PROYECTO FIN DE CURSO DATA & DESARROLLO EN PYTHON

Ele-Task

Autora: Sandra Requena FIliu

https://ele-task.onrender.com/

29 de septiembre de 2023

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

SISTEMA NACIONAL DE GARANTÍA JUVENIL

Contenido

Introducción

Ele-Task es una aplicación web en la cual los usuarios registrados pueden añadir tareas y hacer un seguimiento de estas. Las tareas solo podrán ser vistas y manipuladas por sus creadores para aumentar así la seguridad de esta aplicación. Además, los usuarios registrados en ***Ele-Task*** podrán gestionar su cuenta en cualquier dispositivo móvil. Para desarrollar ***Ele-Task*** se han utilizado los siguientes lenguajes y programas:

- **Backend:** Python, MySQL
- **Frontend:** Angular (HTML, CSS, TypeScript), Bootstrap (Bootswatch)
- **Programas:** Visual Studio Code, phpMyAdmin
- **Testing:** Postman

Backend

Todos los datos manejados (usuarios y tareas) se guardan en una base de datos MySQL, la cual contiene dos tablas, una para cada tipo de dato. La estructura de estas tablas es la siguiente:

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
1	id	int			No	*Ninguna*		AUTO_INCREMENT	Cambiar Eliminar Más
2	nombre	varchar(255)	utf8mb4_0900_ai_ci		No	*Ninguna*			Cambiar Eliminar Más
3	email	varchar(255)	utf8mb4_0900_ai_ci		No	*Ninguna*			Cambiar Eliminar Más
4	usuario	varchar(255)	utf8mb4_0900_ai_ci		No	*Ninguna*			Cambiar Eliminar Más
5	password	varchar(255)	utf8mb4_0900_ai_ci		No	*Ninguna*			Cambiar Eliminar Más
6	imagen	varchar(255)	utf8mb4_0900_ai_ci		Sí	*NULL*			Cambiar Eliminar Más

Ilustración 1. Tabla 'usuario' en la base de datos. El valor del id es autoincremental y la imagen puede ser nula. Tanto los valores del email y del usuario son únicos para evitar usuarios duplicados.

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
1	id	int			No	*Ninguna*		AUTO_INCREMENT	Cambiar Eliminar Más
2	descripcion	varchar(255)	utf8mb4_0900_ai_ci		No	*Ninguna*			Cambiar Eliminar Más
3	realizada	tinyint(1)			No	*Ninguna*			Cambiar Eliminar Más
4	id_usuario	int			No	*Ninguna*			Cambiar Eliminar Más
5	fecha	datetime			Sí	*NULL*			Cambiar Eliminar Más

Ilustración 2. Tabla 'tarea' en la base de datos. El valor del id es autoincremental y la fecha puede ser nula.

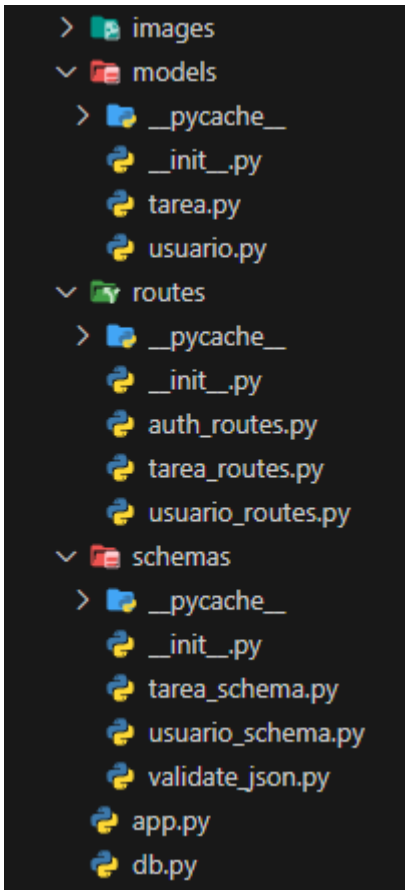
Estas tablas se generaron automáticamente gracias al código desarrollado en Python. Los datos insertados se muestran de la siguiente manera:

	id	nombre	email	usuario	password	imagen
Editar Copiar Borrar	1	Sandra Requena	sandrarequena1999@gmail.com	trafasan	DLaw5039	http://127.0.0.1:5000/images/trafasan.jpg
Editar Copiar Borrar	2	Jorge	alterWolf@gmail.com	alterWolf	Wolf0701	http://127.0.0.1:5000/images/alterWolf.jpg
Editar Copiar Borrar	3	María	darks13@gmail.com	darks13	Lolaso13	*NULL*
Editar Copiar Borrar	4	Helena	helenlumen@gmail.com	lumen124	Fontana8	http://127.0.0.1:5000/images/lumen124.jpg

Ilustración 3. Vista de los datos de los usuarios en la base de datos.

	id	descripcion	realizada	id_usuario	fecha
Editar Copiar Borrar	1	Comprobar si la actualización de tareas funciona c...	1	1	*NULL*
Editar Copiar Borrar	2	Finalizar proyecto	0	1	2023-09-29 14:00:00
Editar Copiar Borrar	3	Prueba de inserción en la web	1	1	*NULL*

Ilustración 4. Vista de los datos de las tareas en la base de datos.



El código desarrollado con Python se ha organizado en las carpetas que se pueden ver en la imagen de la izquierda. Para empezar, en la raíz de la carpeta principal se encuentran los siguientes archivos:

- **app.py:** Es este archivo se crea una instancia de la clase **Flask**, se habilita **CORS** en la aplicación, se definen las rutas mediante *blueprints* y se configura la base de datos. Por último, se crean las tablas en la base de datos si no existen y se ejecuta la aplicación.

- **db.py:** En este archivo solo se importa y crea una instancia de **SQLAlchemy** en la aplicación **Flask**. Si este código estuviera en el archivo **app.py**, se generarían errores.

También se puede ver una carpeta llamada images, en la cual se guardarán las fotos de perfil de los usuarios.

El resto de código se ha organizado en las diferentes carpetas que se muestran. En todas ellas aparece un archivo .py llamado _init_. Este archivo sirve para que las importaciones de variables y clases se simplifiquen.

models

En esta carpeta podemos visualizar los *dataclasses* de Usuario y Tarea.

En el dataclass de Usuario, se ha definido el parámetro *cascade* para que, cuando se elimine un usuario, se eliminen automáticamente sus tareas. También están definido el parámetro *unique* para los atributos email y usuario.

En el dataclass de Tarea, se ha enlazado el atributo *id_usuario* con el usuario, realizando así una relación uno a muchos.

```python
from dataclasses import dataclass
from typing import List
from sqlalchemy.orm import Mapped, mapped_column, relationship
from sqlalchemy import String
from db import db

You, hace 1 segundo | 1 author (You)
@dataclass
class Usuario(db.Model):
    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    nombre: Mapped[str] = mapped_column(String(255))
    email: Mapped[str] = mapped_column(String(255), unique=True)
    usuario: Mapped[str] = mapped_column(String(255), unique=True)
    password: Mapped[str] = mapped_column(String(255))
    imagen: Mapped[str] = mapped_column(String(255), nullable=True)
    tareas: Mapped[List["Tarea"]] = relationship(back_populates="usuario",
                                                 cascade="all, delete-orphan")
```

```python
from dataclasses import dataclass
from sqlalchemy.orm import Mapped, mapped_column, relationship
from sqlalchemy import String, Boolean, DateTime, ForeignKey
from db import db

You, ayer | 1 author (You)
@dataclass
class Tarea(db.Model):
    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    descripcion: Mapped[str] = mapped_column(String(255))
    realizada: Mapped[bool] = mapped_column(Boolean())
    id_usuario: Mapped[int] = mapped_column(ForeignKey("usuario.id"))
    fecha: Mapped[DateTime] = mapped_column(DateTime(), nullable=True)
    usuario: Mapped["Usuario"] = relationship(back_populates="tareas")
```

Ilustración 6. Dataclasses de Usuario (arriba) y Tarea (abajo)

schemas

En esta carpeta se encuentran los diferentes *schemas* de Usuario y Tarea. Estos *schemas* controlarán tanto el visionado como la introducción de datos, siendo estos validados gracias a la función *validate_json*. Se destacan las siguientes validaciones:

- **Usuario:** El *schema* básico define las validaciones para el id, el usuario y la imagen. El segundo *schema* define además el nombre, el email y la contraseña (*password*). El tercero *schema* suma el atributo tareas al resto. Existe un cuarto *schema* para la actualización de la contraseña, el cual tiene los atributos *password* y *new_password*.
  Para todos los parámetros relacionados con contraseñas se ha definido una expresión regular para limitar sus valores. Para el atributo nombre se ha utilizado la validación **Length** para definir su longitud mínima. Para el atributo email se ha usado la validación **Email** presente en la librería *marshmallow*.

```
# Cuarto schema que une el resto de atributos con el atributo tareas
class UsuarioConTareasSchema(UsuarioLoggedSchema):
   tareas = fields.Nested("TareaSchema", many=True)

# Validación del nombre
validate.Length(min=4, error="El nombre debe tener al menos 4
caracteres")
validate.Regexp(regex=" ^\\D+$", error="El nombre no puede contener
números")

# Validación de las contraseñas
validate=validate.Regexp(regex="^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)[a-zA-
Z\d]{8,}$", error="La contraseña no tiene un formato válido")

# Validación del email
validate=validate.Email(error="El correo electrónico no tiene un
formato válido")
```

- **Tarea:** El *schema* básico define las validaciones para el id, la descripción, el estado de realización, el id del usuario y la fecha. El segundo *schema* añade el atributo del Usuario. Los valores que puede tomar el atributo fecha se han limitado para que sean posteriores a la fecha y hora actuales.

```
# Función que valida la fecha
   def validate_datetime(value:datetime):
   value = datetime.fromisoformat(str(value))
   value = value.replace(tzinfo=pytz.UTC)
   if value < datetime.now(pytz.UTC):
        raise ValidationError("La fecha no puede ser menor que la
fecha actual")
```

routes

En esta carpeta se encuentran todas las rutas que maneja nuestra API. Las rutas principales son las siguientes:

- **auth:** En esta ruta se manejan tanto el registro como el inicio de sesión de un usuario. En el método de registro, puede ocurrir la excepción *IntegrityError* si ya existe los datos de email y/o usuario en la base de datos, por lo que se maneja dicha excepción para devolver un mensaje de error.

```
try:
    schema = UsuarioLoggedSchema()
    db.session().add(usuario)
    db.session().commit()
    return jsonify({"usuario": schema.dump(usuario)}), 201
except IntegrityError as e:
    mensaje_error = {
        "error": "Datos de entrada no válidos",
        "messages": {}
    }
    if 'usuario.email' in str(e):
        mensaje_error['messages']['email'] = ['Este correo ya está registrado']
    elif 'usuario.usuario' in str(e):
        mensaje_error['messages']['usuario'] = ['Este usuario ya existe']
    return jsonify(mensaje_error), 500
```

Ilustración 7. Manejo de la excepción IntegrityError. Esto evita que la aplicación colapse. El schema UsuarioLoggedSchema valida los datos del usuario, imagen, nombre, email y password.

Para guardar las imágenes de los usuarios se utiliza la función *guarda_imagen* presente en el mismo archivo. Esta función solo se utilizará si se ha mandado un valor no nulo de imagen. Como se ha podido comprobar en la visualización de los datos de la tabla **usuario** en la base de datos, el valor de imagen guardado será la URL donde se encuentra el archivo si no es nulo. Para acceder a las imágenes a través de ese URL se ha declarado un método *GET* en app.py.

```
def guarda_imagen(json):
    imagen_str = json["imagen"].split(",")[1] if json["imagen"].startswith("data") else json["imagen"]
    img = Image.open(io.BytesIO(base64.decodebytes(bytes(imagen_str, "utf-8"))))
    ruta = f"images/{json['usuario']}.jpg"
    img.convert('RGB').save(f"{os.path.dirname(__file__)}/../{ruta}")
    return ruta
```

```
@app.get('/images/<filename>')
def serve_image(filename):
    print(filename)
    image_path = 'images/' + filename
    return send_file(image_path, mimetype='image/jpeg')
```

Ilustraciones 8 y 9. Función que guarda la imagen del usuario (arriba) y método para acceder a las imágenes de los usuarios (abajo).

Además, se ha añadido una subruta para que el usuario actualice su contraseña. Este método comprueba si la contraseña actual recibida es la misma que la contraseña guardada en la base de datos y, en caso afirmativo, actualiza el dato con la nueva contraseña enviada.

- **usuarios:** Esta ruta maneja la obtención de todos los usuarios y del usuario que haya iniciado sesión; la actualización de sus datos y el borrado de su cuenta. Todos estos métodos excepto el de la obtención de los usuarios están protegidos, por lo que solo se puede acceder a ellos si se proporciona un token JWT (**JSON Web Token**) válido en la solicitud HTTP.

```python
@usuarios.get('/logged')
@jwt_required()
def get_usuario_logueado():
    id_usuario=get_jwt_identity()
    usuario = db.get_or_404(Usuario, id_usuario)
    schema = UsuarioConTareasSchema()
    return jsonify(schema.dump(usuario))
```

Ilustración 10. Método de obtención del usuario logueado. El decorador @jwt_required de Flask-JWT-extended marca este método como protegido.

La actualización del usuario también incorpora la función *guarda_imagen* vista en el punto anterior.

El código del método GET de usuarios es el siguiente:

```python
@usuarios.get('') # type: ignore
def get_usuarios():
    select = db.select(Usuario)
    usuarios = db.session().execute(select).scalars().all()
    # Paginación de la API
    pag = request.args.get('pag', default=1, type=int)
    size = request.args.get('size', default=6, type=int)
    schema = UsuarioSchema(many=True)
    url = f'{request.host_url}usuarios'
    params = {}
    if size != 6:
        params['size'] = size
    params_next = params.copy()
    params_prev = params.copy()
    params_next['pag'] = pag+1
    if pag != 2:
        params_prev['pag'] = pag-1
    response = {
        'count': len(usuarios),
        'next': f'{url}?{urlencode(params_next)}' if len(usuarios) > pag*size else None,
        'previous': f'{url}?{urlencode(params_prev)}' if pag == 2 | ('pag' in params_prev and params_prev['pag'] != 0) else None,
        'result': schema.dump(usuarios[(pag-1)*size:pag*size])}
    return jsonify(response)
```

Como se puede ver, este método devuelve una lista de usuarios paginada, en la que se encuentras estos cuatro datos:

- **count:** Número de usuarios en la base de datos. Este dato es constante en todas las páginas.
- **next:** URL de la siguiente página. Si es la última página, su valor es nulo.
- **previous:** URL de la página anterior. Si es la primera página, su valor es nulo.
- **result:** Lista de usuarios del tamaño especificado. El tamaño por defecto es de 6 usuarios por página.


```
{
    "count": 12,
    "next": "http://127.0.0.1:5000/usuarios?size=2&pag=4",
    "previous": "http://127.0.0.1:5000/usuarios?size=2&pag=2",
    "result": [
        {
            "id": 5,
            "imagen": null,
            "usuario": "chef987"
        },
        {
            "id": 6,
            "imagen": "http://127.0.0.1:5000/images/ornitier9.jpg",
            "usuario": "ornitier9"
```


Ilustración 12. Resultado de la URL http://127.0.0.1:5000/usuarios?pag=3&size=2

A la URL se le pueden añadir los parámetros de consulta *pag* y *size* para cambiar de página y controlar el tamaño de la lista de usuarios, respectivamente. Si se consulta la página 3 de una lista de usuarios con un tamaño de 2 usuarios por página, el resultado sería el de la imagen de la derecha.

- **tareas:** Esta ruta maneja la obtención de todas las tareas, de una tarea y de todas las tareas del usuario. Además, se habilita la creación, actualización y borrado de tareas. En el método para la obtención de tareas del usuario se devuelve una lista paginada al igual que el método GET de usuarios. A esta URL, además de los parámetros de consulta *pag* y *size*, también se le puede añadir el parámetro *realizada*. Este parámetro filtra las recetas en función de su estado de realización y, si se especifica, el orden de la lista varía:
    - Si se filtra por tareas realizadas, estas se ordenan por el valor de su id de forma descendente.
    - Si se filtra por tareas no realizadas, estas se ordenan por el valor de la fecha ascendentemente. Las tareas a las que no se les haya establecido este atributo aparecerán después de las tareas que sí tiene fecha.

```python
@tareas.get('/usuario') # type: ignore
@jwt_required()
def get_tareas_usuario():
    id_usuario=get_jwt_identity()
    select = db.select(Tarea).where(Tarea.id_usuario == id_usuario)
    realizada = request.args.get('realizada', type=int)
    # Paginación de la API
    pag = request.args.get('pag', default=1, type=int)
    size = request.args.get('size', default=3, type=int)
    schema = TareaSchema(many=True)
    url = f'{request.host_url}tareas/usuario'
    params = {}
    if size != 3:
        params['size'] = size
    if realizada != None:
        select = select.where(Tarea.realizada == realizada).order_by(asc(Tarea.fecha) if realizada == 0 else desc(Tarea.id))
        params['realizada'] = realizada
    params_next = params.copy()
    params_prev = params.copy()
    params_next['pag'] = pag+1
    if pag != 2:
        params_prev['pag'] = pag-1
    tareas = db.session().execute(select).scalars().all()
    if realizada == 0: # Se reordena la lista de tareas para mover las tareas sin fechas a los últimos lugares      You, hace 58 
        tareas = sorted(tareas, key=lambda tarea: (0, tarea.fecha) if tarea.fecha is not None else (1,))
    response = {
        'count': len(tareas),
        'next': f'{url}?{urlencode(params_next)}' if len(tareas) > pag*size else None,
        'previous': f'{url}?{urlencode(params_prev)}' if pag == 2 | ('pag' in params_prev and params_prev['pag'] != 0) else None,
        'result': schema.dump(tareas[(pag-1)*size:pag*size])}
    return jsonify(response)
```

Ilustración 13. Código del método GET de la lista de tareas del usuario logueado.

Frontend

Sabrás que estás en la web de ***Ele-Task*** ya que la web tiene un icono propio en la pestaña del navegador. A continuación, podrás ver los apartados de esta web.

Menú de inicio

En el menú de inicio se pueden consultar una explicación breve de las acciones que puedes efectuar en esta web. Además, se ha implementado una barra de navegación que aparecerá en todos los enlaces para navegar por las diferentes rutas de ***Ele-Task***.

Lista de usuarios

Aquí se pueden ver todos los usuarios registrados con parte de sus datos. Los usuarios que no tengan foto de perfil tendrán un icono predeterminado (como el usuario *darks13*). Además, se han añadido la paginación y la opción de cambiar los usuarios mostradas por página. Debajo de la lista, aparece el link que redirecciona a la página de registro y login.

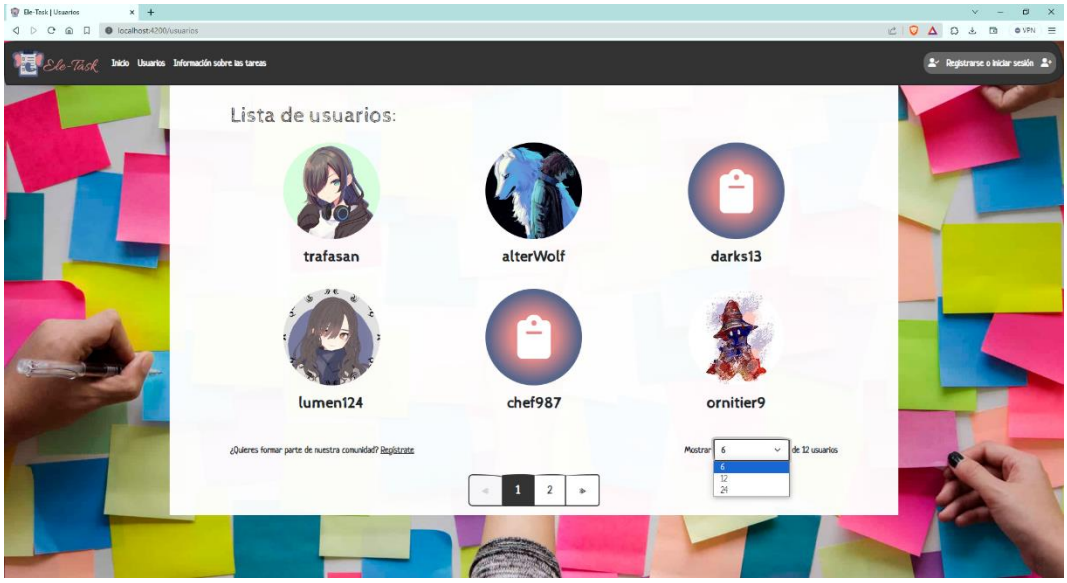


Ilustración 15. Lista de todos los usuarios registrados en Ele-Task.

Información sobre tareas

En este apartado aparece una vista previa de las tareas que puede crear el usuario. Además, se puede consultar el gráfico que cuenta el número de recetas sin realizar, vencidas y realizadas. Por último, se puede encontrar un enlace que redirecciona al apartado *Registrarse o iniciar sesión*.

Ilustración 16. Apartado de Información sobre tareas. Si se mantiene el cursor en el gráfico, se puede consultar el número de tareas en cada sección.

Registro y login

Se puede acceder a este menú desde la barra de navegación y desde los enlaces que se encuentran en el apartado *Información sobre las tareas* y debajo de la lista de usuarios.

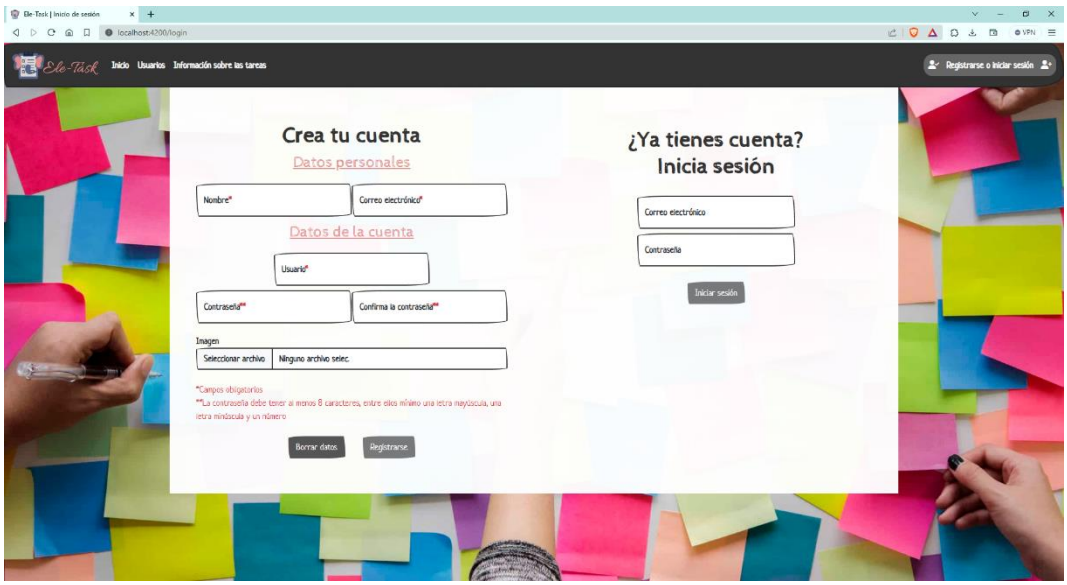


Ilustración 17. Formulario de registro de Ele-Task.

Ilustración 18. Formulario de registro rellenado incorrectamente.

La persona que quiera registrarse en *Ele-Task* debe rellenar al menos los campos obligatorios. Si los rellena mal, saltarán varios errores. Asimismo, si el email ya está registrado y/o el usuario ya existe también saltaran errores. Cuando el formulario se haya rellenado correctamente, se habilitará el botón de registro.

Si la persona se ha equivocado en rellenar los datos, puede borrarlos pulsando el botón “Borrar datos”. Cuando se envía el formulario, la web lo redirecciona.

En este formulario de inicio de sesión, si el usuario pulsa el botón “Iniciar sesión” y al menos uno de los datos no es correcto, aparecerá una alerta. Solo cuando los datos sean válidos, la web le mandará a su menú de usuario.

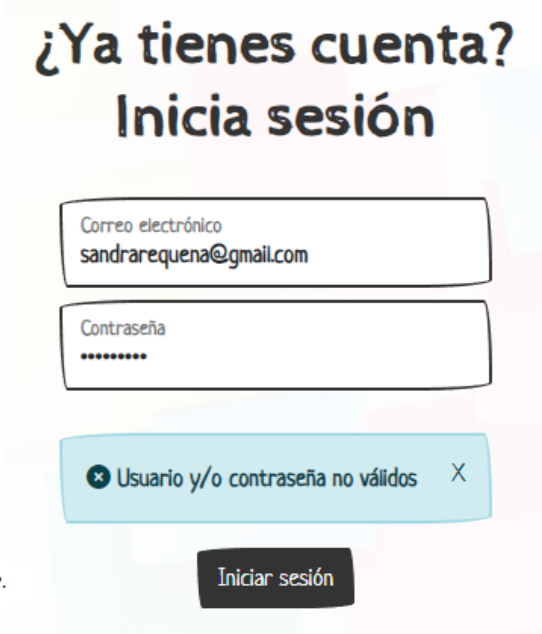


Ilustración 19. Formulario de login rellenado incorrectamente.

Menú de usuario

Cuando el usuario ha iniciado sesión, en la barra de navegación desaparecen los enlaces a los menús de login y registro. En su lugar, aparecerá la foto de perfil del usuario (como es el caso de *trafasan*) o la primera letra de su usuario en caso de no haber guardado foto (como es el caso de *darks13*).

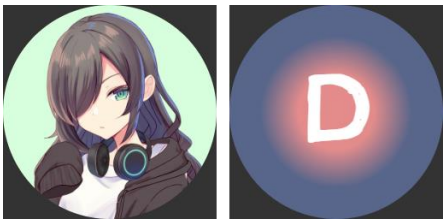
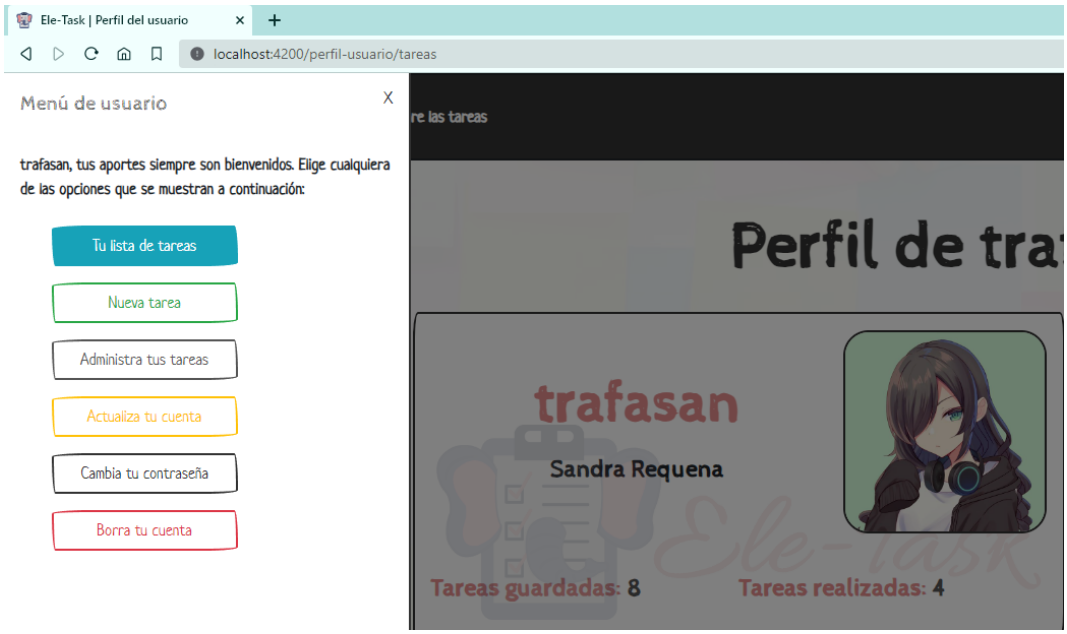


Ilustración 20. Fotos de perfil de trafasan (izquierda) y darks13 (derecha).

En este menú, el usuario puede consultar el número de recetas creadas y realizadas y realizar varias acciones que puede elegir en el menú lateral que aparece cuando se pulsa el botón “¿Quieres hacer algo hoy?”



Cuando se mantiene el cursor en uno de los botones, aparece una ventana con una pequeña explicación de la opción y, si se pulsa, debajo de los botones “¿Quieres hacer algo hoy?” y “Cerrar sesión” aparece el componente seleccionado:

Lista de tareas

Componente seleccionado por defecto. La lista de tareas se divide en dos partes según el estado de realización de estas. El formato de las tareas es el mismo que el visto en el apartado *Información sobre las tareas*. Estas listas aparecen paginadas al igual que la lista de usuarios.

Nueva tarea

Este formulario solo está disponible para los usuarios de ***Ele-Task***, ya que la tarea debe estar vinculada a un usuario. Al igual que los otros formularios, saltarán errores si el formulario se ha rellenado incorrectamente. Cuando se envíe el formulario con los datos correctos, aparecerá un mensaje avisando al usuario de la creación de la tarea.

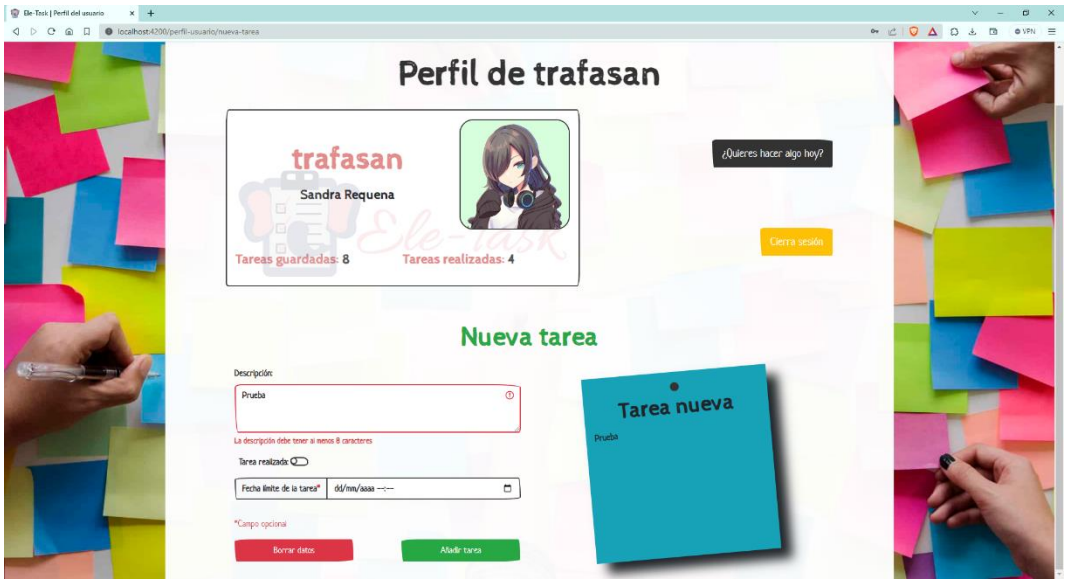
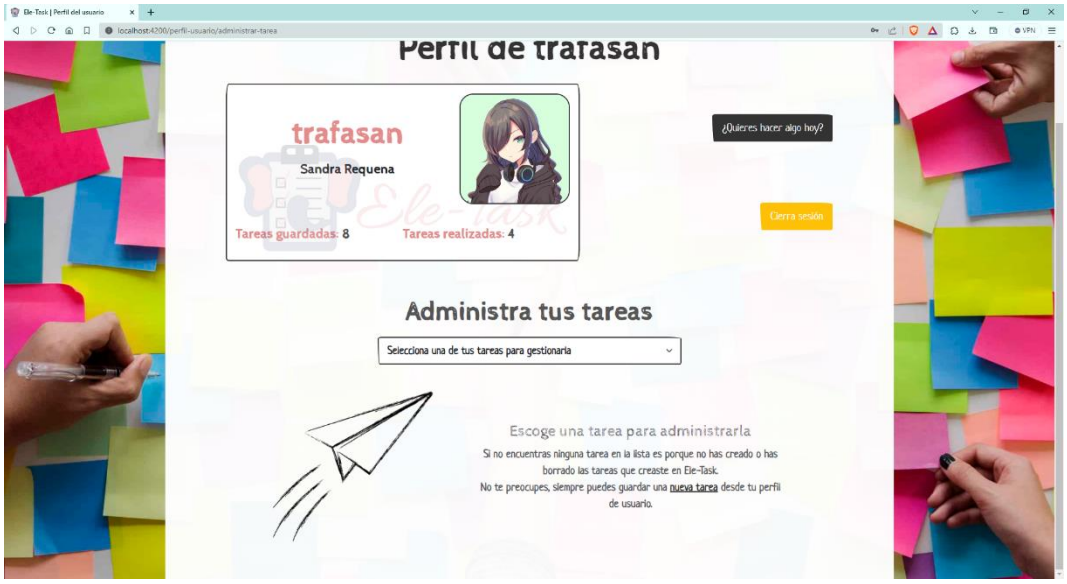


Ilustración 23. Formulario para crear una receta nueva. En este caso, se ha enviado incorrectamente el dato de la descripción.

Administra tus tareas

En este apartado aparece un desplegable con todas las tareas creadas del usuario. Este desplegable se encontrará deshabilitado si el usuario no ha creado ninguna receta.



Al seleccionar una tarea, aparecerá un formulario idéntico al del apartado anterior. Además, también aparece la opción de borrar la tarea.

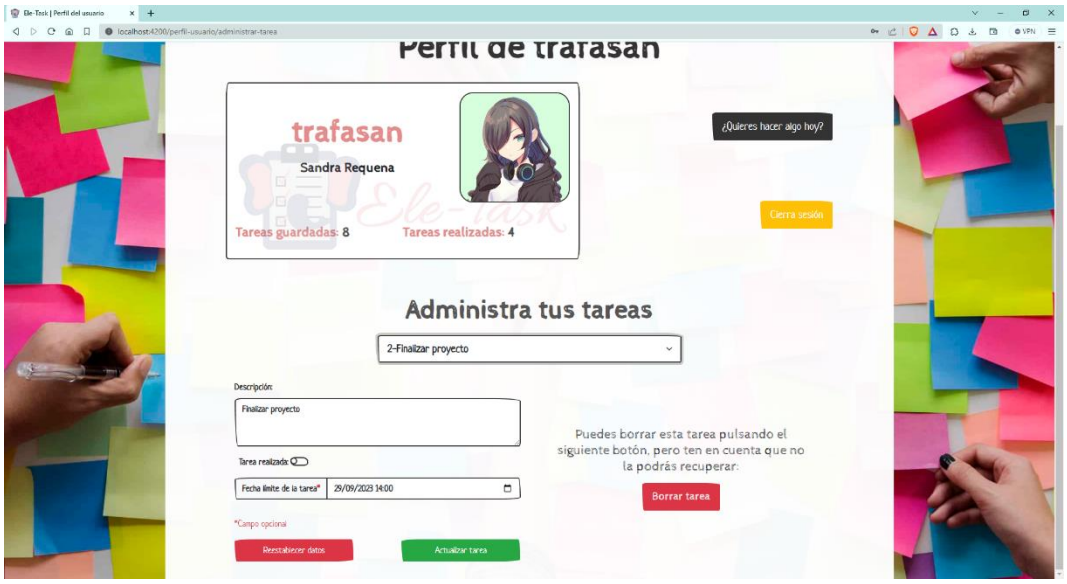


Ilustración 25. Menú de gestión de la tarea nº2.

Actualiza tu cuenta

En este formulario aparecen ya los campos rellenados con los datos actuales del usuario excepto el de la contraseña. Funciona de la misma manera que el formulario de registro en cuanto a errores se refiere. Además, se debe insertar la contraseña actual para poder actualizar los datos correctamente. Nada más enviarlo, los cambios se verán reflejados en el menú de usuario y aparecerá una alerta confirmando que la acción se ha realizado.

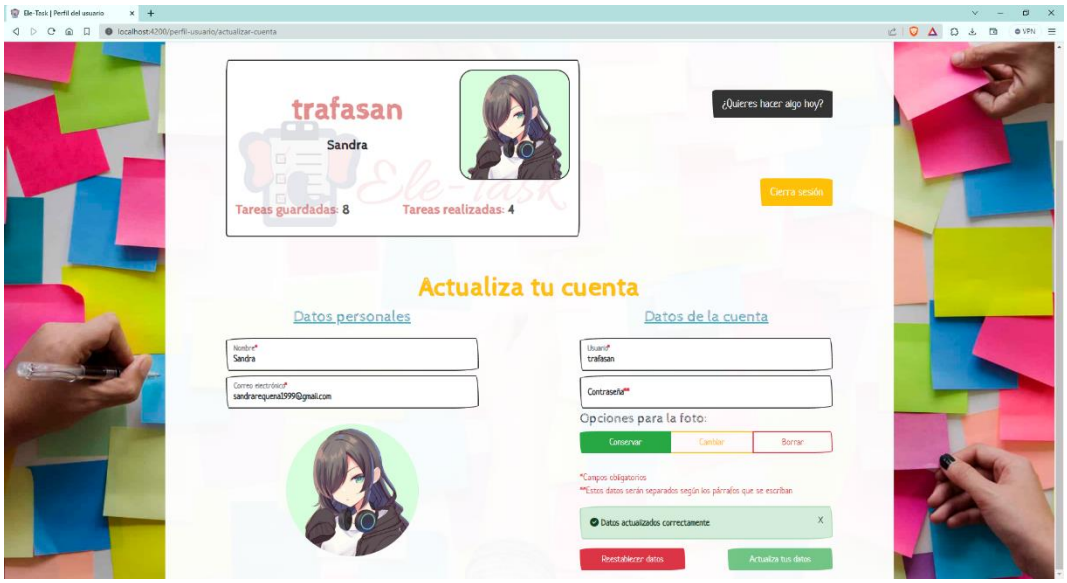
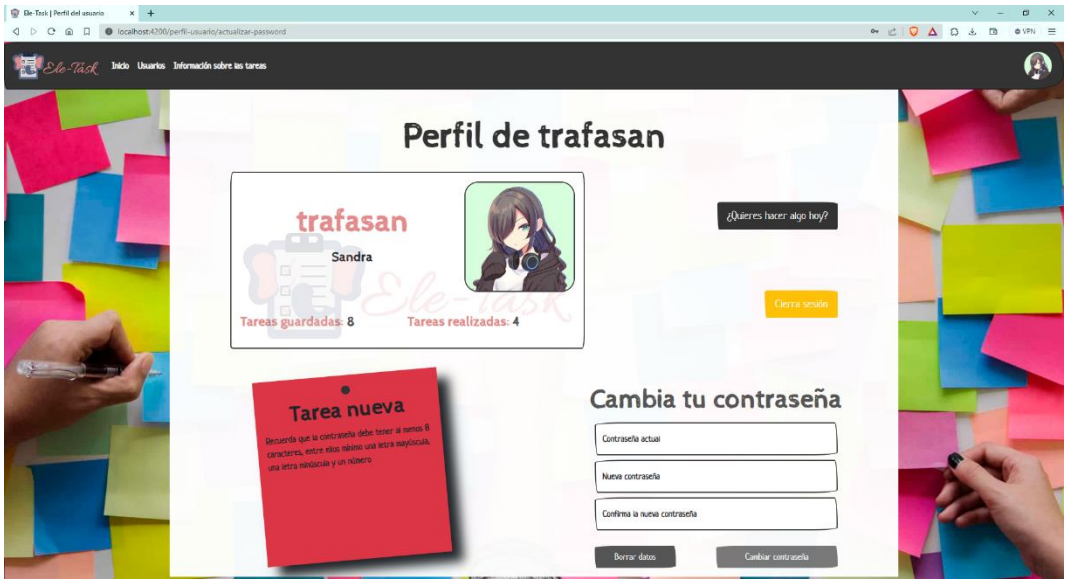


Ilustración 26. Formulario para actualizar la cuenta del usuario. En este caso, se ha enviado el formulario correctamente.

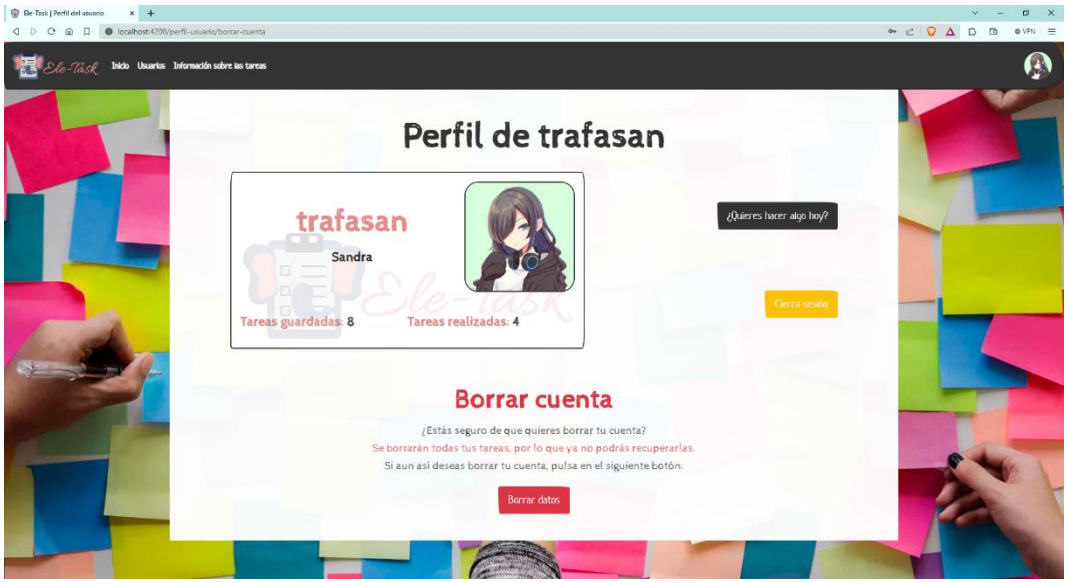
Cambia tu contraseña

Este formulario es específico para actualizar la contraseña. Solo se actualizará cuando la contraseña actual sea la correcta, la nueva contraseña se ajuste al formato y las contraseñas coincidan. Cuando se cumplimente adecuadamente, aparecerá una alerta confirmando que la actualización se ha realizado.



Borra tu cuenta

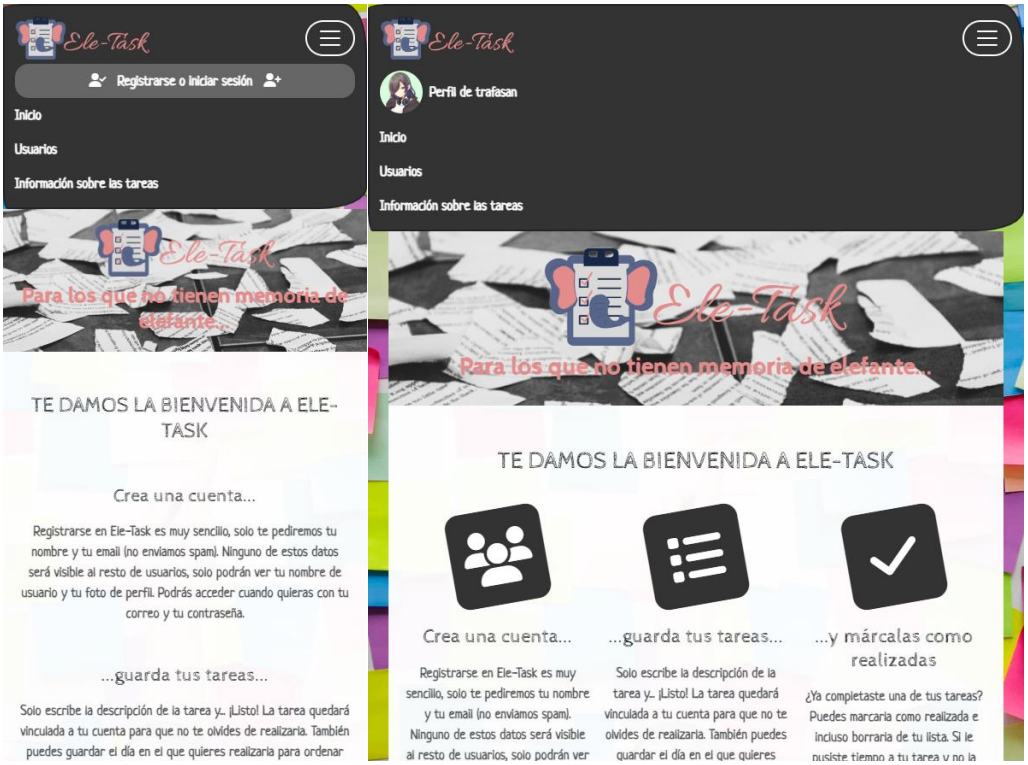
En esta última opción el usuario puede borrar su cuenta junto con las tareas que ha creado en ***Ele-Task***. Si borra su cuenta, la web le redireccionará al menú de inicio.



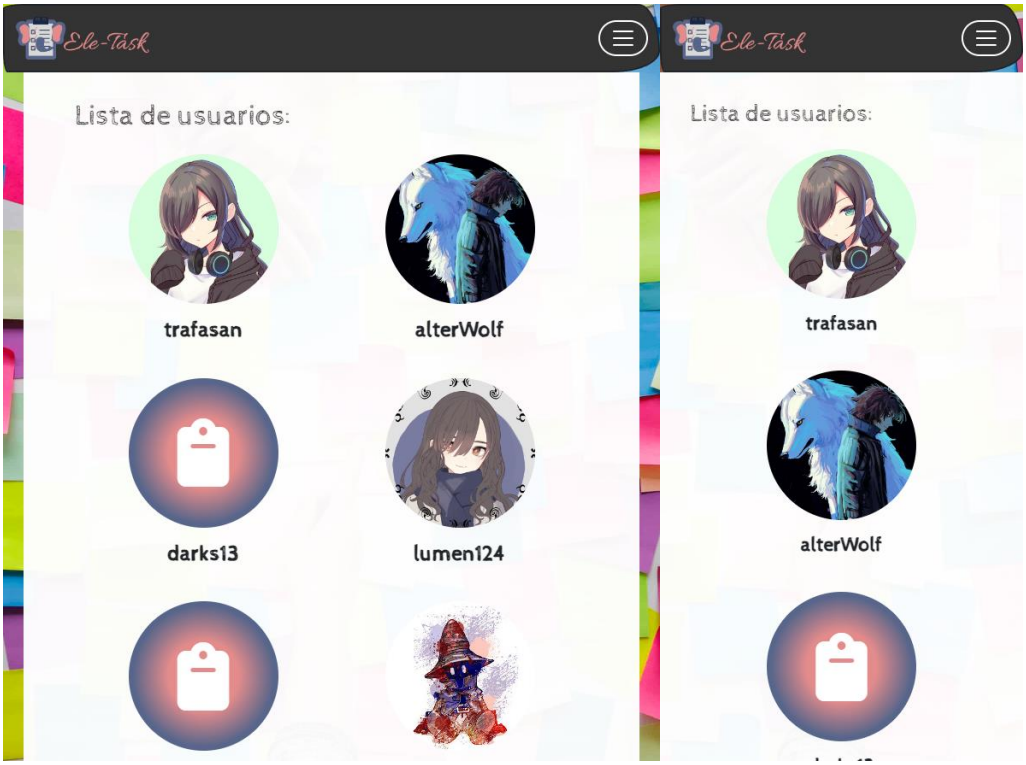
Anexo I: Otros dispositivos

La experiencia de ***Ele-Task*** también se puede vivir en tablet y en móvil. En este apartado se pueden ver algunos menús en formato tablet (768x886) y móvil (425x886).

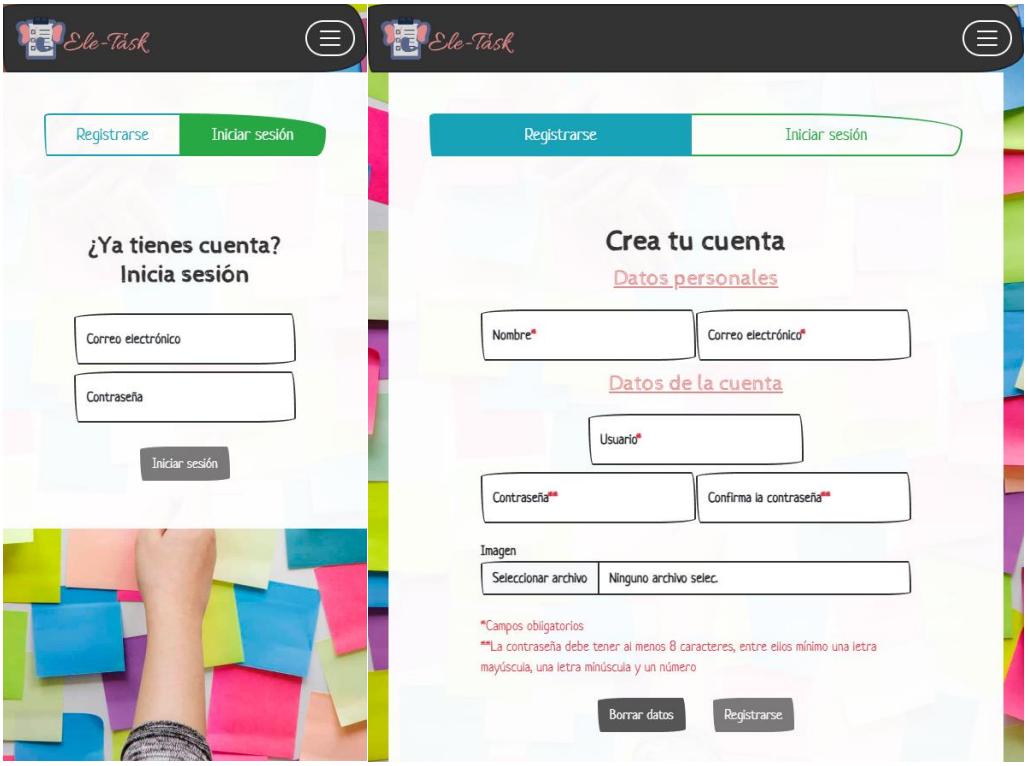
- **Menú de inicio:**



- **Lista de usuarios:**



- **Registro y login:**



Anexo II: Instalaciones

Para hacer funcionar ***Ele-Task*** se deben instalar los siguientes paquetes:

Backend

- pip install flask
- pip install sqlalchemy
- pip install flask-sqlalchemy
- pip install marshmallow
- pip install flask_jwt_extended
- pip install flask-cors
- pip install pillow
- pip install mysql-connector-python
- pip install pytz

Frontend

- npm i bootstrap
- npm i bootswatch@5.3.0
- npm i @ng-bootstrap/ng-bootstrap
- npm i @fortawesome/fontawesome-free
- npm install chart.js