

Introduzione al PHP: il linguaggio e l'ambiente

- Installazione
- Concetti
- Impiego

NB: slide con sfondo colorato così possono essere saltate ai fini della preparazione per l'esame

PHP/Linux: installazione in pillole

Il modo forse più basilare di **installare** l'interprete PHP è usare un gestore di **pacchetti** come *brew* di OSX, *apt* o *yum* di Linux:

```
# apt/yum/brew install php           # installa PHP
```

NB: in **Ubuntu** la "pacchettizzazione" ha una granularità assai **fine**...

Occorrono quindi molti pacchetti per usare *php*: verranno installati automaticamente in quanto dipendenze del pacchetto *php*

NB: l'output qui sotto presuppone che il pacchetto *apache2* sia già installato

```
# apt install php           # il pacchetto php di Ubuntu è  
...                         # quasi vuoto, ma ne richiede altri  
The following NEW packages will be installed: # che apt installerà, come si vede...  
  libapache2-mod-php7.2 php-common php7.2 php7.2-cli php7.2-common  
  php7.2-json php7.2-opcache php7.2-readline
```

NB: le dipendenze di un pacchetto si possono determinare così:

```
# apt-cache depends php  
...
```

PHP: invocazione e versioni recenti (Ubuntu)

Una volta installato PHP, lo si può invocare, vedendone la versione:

```
$ php -v
PHP 7.2.24-0ubuntu0.18.04.10 (cli) (built: Oct 25 2021 17:47:59)...
```

La v. 7.2 non è stata aggiornata con prontezza per Ubuntu LTS (18.04)

```
$ grep DISTRIB_DESCRIPTION /etc/lsb-release # vediamo quale versione di Ubuntu è in uso:
DISTRIB_DESCRIPTION="Ubuntu 18.04.6 LTS"
```

Per avere sempre versioni PHP fresche, senza upgrade di Ubuntu:

```
$ sudo add-apt-repository ppa:ondrej/php # ondrej/php è il repository deb semi-ufficiale per PHP
$ sudo apt update
$ sudo apt upgrade php
```

Bisogna anche cambiare l'engine PHP di default nella più recente disponibile (v. <https://computingforgeeks.com/how-to-install-php-on-ubuntu/>):

```
$ sudo update-alternatives --config php
... # dal menu proposto, si scelga PHP 8.x come nuovo default
```

Ora si può eseguire con `php` la nuova versione installata:

```
$ php -v
PHP 8.1.2 (cli) (built: Jan 24 2022 10:42:15) (NTS) ...
```

Windows e XAMPP: installazione "in pillole"

In **Windows**, PHP si può installare in WSL, cioè in effetti in Linux; oppure, **nativamente**, come parte di uno stack integrato *WAMP*, p.es. **XAMPP**

- NB: conviene installare XAMPP nella directory C:\XAMPP, v.
<https://stackoverflow.com/questions/34481427> <https://php.tutorials24x7.com/blog/how-to-install-xampp-on-windows>

In effetti XAMPP è *cross-platform*, disponibile anche per OSX e Linux

Il **portale ufficiale** di PHP fornisce istruzioni di installazione per Windows: <https://www.php.net/manual/en/install.windows.php> e, in particolare, suggerisce XAMPP: <https://www.apachefriends.org/index.html>

In italiano: <https://www.nassaro.com/guide-e-tutorials/software-guide-e-tutorials/come-installare-e-utilizzare-xampp/>

È opportuno attivare la libreria zip:

<https://stackoverflow.com/questions/23564138/how-to-enable-zip-dll-in-xampp>

PHP: installazione "ufficiale"

Se le poche indicazioni precedenti riguardanti l'installazione non bastassero...

il **portale ufficiale** di PHP fornisce istruzioni dettagliate:

<https://www.php.net/manual/en/install.general.php>

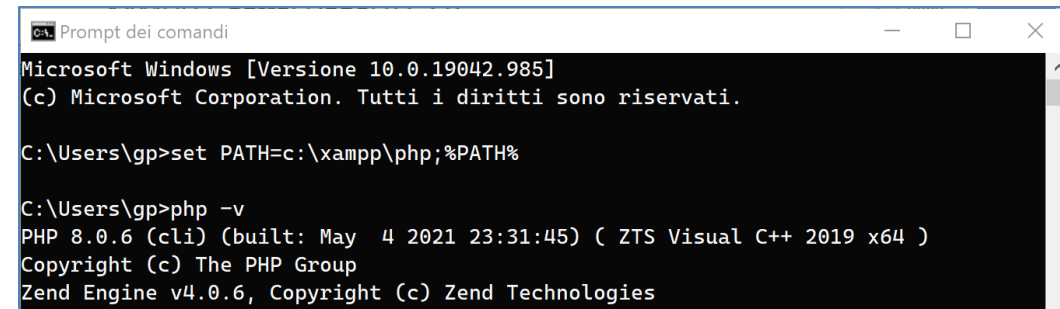
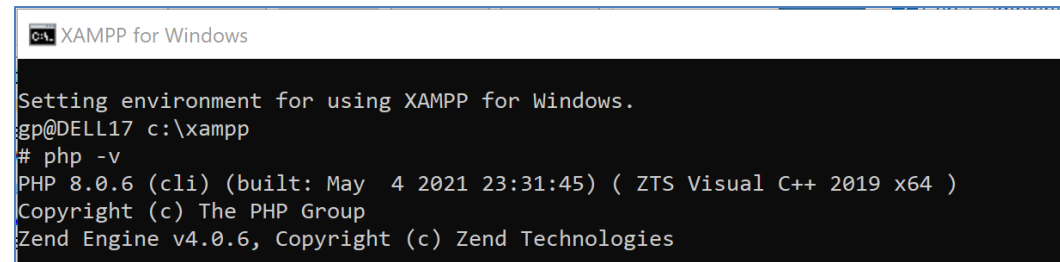
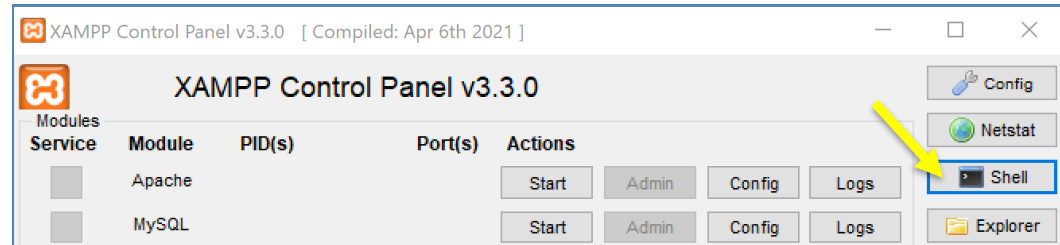
PHP: avvio con Windows

Installato XAMPP, dal pannello di controllo (NB: di XAMPP), clic su *Shell*:

Appare quindi una shell, da cui si può invocare la CLI PHP (p.es. qui con `php -v`)

Lo stesso si può fare da un "*Prompt dei comandi*" qualsiasi, se si inserisce `C:\XAMPP\PHP` nel *PATH*

Il nuovo *PATH* (una *variabile d'ambiente*) può rendersi permanente dal *pannello di controllo* (di Windows!)



La CLI PHP

Si è così installato il comando *php*, tool di tipo CLI (Command Line Interface), invocabile da shell

Il comando *php* accetta varie opzioni, p.es. *-a*

php -a è un **read-eval-print loop** (REPL), che esegue (interpreta) istruzioni PHP, p.es. *echo* (output del proprio argomento), valuta espressioni, e assegna valori a variabili, come negli esempi qui a destra






```
$ php -a
php > echo "Hello!";
Hello!

php > echo 2+3;
5

php > $nome = "pippo";
php > echo $nome;
pippo
```

NB: nella CLI e nei file del codice, le istruzioni PHP vanno **sempre** terminate con punto e virgola: quindi `echo 2+3;` e non: `echo 2+3` .

Interazione user-friendly con la CLI REPL (se il modulo *readline* è attivo):

- navigazione nella *history* dei comandi (con  )
- *editing* della command line (si può "navigare" con   , cancellare, anche tutto il rigo con Control-U (Control-C fa terminare la REPL)
- completamento automatico di identificatori (tasto  (Tab), doppio Tab se vi sono più opzioni di completamento)

REPL PHP alternativo (e più avanzato): <https://psysh.org>

PHP è (anche) un linguaggio per il Web

- **PHP** (acronimo che sta per "**PHP: Hypertext Preprocessor") è un linguaggio di scripting popolare, open source e *general-purpose***
- Un programma o **script** PHP viene eseguito da un **interprete** o **engine** PHP (prima ne abbiamo discusso l'installazione)
- PHP è specialmente adatto, usato e orientato allo **sviluppo Web**; in quest'uso viene interpretato sul server Web (***server side***)
- Più precisamente, sul server, uno script PHP può essere incorporato (*embedded*) all'interno di codice HTML, come, per esempio, in:

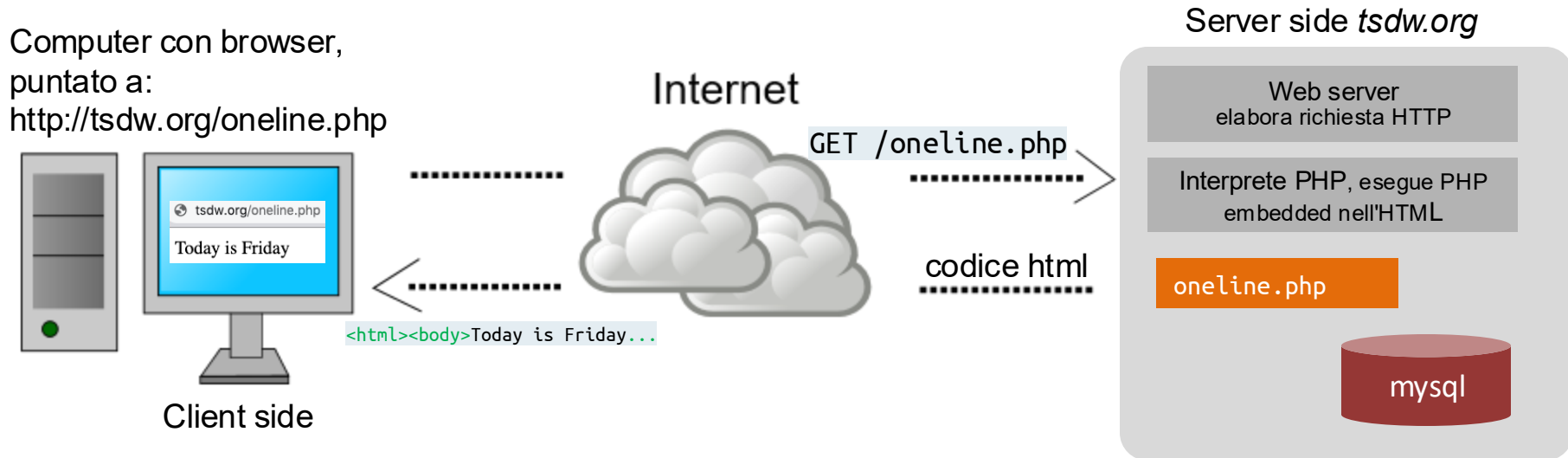
```
<html><body>Today is <?php echo date("l"); ?></body></html>
```

codice che immagineremo sia il contenuto di un file *online.php*

- Quando il client chiede il file *online.php*, il server risponde così:
 - invia al cliente l'HTML nel file così com'è, ma...
 - se incontra nel file uno script PHP, lo passa all'**interprete** PHP,

PHP: interazioni client-server

A grandi linee, ecco le interazioni nell'impiego **server-side** di PHP:



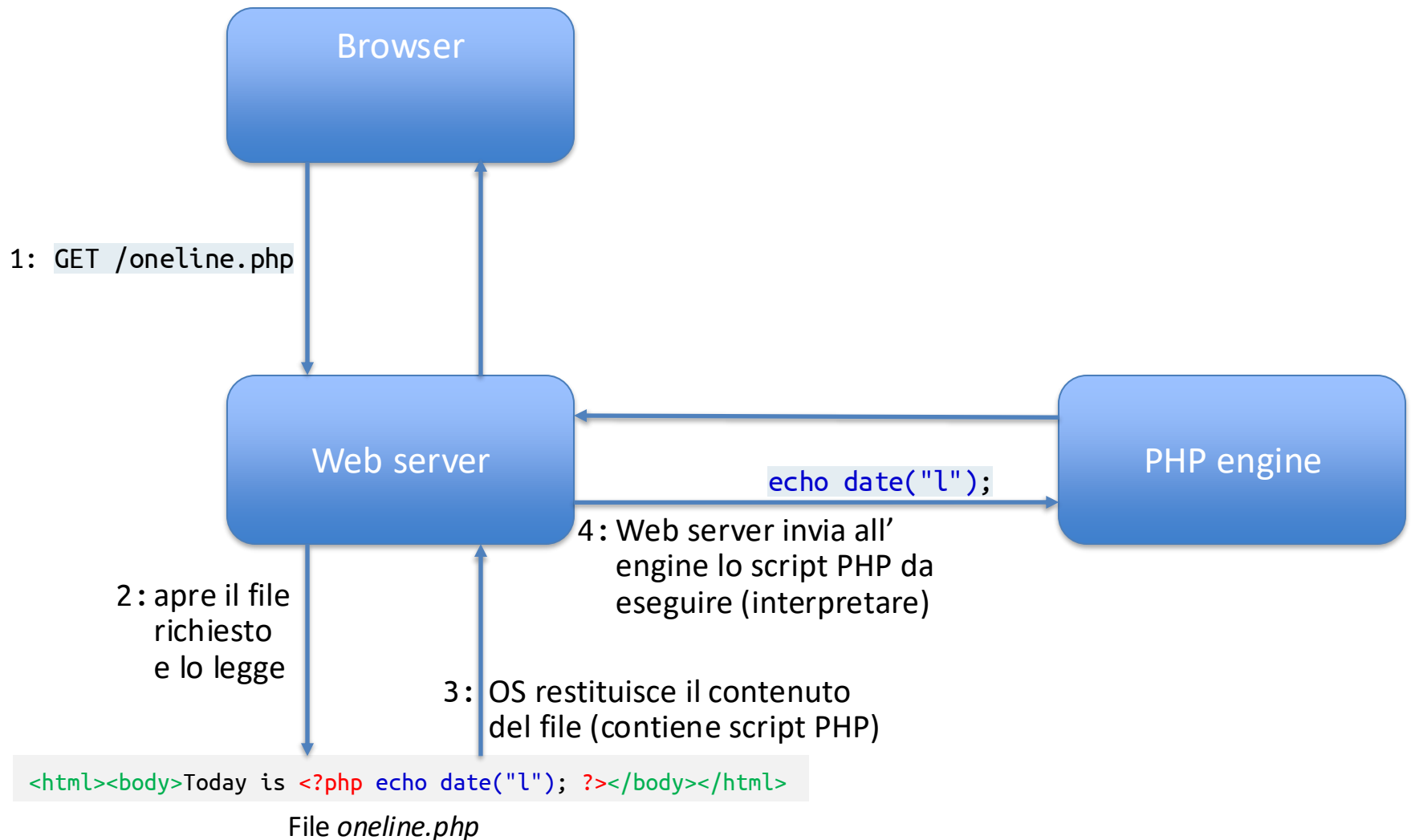
(NB: il termine *cliente*, in questo contesto, può riferirsi sia al browser che al computer su cui gira)

Pagine web, come la *online.php* già vista, sono **dinamiche**, cioè visualizzano sul browser contenuto non già **statico** (fisso), ma variabile e dipendente dall'input dell'utente o dall'ambiente

Nel caso del file *online.php*, il contenuto dinamico (`date()`) viene dal S.O. In altri casi, può provenire da un DB di supporto o "**back-end**" (*mysql* in fig.), DB che può essere ospitato dal server stesso o da altro host

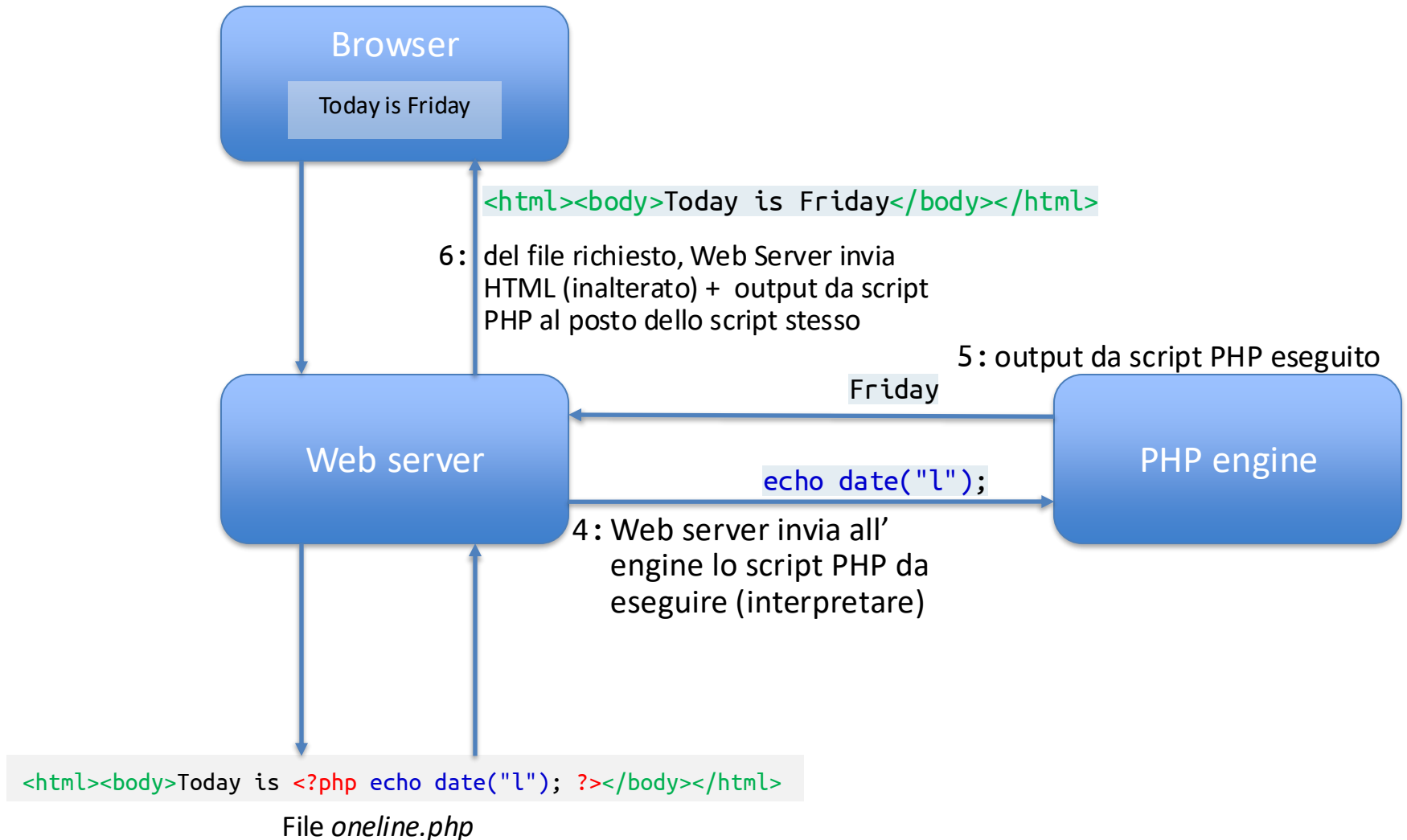
PHP: interazioni client-server / 1

Interazioni tra client e server con PHP in dettaglio (seguire numerazione):



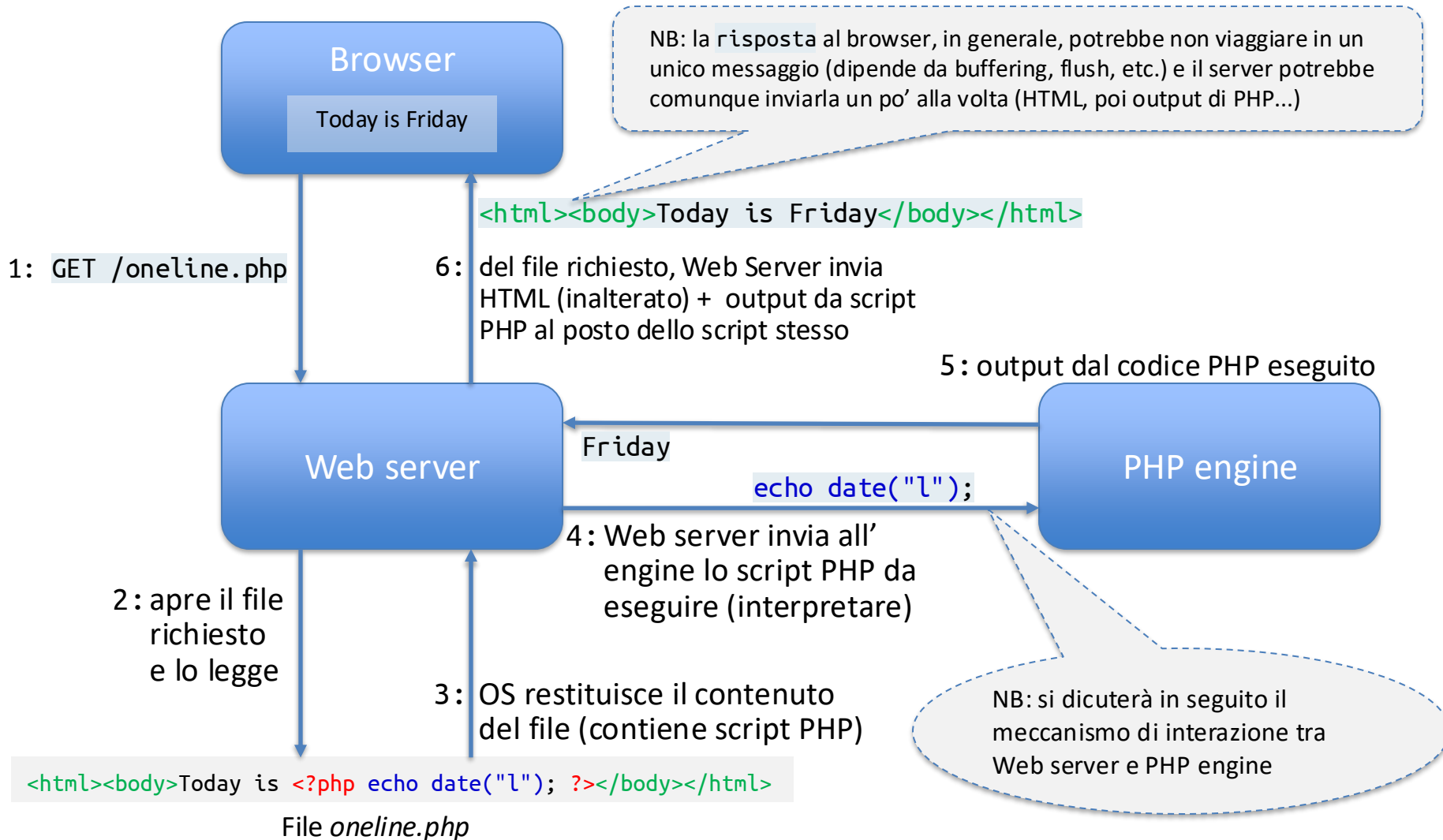
PHP: interazioni client-server / 2

Interazioni tra client e server in dettaglio (seguire numerazione):



PHP: interazioni client-server (tutte)

Interazioni tra client e server con PHP in dettaglio (seguire numerazione):



PHP: embedding nell'HTML vs. PHP puro

In una pagina HTML tipica, le parti statiche sono numerose, anzi per lo più prevalenti (grafica, presentazione, etc.)

Quindi l'**embedding** di PHP in HTML **conviene**, in quanto le parti di HTML statico si scrivono esattamente come per una normale pagina HTML, e come il server le invia al client

```
<!-- embedded.php -->
<html>
<body>
Today is
<?php echo date("l"); ?>
</body>
</html>
```

Si noti come un file *.php* come quello sopra si presenta come una pagina HTML "arricchita", di facile lettura per chi conosce HTML (grafico, etc.)

Alternativa (qui a destra): anche l'HTML statico è generato da codice PHP senza HTML, o "**puro**", (NB: in questo caso si può omettere il tag **?>** di chiusura)
Ma *embedded.php* è più leggibile di *pure.php*, perché ricalca l'HTML che sarà generato

```
<!-- pure.php -->
<?php
echo "<html>";
echo "<body>";
echo "Today is ";
echo date("l");
echo "</body>";
echo "</html>";
```

Generare HTML è, d'altro canto, proprio la prima finalità del PHP!

PHP: embedding vs. puro: varianti

```
<!-- embedded.php -->
<html>
<body>
Today is
<?php echo date("l"); ?>
</body>
</html>
```

a sinistra, il file *embedded.php*,
a destra il corrispondente HTML
generato per il cliente

```
<!-- embedded.php -->
<html>
<body>
Today is Friday
</body>
</html>
```

```
<!-- pure.php -->
<?php
echo "<html>";
echo "<body>";
echo "Today is";
echo date("l");
echo "</body>";
echo "</html>";
```

```
<!-- pure.php -->
<html><body>Today is Friday</body></html>
```

A sinistra e qui sopra: *pure.php* e l'HTML generato (HTML
poco leggibile per mancanza di "a capo")

NB: a fine file si può omettere il tag di chiusura PHP **?>**

```
<?php
echo "<html>\n";
echo "<body>\n";
echo "Today is";
echo date("l");
echo "\n</body>\n";
echo "</html>";
```

Per avere un HTML con più
righe, si inseriscono nel
PHP opportuni **"\n"**, come
qui a sinistra o destra

```
<?php
echo "<html>\n<body>\nToday is";
echo date("l");
echo "\n</body>\n</html>";
```

Qui sotto, con l'operatore **"."** (concatena), un unico *echo*:

```
<?php echo "<html>\n<body>\nToday is" . date("l") . "\n</body>\n</html>";
```

PHP: embedding vs. puro: considerazioni

Si capisce quindi che, per generare, da codice PHP "puro", codice HTML realistico, ricco di grafica, cioè con molti e complessi elementi, servirebbe una miriade di *echo*

Gli *echo* si possono ridurre, al prezzo di dare, agli *echo* residui, argomenti lunghi e illeggibili, come si vede già (in piccolo) in questo esempio di PHP "puro" con un solo *echo*:

```
<!-- pure.php ->
<?php echo "<html>\n<body>\n"Today is" . date("l") . "\n</body>\n</html>";
```

D'altro canto, come già osservato, se, nell'HTML da generare, quasi tutto il codice è statico, è più semplice e leggibile partire dall'HTML desiderato e inserirvi (pochi) script PHP, come nel riquadro a destra

```
<!-- embedded.php ->
<html>
<body>
Today is
<?php echo date("l"); ?>
</body>
</html>
```

Linguaggi e Web: PHP vs. "puro"

L'alternativa è tra: (1) linguaggio che permette l'*embedding* in HTML, come **PHP** (o ASP.NET o JSP), e (2) linguaggio "*puro*" (p. es. C o Java)

L'*embedding* è **vantaggioso** perché:

- l'*HTML statico* si scrive così come sarà inviato al client, mentre...
- in C (o Java) servirebbero apposite (e numerose) istruzioni (*printf()*)

P.es., nel caso visto, in PHP con embedding, le parti statiche si ottengono scrivendole direttamente in HTML: `<html><body>Today is ... </body></html>`

mentre in C servono 2 istruzioni: `printf("<html><body>Today is");` e `printf("</body></html>");`

Si capisce quindi che, per generare dinamicamente HTML "ricco", con molte e complesse parti statiche, in C servirà una miriade di `printf()`! Il codice "puro" risultante tende a essere poco leggibile

NB: per l'approccio "puro" e linguaggi da cui si genera bytecode eseguibile (p.es C o Java), servono poi meccanismi con cui il server Web possa attivare il bytecode in risposta alle richieste ricevute (si vedano CGI per C o le servlet per Java)

Linguaggi e Web: PHP vs. Javascript

Scripting *server-side* (PHP o ASP.NET o JSP) vs. *client-side* (Javascript)

- Nello scripting *client-side*, il server invia al client del codice HTML con gli script embedded (in Javascript) inalterati:
 - il client (browser) **vede** il codice Javascript e lo esegue
 - scopo: miglioramento dell'interazione lato client
- Come già discusso, nello scripting *server-side* al client (browser) arriva l'output generato dall'esecuzione dello script PHP sul server
 - il client (browser) **non** vede il PHP che ha generato l'output
 - scopo: generazione di pagine Web dinamiche (p.es. dipendenti dal contenuto di un database visibile al server);
 - si consideri, p.es., la pagina HTML che visualizza il listino di un sito *e-commerce*

PHP: CLI dalla riga di comando

- Il modo più basilare di invocare PHP è come CLI, dalla riga di comando:

```
$ php --help
Usage: php [options] [-f] <file> [--] [args...]
   php [options] -S <addr>:<port> [-t docroot] [router]
   php [options] -a
-a                Run as interactive shell
-r <code>         Run PHP <code> without using script tags <?...?>
-f <file>         Parse and execute <file>
-S <addr>:<port>  Run with built-in web server.
-t <docroot>      Specify document root <docroot> for built-in web server.
-s              Output HTML syntax highlighted source.
-v              Version number
-c <path>|<file> Look for php.ini file in this directory
-n              No configuration (ini) files will be used
--ini           Show configuration file names
...
```

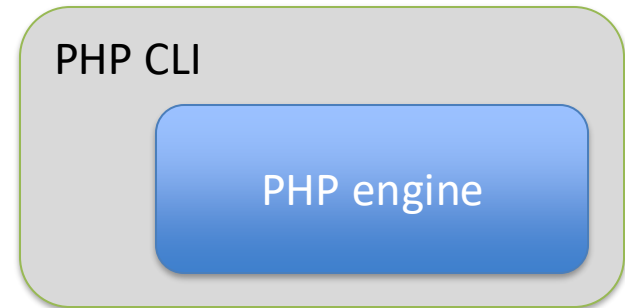
- Qui sopra, si mostra una selezione di opzioni di particolare interesse
- Si vede che *php* può interpretare file contenenti script (*php -f*) o codice inserito interattivamente (*php -a*) o come argomento (*php -r*)

PHP: CLI ed engine

Il comando *php -a* invocato da shell offre (come già visto prima) una REPL, parte della CLI (Command Line Interface) *php*

```
$ php -a
php> echo "Hello!";
Hello!
php>
```

La REPL è in grado di interpretare le istruzioni PHP lette, in quanto la CLI *php* è costruita intorno a un componente *engine* (interprete) PHP



La CLI è anche in grado di elaborare un file *.php*, contenente HTML e script PHP, interpretando gli script PHP attraverso l'engine.

Ciò può avvenire in due modalità: **locale** (*php -f*) o **server** (*php -S*) come illustrato nella prossima slide

Come server, la CLI opera ***stand-alone***, cioè non ha bisogno di un server Web "davanti" per interagire con i clienti (come invece si mostra nell'architettura nelle slide [7](#) e seguenti)

PHP CLI: modalità "locale" e "server"

La CLI può **elaborare un file**, come qui a destra *embedded.php*, contenente HTML e script PHP, in due modalità:

1. *locale*, invocata con `php -f embedded.php`

NB: in questo caso, la CLI scrive sulla **standard output** locale

```
<!-- embedded.php -->
<html><body>
Today is <?php echo
date("l"); ?>
</body></html>
```

```
$ php -f embedded.php
<!-- embedded.php -->
<html><body>
Today is Friday
</body></html>
```

2. *server*; in tal caso la CLI, invocata con `php -S`

- ascolta su socket di rete (qui **192.168.1.8:8000**)

```
$ php -S 192.168.1.8:8000
PHP 7.4.6 Server started ...
127.0.0.1:65527 Accepted
... [200]: GET /embedded.php
```

- accetta richieste di **connessione HTTP**, su cui riceve una **richiesta HTTP** (GET, POST, PUT, DELETE...,) e invia la **risposta**

```
$ telnet 192.168.1.8 8000
Connected ...
GET /embedded.php
HTTP/0.9 200 OK
```

NB: tali richieste possono provenire da browser o da altri clienti come *curl* o, qui a destra, *telnet*

- p.es. a `GET /embedded.php` il server risponde inviando il file con gli script PHP interpretati

```
...
<!-- embedded.php -->
<html><body>
Today is Friday
</body></html>
```

```
Connection closed by host.
```

php -S come server Web

- Se *php* funge da **server**, l'invocazione *php* -S, effettuata sul server, deve specificare nel 1° argomento: l'indirizzo IP o il nome del server stesso e il port su cui esso ascolta le richieste provenienti dai clienti. P.es.:

```
$ php -S 151.97.252.4:7777      # shell del server 151.97.252.4
PHP 7.3.4 Development Server started at Tue May 14 03:12:01 2019
Listening on http://151.97.252.4:7777
Document root is /Users/gp/Dropbox/SD1handouts/php/php2019
Press Ctrl-C to quit.
[Tue May 14 03:12:04 2019] 151.97.50.91:61885 [200]: /
```

Risposta del server a
GET / dal cliente
151.97.50.91

questo presuppone che l'host del server *php* abbia un'interfaccia di rete con l'IP **151.97.252.4** e che il port **7777** sia disponibile

- È più semplice e generale, però, che il server ascolti su **tutte** le interfacce di rete, cioè sull'IP **0.0.0.0** :

```
$ php -S 0.0.0.0:7777
PHP 7.3.4 Development Server started at Tue May 14 03:12:01 2019
Listening on http://151.97.252.4:7777 . . .
```

(in tal modo, non è richiesto che il comando di invocazione debba conoscere gli IP delle interfacce di rete del server)

php -S come server di sviluppo

NB: il server Web *php -S*, integrato nella CLI PHP è un server Web orientato allo **sviluppo**, non alla produzione

Per motivi di **prestazioni** e **sicurezza**, è meglio non esporlo con `$ php -S 0.0.0.0:7777`, se non su una LAN privata

L'uso tipico è invece `$ php -S localhost:7777`, su una macchina di sviluppo, sulla quale girerà anche il browser con cui si effettua il testing degli script PHP sviluppati, visualizzandone l'output

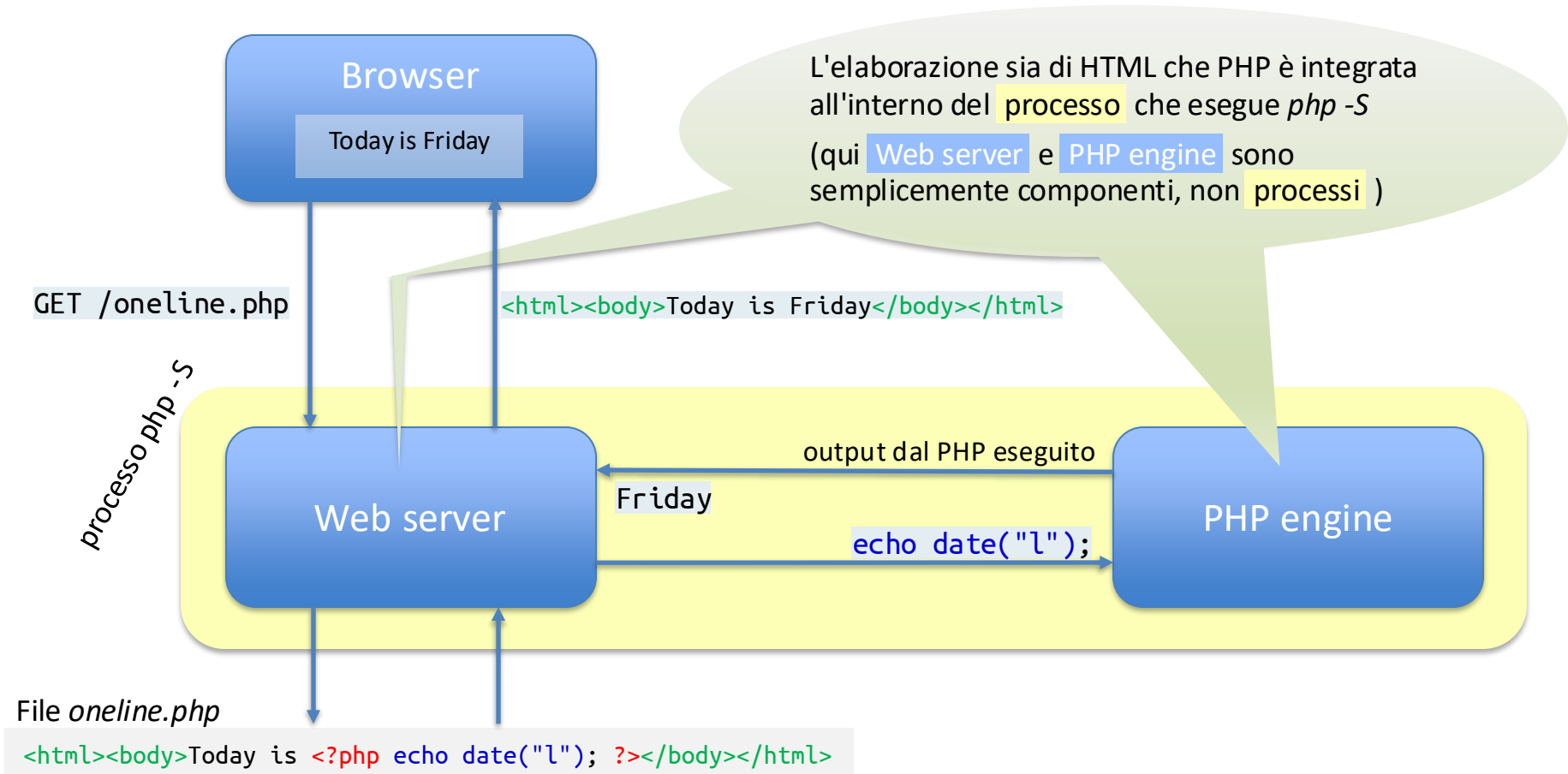
Maggiore **sicurezza** si ha con un server Web di livello *enterprise*, destinato alla produzione, come *Apache*, che richiama l'engine PHP al bisogno, nella configurazione mostrata a pag. [9](#)

php -S non è un server parallelo! Per questo, di nuovo, occorre un server come Apache dotato di un modulo-engine PHP.

(**Prestazioni** ancora migliori usando Apache + **Servizio** php-fpm, discusso in altra lezione)

php -S: il server stand-alone

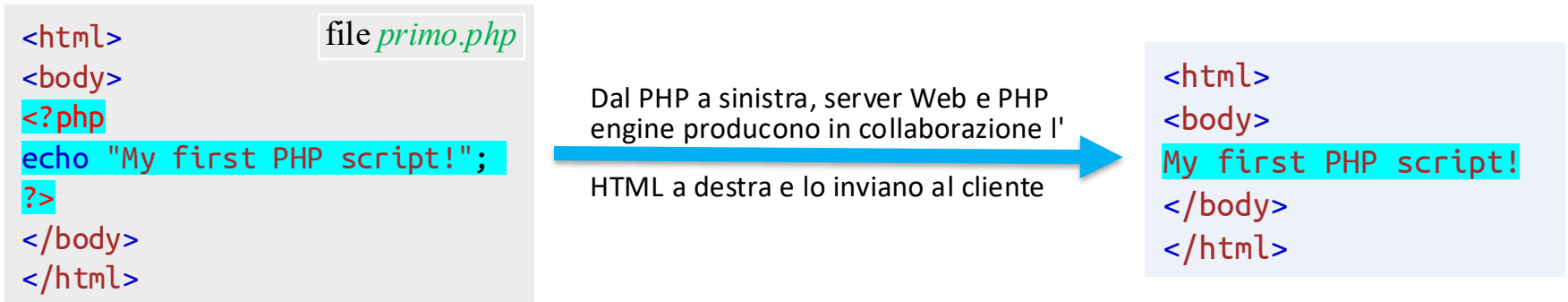
Rispetto alle interazioni già viste tra Browser e Web server + PHP engine, (p. [9](#)), *php -S* funge da **server stand-alone**, cioè li implementa entrambi:



In produzione (v. p. [9](#)) si usano Web server (es. Apache) e PHP engine distinti

Il primo script PHP

- Un file *.php* può contenere HTML, CSS, (JavaScript), e, soprattutto, parti di codice PHP, o *script*, racchiuse ognuna tra i tag `<?php` e `?>`
P. es. l'HTML/PHP nel box a sinistra si può porre in un file *primo.php* nella *document root* di un server Web/PHP engine (tipo *php -S*)



- Se al server arriva, da un cliente, la richiesta HTTP GET `/primo.php`, in risposta (come già visto), il server invia al cliente il contenuto di *primo.php* in cui (ciascuno) script `<?php ... ?>` è rimpiazzato dall'output generato dall'esecuzione dello script stesso
 - a questo scopo, il Web server passa ciascuno script `<?php ... ?>` all'engine PHP e ne attende l'esecuzione e l'output
- Il risultato inviato al cliente è l'HTML nel box di destra

Eeguire il primo script

```
<!-- File primo.php →  
<!DOCTYPE html>  
<html>  
<body><?php  
echo "My first PHP script!";  
>  
</body>  
</html>
```

L'esecuzione di uno script PHP coinvolge (a) il server con l'engine PHP e (b) il cliente con il browser.

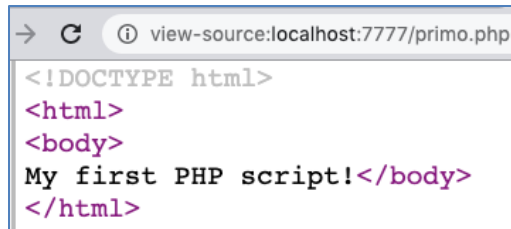
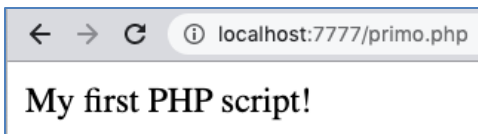
Come detto a p. [17](#), per sviluppo o sperimentazione, conviene far coincidere, come nell'esperimento sotto:

- a) l'host di Server Web+PHP engine (dove gira *php -S*)
- b) l'host cliente, con il browser (verso URL *localhost*)

a) La *document root*, in cui va **primo.php**, è la dir di avvio di *php -S*:

```
$ ls primo.php          # nella directory corrente deve trovarsi il file primo.php  
primo.php  
$ php -S localhost:7777 # ascoltiamo solo richieste dallo stesso host  
...                     # la document root per php -S è la directory corrente
```

b) Ora, sullo stesso host, lanciato un browser, si richiede al server **primo.php** (<http://localhost:7777/primo.php>), causandone l'esecuzione:



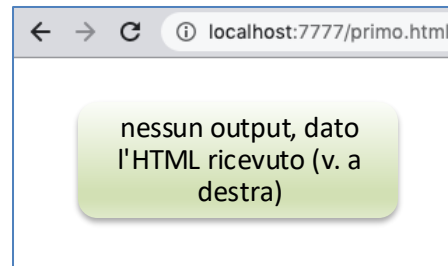
```
$ php -S localhost:7777  
...  
Document root is /Users/gp/.../php2019  
... # il server stampa log delle richieste servite  
[...] [::1]:54378 [200]: GET /primo.php
```

Servire script PHP

Come già visto, *php -S*, per servire al cliente file come **primo.php**, fa eseguire all'engine gli script PHP e li rimpiazza con i rispettivi output. In questo, c'è un aspetto cruciale: **l'estensione .php del file!**

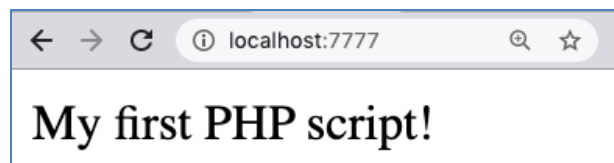
Infatti, se *php -S* servisse un file **primo.html** con lo stesso contenuto di **primo.php**, gli script PHP dentro **primo.html** non verrebbero interpretati e il cliente vedrebbe gli script e non il loro output:

```
$ cp primo.php primo.html
$ $ php -S localhost:7777
... # il server stampa log delle richieste servite
[...][::1]:54378 [200]: GET /primo.html
```

A screenshot of a web browser window showing the source code of 'primo.html'. The code is:

```
<!DOCTYPE html>
<html>
<body>
<?php
echo "My first PHP script!";
?>
</body>
</html>
```

Infine, è utile sapere che *php -S* può anche prendere per **argomento** un file che verrà servito in risposta alla root URL **/** (cioè l'HTTP GET **/**)



```
$ php -S localhost:7777 primo.php
... # il server stampa log delle richieste servite
[Sat May ... 2020] [::1]:57376 Accepted
```

PHP: caratteristiche e punti di forza

- PHP = **PHP: Hypertext Preprocessor** (acronimo ricorsivo)
- Linguaggio di scripting **server-side** per il Web; è **open source**
- Permette di realizzare **pagine web dinamiche**, per lo più con DB di supporto (Web server+PHP e DB sono detti "**back-end**")
- **Efficiente, potente e popolare**: in PHP sono realizzati, p.es., il CMS Wordpress e Facebook (ora su engine *hhvm* per bytecode compilato)
- **Semplice** da imparare: (sintassi C-like, estende HTML in modo naturale)
- Gira su ogni **piattaforma** HW (Intel/...) e OS (Win/Lin/OSX)
- Supportato (come **modulo**) da **web server** come *Apache, IIS, nginx*
- Supporta pressoché tutti i **database** di *back end*, sia relazionali (*mySql, Oracle, Sql server* ...) che non relazionali (*MongoDB* ...)
- Molteplici capacità di **interazione** con l'ambiente (oltre a DB, audio, immagini, email, crittografia...) grazie a librerie interne ed esterne
- Ricca dotazione di librerie o **estensioni**, praticamente per ogni scopo

Cosa può fare PHP

- Può generare pagine dinamiche, il cui contenuto sarà determinato al momento dell'esecuzione del codice PHP, secondo l'input dell'utente e l'"ambiente" (DB, etc.)
- Può gestire (read/write) file sul server
- Può interagire con backend Database (sullo stesso o altri server)
- Può raccogliere dati inseriti in form HTML sul browser
- Può gestire cookie e sessioni
- Supporta la gestione del controllo dell'accesso da parte degli utenti
- Ha capacità crittografiche
- Può generare, in risposta alle richieste dei clienti, sia HTML che altri formati testo o anche binari (PDF, immagini, video...)

PHP: Framework e CMS

Per PHP sono disponibili vari **framework** (complessi di componenti, librerie, pacchetti...) di **sviluppo**, potenti e produttivi, p.es.:

- *Zend* (a oggetti, MVC, con Unit testing e continuous integration)
- *Laravel* (a oggetti, MVC, lo tratteremo in dettaglio)
- *Symfony* (ispirato a *Spring* di Java)

PHP, spesso con lo stack **LAMP** (**L**inux, **A**pache, **M**ySql, **P**HP) ed eventuali framework, supporta i più popolari **CMS** (Content Management Systems, sistemi di gestione di contenuti Web), tra cui:

- *Wordpress*, CMS orientato a pubblicare articoli e blog sul Web
- *Drupal*, CMS general purpose basato sul framework *Symfony*
- *Joomla*, CMS general purpose
- *Magento*, CMS per e-commerce, basato su *Zend*
- *MediaWiki*, CMS di Wikipedia

Considerazioni su PHP

- La comunità di sviluppatori/utenti e l'expertise disponibile, il corredo di framework, librerie, plugin, etc. sono amplissimi 😊
- Molta documentazione disponibile 😊
- Interprete e software a corredo sono gratuiti e open source
- La stragrande maggioranza (>75%, ~5.000.000) dei siti Web è scritta in PHP, tra cui quelli, numerosissimi, basati su Wordpress 😊
- Il 90% dei top 100, 1000 e 1000000 siti usa PHP (p.es. FB e Wikipedia)
- PHP non mantiene alcuno stato tra le richieste (a ogni richiesta ricevuta, lo script viene ricaricato) 😞
- PHP non è brillante nelle prestazioni (cf. Java) 😞
- PHP non ha costrutti nativi per la gestione di richieste concorrenti (cf. NodeJs), che è delegata al server Web), v. <https://www.quora.com/Can-PHP-handle-concurrent-requests>, <https://stackoverflow.com/questions/1430883/simultaneous-requests-to-php-script/> 😞

V. anche www.codemotion.com/magazine/frontend/web-developer/php-for-web-development-in-2022-dead-alive-or-missing-in-action/, www.html.it/articoli/php-vs-nodejs, www.similartech.com/compare/php-vs-python

Risorse online

- <https://www.w3schools.com/php> (seguito per le lezioni)
- https://www.w3schools.com/php/php_oop_what_is.asp
- <https://developer.hyvor.com/tutorials/php>
- <https://www.phptutorial.net> (completo e accurato)
- <https://www.php.net/manual/>
- <https://www.php.net/manual/language.oop5.php> (su oggetti PHP 5)
- <https://www.tutorialspoint.com/php>
- https://www.tutorialspoint.com/php/php_object_oriented.htm
- https://web.archive.org/web/20230408174804mp_/https://www3.ntu.edu.sg/home/ehchua/programming/index.html#php, tutorial conciso ma ricco: [setting up](#), [basics](#), [PHP/MySQL Webapps](#), [OOP in PHP](#), [PHP Miscellaneous](#), [PHP Unit Testing](#)
- <https://www.html.it/guide/guida-php-di-base/>
- <https://www.youtube.com/playlist?list=PL101314D973955661> (fcamuso su youtube)