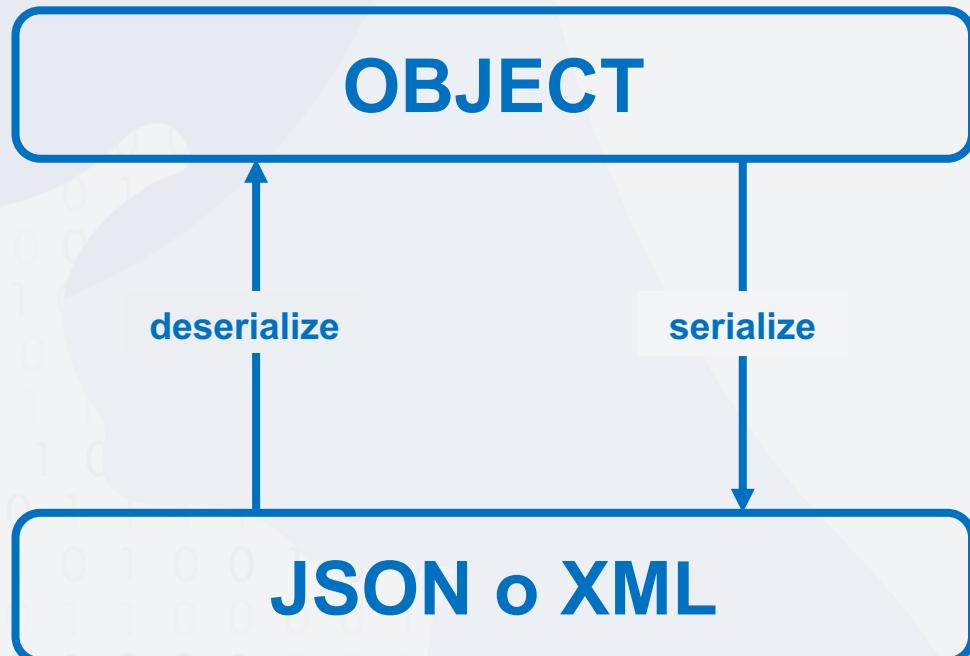
Dentro del **patrón arquitectónico MVC** el **Controller** o **Controlador** se encarga de **mantener** una **comunicación** activa y **manejar** las **interacciones** entre el **usuario** a través de las **vistas** y la **base de datos**, a través del **modelo**.

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**.

