

UNIVERSIDAD AUTÓNOMA DE MADRID

DEPARTAMENTO DE INFORMÁTICA

Sistemas Informáticos II

Práctica - 1A

Roberto MARABINI

Registro de Cambios

Versión ¹	Fecha	Autor	Descripción
1.0	10.12.2024	RM	Primera versión.
1.0	17.02.2025	RM	Introducir comentarios Daniel Lobato

¹La asignación de versiones se realizan mediante 2 números $X.Y$. Cambios en Y indican aclaraciones, descripciones más detalladas de algún punto o traducciones. Cambios en X indican modificaciones más profundas que o bien varían el material suministrado o el contenido de la práctica.

Índice

1. Objetivos	4
2. Descripción del Proyecto	4
2.1. Propósito	5
2.2. Componentes	5
2.3. Flujo General	7
3. Instrucciones para Configuración y Despliegue	8
3.1. Requisitos Previos	9
3.2. Estructura de directorios	10
3.3. Creación y Configuración del proyecto	10
3.4. Endpoints	13
4. Despliegue de la aplicación en las máquinas virtuales	14
4.1. Requisitos Previos	14
5. Introducción a la API REST	16
5.1. Características de una API REST	17
5.2. Implementación del API REST usando <i>Django</i>	17
5.3. Implementación del cliente del API REST usando <i>Django</i>	22
6. Criterios de evaluación	25
A. Configuración de un Repositorio Git Remoto Usando SSH	26
A.1. Crear un Repositorio Git Bare en el Equipo Remoto VM2	26
A.2. Conectar el Repositorio Local al Remoto	26
A.3. Realizar un Push Desde el Repositorio Local	27
A.4. Actualizar Archivos Automáticamente en el Equipo Remoto	27
B. Configuración de gunicorn	28
B.1. Crear un Archivo de Configuración para el Servicio Gunicorn	29
B.2. Explicación de los Parámetros Importantes	30

B.3. Recargar systemd y Activar gunicorn	30
C. Script que lee la tabla voto	31

1. Objetivos

En el ámbito de la informática, la interacción eficiente entre clientes y servidores es un aspecto fundamental para el diseño de aplicaciones distribuidas. La selección de la tecnología adecuada para esta comunicación puede impactar directamente en el rendimiento, la escalabilidad y la latencia del sistema. Por ello, evaluar y comparar diferentes enfoques de comunicación es una tarea crucial para tomar decisiones informadas en el desarrollo de software.

El objetivo de las prácticas de esta asignatura es medir y analizar la eficiencia de la comunicación entre clientes y servidores utilizando diversas soluciones ampliamente empleadas en la industria, tales como:

- **API REST:** basada en el intercambio de mensajes HTTP, ampliamente adoptada por su simplicidad y compatibilidad.
- **RPC (Remote Procedure Call):** un protocolo que permite invocar métodos remotos como si fueran locales, ofreciendo mayor integración y menor sobrecarga en ciertos casos.
- **Sistemas de colas de mensajes:** que introducen una capa de asincronía y desacoplamiento entre los componentes, favoreciendo la tolerancia a fallos y la escalabilidad.

En esta primera práctica nos centraremos en la implementación de una API REST y su despliegue en las máquinas virtuales.

2. Descripción del Proyecto

En esta práctica vamos a partir de un proyecto de ejemplo que tendrá que ser modificada para crear una aplicación web distribuida basada en una API REST.

La aplicación web implementada es un sistema de votación electrónica diseñado para crear/borrar/listar votos. Su propósito principal es garantizar la integridad del proceso electoral mediante la verificación del censo de votantes y el control del registro de votos. A continuación, se detallan sus componentes principales:

2.1. Propósito

El sistema tiene los siguientes objetivos:

- Garantizar que un votante esté registrado en el censo antes de que pueda emitir un voto.
- Asegurarse de que cada votante pueda emitir un único voto por proceso electoral.
- Proveer herramientas para registrar, eliminar y consultar votos.

2.2. Componentes

Modelo de datos (`models.py`)

El proyecto utiliza dos modelos llamados **censo** y **voto** los cuales se describen a continuación. La figura 1 muestra el diagrama relacional de la aplicación.

- **Modelo Censo:**

- Representa a las personas habilitadas para votar.
- Principales campos:
 - `numeroDNI`: Identificador único del votante (clave primaria).
 - `nombre`: Nombre completo del votante.
 - `fechaNacimiento`: Fecha de nacimiento del votante.
 - `anioCenso`: Año de registro del votante en el censo.
 - `codigoAutorizacion`: Código único de autorización.

- **Modelo Voto:**

- Representa un voto emitido por un votante.
- Principales campos:
 - `id`: identificador (clave primaria)
 - `idCircunscripcion`: Circunscripción donde se emite el voto.

- **idMesaElectoral**: Mesa electoral correspondiente.
- **idProcesoElectoral**: Identificador del proceso electoral (clave extranjera).
- **nombreCandidatoVotado**: Candidato al que se dirige el voto.
- **censo**: Relación con el modelo **Censo**, representando al votante.
- **marcaTiempo**: Fecha y hora del registro del voto.
- **codigoRespuesta**: Estado del voto (e.g., éxito o error).
- Restricciones implementadas en la base de datos:
 - Un votante (**censo**) no se puede emitir más de un voto por proceso electoral (**idProcesoElectoral**).

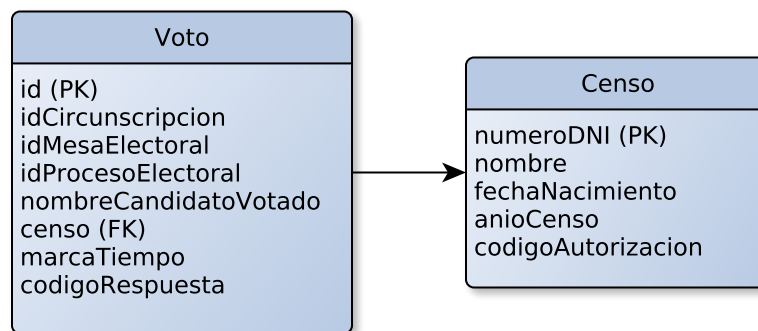


Figura 1: Diagrama relacional del modelo de datos. **PK** y **FK** significan clave primaria y clave extranjera respectivamente.

Lógica de la aplicación (views.py y votoDB.py)

A continuación describimos las funciones que contienen la lógica de la aplicación base entregada al alumno.

▪ **aportarinfo_censo**:

- Permite autenticar a un usuario comprobando su registro en el censo.

- Almacena el **numeroDNI** del votante en una variable de sesión para su uso posterior en el método **aportarinfo_voto**.
- **aportarinfo_voto:**
 - Maneja el registro de votos.
 - Verifica que el votante esté autenticado mediante el **numeroDNI** almacenado en una variable de sesión.
 - Registra el voto utilizando la función **registrar_voto**.
- **testbd:**
 - Integra la funcionalidad de validar un registro en el censo (**aportarinfo_censo**) y el registro de votos (**aportarinfo_voto**).
 - Verifica que el votante esté en el censo antes de registrar su voto.
- **delvoto:**
 - Permite eliminar un voto identificado por un ID único.
- **getvotos:**
 - Recupera los votos registrados en un proceso electoral específico.

Las funciones descritas se implementan en **views.py** y están disponibles al público. Estas son responsables de procesar los formularios con los datos recibidos y generar las páginas web que se devuelven al navegador. Internamente, delegan en diversos métodos definidos en **votoDB.py**, los cuales se encargan del acceso a la base de datos.

2.3. Flujo General

1. **Autenticación:** El usuario ingresa su **numeroDNI** y otros datos mediante un formulario. Si está registrado en el censo, su DNI se guarda en una variable de sesión.

2. **Registro de Voto:** El usuario selecciona un candidato y registra su voto. El sistema verifica que no existan duplicados en el proceso electoral. Esto es, que el mismo votante no ha emitidos dos votos en la misma consulta.
3. **Gestión Adicional:** Los votos pueden ser eliminados o consultados para procesos electorales específicos.

3. Instrucciones para Configuración y Despliegue

En esta sección describimos como desplegar el proyecto **P1-base** en el entorno de los laboratorios. Tal y como se muestra en la figura 2 serán necesarias dos máquinas virtuales a las que llamaremos **VM1** y **VM2** respectivamente. En **VM1** se desplegará la base de datos y en **VM2** la aplicación de *Django*. Si la dirección IP del ordenador **host** es XX.YY.ZZ.II la base de datos debería estar accesible en XX.YY.ZZ.II:15432 y el servidor de *Django* en XX.YY.ZZ.II:28000. En este documento llamaremos **host** al ordenador donde se ejecutan las máquinas virtuales. Ambas máquinas virtuales son accesibles por ssh en las direcciones XX.YY.ZZ.II:12022 (**VM1**) y XX.YY.ZZ.II:22022 (**VM2**). (Recuérdese que diversos puertos de la máquina **host** han sido redirigidos (port forwarding) a diversos servicios ejecutados en las máquinas virtuales).

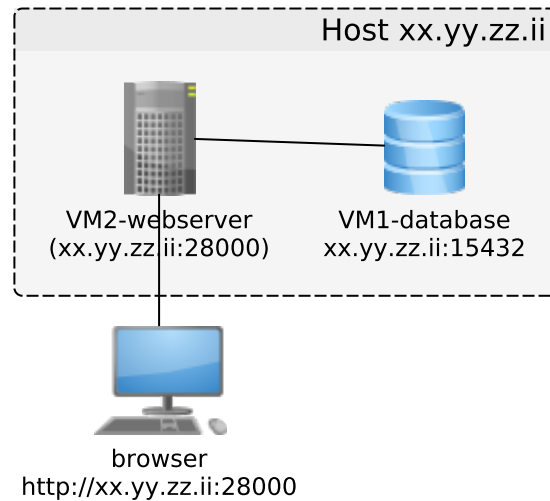


Figura 2: Descripción esquemática del despliegue del proyecto **P1-base**.

3.1. Requisitos Previos

Aunque queda a vuestra elección, recomendamos desarrollar y desplegar el proyecto localmente (en la máquina **host**) usando el servidor de bases de datos instalado en **VM1**. Una vez que la aplicación web funcione se desplegaría en su totalidad en las máquinas virtuales. En las máquinas virtuales ya se encuentra instalado el entorno pero ese no es el caso para el ordenador principal (**host**). El primer paso a realizar sería clonar el proyecto **P1-base** y conseguir que funcione localmente.

Antes de ejecutar la aplicación localmente, asegúrese de cumplir con los siguientes requisitos en el **host**:

- Tener instalado **Python** (versión 3.12).
- Tener un entorno virtual configurado (**venv** o equivalente).
- Haber instalado las dependencias descritas en el fichero **requirements.txt**.

3.2. Estructura de directorios

En la siguiente sección os solicitaremos que realicéis una copia (`git clone`) del repositorio `https://github.com/rmarabini/si2_alumnos.git` el cual contiene el proyecto **P1-base** junto con una colección de directorios donde se irán implementando el resto de las prácticas. La estructura de directorios es:

```
si2_alumnos
|
+-- P1-base
|   +-- env
|   +-- manage.py
|   +-- requirements.txt
|   +-- votoApp
|   +-- votoSite
+-- P1-rpc-client
|   +-- votoAppRPCClient
+-- P1-rpc-server
|   +-- votoAppRPCServer
+-- P1-ws-client
|   +-- votoAppWSClient
+-- P1-ws-server
    +-- votoAppWSServer
```

El proyecto **P1-base** sólo necesita pequeños ajustes pero los otros cuatro proyectos restantes (p.e. **P1-ws-server**) deben ser implementados en su totalidad.

3.3. Creación y Configuración del proyecto

1. Duplica el proyecto en *github*:

```
# Clonar el repositorio original
git clone git@github.com:rmarabini/si2_alumnos.git
```

```
# Cambiar al directorio del repositorio clonado
cd si2_alumnos

# Renombrar el repositorio remoto original
git remote rename origin old-origin

# crea un nuevo repositorio PRIVADO en github
# supongamos que se llama tu_nuevo_repositorio

# Agregar el nuevo repositorio como remoto
git remote add origin git@github.com:tu_usuario/tu_nuevo_repositorio.git

# Empujar todas las ramas al nuevo repositorio
git push -u origin --all
```

2. Configurar el Entorno Virtual en el host.

```
python3 -m venv venv
source venv/bin/activate    # En Windows: venv\Scripts\activate
pip install -r P1-base/requirements.txt
```

3. Configurar la Base de Datos

- El código asume que existe un fichero llamado **env** en la raíz del proyecto en el que se ha definido la variable `DATABASE_SERVER_URL`:

```
1 # env
2 DATABASE_SERVER_URL =
    ↪ 'postgres://alumnodb:alumnodb@localhost:XXXX/'
    ↪ voto'
```

Donde XXXX será 5432 si la base de datos se encuentra en el ordenador **host** y 15432 si la base de datos se encuentra en **VM1**.

4. Ejecutar Migraciones

```
# en el directorio P1-base
python manage.py makemigrations
python manage.py migrate
```

Nota 2: Si usáis el *PostgresSQL* de los ordenadores de los laboratorios la base de datos puede que ya exista, si este es el caso os recomendamos borrarla y volver a crearla antes de ejecutar “migrate”.

5. Crear un Superusuario (Opcional)


```
python manage.py createsuperuser
```

El superusuario os da acceso al interfaz de administracion. Si queréis usarlo no os olvidéis de inicializar el fichero **admin.py** adecuadamente.

Listado 1: Listado del fichero **admin.py** mostrando como incorporar los modelos **voto** y **censo** al sistema de administración de *Django*

```
1 # admin.py
2 from django.contrib import admin
3 from .models import Censo, Voto
4
5 # Register Censo model
6 @admin.register(Censo)
7 class CensoAdmin(admin.ModelAdmin):
8     pass
9
10 # Register Voto model
11 @admin.register(Voto)
```

```
12 class VotoAdmin(admin.ModelAdmin):  
13     pass
```

Código disponible en este enlace: 

6. **Poblad la base de datos** con los datos de censo.

```
python manage.py populate
```

7. **Probar Localmente la Aplicación**

```
python manage.py runserver
```

Accede a `http://127.0.0.1:8000/votoApp/nombre_endpoint` para verificar que la aplicación funcione correctamente. Los **endpoints** definidos se describen a continuación. Recordad que en los ordenadores de los laboratorios el puerto 8000 no está disponible y debéis usar otro puerto como pueda ser el 8001 (`python manage.py runserver 8001`). Igualmente recordad que en los ordenadores de los laboratorios *PostgreSQL* no está arrancado por defecto y que debéis ejecutar la orden `sudo systemctl restart postgresql` para arrancarlo. (NOTA: un elevado porcentaje de los alumnos intenta ejecutar la orden usando “start”, en lugar de “restart” lo cual no funciona).

3.4. Endpoints

El fichero `urls.py` define los **endpoints** que permiten interactuar con las funcionalidades principales de la aplicación. A continuación, se detallan algunos de ellos:

1. **censo/**: utilizado para manejar la validación de los usuarios en el censo. Es funcionalmente equivalente al endpoint raíz (/).
2. **voto/**: Permite a los usuarios registrados (cuyo DNI ha sido validado previamente en el censo) registrar un voto para un candidato en un proceso electoral específico.

3. **testbd/**: Permite realizar pruebas en la base de datos. Combina funcionalidades de verificación de votantes en el censo y de votos, facilitando la integración de ambas operaciones en un entorno de pruebas. Igualmente permite borrar y listar los votos asociados a un cierto proceso electoral.

Ejercicio 1: incluya en la memoria de la practica pruebas de que se ha desplegado la aplicación web localmente y de que esta funciona. Al menos se debe incluir una copia del contenido del fichero **env** así como una captura de pantalla en la que se muestre el resultado de interaccionar con cada uno de los “endpoints”. Las capturas deben mostrar el URL al que se conecta el navegador.

Ejercicio 2: ejecuta los test proporcionados con el proyecto base y comprueba que no devuelven errores. Adjunta en la memoria una captura de pantalla en la que se muestre el resultado de ejecutar los test (`python manage.py test`). Los test proporcionados deben tomarse como requisitos extras del sistema.

4. Despliegue de la aplicación en las máquinas virtuales

Como se comentó en la introducción de la práctica deseamos desplegar la aplicación en la máquina virtual **VM2** manteniendo la base de datos en **VM1** (véase figura 2).

4.1. Requisitos Previos

En la máquina host

Antes de proceder, asegúrate de tener:

- acceso SSH a las máquinas virtuales en los puertos 12022 (**VM1**) y 22022 (**VM2**).
- *git* instalado en la máquina local.

En la máquina VM1

- crea la base de datos **voto**.
- inicialízala desde el **host** modificando la variable **DATABASE_SERVER_URL** en el fichero **env** y ejecutando `python manage.py migrate`

En la máquina VM2

- crea un repositorio *git* tal y como se describe en el apéndice A.
- configura la máquina para que la aplicación web se sirva usando **gunicorn** y para que este se relance cada vez que se modifique el código (véase apéndice B).
- actualiza el fichero **env** para que acceda a la base de datos situada en **VM1**. Recuerda que por razones de seguridad el fichero **env** no debe añadirse al repositorio.
- añade los siguientes comandos al fichero `/home/si2/repo/p1base.git/hooks/post-receive`

```
1 python $TARGET/manage.py migrate
2 python $TARGET/manage.py collectstatic --noinput
```

- **python \$TARGET/manage.py migrate**: Aplica los cambios pendientes en la base de datos basándose en las definiciones de los modelos, asegurando que su estructura esté sincronizada con el código del proyecto.
- **python \$TARGET/manage.py collectstatic --noinput**: Recopila los archivos estáticos (CSS, JavaScript, imágenes) en un único directorio central (**STATIC_ROOT**) para ser servidos en entornos de producción.

Ejercicio 3: incluya en la memoria de la practica pruebas de que se ha desplegado la aplicación web **P1-base** usando las máquinas virtuales y de que esta funciona. Al menos se debe incluir una copia del contenido del fichero **env** así como una captura de pantalla en la que se muestre el resultado de registrar un voto. Esta última debe mostrar claramente el URL al que se conecta el navegador.

Ejercicio 4: El script de python incluido en el apéndice C lee 1000 entradas de la base de datos “voto” una a una e imprime el tiempo de lectura. Ejecútalo desde el ordenador **host** y reporta en la memoria de la práctica los siguientes casos:

1. la base de datos está en la máquina virtual **VM1**.
2. la base de datos no esta en la red local. Crea un base de datos llamada **voto** en <http://www.neon.tech> y ejecuta el script contra esa base de datos
3. no se accede a la base de datos directamente. En Django, el mapeo de clases a tablas se implementa mediante el ORM (Object-Relational Mapping), re-escribe el script para que los votos se lean usando el sistema de modelos de *Django*. Por ejemplo `'SELECT * FROM censo WHERE "numeroDNI" = %s'` debería ser `Voto.objects.get(pk=%s)`. En este nuevo script no debe aparecer código SQL).

Repite cada medida 7 veces y reporta el valor medio de las mismas así como su desviación estándar. Además de los tiempos de lectura incluye en la memoria el script reescrito, el contenido del fichero **env** en cada caso así como un comentario sobre los resultados.

Question 1: ¿En el ejercicio anterior 7 repeticiones de cada medida proporcionan una estimación fiable? ¿Porqué?.

Ejercicio 5: ejecuta los test proporcionados con el proyecto **P1-base** y comprueba que no devuelven errores. Adjunta en la memoria una captura de pantalla en la que se muestre el resultado de ejecutar los test incluyendo el comando utilizado para lanzarlos. En está ejecución la base de datos debe estar en **VM1** y la aplicación web en **VM2**.

5. Introducción a la API REST

Una **API REST** (*Representational State Transfer*) es una interfaz que permite la comunicación entre aplicaciones mediante el intercambio de datos, siguiendo los prin-

cipios y convenciones del protocolo HTTP. REST es una arquitectura ampliamente adoptada debido a su simplicidad, escalabilidad y compatibilidad con diferentes sistemas.

5.1. Características de una API REST

- **Recursos:** Los datos se exponen como recursos accesibles a través de URLs únicas.
- **Operaciones estándar:** Se utilizan métodos HTTP como `GET` (listado de modelos y detalles de un modelo concreto), `POST` (creación de un modelo), `PUT` (actualización de un modelo) y `DELETE` (borrado de un modelo) para interactuar con los recursos.
- **Formato de datos:** Los datos suelen intercambiarse usando JSON o formatos equivalentes.
- **Stateless:** Cada solicitud a la API es independiente, lo que significa que no se almacena el estado de las interacciones en el servidor.

En esta sección implementaremos primero el servidor API REST y posteriormente el cliente que consumirá este API.

5.2. Implementación del API REST usando *Django*

Primero describiremos la funcionalidad de la API REST y tras acabar con la descripción de los diferentes endpoints ofreceremos algunas ideas sobre como implementarlos. La API REST solicitada se desarrollará usando el **Django REST Framework** (DRF) con los siguientes “endpoints” principales.

- Validación de un usuario en el censo electoral.
- Emisión de un voto en procesos electorales específicos.
- Consulta (listado de votos emitidos para un proceso electoral dado) y eliminación de votos por ID.

Los “endpoints” se definen mediante el fichero `urls.py`. Para que los test funcionen correctamente se debe conservar las etiquetas “name=’xxxx’ ”

```
1  urlpatterns = [  
2      # check if person is in "censo"  
3      path('censo/', CensoView.as_view(), name='censo'),  
4      # create "voto"  
5      path('voto/', VotoView.as_view(), name='voto'),  
6      # get list of "votos" associated with a given  
7      # ↪ idProcesoElectoral  
8      path('procesoelectoral/<str:idProcesoElectoral>',  
9          ProcesoElectoralView.as_view(),  
10         # ↪ name='procesoelectoral '),  
11         # delete "voto" with id id_voto  
12         path('voto/<str:id_voto>', VotoView.as_view(),  
13             # ↪ name='voto '),  
14     ]
```

Definid el API de forma que los URLs a los que se conectará el navegador sean del tipo `http://<host_name>/restapiserver/<endpoint>/`

- la validación de la existencia del votante en el censo se realiza en el “end-point” `censo` al cual se accederá mediante una llamada de tipo `POST` que contendrá la información del votante. La llamada debe devolver el estado **HTTP_200_OK** en caso de éxito (el votante existe) o **HTTP_404_NOT_FOUND** en caso de que el votante no exista en el censo. Igualmente debe devolver un diccionario con un mensaje bien sea de éxito o de fracaso. Por ejemplo

```
1  return Response(  
2      {'message': 'Datos no encontrados en Censo.'},  
3      status=status.HTTP_404_NOT_FOUND)
```

Nótese que este API es stateless y no crea variables de sesión en la que almacenar el identificador de censo.

- de forma similar la emisión del voto se realiza en el endpoint `voto` al cual se accederá mediante una llamada de tipo `POST` que contendrá la información del voto. Esta información debe contener el identificador de censo. El API devolverá el estado **HTTP_200_OK** en caso de éxito, **HTTP_404_NOT_FOUND** si el votante no existe en el censo o **HTTP_400_BAD_REQUEST** en el resto de los casos. Igualmente devolverá el voto registrado en caso de éxito y un mensaje de error en el resto de los casos. Por ejemplo:

```
1 if voto is None:
2     return Response({'message': 'Entry not found in
    ↪ Censo.'},
3                     status=status.HTTP_404_NOT_FOUND)
4 voto_dict = model_to_dict(voto)
5 return Response(voto_dict, status=status.HTTP_200_OK)
```

- Para implementar las consultas (listado de votos de un proceso electoral) se utilizará una llamada `GET` a `procesoelectoral` en el que se pasará el identificador del proceso electoral para el cual se solicita el listado de votos. El API devolverá el estado **HTTP_200_OK** en caso de existir votos asociados al proceso electoral solicitado o **HTTP_404_NOT_FOUND** si no existen. Igualmente devolverá el listado de votos en caso de éxito y un mensaje de error en el resto de los casos.
- Finalmente para eliminar un voto se realizará una llamada `DELETE` a `voto` pasando el identificador del voto que se desea borrar. El API devolverá el estado **HTTP_200_OK** en caso de haberse borrado el voto y **HTTP_404_NOT_FOUND** si el voto no existe y por tanto no es posible borrarlo.

Se solicita que implementéis el API REST usando clases que hereden de `APIView` tal y como se describe en <https://davidcasr.medium.com/construir-un-api-rest-con-django-rest-framework-y-apiview-5ea4b2823307> en el paso quinto y siguientes.

El primer paso para crear esta nueva versión de nuestro web site será crear un nuevo proyecto, partiendo del proyecto P1-base. Para ello, copiaremos el directorio

P1-base a **P1-ws-server** con el comando:

```
cp -r P1-base/* P1-ws-server/*
```

El siguiente paso, será renombrar la aplicación, que en P1-base se llamaba votoApp a votoAppWSServer. Para ello, ejecutaremos los comandos:

```
cd P1-ws-server
mv votoApp votoAppWSServer
find ./ -type f -exec sed -i "s/votoApp/votoAppWSServer/g" {} \;
```

El siguiente paso será eliminar los ficheros no necesarios en la parte del servidor. En particular, aquellos relacionados con la interfaz web. Para ello ejecutaremos estos comandos:

```
rm votoAppWSServer/forms.py
# views.py necesita ser reescrito
rm votoAppWSServer/views.py
rm -rf votoAppWSServer/templates
# sobrescribe test_views.py con el fichero test_views.py
# existente en el repositorio para este proyecto.
rm votoAppWSServer/tests_views.py
rm votoAppWSServer/tests_models.py
```

El fichero `views.py` necesita ser reescrito puesto que ahora algunas funciones se crean de forma automática y el resto en la versión original devuelve código HTML y ahora debe devolver JSON. Una vez que esté implementado el API y el servidor esté arrancado al conectaros al mismo (p.e. `http://<hostname>:8000/restapiserver/voto/`) usando el navegador deberíais obtener un resultado similar a:

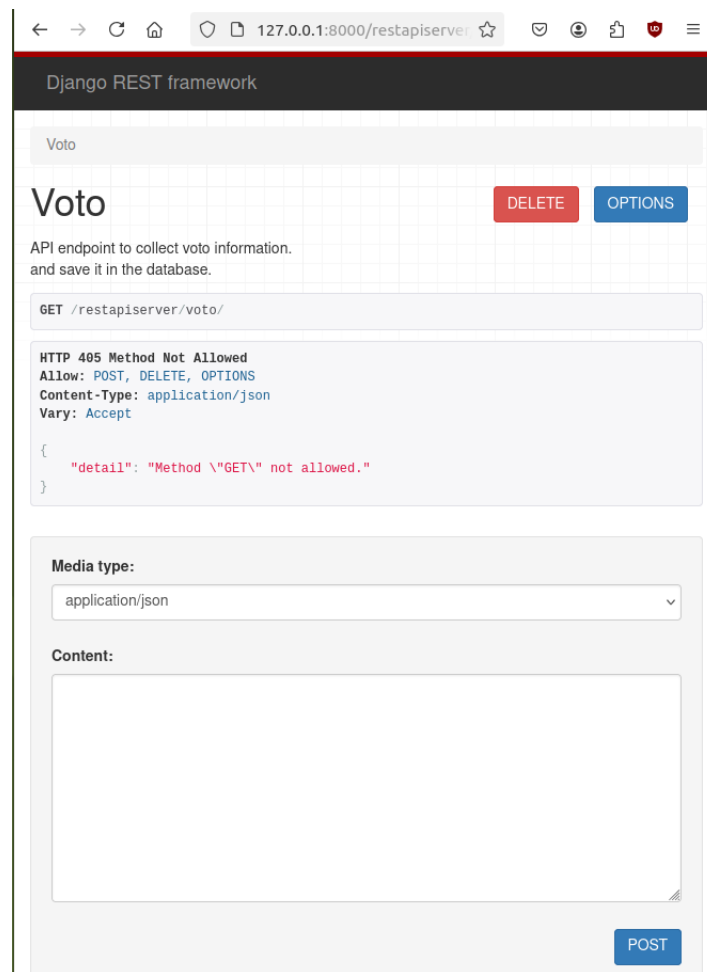


Figura 3: Pantalla mostrando el resultado de conectarse al URL `http://<hostname>:8000/restapiserver/voto/` donde se ofrecen conexiones de tipo `POST` (crear) o `DELETE` (borrar) y se deniegan las de tipo `GET` (ver detalles de los votos). En este ejemplo el servidor está siendo ejecutado en el ordenador **host** sirviendo en el puerto 8000.

Como en el caso anterior despliega el servidor en **VM2** y mantén la base de datos en **VM1**.

Ejercicio 6: ejecuta los test proporcionados con el proyecto **P1-ws-server** aplicación **votoAppWSServer** y comprueba que no devuelven errores. Adjunta en la memoria una captura de pantalla en la que se muestre el resultado de ejecutar los test junto con el comando utilizado. En esta ejecución la base de datos debe estar en **VM1** y la aplicación web en **VM2**.

Ejercicio 7: conéctate a los diferentes endpoints usando el navegador (`http://<hostname>:28000/restapiserver/xxxx/`) y muestra las pantallas resultantes de usar cada uno de ellos. Suministra al sistema datos válidos e inválidos. Junto a cada imagen debes incluir: (1) el URL al que te conectas, (2) el método usado (POST/GET/DELETE) y (3) en caso de usar el método POST los datos enviados en la petición

5.3. Implementación del cliente del API REST usando *Django*

El siguiente paso será implementar el cliente que consuma nuestro API REST. Para implementar el sistema entero (cliente-servidor) usaremos 2 PCs y un total de tres máquinas virtuales tal y como se describe en la figura 4.

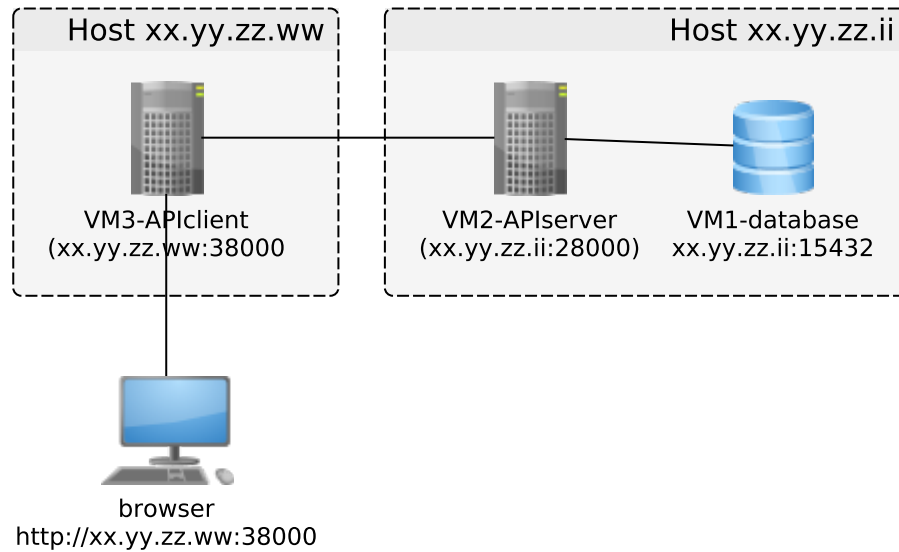


Figura 4: Descripción esquemática del despliegue de las aplicaciones **P1-ws-server** y **P1-ws-client P1-base**.

Crearemos un nuevo proyecto llamado **P1-ws-client** el cual contendrá la aplicación **votoAppWSClient** la cual usará los mismos modelos y el mismo fichero **urls.py** que se definió en el proyecto **P1-base**. En esta implementación el navegador llamará al proyecto **P1-ws-client**, este proyecto sólo puede acceder a la base de datos haciendo peticiones al API REST implementado en el proyecto **P1-ws-server**. Una manera sencilla de implementar este proyecto será, como en el caso anterior, duplicar el proyecto **P1-base** como **P1-ws-client**. A continuación de borrarán los ficheros:

```
rm -rf votoAppWSClient/models.py
rm -rf votoAppWSClient/migrations
rm -rf votoAppWSClient/management
rm -rf votoAppWSClient/tests_models.py
# sobrescribe test_views.py con el fichero test_views.py
# existente en el repositorio para este proyecto.
```

A continuación habrá que modificar el fichero **votoDB.py** para que llame al API REST en lugar de acceder a la base de datos directamente a través de los

modelos de *Django*. Para conseguir este objetivo hay que realizar llamadas de tipo `response = requests.post(api_url, json=voto_dict)` en lugar de `voto = Voto.objects.create(**voto_dict)`. En este ejemplo se realiza una conexión al URL `api_url` al cual se le pasa un diccionario con la información relacionado con el voto `voto_dict` que se desea crear.

Igualmente es necesario añadir una nueva variable llamada `RESTAPIBASEURL` conteniendo la dirección en la que el API REST está escuchado (p.e. `http://localhost:8000/restapiserver/`) al archivo `env`. Esta variable debe ser leída por el fichero `settings.py` de forma que sea accesible en el resto del proyecto.

Nótese que tal y como se hace en el proyecto `P1-base` el método que se encarga de verificar la inclusión del votante en el censo para el proyecto `P1-ws-client` debe almacenar en una variable de sesión el identificador del censo que será utilizado posteriormente para almacenar el voto.

A la hora de implementar los proyectos `P1-ws-client` y `P1-ws-server` reutilizar los directorios del mismo nombre existentes en el repositorio `si2_alumnos`. Para crear un proyecto de *Django* dentro de un directorio ya existente usad el comando `cd P1-ws-xxxxx; django-admin startproject P1-ws-xxxxx` y para crear una aplicación `django-admin startapp votoAppWSxxxxxx votoAppWSxxxxx`.

Ejercicio 8: ejecuta los test proporcionados con el proyecto **P1-ws-client** aplicación **votoAppWSclient** y comprueba que no devuelven errores. Adjunta en la memoria una captura de pantalla en la que se muestre el resultado de ejecutar los test. En esta ejecución la base de datos debe estar en **VM1** y la aplicación web en **VM2**.

Ejercicio 9: incluya en la memoria de la practica pruebas de que se ha desplegado las aplicaciones web `P1-ws-client` y `P1-ws-server` y de que estas funcionan. Al menos se debe incluir una copia del contenido del fichero `env` así como una captura de pantalla en la que se muestre el resultado de interaccionar con cada uno de los “endpoints” del cliente. Las capturas deben mostrar el URL al que se conecta el navegador.

Material a entregar al finalizar la práctica: se debe subir a *Moodle* un fichero zip que contenga la memoria (en formato pdf) respondiendo a todos los ejercicios y cuestiones planteadas en esta guía junto con el código utilizada para implementar los diferentes proyectos. Se considerará que el código funciona si lo hace en los ordenadores del laboratorio. El código entregado en *Moodle* será el evaluado, bajo ninguna condición se evaluará el código existente en los diversos repositorios.

6. Criterios de evaluación

Para aprobar es necesario haber implementado el API REST y satisfacer los ejercicios 6 y 7. En caso de satisfacer este requerimiento se optiene una nota de 5 a la que se sumarán las cantidades siguientes:

Ejercicio	Puntuacion
-----------	------------

1	0.5
---	-----

2	0.5
---	-----

3	0.5
---	-----

4	0.5
---	-----

5	0.5
---	-----

8	1.0
---	-----

9	1.0
---	-----

Question

1	0.5
---	-----

A. Configuración de un Repositorio Git Remoto Usando SSH

Este apéndice explica cómo crear un repositorio Git remoto en otro equipo mediante SSH y habilitar el uso de `git push`.

A.1. Crear un Repositorio Git Bare en el Equipo Remoto VM2

1. Conéctate al equipo remoto mediante SSH:

```
1 # host
2 # el puerto (22022) depende de la maquina a la que
  ↪ desees acceder
3 ssh si2@localhost -p 22022
```

2. Crea un repositorio bare (sin árbol de trabajo):

```
1 # MV2
2 mkdir -p /home/si2/repo
3 cd /home/si2/repo
4 git init --bare p1base.git -b main
```

El último comando debe devolver el mensaje `Initialized empty Git repository in /home/si2/repo/p1base.git/`

A.2. Conectar el Repositorio Local al Remoto

1. Navega al directorio de tu proyecto en el **host** el cual ya es un repositorio de git.
2. Agrega el repositorio remoto **p1base.git** al listado de repositorios remotos con el alias **vm2**:

```
1 # host
2 git remote add vm2
   ↪ ssh://si2@localhost:22022/~/.repo/p1base.git
```

3. Verifica que el remoto se agregó correctamente:

```
1 # host
2 git remote -v
```

A.3. Realizar un Push Desde el Repositorio Local

1. Los cambios se envían al repositorio remoto con el comando:

```
1 git push -u vm2 main
2                                     # lee el siguiente apartado
   ↪ antes de ejecutar este
   ↪ comando
3                                     # -u establece esta
   ↪ configuracion por defecto
   ↪ de forma que
4                                     # de ahora en adelante no es
   ↪ necesario teclear "-u vm2"
   ↪ main"
```

A.4. Actualizar Archivos Automáticamente en el Equipo Remoto

Cuando ejecutas el comando `git push`, estás transfiriendo los **commits** desde el repositorio local hacia el repositorio remoto. Sin embargo, esto **no actualiza directamente los archivos de trabajo en el servidor remoto**, únicamente la base de datos de *git* se actualiza. En algunos casos, como cuando usas *git* para desplegar un sitio web, el repositorio remoto puede estar asociado a un área de

trabajo. Para que los archivos de trabajo reflejen los cambios más recientes, deberás configurar el repositorio remote como se describe a continuación.

1. En **VM2** crea o edita el archivo **post-receive** en el repositorio bare del equipo remoto). Como ya sospecharás el fichero **post-receive** se ejecuta cada vez que la máquina recibe un **git push**:

```
1 #VM2
2 nano /home/si2/repo/p1base.git/hooks/post-receive
```

2. Agrega las siguientes líneas al archivo:

```
1 #!/bin/bash
2 GIT_WORK_TREE=/home/si2/repo/p1base git checkout -f
```

Si el directorio `/home/si2/repo/p1base` no existe créalo. Esta será la ubicación del directorio de trabajo.

3. Haz que el archivo sea ejecutable:

```
1 chmod +x /home/si2/repo/p1base.git/hooks/post-receive
```

4. Finalmente en el **host** ya puedes ejecutar el comando **git push**.

```
1 # host
2 git push -u vm2 main
```

y el código se actualizará con la última versión.

B. Configuración de gunicorn

Cuando se despliega la aplicación web en modo “producción” no se utiliza el servidor de desarrollo que acompaña a *Django* sino un contenedor de aplicaciones como **gunicorn**.

Para que **gunicorn** se relance cada vez que se despliega código nuevo debemos modificar el archivo **post-receive** en **VM2** y añadir la línea

```
1 # Reiniciar Gunicorn
2 sudo systemctl restart gunicorn
```

Para que el comando `systemctl restart gunicorn` funcione correctamente en el contexto de un “hook” de *git*, es importante configurar y asegurarte de que el servicio **gunicorn** esté gestionado adecuadamente por **systemd**. Aquí está una guía completa para configurarlo.

B.1. Crear un Archivo de Configuración para el Servicio Gunicorn

Crea un archivo de configuración para **gunicorn** en el directorio `/etc/systemd/system/` de **VM2**:

```
1 sudo nano /etc/systemd/system/gunicorn.service
```

Agrega la configuración del servicio. Aquí hay un ejemplo para una aplicación basada en *Django*:

```
[Unit]
Description=Gunicorn WSGI Application Server
After=network.target

[Service]
User=si2
Group=si2
WorkingDirectory=/home/si2/repo/<nombre-del-proyecto>
Environment="PATH=/home/si2/venv/bin"
ExecStart=/home/si2/venv/bin/gunicorn \
    --workers 1 \
    --bind 0.0.0.0:8000 \
    votoSite.wsgi:application
ExecReload=/bin/kill -s HUP $MAINPID
```

```
KillMode=mixed
TimeoutStopSec=5
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

B.2. Explicación de los Parámetros Importantes

- **User y Group:** Define al usuario y grupo que ejecutarán **gunicorn**. Normalmente se utiliza **www-data** aunque aquí usemos **si2**.
- **WorkingDirectory:** Ruta al directorio raíz de tu proyecto
- **ExecStart:** El comando para iniciar Gunicorn. Incluye:
 - Ruta al binario de **gunicorn** dentro del entorno virtual.
 - **-workers:** Número de procesos de trabajadores.
 - **-bind 0.0.0.0:8000:** escucha en el puerto 8000 y admite conexiones desde cualquier host.
 - **votoSite.wsgi:application:** Especifica la entrada WSGI de tu aplicación.

B.3. Recargar systemd y Activar gunicorn

Después de crear el archivo, recarga **systemd** para aplicar los cambios:

```
sudo systemctl daemon-reload
```

Habilita **gunicorn** para que se inicie automáticamente con el sistema:

```
sudo systemctl enable gunicorn
```

Inicia el servicio **gunicorn**:

```
sudo systemctl start gunicorn
```

Verifica que **gunicorn** esté funcionando:

```
sudo systemctl status gunicorn
```

C. Script que lee la tabla voto

Script en Python que leer las primeras 1000 entradas de una tabla llamada censo en una base de datos llamada voto y muestra el tiempo invertido en la lectura:

Listado 2: código del script `read_1000_entries_from_db.py`

```
1 # connect to the database, read the first 1000 entries
2 # then perform 1000 queries retrieving each one of the
   ↪ entries
3 # one by one. Measure the time requiered for the 1000
   ↪ queries
4
5 import psycopg2
6 import time
7
8 # Configuracion de la base de datos
9 db_config = {
10     'dbname': 'voto', # Nombre de la base de datos
11     'user': 'alumnodb', # Reemplaza con tu usuario de
   ↪ PostgreSQL
12     'password': 'alumnodb', # Reemplaza con tu
   ↪ contraseña
13     'host': 'localhost', # Cambia si el host es diferente
14     'port': 5432, # Cambia si tu puerto es diferente
15 }
16
17 try:
18     # Conexion a la base de datos
19     conn = psycopg2.connect(**db_config)
20     cursor = conn.cursor()
```



```
21
22     # Leer las primeras 1000 entradas de la tabla censo
23     query_fetch_1000 = "SELECT * FROM censo LIMIT 1000"
24     cursor.execute(query_fetch_1000)
25     rows = cursor.fetchall()
26
27     # Preparar para las búsquedas individuales
28     search_query = 'SELECT * FROM censo WHERE "numeroDNI"
    ↪ = %s' # Asumiendo que hay una columna 'id'
    ↪ para identificar las filas
29
30     # Medir el tiempo de inicio
31     start_time = time.time()
32
33     # Realizar búsquedas una a una
34     for row in rows:
35         id_value = row[0] # Suponiendo que la primera
    ↪ columna es el ID
36         cursor.execute(search_query, (id_value,))
37         cursor.fetchone() # Obtener la fila encontrada
38
39     # Medir el tiempo de finalización
40     end_time = time.time()
41
42     # Mostrar los resultados
43     print(f"Tiempo invertido en buscar las 1000 entradas
    ↪ una a una: {end_time - start_time:.6f}
    ↪ segundos")
44
45 except Exception as e:
46     print(f"Error: {e}")
47
48 finally:
49     # Cerrar el cursor y la conexión
```

```
50     if 'cursor' in locals():
51         cursor.close()
52     if 'conn' in locals():
53         conn.close()
```

Código disponible en este enlace: 