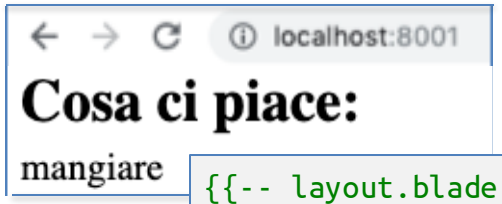
View parametriche

- Consideriamo una view `welcome`, col suo codice, e la relativa route:




```
{{-- layout.blade.php --}}
<html><head></head><body>
@yield('contenuto')
</body></html>
```

```
{{-- welcome.blade.php --}}

@extends('layout')

@section('contenuto')
<h2>Cosa ci piace:</h2>
mangiare
@endsection
```

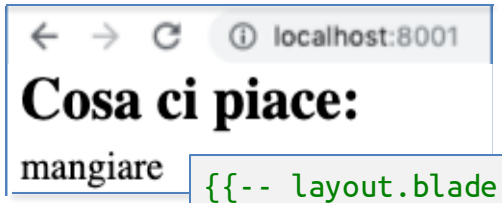
```
<!-- web.php -->

<?php
// Web Routes

Route::get('/', function () {
    return view('welcome');
});
```

- La view `welcome` è evidentemente **statica**: ogni volta che la si **serve** (cioè la si invia) a un browser, in risposta a `GET /` , l'azione preferita resta quella ("mangiare") scritta nell'HTML di *welcome.blade.php*

- È utile rendere la view `welcome` **parametrica** rispetto alla chiamata in *web.php* che la fa servire, cioè `view('welcome',...)` (NB `...`), così:
  1. In *welcome.blade.php* rimpiazzare l'azione specifica "mangiare", con un **parametro** *azione_pref* (sta per *"azione preferita")*
  2. Con il 2° argomento `...` di `view('welcome',...)` assegnare, di volta in volta, al parametro *azione_pref* uno specifico valore mangiare, bere etc.

View parametriche / 1

- Consideriamo quindi la view `welcome`, col suo codice, e la relativa route:




```
{{-- layout.blade.php --}}
<html><head></head><body>
@yield('contenuto')
</body></html>
```

```
{{-- welcome.blade.php --}}

@extends('layout')

@section('contenuto')
<h2>Cosa ci piace:</h2>
mangiare
@endsection
```

```
<!-- web.php -->

<?php
// Web Routes

Route::get('/', function () {
    return view('welcome',...);
});
```

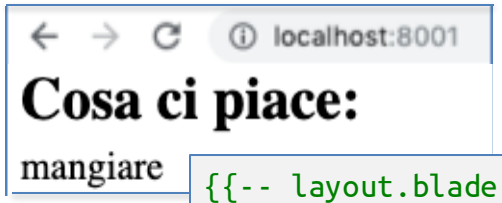
- Come detto, si vuole rendere la view `welcome` parametrica introducendo in *welcome.blade.php* un **parametro** *azione_pref* al posto di "mangiare"
- Ricordando che il file della view può contenere, oltre che codice HTML e notazione blade anche codice PHP, si può introdurre una variabile `$azione_pref` a mo' di parametro, come qui a destra

```
{{-- welcome.blade.php --}}

@extends('layout')

@section('contenuto')
<h2>Cosa ci piace:</h2>
<?php echo $azione_pref ?>
@endsection
```

- p.es. `$azione_pref` nella view `welcome`
  - NB: `<?= expr ?>` equivale a: `<?php echo expr ?>`

View parametriche

- Consideriamo una view `welcome`, con il suo codice, e la relativa route:




```
{{-- layout.blade.php --}}
<html><head></head><body>
@yield('contenuto')
</body></html>
```

```
{{-- welcome.blade.php --}}
sss
@extends('layout')

@section('contenuto')
<h2>Cosa ci piace:</h2>
mangiare
@endsection
```

```
<!-- web.php -->

<?php
// Web Routes

Route::get('/', function () {
    return view('welcome');
});
```

- La view `welcome` è **statica**: ogni volta che la si serve a un browser, in particolare, l'azione preferita ("mangiare") resta quella scritta nel codice
- Sarebbe utile che la view `welcome` fosse **dinamica**, in modo parametrico, cioè "mangiare" possa cambiare, ma non intervenendo sull' HTML in *welcome.blade.php*, bensì sull'invocazione `view('welcome',...)` in *web.php*
- Si può ottenere ciò, introducendo un parametro nella view, p.es. `$azione_pref` nella view `welcome`
  - NB: `<?= expr ?>` equivale a: `<?php echo expr ?>`
- L'invocazione `view('welcome',...)` determina con un oportuno parametro `...` il valore di `$azione_pref` in *welcome.blade.php*

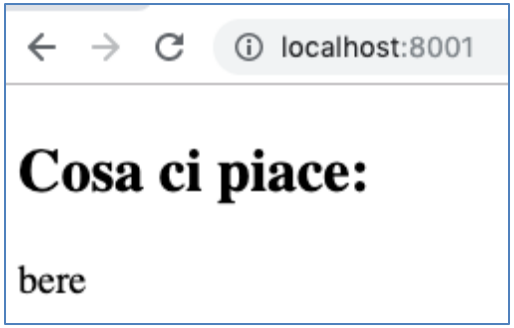
```
{{-- welcome.blade.php --}}

@extends('layout')

@section('contenuto')
<h2>Cosa ci piace:</h2>
<?= $azione_pref ?>
@endsection
```

View parametriche / 2

- Le view Blade possono dunque contenere parametri (variabili), come `$azione_pref`
- Come dare un valore al parametro ogni volta che la view `welcome` (qui a destra) è servita?
- Ricordando che la view `welcome` era attivata da una route invocando la funzione `view('welcome')`, occorre che tale chiamata abbia anche, come 2° argomento, un array hash con chiave `'azione_pref'`
- Il valore `'bere'` corrispondente alla chiave sarà assunto da `$azione_pref` nell'HTML che l'app invierà al browser
- L'HTML inviato è quindi un'*istanza* della view parametrica (ha parametro `$azione_pref`)

```
{{-- welcome.blade.php --}}

@extends('layout')

@section('contenuto')
<h2>Cosa ci piace:</h2>
<?= $azione_pref ?>
@endsection
```

```
<!-- web.php -->

<?php
// Web Routes

Route::get('/', function () {
    return view('welcome',
    ['azione_pref' => 'bere']);
});
```



NB: Se il valore di `$azione_pref` potesse dipendere, p.es., da parti della URL di richiesta (v. oltre), avremmo view oltre che *parametriche*, anche *dinamiche*, ossia dipendenti dalla richiesta HTTP!

View con parametri array

- Il valore passato dalla funzione *view()* al parametro può essere anche un *array*, sul quale la view potrà poi eseguire un ciclo con l'appropriato codice PHP

```
<!-- web.php -->

<?php

Route::get('/', function () {
  return view('welcome', [// view() ha un solo 2° parametro,
    'azioni_pref' =>    // cioè un array con chiave 'azioni_pref'
    ['bere','mangiare'] // e valore ['bere','mangiare'], a sua
  ]);                   // volta un array che sarà assegnato a
});                     // $azioni_pref nella view welcome (sotto)
```

- Ecco come la view *welcome* sfrutta l'array che le viene passato dalla route come valore del parametro *$azioni_pref*

```
{{-- welcome.blade.php --}}

@extends('layout')

@section('contenuto')
<h2>Cosa ci piace:</h2>
<?php foreach ($azioni_pref as $una_azione) : ?>
  <br> <?= $una_azione; ?>
<?php endforeach; ?>
@endsection
```

- NB: la sintassi del blocco PHP (`foreach`) aperto nel primo script `<?php` viene completata nel successivo – è una caratteristica utile ma poco elegante!

Cosa ci piace:

bere
mangiare

Annotazioni/abbreviazioni in blade

- Il componente *blade* di Laravel permette annotazioni equivalenti ai tag `<?php... ?>` , ma più concise ed eleganti
- Si confrontino, p.es., le due versioni di *welcome.blade.php*:

```
{{-- welcome.blade.php --}}

@extends('layout')

@section('contenuto')
<h2>Cosa ci piace:</h2>
<?php foreach ($azioni_pref as $una_azione) : ?>
  <br> <?= $una_azione ?>
<?php endforeach; ?>
@endsection
```

```
{{-- welcome.blade.php --}}

@extends('layout')

@section('contenuto')
<h2>Cosa ci piace:</h2>
@foreach ($azioni_pref as $una_azione)
<br> {{ $una_azione }}
@endforeach
@endsection
```

- La traduzione in PHP dei tag Blade `@foreach` e simili avviene solo la prima volta che la pagina è servita (per efficienza)
- Ulteriore concisione si ottiene rimpiazzando, come qui sopra a destra, l'espressione `<?= ... ?>` con `{{...}}`

Accorgimenti per il parametro

- Nel file delle rotte il valore del parametro (come l'array `['bere','mangiare']`, valore di `$azioni_pref`), se molto strutturato, si può specificare in modo più chiaro assegnandolo *prima* a una variabile locale alla funzione callback della route /

```
<!-- web.php -->

<?php

Route::get('/', function () {
  return view('welcome', [ // un solo 2° parametro
      'azioni_pref' =>   // chiave e parametro view
      ['bere','mangiare']// valore passato alla view
  ]);                    // per il parametro
});                      // $azione_pref
```

- Cioè come per esempio la variabile *$lista_azioni* qui a destra:

```
<!-- web.php -->

<?php

Route::get('/', function () {
  $lista_azioni = ['bere','mangiare'];
  return view('welcome', [ // un solo 2° parametro
      'azioni_pref' =>   // chiave e parametro view
      $lista_azioni      // valore passato alla view
  ]);                    // per il parametro
});                      // $azione_pref
```

Due parametri nella view

- Vediamo un esempio, con **due parametri** *$azioni_pref* e *$titol* nella view *welcome* istanza del template *layout*
  - adesso, però, nel callback del routing, la view *welcome* dovrà essere invocata in modo da legare un valore a *$azioni_pref* e un altro a *$titol* ...
- Per questo, nel callback la chiamata a *view()* ha ora, rispetto a prima, anche un 3° argomento:
  la coppia hash con chiave *'titol'* e valore *'Benvenuti'*, che sarà assegnato al parametro *$titol* della view *welcome*

```
{{-- layout.blade.php --}}
<html><head><title>
@yield('titolo')
</title></head>
<body>
@yield('contenuto')
</body>
</html>
```

```
{{-- welcome.blade.php --}}

@extends('layout')
@section('titolo', $titol)

@section('contenuto')
<h2>Cosa ci piace:</h2>
@foreach ($azioni_pref as $una_azione)
<br> <?= $una_azione; ?>
@endforeach
@endsection
```

```
<!-- web.php -->

<?php
Route::get('/', function () {
  $lista_azioni = ['bere','mangiare'];
  return view('welcome',
    [ 'azioni_pref' => $lista_azioni ],
    [ 'titol' => 'Benvenuti' ]
  );
});
```

Parametri multipli nella view

- Vediamo un altro esempio, con un terzo parametro *$quando* nella view *welcome*...
- Ma, per dargli un valore, non si può aggiungere a *view()*, nel callback in *web.php*, un 4° argomento (errore!)

```
{{-- welcome.blade.php --}}

@extends('layout')
@section('titolo', $titol)

@section('contenuto')
<h2>Cosa ci piace {{$quando}}:</h2>
@foreach ($azioni_pref as $una_azione)
<br> <?= $una_azione; ?>
@endforeach
@endsection
```

- Bisogna invece dare a *view()*, come 2° argomento, un **array di coppie hash**: ognuna di queste ha una
  - **chiave** nome di un parametro della view (p.es. *'quando'*) e un
  - **valore** che verrà assegnato al parametro (p.es. *'oggi'*)

```
<!-- web.php -->

<?php
Route::get('/', function () {
  $lista_azioni = ['bere','mangiare'];
  return view('welcome', [
    'azioni_pref' => $lista_azioni ,
    'titol' => 'Benvenuti',
    'quando' => 'oggi'
  ]);
});
```

- Così è possibile gestire parametri multipli nella view!

Parametri della URL per view dinamiche

- Il valore da assegnare a una variabile di view come *$quando* può anche essere "estratto" dalla URL, rendendo così l'HTML generato dalla view veramente dinamico (cioè dipendente dalla URL di richiesta)
- Per l'estrazione, nel callback di route, si usa la funzione PHP *request('id')*, che restituisce il valore del parametro *id* nella URL
- Nell'esempio qui, *request('tempo')* estrae dalla URL il parametro *?tempo=oggi*

```php
<!-- web.php -->

<?php
Route::get('/', function () {
   $azioni = ['bere','mangiare'];
   return view('welcome', [
      'azione_pref' => $azioni ,
      'quando' => request('tempo')
   ]);
});
```

```blade
{{-- welcome.blade.php --}}

@extends('layout')

@section('contenuto')
<h2>Cosa ci piace {{$quando}}:</h2>
...
```

Parametri da richiesta HTTP per view dinamiche

- Nell'esempio precedente il valore della variabile di view *$quando*, è estratto con la chiamata *request('tempo')* nel callback di route in *web.php*
- Il parametro *'tempo'* proviene indirettamente dalla URL nel browser ma in realtà è nel messaggio di richiesta HTTP GET, come illustrato qui sotto:

```php
<!-- web.php -->
<?php
Route::get('/', function () {
   $azioni = ['bere','mangiare'];
   return view('welcome', [
      'azione_pref' => $azioni ,
      'quando' => request('tempo')
]); });
```

```
{{-- welcome.blade.php --}}
@extends('layout')
@section('contenuto')
<h2>Cosa ci piace {{$quando}}:</h2>
...
```

```
$ telnet localhost 8000
Connected to localhost.
Escape character is '^]'.
GET /?tempo=domani          #  inserire GET... e battere due RETURN

HTTP/1.1 200 OK
...
<!DOCTYPE html>
...
<h2>Cosa ci piace domani </h2>
...
Connection closed by foreign host.
```

Attacchi XSS e Blade

1. Dunque Blade (in generale PHP) consente a chi si collega a pagine come *welcome.blade.php* di determinare, con la *request* GET, il contenuto di una variabile, come *$quando*
2. Spesso accade che il codice PHP lato server invii al browser il contenuto della variabile per visualizzarlo, come fa (qui a destra) *welcome.blade.php* con *$quando*

```php
<!-- web.php -->
<?php
Route::get('/', function () {
  $azioni = ['bere','mangiare'];
  return view('welcome', [
    'quando' => request('tempo')]);
});
```

```
{{-- welcome.blade.php --}}
<h2>Cosa ci piace {{$quando}}:</h2>
...
```

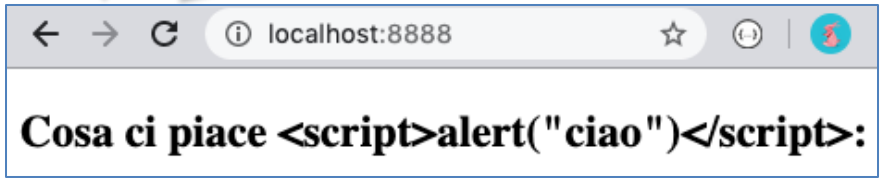
- Qui, in un uso perfettamente legittimo di (1,2), *www.passatempi.org* mostra un link verso sé:
- Gli attacchi *XSS* (cross-site scripting) sfruttano la fase (1) sopra per causare l'*iniezione* di codice Javascript malevolo nella variabile, nella speranza che nella fase (2) l'invio del Javascript al browser ne determini l'esecuzione da parte di questo
- Per attivare XSS, basta indurre un utente a cliccare un link adatto...

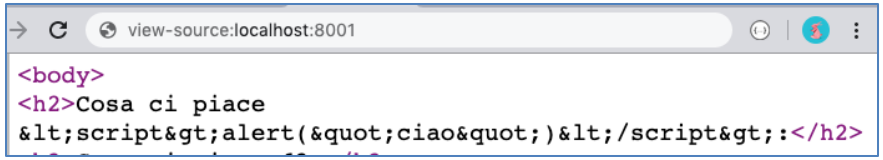
{{...}} - protegge

- {{...}} di Blade non equivale *esattamente* a <?= ... ?>
- Anche se, come <?= ... ?>, Blade passa ... a PHP per la valutazione...
- Il valore restituito da PHP viene filtrato con la funzione PHP *htmlspecialchars*(), che elimina, in particolare, i segni "minore" e "maggiore" che delimitano i tag
- Insomma, {{...}} valuta ... ma elimina i tag dal risultato!

```php
<!-- web.php -->
<?php
Route::get('/', function () {
    $azioni = ['bere','mangiare'];
    return view('welcome', [
        'azione_pref' => $azioni ,
        'quando' =>
        '<script>alert("ciao")</script>'
    ]);
});
```

```
{{-- welcome.blade.php --}}
@extends('layout')
@section('contenuto')
<h2>Cosa ci piace {{$quando}}:</h2>
```





- Quindi, anche se ... fosse una variabile contenente una stringa con del javascript, in Blade {{...}} **protegge**, evita che il javascript sia eseguito!
- Nell'esempio in figura, si vede che la view *welcome* riceve una variabile *$quando* contenente javascript, la valuta, ma nel risultato il javascript viene visualizzato, non già eseguito (v. finestra *view-source*)

{{...}} vs. <?= ... ?>

- {{...}} di Blade non equivale *esattamente* a <?= ... ?> come si vede qui a fianco
- Se la variabile PHP in ... contiene una stringa con codice javascript!
- {{...}} protegge, cioè evita l'esecuzione del javascript
- Invece <?= ... ?> lascia il javascript in ... inalterato...

così, in *welcome.blade.php*

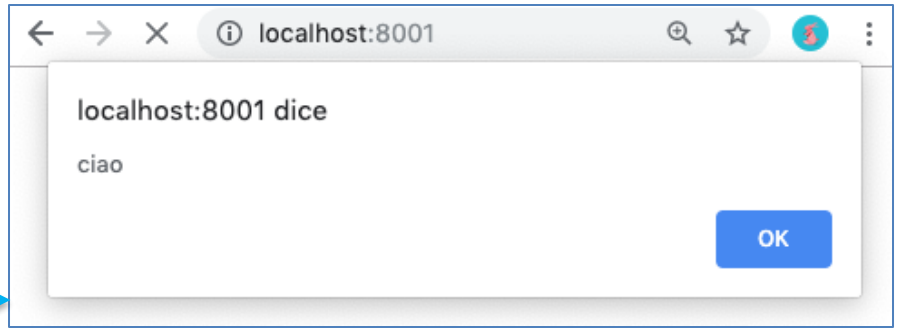
```
<h2>Cosa ci piace <?= $quando ?>:</h2>
```

produce

eseguendo il javascript (v. *view-source:*)

```
<!-- web.php -->
<?php
Route::get('/', function () {
    $azioni = ['bere','mangiare'];
    return view('welcome', [
        'azione_pref' => $azioni ,
        'quando' =>
        '<script>alert("ciao")</script>'
    ]);
});
```

```
{{-- welcome.blade.php --}}
@extends('layout')
@section('contenuto')
<h2>Cosa ci piace <?= $quando ?>:</h2>
```



{{...}} e {!! ... !!}

- Dunque, `{{...}}` protegge (da) il proprio argomento (espressione `...`) che potrebbe contenere codice Javascript non previsto o, peggio, malizioso, con effetti comunque indesiderati
- Cautele simili si dovrebbero avere con il codice che verrà valutato all'interno di tag `<?php` e `?>`
  - di norma esso dovrebbe essere innocuo, ma potrebbe non esserlo se proviene in parte da parametri di GET (`?XXX=YYY`) o da form inviati con POST
  - in questi casi `{{...}}` dà protezione, perché fa valutare `...` da PHP, ma filtra il valore prodotto con *htmlspecialchars()* (si dice che "quota" o "protegge" il proprio argomento)
  - tale filtro non opera, invece, con il costrutto Blade `{!!...!!}` che presenta quindi gli stessi rischi di `<?= ... ?>`
  - si suole dire che il codice in `...` "va considerato colpevole finché non lo si dimostra innocente"!

Facility per variabili nelle view: ->*with()*

- Come sappiamo, in una route, la funzione *view()* usa il 2° parametro, un array `['k1'=>'V1','k2'=>'V2',...]` per assegnare V1 alla var *$k1*, V2 a *$k2* ... nella view blade che è il 1° parametro di *view()*

```php
<!-- web.php -->
<?php
Route::get('/', function () {
    return view('welcome', [
        'azione' => 'bere',
        'quando' => 'oggi' ]);
});
```

- Lo stesso effetto si ottiene applicando a *view*, col solo 1° parametro, il metodo *with()* applicato all'array già usato come 2° arg
  - NB: *with()* è un metodo della classe (*View*) restituita dalla chiamata *view()*

```php
Route::get('/', function () {
    return view('welcome')->with( [
        'azione' => 'bere',
        'quando' => 'oggi' ] );
});
```

- L'uso di *with()* è molto libero e ammette diverse varianti...

Facility per variabili nelle view: ->*with*

- L'uso base di *with()* mostrato di nuovo qui a destra, ammette molte varianti, semplificazioni, etc.xx

```
Route::get('/', function () {
   return view('welcome')->with( [
      'azione' => 'bere',
      'quando' => 'oggi' ] );
});
```

- può convivere col 2° parametro
- si può applicare più volte (perché restituisce sempre View)
- se l'argomento array di *with()* ha solo una coppia, lo si può sostituire con i due membri della coppia:
- NB: è bene che il nome della chiave (p.es. xyz) contenga solo minuscole

```
Route::get('/', function () {
   return view('welcome', [
      'azione' => 'bere'] )->with( [
      'quando' => 'oggi' ] );
});
```

```
Route::get('/', function () {
   return view('welcome')->with( [
      'azione' => 'bere'])->with( [
      'quando' => 'oggi']);
});
```

Facility per variabili nelle view: *compact()*

- Come già detto, in una route, la funzione *view()* usa il 2° parametro per associare un valore alla chiave '*xyz*', in modo che *$xyz* figuri come variabile nella view blade che è il 1° parametro di *view()*

- Si consideri ora un caso come qui a destra, in cui:
  - il 2° parametro definisce una sola chiave '*azioni*'
  - il valore assegnato alla chiave sia quello di una variabile chiamata anche *$azioni*

```
<!-- web.php -->
<?php
Route::get('/', function () {
   $azioni = ['bere','mangiare'];
   return view('welcome',
      ['azioni' => $azioni ]);
});
```

- L'array hash, in questo caso, si può esprimere mediante la funzione PHP *compact()*, come qui a destra:

```
<!-- web.php -->
<?php
Route::get('/', function () {
   $azioni = ['bere','mangiare'];
   return view('welcome',
                compact('azioni'));
});
```