

Sviluppo di Applicazioni Web con Flask

Programmazione di Applicazioni Data Intensive

Laurea in Ingegneria e Scienze Informatiche

DISI – Università di Bologna, Cesena

Proff. Gianluca Moro, Roberto Pasolini

nome.cognome@unibo.it



Outline

- Introduzione alle applicazioni Web
- Standard usati nel Web (HTTP, HTML, ...)
- Sviluppo Web lato server
- Basi di Flask: routing ed esecuzione applicazione
- Introduzione ai template
- URL applicazione e file statici
- Funzionalità principali dei template
- Invio di dati tramite form
- Gestione dati di richieste e sessioni
- Deployment



Applicazioni Web

- Un'**applicazione web** è un servizio fruibile **attraverso un browser web** con elaborazione lato browser e/o lato server
- Qualsiasi utente, eventualmente autenticato, può usufruire dell'applicazione conoscendone il relativo URL
- L'applicazione si basa su **tecnologie standard W3C** implementate dai browser e utilizzate nel World Wide Web
 - protocollo **HTTP** per lo scambio di informazioni (richiesta-risposta)
 - linguaggi **HTML e CSS** per la presentazione
 - linguaggio **JavaScript** per l'interattività e la logica dell'applicazione
 - formato **XML** (o alternative, es. JSON) per la rappresentazione di documenti e metadati



Vantaggi delle Applicazioni Web

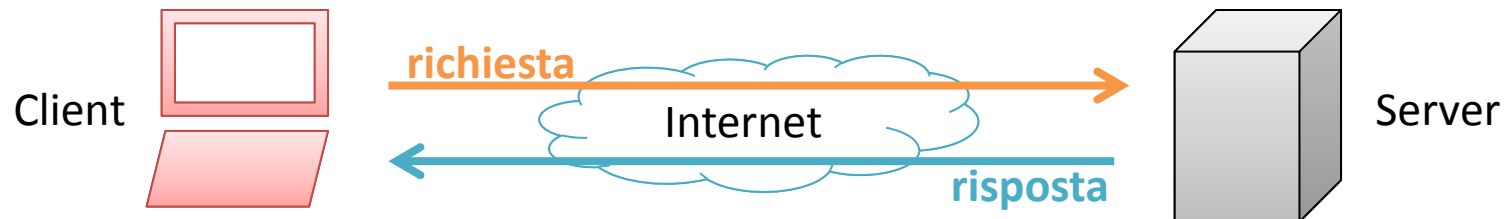
Rispetto ad applicazioni tradizionali installate ed eseguite sul proprio dispositivo (es. app per smartphome)...

- Un'applicazione Web si scrive **una volta per tutte le piattaforme** utilizzando **tecnologie standard**
 - gli standard (es. HTML5) stanno sempre più fornendo funzionalità in precedenza specifiche per ogni S.O. (localizzazione, notifiche ecc.)
- Si può usare da **qualsiasi dispositivo** dotato di un browser
 - PC, smartphome, tablet, smart TV ecc.
- L'avvio iniziale e gli aggiornamenti dell'applicazione **non richiedono installazione** o altri interventi sui client



Modello Client-Server e HTTP

- Le applicazioni web si basano sul **modello client-server**
 - un **server** ospita una o più applicazioni
 - molteplici **client** si connettono al server per utilizzarne le applicazioni
- La comunicazione avviene attraverso **HTTP** (*HyperText Transfer Protocol*), basato su scambio di richieste e risposte
 - il client invia una **richiesta** al server per una risorsa (es. una pagina)
 - il server **risponde** inviando la risorsa o un errore (es. “404 Not found”)
- HTTP è **stateless**: ogni richiesta è indipendente dalle precedenti poiché non mantiene lo stato della connessione
 - nella pratica, si usano le *sessioni* per mantenere memoria dello stato



HTTP: Richieste

- Una richiesta HTTP è diretta da un certo URL, ad esempio:

`https://io1.unibo.it/course/view.php?id=18284`

Il protocollo è indicato all'inizio dell'URL (HTTPS = HTTP Sicuro, connessione cifrata)

L'host identifica il web server da contattare

Il *percorso* identifica l'applicazione a cui accedere e/o una specifica risorsa al suo interno

- Ad ogni richiesta è associato un *metodo*, che indica **l'azione da compiere** sulla risorsa identificata dall'URL
 - GET** è il metodo usato comunemente dal browser per **reperire il contenuto** di una pagina (statica o generata dinamicamente)
 - POST** si usa per **inviare dati** (es. da un form) alla webapp, causandone potenzialmente un **cambiamento nello stato** (es. pubblicare una foto)
- Ad es., digitando l'URL sopra, il browser invia al server `io1.unibo.it` la richiesta **"GET /course/view.php?id=18284"**



HTTP: Risposte

- La risposta del server ad una richiesta contiene un *codice* di 3 cifre e un messaggio breve che ne indicano l'esito, ad es.
 - 200 OK: la richiesta è stata accettata
 - 404 Not Found: il percorso indicato nella richiesta non è valido
 - 500 Internal Server Error: c'è un problema lato server
- Se atteso, viene riportato anche il *contenuto* della risposta
 - ad es. se la richiesta era una GET ed è riuscita (codice 200)
- Sia richiesta che risposta contengono spesso degli *header* che riportano alcuni dettagli su di esse
 - dimensione e tipo del contenuto (HTML, JSON, immagine, ...), per quanto può essere mantenuto in una cache, dati di autenticazione, ...



Tecnologie Lato Client

- Esistono diversi linguaggi interpretati dal browser (client) che permettono la fruizione di contenuti e servizi del WWW
- **HTML** (*HyperText Markup Language*) è usato per strutturare e definire il **contenuto** di una pagina
 - può includere anche altri aspetti (stile di presentazione ecc.), che però sono per buona prassi gestiti in file separati
- **CSS** (*Cascading Style Sheet*) si usa per definire lo **stile** di presentazione e il **layout** della pagina
- **JavaScript** permette di aggiungere **interattività** alla pagina e fornisce **capacità di calcolo** client-side
 - **AJAX** (*Asynchronous JavaScript and XML*) prevede l'uso di JavaScript per aggiornare in modo asincrono singole parti di una pagina




Esempio: Una Semplice Pagina HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>My webapp</title>
    <link rel="stylesheet" type="text/css" href="style.css">
  </head>
  <body>
    <h1>My webapp</h1>
    <p>Hi, user!</p>
    <p><a href="http://www.unibo.it/">
      Go to Unibo.it
    </a></p>
  </body>
</html>
```

Nell'head sono dichiarate alcune generalità della pagina, es. il titolo e l'importazione di file esterni (in questo caso un foglio di stile CSS)

Nel body c'è il contenuto effettivo della pagina, in questo caso un'intestazione di primo livello (h1), un paragrafo (p) di testo semplice e un altro paragrafo con un link (a)

Così è come la pagina appare nel browser se il file style.css è assente



Esempio: Un Semplice Foglio di Stile CSS

```
body {  
  background-color: #7777ff;  
  font-family: serif;  
}
```

```
h1 {  
  color: blue;  
  align: center;  
}
```

```
a {  
  text-decoration: none;  
}
```

Ogni sezione si riferisce ad un tipo di tag HTML; le dichiarazioni in body (colore sfondo e tipo di carattere) hanno effetto sull'intera pagina; le dichiarazioni su h1 (colore e allineamento testo) e a (rimozione sottolineatura) hanno effetto su quei tag

Questa è la pagina di prima con applicato questo foglio di stile



Elaborazione Lato Server

- I siti web più semplici (“Web 1.0”) sono *statici*
 - il browser riceve direttamente i file HTML e altri contenuti memorizzati sul server, gestiti dagli amministratori del sito
- La maggior parte dei siti moderni (“Web 2.0”) sono *dinamici*
 - i contenuti sono generati **con elaborazioni lato server e/o client**
 - **gli utenti forniscono dei contenuti** mediante piattaforme di blogging, social network ecc. che le elaborano e le pubblicano
- Nei siti dinamici, il server deve **generare dinamicamente una pagina di risposta** ad ogni richiesta
 - più risposte alla stessa richiesta possono essere differenti, poiché lo stato dell’applicazione può influenzare il risultato
 - ad es. la pagina di una discussione su un forum mostra sempre più messaggi man mano che questi sono pubblicati dagli utenti



Esempi di Tecnologie Lato Server

- Nel tempo si sono diffuse diverse tecnologie per l'implementazione lato server di applicazioni Web
- **PHP** (*PHP: Hypertext Preprocessor*) è stato tra i primi linguaggi dedicati per lo sviluppo server-side ed è usato ancora oggi
 - un file PHP è costituito da un mix di contenuto HTML statico e di codice eseguito lato server che genera il contenuto dinamico
- In Java si utilizza la ***Servlet API*** come tecnologia di base per la generazione di contenuto Web dinamico
 - i file **JSP** (*JavaServer Pages*) mixano contenuto statico e codice Java eseguito lato server in modo simile a PHP e sono compilati in servlet
- Più recente, ***Node.js*** è un interprete JavaScript lato server, che permette di usare lo stesso linguaggio già diffuso lato client



Applicazioni Web in Python

- Anche Python è tra i linguaggi più usati per lo sviluppo Web
- Python è particolarmente indicato per la **realizzazione rapida di piccole applicazioni** e di prototipi
 - sfruttando la grande quantità di librerie disponibili per Python, si possono creare rapidamente applicazioni con *business logic* complesse
- Viene però **usato anche in servizi a traffico elevato**, ad es. YouTube, Instagram, Pinterest, Dropbox, ...
 - *While many teams are moving on to more complex ecosystems, YouTube really does keep it simple. They program primarily in Python, use MySQL as their database, they've stuck with Apache, and **even new features for such a massive site start as a very simple Python program.***
<http://highscalability.com/blog/2012/3/26/7-years-of-youtube-scalability-lessons-in-30-minutes.html>



Framework Web

- Per lo sviluppo Web in Python, così come in altri linguaggi general-purpose, si usa in genere un *framework*
- Un framework fornisce un'API che lo sviluppatore usa per **definire gli aspetti specifici** della sua webapp
 - modello dei dati, contenuti, *business logic*, ...
- Il framework **gestisce gli aspetti comuni** a tutte le webapp, semplificando notevolmente lo sviluppo
 - interpretazione degli URL, gestione delle sessioni, autenticazione, ...
- Esistono decine di framework Web, che offrono livelli di astrazione e funzionalità differenti



Flask

- **Flask** è un *microframework* Python per la realizzazione di semplici applicazioni Web
- Offre un set di funzionalità essenziali, basandosi su altre librerie Python per gestire aspetti specifici
 - al contrario ad es. di *Django*, altro framework Web che fornisce invece una propria API per gestire funzionalità di alto livello
- Da un unico file con poche righe di codice si può abbozzare una semplice webapp, da espandere poi progressivamente
- Tramite la libreria di templating *Jinja2* integrata, è possibile generare dinamicamente pagine di risposta alle richieste
- Le applicazioni si possono testare sul webserver minimale integrato in Flask e poi installare su server ad alte prestazioni



Un'Applicazione Flask

- Il codice di una webapp Flask `XYZ` è contenuto
 - in un modulo (singolo file principale `XYZ.py`) oppure
 - in un package (directory `XYZ` con un file principale `__init__.py`)
- Nel file principale del progetto va creato un oggetto della classe `Flask` che rappresenti l'applicazione
 - nel creare l'oggetto va indicato il nome del modulo o package, necessario a Flask per reperire le risorse della webapp
 - per questo si usa di solito l'attributo `__name__` del modulo

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

- Tramite l'oggetto `app` si può configurare l'applicazione



Logica dell'Applicazione

- Per definire la *business logic* della webapp, scriviamo funzioni che **restituiscano le risposte** da dare a determinate **richieste**
- Poniamo ad esempio una webapp accessibile all'indirizzo <http://example.com>
- Vogliamo che una richiesta a <http://example.com/hello> fornisca come risposta "Hello, world!"
- Scriviamo una funzione che **restituisca la risposta**
 - in questo caso specifico la risposta non dipende da alcun parametro

```
def say_hello():  
    return "Hello, world!"
```

- Come collegare tale funzione al percorso **"/hello"** della webapp?



Routing

- Nei framework Web, il *routing* associa al **percorso** di una **richiesta** un'**azione** da eseguire che genera una **risposta**
- Per associare una funzione ad un **percorso della webapp**, usiamo il decoratore **route** dell'oggetto **app**

```
@app.route("/hello")  
def say_hello():  
    return "Hello, world!"
```

Un *decoratore* è una funzione/metodo applicato ad un'altra funzione con la sintassi "@" (simile ad annotazioni Java)

- In questo modo la funzione **say_hello** sarà chiamata ad ogni **richiesta GET al percorso "/hello"**
- Una webapp può contenere un numero arbitrario di route
- Per definire il contenuto della home page (**http://example.com**) si crea un route sul percorso **"/"**



Esempio di Applicazione Flask Minimale

- Si supponga di avere in una directory `mywebapp` il file `mywebapp.py` contenente il seguente codice

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

```
@app.route("/hello")  
def say_hello():  
    return "Hello, world!"
```



mywebapp



mywebapp.py

- Questo è un primo esempio di applicazione Flask minimale ma funzionante, vediamo come eseguirla...



Testare un'Applicazione Flask

- Flask integra un web server basilare eseguibile da linea di comando per testare il funzionamento dell'applicazione
- Prima di eseguirlo, va configurato in una variabile d'ambiente `FLASK_APP` il nome del file

```
$ export FLASK_APP=mywebapp.py      (Linux / Mac OS X)
```

```
> set FLASK_APP=mywebapp.py         (Windows)
```

- Si può quindi eseguire l'applicazione lanciando dalla directory dell'applicazione il comando:

```
$ flask run
```

- In output viene stampato **l'indirizzo** dell'applicazione

```
* Serving Flask app "mywebapp"
```

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



Accedere ad un'Applicazione nel Browser

- Possiamo accedere all'app dal browser digitandone un URL composto dall'**indirizzo** e da un **percorso valido**

```
Prompt dei comandi - flask run

C:\Users\roberto.pasolini2\mywebapp>set FLASK_APP=mywebapp.py

C:\Users\roberto.pasolini2\mywebapp>flask run
* Serving Flask app "mywebapp"
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```



Hello, World!

richiesta

risposta

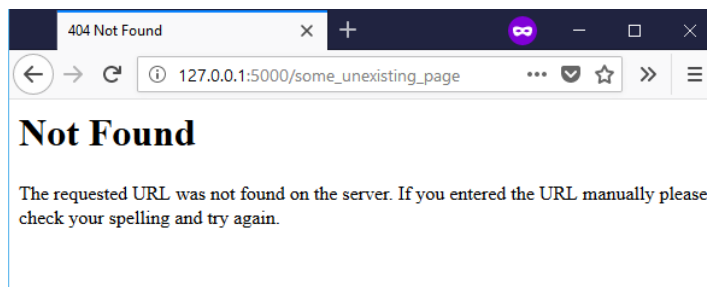
```
from flask import Flask
app = Flask(__name__)
```

```
@app.route("/hello")
def say_hello():
    return "Hello, world!"
```

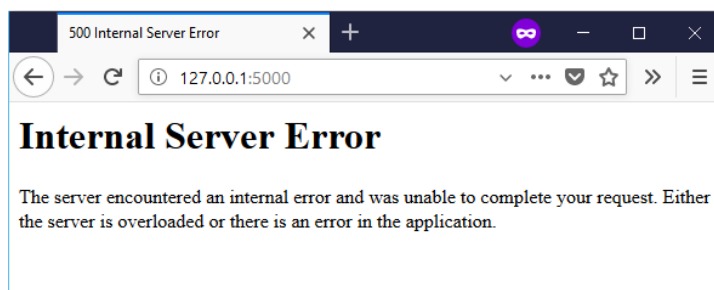


Pagine di Errore

- Su richiesta a percorso non definito nella webapp, Flask invia una risposta **404 Not Found** con pagina d'errore standard



- Se nella funzione invocata da una richiesta si verifica un errore, la risposta sarà **500 Internal Server Error**
 - la traccia dell'errore è stampata nel terminale



Modalità Sviluppo

- Le pagine d'errore mostrate sono usate dal server Flask quando è eseguito nella modalità standard
- Possiamo configurare il server in modo da abilitare alcune funzionalità utili durante lo sviluppo della webapp
 - riavviare in automatico la webapp ad ogni modifica di un file sorgente
 - normalmente bisogna terminare (Ctrl+C) e riavviare la webapp
 - in caso di errore, mostrarne la traccia direttamente nel browser
- Per attivare la modalità sviluppo, impostare la variabile d'ambiente `FLASK_ENV` a “development”

\$ `export FLASK_ENV=development` *(Linux / Mac OS X)*

> `set FLASK_ENV=development` *(Windows)*

- Per la modalità standard impostare la variabile a “production”



Configurazione di un'Applicazione Flask

- Un'applicazione ha spesso diversi parametri che l'utente può configurare prima di eseguirla
 - parametri definiti da Flask, es. per gestire le sessioni (vedi dopo)
 - parametri specifici dell'applicazione, definiti dallo sviluppatore
- L'attributo `config` dell'oggetto `Flask` permette di accedere ai parametri come un dizionario
- `config` offre inoltre metodi per impostare i parametri da diverse fonti
 - `from_pyfile`: legge configurazione da un file con nome dato
 - `from_mapping`: specifica parametri direttamente nel codice
 - `from_envvar`: legge configurazione da un file il cui nome è indicato in una variabile d'ambiente (simile alla var. `FLASK_APP`)



Configurazione Tipica

- Un pattern tipico per la configurazione di una webapp è:
 1. caricare i valori predefiniti dei parametri da un file distribuito insieme al codice (o direttamente dal codice stesso)
 2. caricare un file esterno, se presente, sovrascrivendo i valori di alcuni parametri con quelli definiti in esso

```
app = Flask(__name__)  
app.config.from_mapping(foo=10)  
app.config.from_envvar(MYAPP_CONFIG)
```

- Chi avvia la webapp può così cambiare il valore di `foo` creando un file e indicandone il nome nella var. `MYAPP_CONFIG`

```
$ export MYAPP_CONFIG=myconf.cfg
```

```
foo = 20
```

```
myconf.cfg
```



Routing con Parametri

- Il percorso associato ad una funzione può contenere dei **segnaposto** “<...>” che rappresentano dei **parametri**
- Quando deve gestire una richiesta, alla funzione sono passati i **valori dei parametri presenti nel percorso**, in forma di stringhe
- Sia definita ad es. questa funzione con un parametro **name**:

```
@app.route("/user/<name>")
```

```
def user_page(name):
```

```
    return "This is {}'s page".format(name)
```

- Alla richiesta all'URL...

`http://127.0.0.1:5000/user/bob`

- ...la webapp risponderà...

This is **bob**'s page



Tipi di Parametri

- Ad un **parametro** può essere associato un **tipo**
- Definiamo ad esempio una funzione con parametro di tipo **int**:

```
@app.route("/aaah/<int:length>")
```

```
def say_aaah(length):  
    return length * "A" + "H"
```

- In questo modo Flask verifica che l'URL contenga un numero intero e lo converte a tale quando lo passa alla funzione
- Alla richiesta all'URL...

```
http://127.0.0.1:5000/aaah/5
```

- ...la webapp risponderà "AAAAAH"
- Tra gli altri tipi disponibili ci sono "float" (numero decimale) e "path" (stringa che può contenere caratteri "/")



Metodi delle Richieste

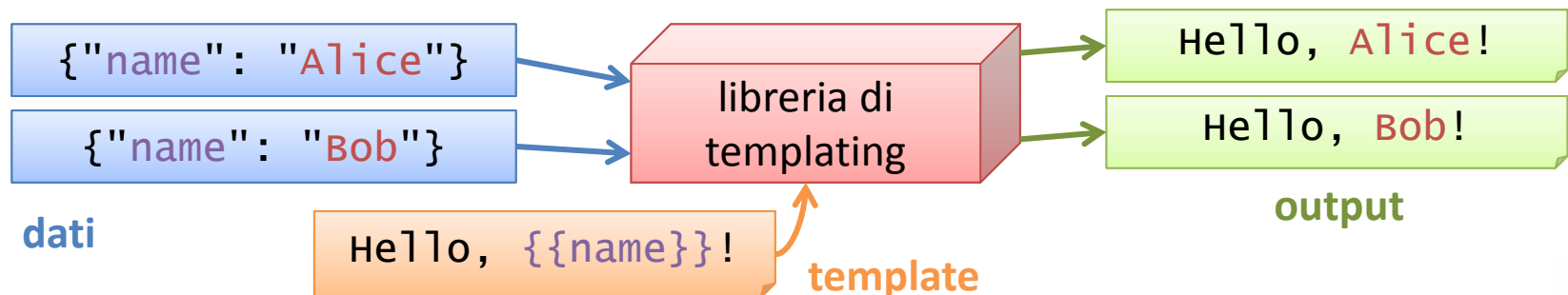
- Di default, una funzione decorata con `route` è eseguita a fronte di una **richiesta GET al percorso indicato**
 - con GET si ottiene una risorsa dalla webapp senza modificarne lo stato
 - quando si naviga ad un URL, il browser invia una richiesta GET
- Per far sì che la funzione sia invocata da altri metodi, questi vanno specificati col parametro `methods` come lista di stringhe
- Ad es., per una richiesta che esegue un'azione modificando lo stato della webapp, si richiede l'uso del metodo POST
 - una richiesta POST è comunemente inviata tramite un form HTML, di cui vedremo un esempio più avanti

```
@app.route("/place-order", methods=["POST"])  
def place_order():  
    ...
```



Templating

- Le librerie di *templating* sono usate per la **generazione di contenuti dinamici** basati su modelli
- Un modello (*template*) è un file con contenuto statico (es. HTML) con inseriti dei **segnaposto per parti variabili**
 - tramite “tag” definiti dalla libreria si possono anche eseguire azioni quali ripetere più volte un contenuto, innestare un altro file ecc.
 - non** si può inserire codice eseguibile arbitrario come in PHP o JSP
- Dall’unione di un **modello** e di un **insieme strutturato di dati**, la libreria genera una **pagina completa**



Jinja2

- Flask integra *Jinja2* come libreria di templating
- Un template Jinja2 è un file HTML che contiene la **struttura comune** di pagine generate dinamicamente
- Nel template sono inseriti tag “**{{ ... }}**” recanti all’interno delle **espressioni**, con sintassi simile a quella di Python
- Le espressioni contengono **variabili**, i cui **valori sono forniti dall’applicazione** quando invoca il *rendering* del template
- Ad ogni rendering, le espressioni sono valutate e i **risultati sono inseriti in output** al posto dei rispettivi tag
 - qualsiasi contenuto fuori dai tag è riportato in output così com’è
- Si possono anche usare tag “**{# ... #}**” per inserire **commenti** non riportati in output



Esempio: Template di Pagina HTML

```
{# a minimal HTML page #}  
<!DOCTYPE html>  
<html>  
<head>  
  <title>My webapp</title>  
</head>  
<body>  
  <h1>My webapp</h1>  
  <p>Hi, {{ name }} !</p>  
</body>  
</html>
```

- Questo template contiene una semplice pagina HTML con gli elementi di base
- Il **tag commento** nella prima riga viene ignorato, non appare nell'output
- Nel paragrafo `<p>` è contenuto un **tag** con l'espressione "`name`"
- A ogni rendering del template, il tag sarà sostituito col valore di `name` passato dall'applicazione



Uso di un Template in Flask

- Jinja2 può essere richiamato da Flask per generare pagine dinamiche in risposta alle richieste
- Da una funzione di risposta si può usare la funzione `render_template` per generare una pagina da un template
- Alla funzione va fornito il **nome del file del template** e i **valori da assegnare alle variabili** (tramite argomenti con nome)

```
@app.route("/hello/<user>")
```

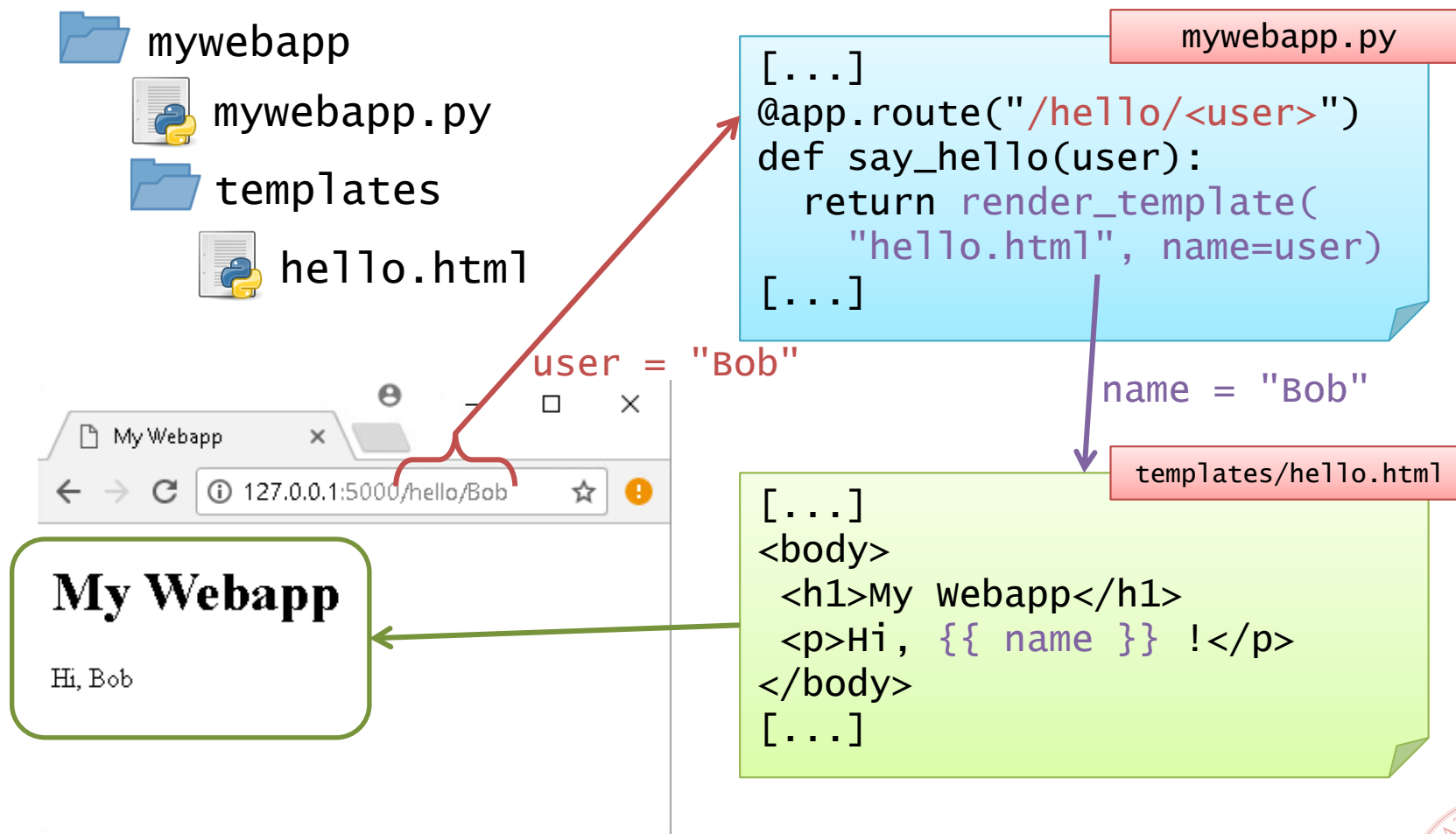
```
def say_hello(user):
```

```
    return render_template("hello.html", name=user)
```

- I file dei template vanno posti in una directory `templates` insieme al file Python dell'applicazione



Esempio: Risposta ad una Richiesta con Rendering di un Template



Gestione di una Richiesta

- I passaggi tipici di una funzione decorata con `route` che risponde ad una richiesta sono:
 1. **raccogliere e verificare parametri** e/o dati passati in input
 - es. verificare che un numero dato dall'utente sia incluso in un range specifico, che una stringa non superi certe dimensioni, ...
 2. **compiere elaborazioni** su essi per ottenere dei dati di output
 - può includere leggere/scrivere dati da/a file su disco, database, ...
 3. **generare una risposta** integrando i dati di output in un opportuno template
- L'uso del templating permette di isolare **l'elaborazione** dei dati (definita in Python) dalla loro **presentazione** (nei template)
 - prendendo ad es. come riferimento il pattern *Model-View-Controller*, le funzioni `route` sono i *controller* e i template sono le *view*



Restituire una Risposta in JSON

- Flask di default considera tutte le risposte fornite da una webapp di tipo HTML
 - ogni risposta ha un header “Content-Type: text/html”
 - la risposta è quindi destinata alla visualizzazione in un browser
- La funzione `jsonify` invece converte un oggetto dato in una stringa JSON e fa sì che il formato della risposta sia JSON
 - di solito viene usato un dizionario, trasformato in un oggetto JSON con le corrispondenti coppie chiave-valore
 - questo può essere usato per creare una semplice API Web, utilizzabile da altre applicazioni

```
from flask.json import jsonify
@app.route("/api/test")
def json_api_test():
    return jsonify({"test": ["some", "values"]})
```



Ottenere gli URL delle Pagine

- Nelle pagine di una webapp è solito inserire link alle altre pagine, delle quali va specificato l'URL
- Inserire direttamente l'URL d'una pagina nell'app è sconsigliato
 - l'URL completo dipende da indirizzo e configurazione del server
 - cambiando l'URL di una pagina, ne andrebbero cambiati tutti i link
- Flask fornisce la funzione `url_for` per generare gli URL corretti per ciascuna pagina della webapp, specificando
 - il **nome della funzione** annotata con `route` come primo argomento
 - eventuali **parametri del percorso** come argomenti con nome

```
from flask import url_for
```

```
hello_url = url_for("say_hello", user=name)
```

- La funzione `url_for` può essere utilizzata allo stesso modo anche all'interno di un template



Esempio: Inserire Link ad Altre Pagine in un Template

mywebapp.py

"/" è la home page

```
@app.route("/")
```

```
def home():
    return ...
```

```
@app.route("/user/<name>")
```

```
def user_page(name):
    return render_template(
        "user.html", name=name)
```

```
@app.route("/recs/<name>")
```

```
def recomms(name):
    return ...
```

templates/user.html

```
[...]
```

```
<p>Hi, {{ name }}!</p>
```

```
<p>
```

{# link semplice #}

```
<a href="{{ url_for("home") }}">
```

Return to home page

```
</a>
```

```
</p>
```

```
<p>
```

{# link con un parametro #}

```
<a href="{{ url_for("recomms", name=name) }}">
```

See products recommended for you

```
</a>
```

```
</p>
```

```
[...]
```



File Statici

- Nelle pagine HTML è spesso necessario **importare altri file**
 - fogli di stile CSS, script JavaScript, immagini, font, ...
- Al contrario delle pagine, questi file sono di solito *statici*: **i loro contenuti sono fissi**, non serve generarli dinamicamente
- In un'applicazione Flask possono essere inclusi dei file statici inserendoli in una directory **static**
- Ciascuno di questi file sarà accessibile ad un URL del tipo:
`http://127.0.0.1:5000/static/nomeFile`
- L'URL si ottiene dalla funzione `url_for` usando `"static"` come nome funzione e indicando un argomento `filename`:
`url_for("static", filename="nomeFile")`



Esempi: Usare File Statici nei Template

- Per applicare un CSS:

```
<head>
  [...]
  <link rel="stylesheet" type="text/css"
    href="{{ url_for("static", filename="style.css") }}">
</head>
```

- Per inserire uno script JavaScript:

```
<body>
  [...]
  <script type="text/javascript"
    src="{{ url_for("static", filename="mywebapp.js") }}">
  </script>
</body>
```



Template: Attributi di Oggetti

- I valori dati alle variabili dei template possono essere **oggetti Python arbitrari** (numeri, stringhe, liste, dizionari, ...)
- Dai template è possibile **accedere agli attributi** di questi oggetti e chiamarne metodi con la sintassi ordinaria di Python
- In più, si possono usare indici (notazione “**x[indice]**”) come se fossero attributi (notazione “**x.attributo**”) e viceversa
- Ad es. passando un **dict** *d* ad un template, si può accedere ad un valore in esso con “**d.chiave**”

```
userdata = {"id": 0, "first": "John", "last": "Doe"}  
render_template("page.html", user=userdata)
```

```
<p>Hello, {{ user.first }}</p>    → Hello, John
```



Template: Blocchi Condizionali

- Nei template Jinja2 si possono usare tag “`{% ... %}`” per attivare funzionalità avanzate durante il rendering
- Con i tag `if` e `endif` si delimita una parte di contenuto inserita solamente se una condizione data è soddisfatta
 - ad es. con “`if nomevariabile`” si verifica che una variabile di nome dato esista e non sia valutata a `False` (es. lista vuota)
- All’interno di `if` e `endif` si possono usare tag `elif` ed `else` con semantica analoga alle omonime parole chiave Python

```
{% if user %}      {# i.e. se user esiste non vuota #}
<p>You are logged in as {{ user }}</p>
{% else %}         {# il blocco else è facoltativo #}
<p>You are not logged in</p>
{% endif %}
```



Template: Iterazione (1)

- I tag `for` e `endfor` delimitano del **contenuto ripetuto una volta per ciascun elemento** in una collezione
- Il `for` si usa spesso per creare **elenchi o tabelle** con un elemento o riga per ciascun oggetto in una collezione
- È comune iterare su collezioni di oggetti strutturati (es. dizionari), da ciascuno dei quali si estraggono più attributi
 - ad es. nella riga di una tabella è mostrato un attributo per cella

```
<table>
```

```
<tr><th>Name</th><th>Price</th></tr>
```

```
{% for item in items %}
```

```
<tr><td>{{item.name}}</td><td>{{item.price}}</td></tr>
```

```
{% endfor %}
```

```
</table>
```



Template: Iterazione (2)

- Come in Python, al costrutto `for...in` si può accodare una clausola `if` per prendere solo alcuni elementi dalla collezione

```
<ul>{% for item in items if not item.hidden %}  
  <li>{{ item }}</li>  
{% endfor %}</ul>
```

- Nel blocco `for` si può inserire un blocco `else`, riportato in output come alternativa se la collezione è vuota

```
<ul>{% for item in items %}  
  <li>{{ item }}</li>  
{% else %}  
  <li>No item found</li>  
{% endfor %}</ul>
```



Template: Iterazione (3)

- All'interno di un blocco `for`, è possibile accedere ad un oggetto `loop` che fornisce alcune informazioni e funzioni
 - `index` e `index0` indicano il numero di iterazione corrente partendo da 1 e 0 rispettivamente, senza contare elementi esclusi da `if`
 - `first` e `last` indicano se l'elemento corrente è il primo o l'ultimo
 - `cycle`, dato un elenco di valori, cicla tra essi attraverso le iterazioni del `for`; utile ad es. per creare una tabella con righe a colori alterni

```
<table>
<tr><th>#</th><th>Item</th></tr>
{% for item in items %}
  <tr class="{{ loop.cycle("odd", "even") }}">
    <td>{{ loop.index }}</td><td>{{ item }}</td></tr>
{% endfor %}
</table>
```



Template: Filtri

- I *filtri* in Jinja2 sono trasformazioni (funzioni unarie) applicabili a variabili o espressioni in un template
- Si applicano con la sintassi “*valore|filtro*”
 - un filtro può avere argomenti: “*valore|filtro(x, y)*”
 - si possono concatenare: “*valore|filtro1|filtro2|...*”
- I filtri disponibili includono:
 - *length*: converte una lista o stringa nella sua lunghezza
 - es. “*There are {{ items|length }} items*”
 - *escape* (o *e*): converte caratteri speciali HTML (es. “<”) in sequenze di escape (“<”), va usato su stringhe che potrebbero contenerli
 - *default(x)*: converte una variabile non definita nel valore *X* dato
 - *round(N)*: arrotonda un numero a *N* cifre decimali
 - *join(x)*: concatena elementi di una lista inserendo stringa *X* tra essi



Template: Test

- I test sono condizioni che si possono verificare su un'espressione tramite la sintassi “*valore is test*”
 - anche i test possono avere argomenti: “*valore is test(x, y)*”
- Si possono usare ovunque sia atteso un valore booleano
 - es. come condizione in blocco *if* o in clausola *if* di *for*
- I test disponibili includono:
 - *defined*: la variabile è definita?

```
{% if user is defined %}  
  <p>Logged in as {{ user }}</p>  
{% endif %}
```
 - *even/odd*: il valore è pari/dispari?
 - *divisibleby(N)*: il valore è divisibile per *N*?



Template: Macro

- Una macro definisce una **porzione di contenuto riutilizzabile**
- Una macro può contenere **parametri** con eventuali valori di default, **utilizzabili come variabili**

- Una macro è definita all'interno di un template...

```
{% macro colored(text, color="blue") %}
    <span style="color: {{ color }};">{{ text }}</span>
{% endmacro %}
```

- ...è può essere richiamata successivamente come una funzione
 - ci si può riferire ai parametri per posizione o nome come in Python

```
<p>{{ colored("This", "red") }} is red</p>
<p>{{ colored(text="This") }} is blue</p>
```



Template: Macro Importate

- Una macro definita in un template può essere utilizzata in un altro template importandola, con una logica simile a Python
 - si usa creare uno o più template dove sono solamente definite delle macro, utilizzate negli altri template
- Come in Python, si può importare un template con un nome e accedere alle sue macro come attributi...

```
{% import "macros.html" as macros %}
```

```
<p>{{ macros.colored("This", "#ff0000") }} is red</p>
```

- ...oppure importare direttamente una o più macro

```
{% from "macros.html" import colored %}
```

```
<p>{{ colored("This", "#00ff00") }} is green</p>
```



Template: Ereditarietà

- In una webapp è comune che le pagine, seppur diverse nei contenuti, condividano una **stessa struttura generale**
 - layout generale, header e footer, stili in CSS, ...
- In Jinja2 **ogni template può estenderne un altro**, in modo da **riutilizzarne del contenuto** e aggiungerne altro
 - ogni template *A* può definire dei *blocchi* di contenuto (eventualmente vuoti), ciascuno con un nome
 - se un template *B* estende *A* di default ne replica esattamente i contenuti, ma può **ridefinire i contenuti dei singoli blocchi**
- L'organizzazione tipica dei template in una webapp prevede
 - un template “master” con la struttura comune di tutte le pagine che contenga uno o più blocchi (vuoti) per i contenuti specifici
 - un template per ciascuna pagina che estenda il master e ridefinisca solamente i blocchi con i contenuti specifici della pagina



Esempio di Template Master

- Questo template definisce la struttura di una pagina HTML e inserisce un titolo nel body col nome dell'applicazione
- Al posto del resto del contenuto della pagina, viene inserito un blocco "body" vuoto

```
<!DOCTYPE html>
<html><head>
  <title>My webapp</title>
  <link rel="stylesheet" type="text/css"
    href="{{ url_for("static", filename="style.css") }}">
</head><body>
  <h1>My webapp</h1>
  {% block body %}{% endblock %}
</body></html>
```



Esempio di Template per Pagina Specifica

- Un altro template può dichiarare di ereditare il template master col tag `extends` e ridefinire il blocco “body”
- Ad ogni rendering, risulterà la struttura del template master con i contenuti del blocco “body” definiti in questo template

```
{% extends "master.html" %}  
{% block body %}  
  <p>Products recommended for you are:</p>  
  <ul>  
    {% for i in items %}<li>{{ i }}</li>{% endfor %}  
  </ul>  
{% endblock %}
```



Attributi della Richiesta

- In una webapp Flask, si possono ottenere informazioni sulla richiesta a cui si sta rispondendo dall'oggetto `request`
 - si tratta di un `oggetto globale` da importare dal package `flask`
 - questo oggetto è però un *proxy* che `rappresenta sempre la richiesta corrente`, dedotta dal contesto d'esecuzione del codice
 - l'oggetto `request` è accessibile anche dai template
- `from flask import request`
- `request` ha diversi attributi che forniscono informazioni sulla richiesta corrente, ad es.:
 - `path`: percorso della richiesta (relativo alla webapp)
 - `url`: URL completo della richiesta
 - `method`: metodo della richiesta (es. "GET")
 - `user_agent`: nome del browser usato dal client



Parametri dell'URL della Richiesta

- In un qualsiasi URL possono essere specificati **parametri arbitrari**, secondo la seguente forma:

`http://example.com/mypage?a=foo&b=42`

- Questi parametri sono accessibili dall'attributo `args` dell'oggetto `request`, utilizzabile come un dizionario
 - i valori sono sempre stringhe, da convertire in numeri se necessario

```
@app.route("/mypage")
```

```
def some_page():
```

```
    return "value of 'a' is {}".format(request.args["a"])
```

`{"a": "foo", "b": "42"}`

- Se si accede ad un parametro non specificato nella richiesta, Flask invia una risposta con codice 400 (errore nella richiesta)
- Tali parametri possono essere specificati anche in `url_for`
`url_for("some_page", a="foo", b="42")`



Invio di Dati da un Form con Metodo GET

- In HTML si possono inserire *form* (moduli) che l'utente può compilare per inviare **richieste con parametri** ad una webapp
- Ad ogni form è associato un percorso della webapp ed un metodo di invio (in genere GET o POST)
- All'invio (*submit*) del modulo, viene inviata una richiesta al percorso impostato che **include i dati inseriti dall'utente**
- Nel caso di una richiesta GET, i dati sono **inviati in forma di parametri dell'URL** accessibili con `request.args`
 - ricordare che il metodo GET si usa per richieste che non modificano lo stato dell'app, ad es. per un form di ricerca
 - il nome del parametro a cui è associato ciascun valore è dato dall'attributo `name` del corrispondente campo del modulo

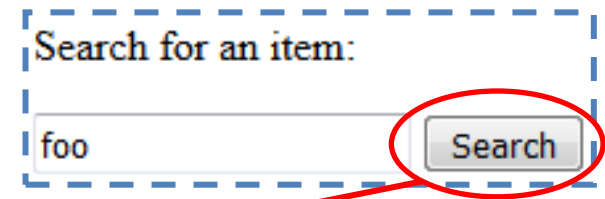


Esempio: Form per Ricerca (GET)

templates/some_page.html

```
<p>Search for an item:</p>
<form action="{{ url_for('search_items') }}"
      method="GET">
  <input type="text" name="query">
  <input type="submit" value="Search">
</form>
```

Dichiaro un form con due controlli (input):
casella di testo e pulsante invio



GET /search?query=foo

mywebapp.py

```
@app.route("/search")
```

```
def search_items():
```

```
    query = request.args["query"]
```

```
    results = do_search(query)
```

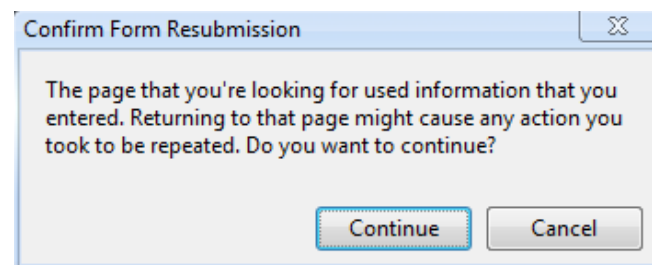
```
    return render_template("results.html", items=results)
```

Reperisco il valore inserito
dall'utente nella casella "query"
dai parametri dell'URL



Invio di Dati da un Form con Metodo POST

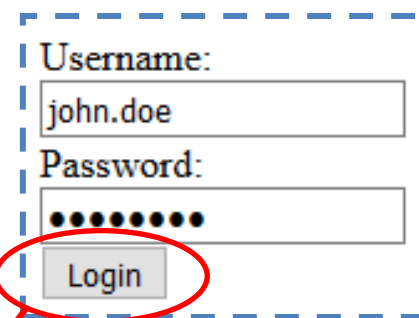
- Se la richiesta inviata da un form comporta un cambiamento di stato nella webapp, si usa per convenzione il metodo POST
 - es. pubblicare un messaggio in un forum
 - spesso i browser avvertono l'utente se tornando ad una pagina indietro tenta di reinviare una richiesta POST già inviata
- Nel caso di POST, i dati del form non sono inviati come parametri dell'URL ma come *payload* (corpo) della richiesta
- Per accedere ad essi si usa l'attributo `form` dell'oggetto `request`, che funziona allo stesso modo di `args`



Esempio: Form per Login (POST)

`templates/some_page.html`

```
<form action="{{ url_for('user_login') }}"
      method="POST">
  Username:<br>
  <input type="text" name="username"><br>
  Password:<br>
  <input type="password" name="password"><br>
  <input type="submit" value="Login">
</form>
```



`mywebapp.py`

```
@app.route("/login", methods=["POST"])
def user_login():
    username = request.form["username"]
    ...
```

POST /login

username=john.doe&password=password



Redirect

- Una webapp può rispondere ad una richiesta con un *redirect*, che indica al browser di **inoltrare una richiesta ad un altro URL**
- Si usa spesso in **risposta a una richiesta POST** per redirezionare il browser ad una **pagina che mostri i risultati** della richiesta
- Per generare una risposta di questo tipo si usa la funzione **redirect**, a cui va passato l'URL di destinazione
 - per indicare URL interni alla webapp usare **url_for**

```
from flask import redirect
```

```
@app.route("/do-stuff", methods=["POST"])
```

```
def do_stuff():
```

```
    execute_some_action(request.form["what"])
```

```
    return redirect(url_for("some_page", ...))
```



Errori

- La funzione `abort` interrompe l'esecuzione e invia una risposta con un codice d'errore HTTP dato

```
from flask import abort
```

```
@app.route("/page")
```

```
def some_unimplemented_page():
```

```
    abort(500)    # errore del server
```

- Per inviare una pagina con un codice diverso da 200 (OK), la funzione può restituire una tupla *(pagina, codice)*

```
@app.route("/page")
```

```
def some_unimplemented_page():
```

```
    return render_template("notimpl.html"), 500
```



Dati Contestuali alla Richiesta

- Una funzione decorata con `route` può chiamare liberamente altre funzioni definite nel codice
- Diverse funzioni richiamate dalla stessa richiesta possono accedere a risorse comuni
 - i contenuti di un file esterno, una connessione ad un database, ...
- In questi casi può essere opportuno avere un riferimento alla richiesta corrente e salvare riferimenti ad oggetti in esso
 - ad es. se una funzione si connette ad un database, un'altra funzione chiamata nella stessa richiesta potrebbe riusare la stessa connessione
- Nell'oggetto `g` di Flask possono essere salvati riferimenti ad oggetti arbitrari e richiamati nel contesto della stessa richiesta
 - similmente a `request`, `g` è un oggetto globale che è però sempre legato alla richiesta corrente ed è richiamabile anche dai template



Esempio: Accesso a Dati in un File

- Scenario: abbiamo diverse funzioni della webapp che accedono a dati in un file, caricandolo in memoria
- Vogliamo che ciascuna riutilizzi i dati già presenti in memoria se il file era già stato caricato nell'arco della stessa richiesta
- Scriviamo una funzione che reperisca i dati del file dal contesto della richiesta, caricandoli se necessario

```
from flask import g
def get_some_data():
    if "some_data" not in g:
        g.some_data = load_some_data_from_file()
    return g.some_data
```



Sessioni

- Come anticipato, il protocollo HTTP è *stateless*: ogni richiesta è gestita a se e la risposta ad essa è indipendente dalle altre
- La webapp deve però tenere traccia dell'attività di un utente
 - ad es. in un e-commerce tipicamente un utente aggiunge prodotti ad un carrello progressivamente prima di effettuare un ordine
 - queste attività richiedono diversi scambi col server, che deve tenere traccia dei carrelli di diversi utenti contemporaneamente
- Come altri framework, Flask può tenere traccia delle *sessioni*, ovvero *sequenze di richieste e risposte con lo stesso client*
- A ciascuna sessione possono essere *associati dei dati specifici*, non condivisi con altre sessioni aperte allo stesso tempo
 - Flask di base usa sessioni *client-side*: i dati di una sessione sono memorizzati direttamente in un cookie, con forti limiti di memoria



Uso delle Sessioni

- Per usare le sessioni, va definita nella configurazione una stringa `SECRET_KEY` usata per firmare i cookie

```
app = Flask(__name__)
```

```
app.config.from_mapping(SECRET_KEY="fn#3_f/9vj")
```

- I dati della sessione corrente sono accessibili dall'oggetto `session`, con funzionamento simile a `g`
 - è un oggetto globale, ma legato al contesto della richiesta corrente
- ```
from flask import session
```

- Possiamo impostare i dati in una sessione...

```
session["name"] = user_name
```

- ...e richiamarli sia da codice che da template

```
<p>Hello, {{ session.name }}!</p>
```



# Mostrare Avvisi all'Utente

- Le sessioni sono usate da Flask per implementare i messaggi *flash*, utili per dare dei feedback temporanei all'utente
- La funzione `flash` inserisce un messaggio in una “coda” nella sessione corrente

```
from flask import flash, get_flashed_messages
```

– ad es. in una richiesta POST riferisco l'esito di un'azione

```
order_id = do_checkout(order_data)
```

```
flash("Checked out order #{}".format(order_id))
```

- La funzione `get_flashed_messages` (utilizzabile da un template) restituisce i messaggi nella coda e la svuota

```
{% for notice in get_flashed_messages() %}
```

```
<div class=notice>{{ notice }}</div>{% endfor %}
```





# Deployment di un'Applicazione Flask

- Il server integrato in Flask è adatto per testare l'applicazione, ma non per metterla realmente in servizio (in “produzione”)
- Flask è conforme a **WSGI** (*Web Server Gateway Interface*), uno standard per l'interazione tra server Web e webapp in Python
  - lo standard prevede che la webapp abbia la forma di una funzione invocata ad ogni richiesta che restituisca la corrispondente risposta
  - molti altri framework Web Python (es. Django) implementano WSGI
  - lo standard è descritto in PEP (*Python Enhancement Proposal*) 3333
- Una volta installata, una webapp WSGI può essere messa in servizio su diversi Web server opportunamente configurati
  - Web server general-purpose: Apache (con *mod\_wsgi*), nginx, ...
  - server specifici per app WSGI: Gunicorn, uWSGI, ...

