CAPÍTULO 2

ESTRUCTURA DE APLICACIÓN BÁSICA

En este capítulo se aprenderá sobre las diferentes partes de una aplicación Flask. También escribirá y ejecutará su primera aplicación web Flask.

Inicialización

Todas las aplicaciones de Flask deben crear una *instancia de aplicación*. El servidor web pasa todas las solicitudes que recibe de los clientes a este objeto para su manejo, utilizando un protocolo llamado de interfaz de puerta de enlace del servidor (WSGI, pronunciado "wiz-ghee"). La instancia de la aplicación es un objeto de la clase Flask, usualmente creado de la siguiente manera:

```
from flask import Flask
app = Flask(__name__)
```

El único argumento requerido para el constructor de la clase Flask es el nombre del módulo principal o paquete de la aplicación. Para la mayoría de las aplicaciones, la variable **__name__** de Python es el valor correcto para este argumento.

Nota: El argumento __name__ que se pasa al constructor de la aplicación Flask es una fuente de confusión entre los desarrolladores de Flask. Flask utiliza este argumento para determinar la ubicación de la aplicación, que a su vez le permite localizar otros archivos como que forman parte de la aplicación, como imágenes y plantillas.

Luego tú aprenderás más formas complejas para inicializar en una aplicación, pero para aplicaciones simples es todo lo que se necesita

Rutas y Funciones de Visualización

Los clientes, como los navegadores web, envían solicitudes al servidor web, que a su vez las envía a la instancia de la aplicación Flask. La instancia de la aplicación Flask necesita saber qué código necesita ejecutar para cada URL solicitada, por lo que mantiene una asignación de URL a las funciones de Python. La asociación entre la URL y la función que la maneja se denomina **ruta**.
La forma más conveniente de definir una ruta en una aplicación Flask es a través del decorador **app.route** expuesto por la instancia de la aplicación. El siguiente ejemplo muestra cómo se declara una ruta con este decorador:

```
@app.route('/')
def index():
    return '<h1>Hola Mundo!</h1>'
```

Nota: Los decoradores son una característica estándar del lenguaje Python. Un uso común de los decoradores es registrar funciones de controlador que se invocarán cuando ocurran ciertos eventos.

El ejemplo anterior registra la función *index()* como el controlador de la URL raíz de la aplicación. Si bien el decorador **app.route** es el método preferido para registrar funciones de vista, Flask también ofrece una forma más tradicional de configurar las rutas de la aplicación con el método *app.add_url_rule(),* que en su forma más básica toma 3 argumentos: la URL, el nombre del endpoint y la función de vista. El siguiente ejemplo usa *app.add_url_rule()* para registrar una función *index()* que es equivalente a la mostrada anteriormente:

```
def index():
    return '<h1>Hola Mundo!</h1>'

app.add_url_rule('/','index', index)
```

Las funciones como index() que maneja las URL de las aplicaciones se denominan funciones de aplicaciones. Si la aplicación se implementa en un servidor asociado con el nombre del dominio www.example.com, navegar a http://www.example.com en su navegador activaría index() para ejecutarse en el servidor. El valor de retorno de esta función de vista es la respuesta que recibe el cliente. Si el cliente es un navegador web, esta respuesta es el documento que se muestra al usuario en la ventana del navegador. Una respuesta devuelta por una función de vista puede ser una cadena simple con contenido HTML, pero también puede tomar formas más complejas, como se verá más adelante.

Nota: Introducir cadenas de respuestas con código HTML en archivos fuente de Python conduce a un código que es difícil de mantener. Los ejemplos en este capítulo lo hacen solo para introducir el concepto de respuestas. Se aprenderá una mejor manera de generar respuestas HTML en el capítulo 3.

Si presta atención a cómo se forman algunas URL para los servicios que utiliza todos los días, notará que muchas tienen secciones variables. Por ejemplo, la URL de su página de perfil de Facebook tiene el formato https://www.facebook.com/<your-name>, que incluye su nombre de usuario. Flask admite estos tipos de URL utilizando una sintaxis especial de el decorador app.route. El siguiente ejemplo define una ruta que tiene un componente dinámico:

```
@app.route('/user/<name>')
def user(name):
    return '<h1>Hola, {}! </h1>'.format(name)
```

La parte de la URL de la ruta entre los corchetes angulares es la parte dinámica. Cualquier URL que coincida con las porciones estáticas se asignará a esta ruta, y cuando se invoque la función de vista, el componente dinámico pasará como un argumento. En este ejemplo anterior, el argumento de nombre se usa para generar una respuesta que incluye un saludo personalizado.

Los componentes dinámicos en las rutas son cadenas de forma predeterminada, pero también puede ser diferentes tipos. Por ejemplo, la ruta **/user/<int:id>** coincidiría solo con las URL que tengan número entero en el segmento dinámico de identificación, como /user/123. Flask admite los tipos String, int, float y path para rutas. El tipo de ruta **path** es un tipo de cadena especial que puede incluir barras diagonales, a diferencia del tipo de cadena.

Una Aplicación Completa

En las secciones anteriores aprendió sobre las diferentes partes de una aplicación web en Flask y ahora es el momento de escribir la primera. El script de aplicación *hello.py* que se muestra en el ejemplo 2-1 define una instancia de aplicación y una única ruta de función de vista, como se describió anteriormente.

Ejemplo 2-1. hello.py: Una aplicación completa de Flask

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
        return '<h1>Hola Mundo!</h1>'
```

*Nota: si ha clonado el repositorio Git de la aplicación en GitHub, ahora puede ejecutar el **git checkout 2a** para ver esta versión de la aplicación.*

Servidor Web de Desarrollo

Las aplicaciones de Flask incluyen un servidor web de desarrollo que se puede iniciar con el comando de ejecución de Flask. Este comando busca el nombre de script de Python que contiene la instancia de la aplicación en la variable de entorno FLASK_APP.
Par iniciar la aplicación hello.py de la sección anterior, primero asegurar que el entorno este activado y tenga instalado flask.

```
(venv) $ export FLASK_APP = hello.py
(venv) $ flask run
* Serving Flask app "hello"
```

```
* Runnig on http://127.0.0.1:5000/ (Press CTRL + C to quit)
```

Una vez que el servidor se inicia, entra en un bucle que acepta solicitudes y las atiende. Este ciclo continúa hasta que se detenga la aplicación presionando Ctrl + C.
Con el servidor en funcionamiento, abra su navegador web y escriba https://localhost:5000/ en la barra de direcciones. La figura 2-1 muestra lo que verá después de conectarse a la aplicación.

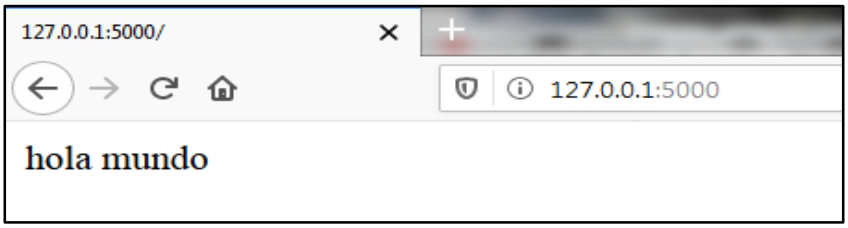

Figure 2-1. hello.py Flask application

Si escribe algo más después de la URL base, la aplicación no sabrá cómo manejarlo y devolverá un código de error 404 al navegador, el error familiar que obtiene cuando navega a una página web que no existe.

Nota: El servidor web proporcionado por Flask está destinado a ser utilizado solo para el desarrollo y pruebas. Aprenderás sobre los servidores web de producción en el capítulo 17.

*Nota: El servidor web desarrollado de Flask también se puede iniciar mediante programación invocando al método **app.run().** Las versiones anteriores de Flask que no tenían el comando de flask requerían que el servidor se inicia ejecutando el script principal de la aplicación, que tenía que incluir el siguiente fragmento final:*

```
if __name__ == “__main__”:
        app.run()
```

*Si bien el comando de ejecución de flask hace que esta práctica sea innecesario, el método **app,run()** puede ser útil en ciertas ocasiones, como pruebas unitarias, como aprenderás en el capítulo 15.*

Rutas Dinámicas

La segunda versión de la aplicación, que se muestra en el Ejemplo 2-2, agrega una segunda ruta que es dinámica. Cuando visitas la URL dinámica en tu navegador, se le presenta con un saludo personalizado que incluye el nombre provisto en la URL.

```
from flask import Flask
app = Flask(__name__)
```

```
@app.route('/')
def index():
        return '<h1>Hola Mundo!</h1>'

@app.route('/user/<name>')
def user(name):
        return '<h1>Hola, {}!</h1>'.format(name)
```

Nota: Si ha clonado el repositorio Git de la aplicación en GitHub, usted ahora puede ejecutar el git checkout 2b para ver esta versión de la aplicación.

Para probar la ruta dinámica, asegúrese de que el servidor se esté ejecutando y luego navegue a *http://localhost:5000/user/Dave*. La aplicación responderá con el saludo personalizado utilizando el argumento dinámico con nombre. Intente usar diferentes nombres en la URL para ver cómo la función de vista siempre genera la respuesta basada en el nombre dado. Un ejemplo se muestra en la Figura 2-2.

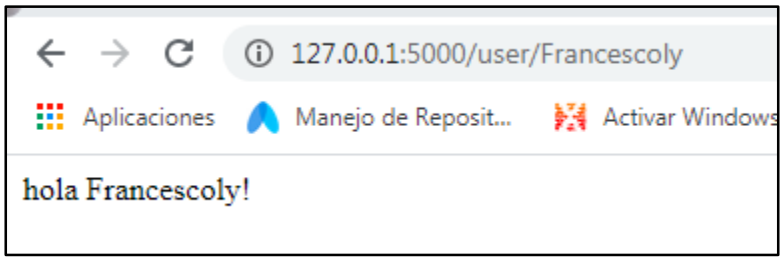

Figura 2-2. Ruta Dinámica.

Modo de Depuración

Las aplicaciones de Flask se pueden ejecutar opcionalmente en modo depuración. En este modo, dos módulos muy convenientes del servidor de desarrollo llamados el recargador y el depurador están habilitados por defecto.

Cuando el re-cargador está habilitado, Flask observa todos los archivos de código fuente de su proyecto y reinicia automáticamente el servidor cuando se modifica alguno de los archivos. Tener un servidor que se ejecuta con el re-cargador habilitado es extremadamente útil durante el desarrollo, porque cada vez que se modifica y guarda un archivo fuente, el servidor se reinicia automáticamente y recoge el cambio.

El depurador es una herramienta basada en la web que aparece en su navegador cuando su aplicación genera una excepción no controlada. La ventana del navegador web se transforma en una traza de pila interactiva que le permite inspeccionar el código fuente y evaluar expresiones en cualquier lugar de la pila de llamadas. Puede ver como se ve el depurador en la Figura 2-3.

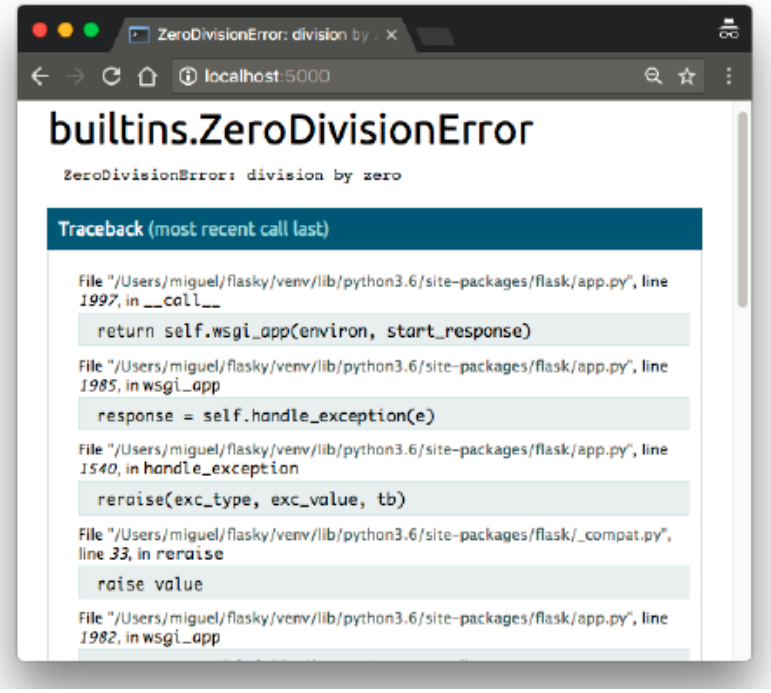

Figure 2-3. Flask debugger

Por defecto, el modo depurador está deshabilitado. Para habilitarlo, establezca una variable de entorno FLASK_APP = 1 antes de invocar la ejecución de flask:

```
(venv) $ export FLASK_APP=hello.py
(venv) $ export FLASK_DEBUG=1
(venv) $ flask run

* Serving Flask app “hello”
* Forcing debug mode on
* Runnig on http://127.0.0.1:5000/
* Restarting with stat
* Debugger is active!
* Debugger PIN: 273-181-528
```

nota: ***Si está utilizando Microsoft Windows, es set en lugar de export para configurar las variables de entorno.***

Nota: Si inicia su servidor con el método app.run(), el FLASK_APP y las variables de entorno FLASK_DEBUG no se utilizan. Para habilitar el modo de depuración mediante programación, use app.run (debug= True).

Nota: Nunca active el modo de depuración en un servidor de producción. El depurador en particular permite que el cliente solicite la ejecución remota de código, por lo que hace que

su servidor de producción sea vulnerable a los ataques. Como medida de protección simple, el depurador debe activarse con un PIN, impreso en la consola mediante el comando de ejecución de flask.

Opciones de Línea de Comando

El comando Flask admite una serie de opciones. Para ver qué hay disponible, puede ejecutar flask --help o simplemente *flask* sin ningún argumento:

```
(venv) $ flask --help
Usage: flask [options] COMMAND [ARGS]...
```

Un script de utilidad general para aplicaciones de Flask. Proporciona comandos para Flask, extensiones y la aplicación. Carga la aplicación definida en la variable de entorno FLASK_APP, o desde un archivo wsgi.py. Establecer la variable de entorno FLASK_ENV en “development” habilitará el modo de depuración.

```
> set FLASK_APP=hello.py
> set FLASK_DEBUG=1
> flask run
```

options:

--version	Muestra la versión de Flask.
--help	Muestra los mensajes y salidas.

Commands:

routes	Muestra las rutas para la aplicación.
run	Ejecuta el servidor de desarrollo.
shell	Ejecuta un shell en el contexto de la aplicación.

El comando de shell flask se usa para iniciar una sesión de shell Python en el contexto de la aplicación. Puede usar esta sesión para ejecutar tareas de mantenimiento o pruebas, o para depurar problemas. Los ejemplos reales donde este comando es útil se presentarán más adelante, en varios capítulos.

Ya está familiarizado con el comando de ejecución de flask, que, como su nombre indica, ejecuta la aplicación con el servidor web de desarrollo. Este comando tiene varias opciones:

```
(myvenv) > flask run --help
Usage: flask run [OPTIONS]

Ejecuta un servidor de desarrollo local para la aplicación Flask.
```

Este servidor local se recomienda sólo para fines de desarrollo, pero también se puede usar para implementaciones de intranet simples. Por defecto, no admitirá ningún tipo de concurrencia para simplificar la depuración. Esto se puede cambiar con la opción *--with-threads* que habilitará el subprocesamiento múltiple básico.

El re cargador y el depurador están habilitados de forma predeterminada si el indicador de depuración de Flask está habilitado y deshabilitado de lo contrario.

Opciones:

-h, --host TEXT	La interface a la cual vincularse.
-p, --port INTEGER	El puerto al cual unirse.
--reload / --no-reload	Habilitar o deshabilitar el cargador. De forma predeterminada, el cargador está activado si la depuración está habilitada.
--debugger /--no-debugger	Habilita o deshabilita el depurador. De forma predeterminada, el depurador está activo si la depuración está habilitada.
--eager-loading / --lazy-loader	Habilita o deshabilita la carga ansiosa. Por defecto, la carga ansiosa está habilitada si el re-cargador está deshabilitado.
--with-threads / --without-threads	Habilita o deshabilita subprocesos múltiples.
--help	Muestra los mensajes y salida.

El argumento *--host* es particularmente útil porque le dice al servidor web qué interfaz de red debe escuchar para las conexiones de los clientes. De forma predeterminada, el servidor web de desarrollo de Flask escucha las conexiones en el host local, por lo que solo se aceptan las conexiones que se originan en la computadora que ejecuta el servidor. El siguiente comando hace que el servidor web escuche las conexiones en la interfaz de la red pública, permitiendo que otras computadoras en la misma red también se conecten:

(myvenv) > **flask run --host 0.0.0.0**

- Serving Flask app “”
- Serving Flask app "hello"
- Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

Ahora se puede acceder al servidor web desde cualquier computadora de la red en http://a.b.c.d:5000, donde a.b.c.d. es la dirección IP de la computadora que ejecuta el servidor en tu red.

Las opciones --reload, --no-reload, --debugger y --no-debugger proporcionan un mayor grado de control además de la configuración de depuración. Por ejemplo, si el modo de depuración está habilitado, --no--debugger puede usarse para apagar el depurador, mientras se mantiene el modo de depuración y el re-cargador habilitado.

El Ciclo de Solicitud - Respuesta

Ahora que ha jugado con la aplicación básica de Flask, es posible que desee saber más sobre cómo Flask hace su magia. Las siguientes secciones describen algunos de los aspectos de diseño del marco.

Aplicaciones y Solicitar Contextos

Cuando Flask recibe una solicitud de un cliente, necesita tener algunos objetos disponibles para la función de vista que lo maneja. Un buen ejemplo es el objeto de solicitud, que encapsula la solicitud HTTP enviada por el cliente.

La forma obvia en la que Flask podría dar acceso a una función de vista al objeto de solicitud es enviándolo como argumento, pero eso requeriría que cada función de vista en la aplicación tenga un argumento adicional. Las cosas se vuelven más complicadas si considera que el objeto de solicitud no es un único objeto que las funciones de visualización pueden necesitar para acceder a una solicitud.

Para evitar el desorden en las funciones de vista con muchos argumentos que no siempre son necesarios, Flask usa contextos para hacer que ciertos objetos sean accesibles globalmente temporalmente. Gracias a los contextos, se pueden escribir funciones de visualización como la siguiente:

```
from flask import request

@app.route(´/´)
def index():
        user_agent = request.headers.get(‘User-Agent’)
        return ‘<p>Your browser is {} </p>’.format(user_agent)
```

Observe cómo en esta función de vista, la solicitud se usa como si fuera una variable global. En realidad, la solicitud no puede ser una variable global; es un servidor multiproceso, varios subprocesos pueden estar trabajando en diferentes solicitudes de diferentes clientes al mismo tiempo, por lo que cada subproceso necesita ver un objeto diferente en la solicitud. Los contextos permiten a Flask hacer que ciertas variables sean accesibles globalmente para un hilo sin interferir con los otros hilos.

Nota: Un hilo es la secuencia más pequeña de instrucciones que se pueden administrar de forma independiente. Es común que un proceso tenga múltiples subprocesos activos, a veces compartiendo recursos como memoria o identificadores de archivos. Los servidores web multiproceso inician un grupo de subprocesos y seleccionan un subproceso del grupo para manejar cada solicitud entrante.

Hay dos contextos en Flask: el contexto de aplicación y el contexto de la solicitud. La tabla 2-1 muestra las variables expuestas por cada uno de estos contextos.

Table 2-1. Contextos Globales de Flask.

Nombre de Variable	**Contexto**	**Descripción**
current_app	Contexto de Aplicación	La instancia de la aplicación para la aplicación activa.
g	Contexto de Aplicación	Un objeto de la misma aplicación puede usar para el almacenamiento temporal durante el manejo de una solicitud. Esta variable se restablece con cada solicitud.
request	Contexto de Solicitud	El objeto de solicitud, que encapsula el contenido de una solicitud HTTP enviada por el cliente.
session	Contexto de Solicitud	La sesión del usuario, un diccionario de la misma aplicación puede usar para almacenar valores que se “recuerdan” entre solicitudes.

Flask activa (o empuja) los contextos de aplicación y solicitud antes de enviar una solicitud a la aplicación, y los elimina después de que se maneja la solicitud. Cuando se empuja el contexto de la aplicación, las variables ***current_app*** y **g** están disponibles. Si se accede a cualquiera de estas variables sin una aplicación activa o un contexto de solicitud, se genera un error. Las 4 variables de contexto se tratarán en detalle en este y los capítulos posteriores, así que no se preocupes si aún no comprende por qué útiles.

La siguiente sesión de shell de Python demuestra cómo funciona el contexto de la aplicación:

```
>>>> from hello import app
>>>> from flask import current_app
>>>> current_app.name
Traceback  (most recent call last):
...
RuntimeError: working outside of application context:
```

>>>> **app_ctx = app.app.context()**
>>>> **app_ctx.push()**
>>>> **current_app.name**
'hello'
>>>> **app.ctx.pop()**

En este ejemplo, *current_app.name* falla cuando no hay un contexto de aplicativo activo, pero se vuelve válido una vez que se empuja un contexto de aplicación para la aplicación. Observe cómo se obtiene el contexto de una aplicación invocando *app.app_context()* en la instancia de la aplicación.

Despacho de Solicitudes

Cuando la aplicación recibe una solicitud de un cliente, necesita averiguar qué función de vista invocar para atenderla. Para esta tarea, Flask busca la URL que figura en la solicitud en el mapa de URL de la aplicación, que contiene una asignación de URL a las funciones de visualización que las manejan. Flask crea este mapa utilizando los datos proporcionados en el decorador *app.route*, o la versión equivalente sin decorador, *app.add_url_rule().*

Para ver cómo se ve el mapa de URL en una aplicación Flask, puede inspeccionar el mapa creado para hello.py en el shell de Python. Antes de intentar esto, asegúrese de que su entorno virtual esté activado.

(myvenv) > **python**
>>> **from index import app**
>>> **app.url_map**
Map([<Rule '/about' (HEAD, GET, OPTIONS) -> about>,
<Rule '/' (HEAD, GET, OPTIONS) -> home>,
<Rule '/static/<filename>' (HEAD, GET, OPTIONS) -> static>,
<Rule '/user/<name>' (HEAD, GET, OPTIONS) -> user>])

```
(myvenv) C:\Users\PEDRITO\Libro Flask\my_flasky>python
Python 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from hello import app
>>> app.url_map
Map([<Rule '/' (OPTIONS, GET, HEAD) -> index>,
 <Rule '/static/<filename>' (OPTIONS, GET, HEAD) -> static>,
 <Rule '/user/<name>' (OPTIONS, GET, HEAD) -> user>])
>>>
```

Las rutas **/** y **/user<name>** fueron definidas por los decoradores *app.route* en la aplicación. La ruta */static/<filename>* es una ruta especial agregada por Flask para dar acceso a archivos estáticos. Aprenderá más sobre los archivos estáticos en el Capítulo 3.

Los elementos (HEAD, OPTIONS, GET) que se muestran en el mapa de URL son los métodos de solicitud que manejan las rutas. Las especificaciones HTTP define que todas

las solicitudes se emiten con un método, que normalmente indica qué acción está pidiendo el cliente que realice el servidor. Flask asocia métodos a cada ruta para que diferentes funciones de vista puedan manejar diferentes métodos de solicitud enviados a la misma URL. Los métodos HEAD y OPTIONS son administrados automáticamente por Flask, por lo que en la práctica se puede decir que en esta aplicación las tres rutas en el mapa de URL se adjuntan al método GET, que se utiliza cuando el cliente desea solicitar información como una página web. Aprenderá a crear rutas para otros métodos de solicitud en la Capítulo 4.

El objeto de la Solicitud

Ha visto que Flask expone un objeto de solicitud como una variable de contexto denominada solicitud. Este es un objeto extremadamente útil que contiene toda la información que el cliente incluyó en la solicitud HTTP. La tabla 2-2 enumera los atributos y métodos más utilizados del objeto de solicitud Flask.

Tabla 2-2 Objeto de solicitud Flask

Atributo o Método	Descripción
form	Un diccionario con todos los campos de formulario enviados con la solicitud.
args	Un diccionario con todos los argumentos pasados en la cadena de consulta de la URL.
values	Un diccionario que combina los valores de formulario y argumentos.
cookies	Un diccionario con todas las cookies incluidas en la solicitud.
headers	Un diccionario con todos los encabezados HTTP incluidos en la solicitud.
files	Un diccionario con todas las cargas de archivos incluidas con la solicitud.
get_data()	Devuelve los datos almacenados en el búfer del cuerpo de la solicitud.
get_json()	Devuelve un diccionario de Python con el JSON analizado incluido en el cuerpo de la solicitud.
blueprint	EL nombre del Flask blueprint que maneja la solicitud. Los planos se presentan en el Capítulo 7.
endpoint	EL nombre del punto final de Flask que maneja la solicitud. Flask usa el nombre de la función de vista como el nombre del punto final para una ruta.

method	El método de solicitud HTTP, como GET o POST
scheme	El esquema de URL (http p https).
is_secure()	Devuelve True si la solicitud se realizó a través de una conexión segura (HTTPS)
host	El host definido en la solicitud, incluido el número de puerto si lo proporciona el cliente.
path	La porción de ruta de la URL.
query_string	La parte de la cadena de consulta de la URL, como un valor binario sin procesar.
full_path	La ruta y las porciones de cadena de consulta de la URL.
url	La URL completa solicitada por el cliente.
base_url	Igual que url, pero sin el componente de cadena de consulta.
remote_addr	La dirección IP del cliente.
environ	El diccionario de entorno WSGI sin procesar para la solicitud.

Request Hooks (solicitar ganchos)

A veces es útil ejecutar código antes o después de que se procese cada solicitud. Por ejemplo, al comienzo de cada solicitud puede ser necesario crear una conexión de base de datos o autenticar al usuario que realiza la solicitud. En lugar de duplicar el código que realiza estas acciones en cada función de vista, Flask le ofrece la opción de registrar funciones comunes que se invocarán antes o después de que se envíe una solicitud.

Los ganchos de solicitud se implementan como decoradores. Estos son cuatro ganchos compatibles con Flask:

- before_request: Registra una función para ejecutar antes de cada solicitud.

- before_first_request: Registra una función para ejecutarse solo antes de que se maneje la primera solicitud. Esta puede ser una forma conveniente de agregar tareas de inicialización del servidor.

- after_request: Registra una función para ejecutarse después de cada solicitud, pero solo si no se produjeron excepciones no controladas.

- teardown_request: Registra una función para ejecutarse después de cada solicitud, incluso si se produjeron excepciones no controladas.

Un patrón común para compartir datos entre las funciones de enlace de solicitud y las funciones de visualización es usar el contexto g global como almacenamiento. Por ejemplo, un controlador *before_request* puede cargar el usuario conectado desde la base de datos y almacenarlo en *g.user*. Más tarde, cuando se invoca la función de vista, puede recuperarlo al usuario desde allí.

Se mostrarán ejemplos de ganchos de solicitud en capítulos futuros, así que no se preocupe si el propósito de estos ganchos todavía no tiene sentido.

Responses (Respuestas)

Cuando Flask invoca una función de vista, espera que su valor de retorno sea la respuesta a la solicitud. En la mayoría de los casos, la respuesta es una cadena simple que se envía de vuelta al cliente como una página HTML.

Pero el protocolo HTTP requiere más que una cadena como respuesta a una solicitud. Una parte muy importante de la respuesta HTTP es el código de estado, que Flask establece por defecto en 200, el código que indica que la solicitud se realizó con éxito.

Cuando una función de vista necesita responder con un código de estado diferente, puede agregar el código numérico como un segundo valor de retorno después del texto de respuesta. Por ejemplo, la siguiente función de vista devuelve un código de estado 400, el código para un error de solicitud incorrecta:

```
@app.route('/')
def index():
    return '<h1>Bad request</h1>', 400
```

Las respuestas devueltas por las funciones de vista también pueden tomar un tercer argumento, un diccionario de encabezados que se agregan a la respuesta HTTP. Verá un ejemplo de encabezados de respuestas personalizados en el Capítulo 14.

En lugar de devolver uno, dos o tres valores como una tupla, las funciones de vista Flask tienen la opción de devolver un objeto de respuesta. La función ***make_response()*** toma uno, dos o tres argumentos, los mismos valores que pueden devolverse desde una función vista, y devuelve un objeto de respuesta equivalente. A veces es útil generar el objeto de respuesta dentro de la función de vista y luego usar sus métodos para configurar aún más la respuesta. El siguiente ejemplo crea un objeto de respuesta y luego establece una cookie en él:

```
from flask import make_response

@app.route('/')
def index():
    response = make_response('<h1>Este documento lleva una cookie</h1>')
    response.set_cookie('answer', '42')
    return response
```

Tabla 2-3 Muestra los atributos y métodos más utilizados disponibles en los objetos de respuesta.

Tabla 2-3 Objeto de respuesta Flask

Atributo o Método	Descripción
status_code	El código numérico de estado HTTP
headers	Un objeto tipo diccionario con todos los encabezados que se enviarán con la respuesta.
set_cookie()	Agrega una cookie a la respuesta.
delete_cookie()	Remueve las cookies
content_length	La longitud del cuerpo de respuesta
content_type	El Tipo media del cuerpo de respuesta
set_data()	Establece el cuerpo de las respuesta como un valor de cadena o bytes
get_data	Obtiene el cuerpo de respuesta

Hay un tipo especial de respuesta llamada redireccionamiento. Esta respuesta no incluye un documento de página; solo le da al navegador una nueva URL para navegar. Un uso muy común de los redireccionamientos es cuando se trabaja con formularios web, como aprenderá en el capítulo 4.

Una redirección generalmente se indica con un código de estado de respuesta 302 y la URL a la que indica en un encabezado de ubicación. Se puede generar una respuesta de redireccionamiento manualmente con retorno de tres valores o con un objeto de respuesta, pero dado su uso frecuente, Flask proporciona una función auxiliar *redirect()* que crea este tipo de respuesta:

```
from flask import redirect
@app.route('/')
def index():
    return redirect('http://www.google.com')
```

Se emite otra respuesta especial con la función *abort(),* que se utiliza para el manejo de errores. El siguiente ejemplo devuelve el código de estado 404 si el argumento dinámico de *id* proporcionado en la URL no representa un usuario válido.

```
from flask import abort

@app.route('/user/<id>')
def get_user(id):
        user = load_user(id)
        if not user:
                abort(404)
        return '<h1>hello {} </h1>'.format(user.name)
```

Tenga en cuenta que *abort()* no devuelve el control a la función porque genera un excepción.

Flask Extensions (Extensiones de flask)

Flask está diseñado para extenderse. Se mantiene intencionalmente fuera de las areas funcionales importantes, como la base de datos y la autentificación de usuarios, lo que le da la libertad de seleccionar los paquetes que mejor se adapten a su aplicación, o de escribir los suyos si así lo desea. La comunidad ha creado una gran variedad de extensiones de Flask para muchos propósitos diferentes, y si eso no es suficiente, también se puede usar cualquier paquete o biblioteca estándar de Python. Utilizará su primera extensión de Flask en el Capítulo 3.

Este capítulo introdujo el concepto de respuestas a las solicitudes, pero hay mucho más que decir sobre las respuestas. Flask proporciona muy buen soporte para generar respuestas usando plantillas, y este es un tema tan importante que el próximo capítulo está dedicado a ello.