Universidad Autónoma de Madrid

Departamento de Informática

# Proyecto de Sistemas Informáticos
# Práctica - 3

Roberto Marabini

# Índice

# 1. Objetivos

En esta práctica se enuncia el proyecto a implementar, se crea el modelo de datos y se realiza una implementación básica del mismo. El problema que queremos resolver es, dada una asignatura, como gestionar la elección de grupos de prácticas.

# 2. Creación de Grupos de Prácticas

A muy grandes rasgos, la aplicación que deseamos crear deberá:

1. Cargar la información relacionada con el curso: alumnos, profesores, grupos de teoría, grupos de prácticas, restricciones de horario a la hora de elegir grupos, etc.

2. Permitir a los alumnos formar parejas (los alumnos pertenecientes a una misma pareja serán asignados al mismo grupo) y solicitar la convalidación de las practicas del año pasado.

3. Permitir a los alumnos seleccionar grupos siguiendo la estrategia FIFO (asignación a los grupos de prácticas por orden de solicitud)

## 2.1. Requerimientos

Antes de hacer publica la aplicación será necesario inicializarla con la información relativa a los alumnos, profesores, grupos, etc. Para ello se creará un script llamado `populate.py`, cuya implementación se describirá en detalle, el cual realiza las siguientes funciones:

- Borrar todos los objetos creados en la base de datos.
- Inicializar la base de datos con la información relaciona con la asignatura EDAT. Tras la inicialización, la base deberá contener información sobre:
  - alumnos
  - profesores

  - notas obtenidas el año pasado por los alumnos repetidores
  - grupos de teoría
  - grupos de prácticas
  - dado un grupo de teoría que grupos de prácticas son elegibles
  - restricciones temporales, esto es, cuando se puede acceder a la distinta funcionalidad ofrecida por la aplicación

Una vez poblada la aplicación se abrirá al público

- Los usuarios se identifican usando un nombre de usuario y una clave.
- Los usuarios no identificados sólo podrán acceder a la página de login y a una "home page" donde se describe el sistema.
- El nombre de usuario y la clave serán el NIE y el DNI del estudiante respectivamente.
- Varios usuarios deben poder conectarse simultáneamente desde navegadores diferentes.
- Los usuarios deberán ser capaces de solicitar la convalidación de las practicas y se les concederá en caso de que superen unos criterios preestablecidos.
- Los usuarios deben ser capaces de agruparse en parejas.
- Los usuarios deben ser capaces de cancelar una pareja.
- Los usuarios deben ser capaces de solicitar grupos de practicas. La asignación se realizará de forma inmediata.

Recuerda que tu código debe satisfacer los criterios de estilo marcados por *Flake8*.

# 3. Modelo de datos

El modelo de datos que dará soporte a la aplicación seguirá el esquema ORM (Object Relational Mapping) de *Django*. A continuación, véase la figura 1, se muestra el diseño mínimo que debes usar en tu solución. A partir de este diseño, puedes añadir cualquier entidad o atributo que consideres necesario, pero **no eliminar** ninguno de los propuestos:

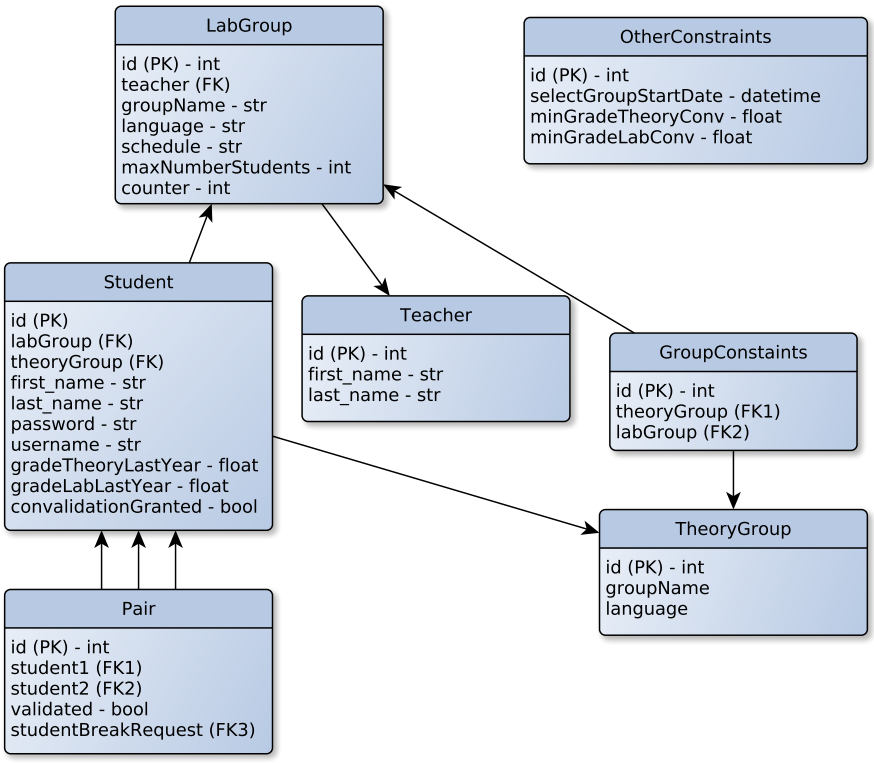


Figura 1: Esquema relacional de la base de datos.

En la figura 1 se ha utilizado la nomenclatura siguiente. Cada modelo esta contenido en una caja donde se muestra el nombre de la clase y los atributos de la misma. PK y FK son las abreviaturas de clave primaria y clave extranjera respectivamente. Las flechas muestran relaciones uno a muchos con el lado de muchos marcado por la cabeza de la fecha. Tras el nombre de cada atributo y separado por un guión se

muestra el tipo de datos del mismo (excepto en el caso de las claves extranjeras). Recuerda que *Django* crea de forma automática una clave primaria llamada `id` para cada uno de los modelos por lo tanto no la crees tu explícitamente.

## 3.1. Requerimientos a satisfacer por el Modelo de Datos

- El modelo debe crearse en un proyecto de *Django* llamado *labassign* en el cual existirá una única aplicación llamada *core*.
- Crea la clase `Student` heredando de la clase `User` definida en *Django* (`django.contrib.auth.models`) de forma que se tenga acceso al sistema de verificación y validación de claves proporcionado por el propio framework. Está aproximación es un alternativa a la mostrado en el libro de referencia donde se crea una relación uno a uno (`models.OneToOneField`) entre `User` y `UserProfile`.
- La variable `validate` se usa para indicar si una pareja (`Pair`) está activa o no. (Sólo las parejas activas pueden ser usadas para solicitar un grupo). Cuando un usuario solicita la creación de una pareja se almacena la misma como un objeto de tipo `Pair` con la variable `validated`=False siendo `student_1` el usuario que ha solicitado la creación de la pareja. Para que esta pareja se active, es necesario que el segundo miembro de la pareja solicite la creación de una pareja formada por los mismos estudiantes. Tras esta segunda petición el valor del atributo `validated` se actualiza a `True` y la pareja queda activada.
- Es posible solicitar la disolución de una pareja (siempre que no se haya elegido grupo). Si `validate`==False la disolución se otorga de forma inmediata (se borra el objeto de tipo `Pair`). En caso contrario cuando uno de los miembros de la pareja solicita la disolución se almacena dicho estudiante en la variable `studentBreakRequest` y cuando el segundo miembro de la pareja solicita la disolución se procede a borrar el objeto de tipo `Pair`.
- `selectGroupStarDate` marca la fecha a partir de la cual se puede elegir grupo.
- No olvides crear una función `__str__` para cada modelo. Este método debe devolver una representación en forma de cadena del objeto. Se le llama cuando

imprimimos una instancia del objeto con print o llamamos a la función interna `str()` sobre el mismo.

- Usando las opciones `meta` de los modelos (`https://docs.djangoproject.com/en/3.1/ref/models/options/`) asegúrate de que cuando se solicite un listado de `Students` o `Teachers` este salga ordenado por los atributos `last_name, first_name`. Igualmente si se solicita un listado de `LabGroups`, `TheoryGroups` o `GroupConstraints` estos deben ordenarse por el nombre del grupo de laboratorio, teoría o ambos.

# 4. Probando los Modelos

Para poder verificar el modelo de datos (y la aplicación en general) se necesita almacenar datos de prueba en la misma.

## 4.1. Poblar la Base de Datos

Crea una script llamado `populate.py` que genere objetos de las diferentes modelos y los persista en la base de datos del proyecto. Usa como guía el fichero llamado `populate.py` disponible en *Moodle*. Este script tiene una estructura que le permite ser invocado usando la línea de comandos `python3 ./manage.py populate all 19-edat.csv 19-edat_2.csv`.

El fichero debe situarse en el directorio `management/commands` (el path está dado desde el directorio que contiene la aplicación *core*). Si el directorio no existe creadlo. Los datos a introducir por el script `populate.py` estarán "hardcodeados" en el fichero excepto los referentes a los estudiantes, estos últimos se leerán de sendos ficheros de texto en formato CSV denominados `19-edat.csv` y `19-edat_2.csv`. Los ficheros se encuentran en *Moodle*. La estructura de `19-edat.csv` está compuesta por cinco campos llamados: `NIE`, `DNI`, `Apellidos`, `Nombre` y `grupo-teoria` mientras que en `19-edat_2.csv` se han añadido dos campos extra con las notas de prácticas y teoría del año pasado. Obviamente solo un subconjunto de alumnos presentes esté curso aparecerán en este segundo fichero y algunos de los alumnos que aparecen en el

segundo fichero no cursarán la asignatura este año. En todos los datos leídos de los ficheros en formato cvs debéis eliminar los espacios en blanco al inicio y final de cada cadena. (NOTA: en los ficheros de datos, los valores de los distintos campos son ficticios y por lo tanto no se corresponden a la realidad).

Para obtener el nombre de los grupos de teoría y prácticas así como los horarios y los nombres de los profesores consultad los valores asignados a la asignatura EDAT en la pagina web de la escuela tanto para el grado de informática como para el doble grado. Finalmente, los valores para los modelos `OtherConstraints` y `GroupConstraints` están sugeridos en la template para el fichero `populate.py`

## 4.2. Makefile

Durante el desarrollo de la práctica es normal ejecutar de forma repetitiva algunos comandos. Para automatizar el proceso hemos usado el comando `make` con el `makefile` que podéis bajar de *Moodle* y que usaremos para corregir vuestras practicas.

En `makefile` se han definido las siguientes operaciones:

- `clear_db`: borra todos los objetos persistidos en la base de datos llamada databasename.
- `create_super_user`: crea un usuario con permisos de administración. El nombre de usuario y la clave son `alumnodb`.
- `populate`: equivalente a `python3 ./manage.py populate all 19-edat.csv 19-edat_2.csv`
- `run`: equivalente a `python3 manage.py runsever`
- `update_db`: equivalente a `python3 manage.py makemigrations; python3 manage.py migrate`
- `clear_update_db`: similar a `update_db` pero borra previamente el contenido del directorio `migrations`

- `reset_db`: equivalente a ejecutar `clear_update_db`, `update_db` y `create_user`
- `shell`: lanzar el cliente de *PostgreSQL* llamado `psql`
- `test_models`: equivalente a ejecutar el comando `python3 manage test core.tests_models`
- `test_services`: equivalente a ejecutar el comando `python3 manage test core.tests_services`

## 4.3. Tests

Para verificar la correcta definición del modelo, se proporciona una batería de tests unitarios y funcionales (no necesariamente completa) que debe satisfacer tu código. En concreto, en el fichero `tests_models.py` se proporcionan la clase, `ModelTests` con tests para las clases `Teacher`, `LabGroup`, `TheoryGroup`, `GruopConstraints`, `Student`, `Pair` y `OtherConstraints`, respectivamente. Existe un segundo fichero de test llamado `tests_services.py` cuyo contenido se describirá más adelante. `tests_models.py` usa una colección de ficheros auxiliares con extensión `.pkl` que se encuentran disponibles en *Moodle*.

Una vez que hayáis conseguido que los test se ejecuten de forma satisfactoria ejecutad `coverage`

```
coverage erase
coverage run --omit="*/test*" --source=core ./manage.py test core.tests_models
coverage report -m -i
```

mirad el "coverage" de los ficheros `models.py` y `populate.py`, y si no es del 100 % añadid al fichero de tests los test que hagan falta para alcanzar este valor. (La solicitud the 100 % de cobertura aplica exclusivamente al codigo que creeis vosotros. Ignorad el codigo creado por django o el suministrado por vuestros profesores)

**NOTA IMPORTANTE**: A la hora de realizar la implementación, se recomienda seguir una estrategia TDD (test-driven development) tratando de satisfacer uno a uno y siguiendo el orden establecido cada uno de los tests. Esto aplica para toda la implementación del proyecto con los tests proporcionados.

# 5. Trabajo a realizar durante la primera y segunda semana

## 5.1. Modelo de datos

Crear un proyecto *Django* llamado *labassign* que incluya la aplicación *core* con el modelo ORM que dará soporte a la aplicación satisfaciendo los siguientes requerimientos:

- El proyecto contendrá una página de administración (interfaz *Django* en la dirección `http://hostname:8000/admin/`) que permita introducir y borrar datos de forma coherente en la base datos. Tanto el nombre de usuario como la c del usuario de administración deben ser *alumnodb*.

  **NOTA IMPORTANTE**: un error típico a la hora de crear la página de administración es no registrar los modelos en `admin.py`.

- Los datos deben persistirse en una base de datos *PostgreSQL* llamada *psi*.

- Si se añade cualquier elemento al esquema de datos descrito en el enunciado, se deberá presentar un diagrama entidad-relación describiendo la base de datos que será usada por la aplicación (formato pdf). Añade este fichero al repositorio para incluirlo en la entrega.

- Ejecutando el comando `python3 ./manage.py populate all 19-edat.csv 19-edat_2.csv` debe ser posible poblar la aplicación con datos tal y como se describe en la subsección 4.1.

- El código creado debe satisfacer los tests definidos en la clase `ModelTests` los cuales asumen la existencia del fichero `populate.py`.

- El "coverage" de los ficheros `models.py` y `populate.py` debe ser del 100 %. La solicitud the 100 % de cobertura aplica exclusivamente al codigo que creeis vosotros. Ignorad el codigo creado por django o el suministrado por vuestros profesores.

No olvides crear un repositorio privado en *Github* donde ir mandando las diferentes versiones de tu código. Añade a tu profesor al repositorio.

## 5.2. Acceso y manipulación de datos

Una vez creado y validado el modelo de datos, crear en la raíz del proyecto un script *Python* llamado `test_query.py` que realice las siguientes tareas:

- Comprueba si existe un usuario (`Student`) con id=1000 y si no existe lo crea (y persiste en la base de datos). En el futuro nos referiremos a este usuario como `user_1000`. Recordar que *Django* añade automáticamente a todas las tablas del modelo un atributo `id` que actúa como clave primaria.
- Comprueba si existe un usuario con id=1001 y si no existe lo crea (y persiste en la base de datos). En el futuro nos referiremos a este usuario como `user_1001`.
- Crea una pareja (`Pair`) usando como `student1` y `student2` los usuarios `user_1000` y `user_1001` respectivamente. Persiste el resultado en la base de datos.
- Busca todas las parejas (`Pair`) en donde `user_1000` figure como `student1`. Imprima el resultado de la búsqueda por pantalla.
- Modifica el valor de `validate` en las parejas resultantes de la búsqueda anterior de forma que valga `True`. Persiste la modificación en la base de datos.
- Crea un objeto de tipo `OtherConstraints` con el valor de `selectGroupStartDate` igual al momento actual más un día.
- Realiza una búsqueda que devuelva todos los objetos de tipo `OtherConstraints` y para el primero de los objetos devueltos compara el valor de `selectGroupStartDate` con el momento actual. Imprime el resultado de la comparación por pantalla de forma que se indique si `selectGroupStartDate` es una fecha en el pasado o en el futuro. El código creado debe ser valido para cualquier valor de `selectGroupStartDate`.

# 6. Trabajo a presentar al final de la segunda semana

- Subid a *Moodle* el fichero obtenido al ejecutar el comando `zip -r ../assign3_first_delivery.zip .git` desde la raíz del proyecto. Recordad que hay que añadir y "comitir" los ficheros a git antes de ejecutar el comando. Si queréis comprobar que el contenido del fichero zip es correcto lo podéis hacer ejecutando la orden: `cd ..; unzip assign3_first_delivery.zip; git clone . tmpDir; ls tmpDir`. A partir de este fichero comprimido debe ser posible obtener: los modelos, los tests y los scripts: `test_query.py` y `populate.py`
- **Desplegar la aplicación en *Heroku* como una aplicación nueva (no uséis la creada en la práctica anterior)** y poblarla ejecutando `python3 ./manage.py populate all 19-edat.csv 19-edat_2.csv`. Recordar que para que el despliegue funcione correctamente es necesario:
  - Añadir la dirección de *Heroku* a la variable `ALLOWED_HOSTS` definida el fichero `settings.py`.
  - Migrar el modelo de datos.
  - Crear el usuario de administración con nombre de usuario y clave *alumnodb*.
  - El script `test_query.py` se puede ejecutar con el comando `heroku run python test_query.py`.

# 7. Trabajo a desarrollar durante la tercera y cuarta semana

Vamos a crear diferentes funciones/servicios en el fichero `views.py` de la aplicación *core*. A la hora de publicar cada una de estas funciones `xxxx`, sugerimos crear una

entrada en `urls.py` con url=`xxxx`, función_a_ejecutar=`views.xxxx` y alias=`xxxx`. Adicionalmente, sugerimos mapear la url `index/` y la url vacía a la función `views.home` usando como alias `index` y `home`, respectivamente. Los distintos servicios se accederán desde un menú funcionalmente equivalente al que puede verse en `https://desolate-sands-30591.herokuapp.com/`.

El catálogo de servicios a desarrollar es:

- **login:**
  Comprueba en el sistema de autenticación de *Django* que un par usuario-clave recibido por POST es válido y, de serlo, crea una nueva sesión para él.
  Entrada: dos cadenas de caracteres con el nombre del usuario y su clave.
  Resultado: si el login es incorrecto repintar el formulario de login junto a un mensaje de error. Si el login es valido mostrar `home` donde se resume la información relacionada con el usuario. La información mostrada debe contener: (1) el nombre del usuario, (2) el estado de la convalidación (3) si el usuario es miembro de alguna pareja nombre de los integrantes de la misma y valor del campo `validated` y (4) si el usuario ha solicitado un grupo de practicas, mostrar el nombre del grupo.
  Accesibilidad: solo puede ser invocado por usuarios anónimos. Si existe un usuario autenticado se debe devolver un mensaje de error. Implementad este requisito usando el sistema de verificación de *Django*.

- **logout:**
  Cierra la sesión de un usuario previamente autenticado. Para ello, borra todas las variables de sesión.
  Entrada: ninguna.
  Resultado: el usuario y todos sus datos asociados dejan de estar almacenados en la sesión.
  Accesibilidad: solo puede ser invocado por usuarios previamente autenticados. Para implementar este requisito se recomienda revisar la documentación referente al decorador `login_required`.

- **convalidation:**
  Solicita la convalidación de las prácticas realizadas el año pasado

Entrada: ninguna. Usa el objeto de tipo `Student` almacenado al realizar el login.
Resultado: (a) se comprueba las notas obtenidas el año pasado, (b) se comparan estas notas con los valores almacenados en los campos `minGradeTheoryConv` y `minGradeLabConv` de la clase `OtherConstraints` y (c) si las notas del año pasado superan los valores almacenados en `OtherConstraints` se concede la convalidación para lo cual es necesario actualizar el campo `convalidationGranted` de la clase `Student`. Se debe devolver al usuario un mensaje comunicándole de forma razonada el resultado de su solicitud. No se aceptaran convalidaciones de usuarios que formen parte de una pareja validada o que hayan iniciado la creación de una pareja `student1`
Accesibilidad: solo puede ser invocado por usuarios previamente autenticados. Un usuario dado sólo puede solicitar la convalidación de sus notas no las de otros alumnos. Implementad este requisito usando el sistema de verificación de *Django*.

- **applypair:**
  Solicita la creación de una pareja.
  Entrada: identificador del estudiante segundo miembro de la pareja. Al estudiante solicitante lo denominaremos `user1` y al segundo miembro de la pareja `user2`. `user2` se seleccionará entre los candidatos ofrecidos en un menú desplegable el cual no debe contener candidatos inválidos como puedan ser candidatos asociados a otra pareja ya validada o `user1`.
  Resultado: Si la pareja (`student1=user2`, `student2=user1`) no existe se crea un objeto de tipo `Pair` con los valores (`student1=user1`, `student2=user2`, `validate=False`). Si la pareja (`student1=user2`, `student2=user1`) existe entonces se actualiza el valor de `validate` a `True`.
  Accesibilidad: solo puede ser invocado por usuarios previamente autenticados. Implementad este requisito usando el sistema de verificación de *Django*. Un usuario dado sólo puede solicitar una pareja en la que el sea uno de los miembros y el segundo miembro no pertenezca a una pareja ya validado o sea el mismo.

- **applygroup:**
  Solicita un grupo de prácticas.
  Entrada: identificador de grupo. Se seleccionará entre los candidatos ofrecidos en un menú desplegable el cual no debe contener candidatos inválidos. Recordad que la clase `GroupConstraints` almacena los grupos de practicas que pueden ser elegidos por los estudiantes dependiendo del grupo de teoría al que pertenezcan. Igualmente el campo `maxNumberStudents` de la clase `LabGroup` almacena el número máximo de estudiantes que puede existir en ese grupo de prácticas. Finalmente `counter` (también de `LabGroup`) almacena el número de estudiantes asignados a ese grupo de laboratorio hasta la fecha.
  Resultado: Si el grupo solicitado es válido y no esta completo el usuario es asignado al mismo al tiempo que la variable `counter` se incrementa. Si el usuario pertenece a una pareja validada su compañero también es agregado al grupo seleccionado y `counter` se incrementa una segunda vez.
  Accesibilidad: Este método solo puede ser invocado por usuarios previamente autenticados. Un usuario dado sólo puede solicitar grupo para el mismo y para su compañero si forman una pareja validada. Implementad la primera parte de estos requisitos usando el sistema de verificación de *Django*.

Para probar la funcionalidad solicitada, el fichero `tests_services.py` proporciona una batería de test (no necesariamente completa) de todos los servicios descritos. `LogInOutServiceTests`, `ConvalidationServiceTests`, `PairServiceTests` y `GroupServiceTests`.

**NOTA**: Desarrollar el proyecto localmente y cuando funcione subirlo a *Heroku* **sin pisar** la aplicación subida en la práctica 2.

# 8. Trabajo a presentar al finalizar la práctica

- Aseguraros que vuestro código satisface todos los tests proporcionados. No es admisible que se modifique el código de los tests excepto las variable situadas al principio del fichero las cuales se encuentran claramente identificadas.
- Implementar todos los tests que consideres necesarios para cubrir la funcionalidad desarrollada. Estos tests se deben incluir en clases dentro de un fichero llamado `tests_additional.py`.
- Incluir en la raíz del proyecto un fichero llamado coverage.txt que contenga el resultado de ejecutar el comando `coverage` para todos los tests.
- Desplegar y probar la aplicación en *Heroku*.
- Subir a *Moodle* el fichero obtenido al ejecutar el comando `zip -r ../assign3_final.zip .git` desde la raíz del proyecto. Recordad que hay que añadir y "comitir" los ficheros a git antes de ejecutar el comando. Si queréis comprobar que el contenido del fichero zip es correcto lo podéis hacer ejecutando la orden: `cd ..; unzip assign3_final.zip; git clone . tmpDir; ls tmpDir`. En este punto es importante asegurarse que la variable `ALLOWED_HOSTS` del fichero `settings.py` incluido en la entrega contiene tu dirección de despliegue en *Heroku* (si no aparece corregiremos la práctica como si el proyecto no estuviera desplegado en *Heroku*). Asimismo, comprobar que tanto el nombre de usuario como la clave del usuario de administración son *alumnodb*.

# 9. Criterios de evaluación

Nota: A la hora de evaluar la práctica 3 **NO** se consideran los test implementados en el fichero `tests_services.py` dentro de la clase `BreakPairServiceTests`. Estos tests acceden a funciones solicitadas en la practica 4.
Para aprobar con 5 puntos es necesario satisfacer en su totalidad los siguientes criterios:

- Todos los ficheros necesarios para ejecutar la aplicación se han entregado a tiempo.
- El código se ha guardado en un repository git y este repositorio es privado.
- Los script `test_query.py` y `populate.py` hacen lo que se solicita.
- La aplicación se puede ejecutar localmente.
- Al ejecutar en local los tests el número de fallos no es superior a seis y el código que los satisface es funcional.
- No se ha modificado el código de los tests.

Cumpliendo los siguientes criterios se puede optar a una nota máxima de 6.9:

- Los criterios enunciados en el párrafo anterior se satisfacen en su totalidad.
- La aplicación está desplegada en *Heroku*. En el fichero `settings.py` se encuentra añadida la dirección de *Heroku* a la variable `ALLOWED_HOSTS`. Además de estar desplegada, la aplicación funciona correctamente en *Heroku*.
- Al ejecutarse los tests el número de fallos no es superior a cuatro y el código que los satisface es funcional.
- Se utiliza correctamente el sistema de autentificación de usuarios de *Django*.
- La aplicación de administración de base de datos se ha desplegado y está accesible en *Heroku* usando el nombre de usuario/clave *alumnodb*.

- Usando la aplicación de administración es posible crear o borrar objetos pertenecientes a todos los modelos solicitados.

Cumpliendo los siguientes criterios se puede optar a una nota máxima de 7.9:

- Los criterios enunciados en el párrafo anterior se satisfacen en su totalidad.
- Al ejecutarse los tests el número de fallos no es superior a dos y el código que los satisface es funcional.
- El código es legible, eficiente, está bien estructurado y comentado.
- Se utilizan las herramientas que proporciona el framework.
- Sirva como ejemplo de los puntos anteriores:
  - Las búsquedas las realiza la base de datos no se cargan todos los elementos de una tabla y se busca en las funciones definidas en `views.py`.
  - Los errores se procesan adecuadamente y se devuelven mensajes de error comprensibles.
  - El código presenta un estilo consistente y las funciones están comentadas incluyendo su autor. Nota: el autor de una función debe ser único.
  - Se es coherente con los criterios de estilos marcados por *Flake8*. *Flake8* no devuelve ningún error al ejecutarse sobre las líneas de código programadas por el estudiante.
- Resulta imposible suplantar a un usuario sin conocer su nombre de usuario y clave. Por ejemplo realizando una llamada directamente a un método usando `curl` o similar.

Cumpliendo los siguientes criterios se puede optar a una nota máxima de 8.9:

- Los criterios enunciados en el párrafo anterior se satisfacen en su totalidad.
- Todos los tests y todas las pruebas ejecutadas dan resultados satisfactorios.

- Si reducimos el tamaño de la ventana del navegador o utilizamos el zoom, todos los elementos de la página siguen resultando accesibles y no se pierde funcionalidad.

Para optar a la nota máxima se debe cumplir lo siguientes criterios:

- Los criterios enunciados en el párrafo anterior se satisfacen en su totalidad.
- Se reporta la cobertura (programa coverage) obtenida por los tests antes y después de añadir tus nuevos tests. La cobertura debe incrementarse.
- La cobertura para los ficheros que contienen los modelos, las vistas y los formularios es superior al 99 %.

Nota: Entrega parcial retrasada → substraer un punto por cada entrega.

Nota: Entrega final fuera de plazo → substraer un punto por cada día (o fracción) de retraso en la entrega.

Nota: El código usado en la corrección de la práctica será el entregado en *Moodle*. Bajo ningún concepto se usará el código existente en *Heroku*, *Github* o cualquier otro repositorio.