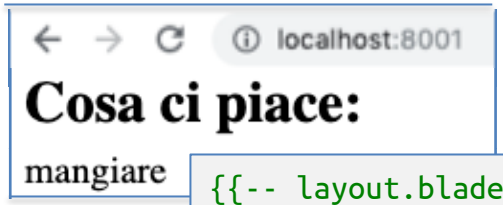
View parametriche

- Consideriamo una view, con il 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 **statica**: ogni volta che si serve *welcome*, p.es., il verbo ("mangiare") resta quello scritto nel codice
- Sarebbe utile che la view *welcome* fosse **dinamica**, in modo parametrico, cioè che il verbo cambi, da un rendering all'altro, senza che cambi il codice *welcome.blade.php*, ma solo l'invocazione *view('welcome',...)* in *web.php*
- Si può, per questo, introdurre un parametro nella view, p.es. *$azione_pref* nella view *welcome*
  - NB: il tag `<?=...?>` viene rimpiazzato dal valore del PHP
- Ma come si assegna un valore ad *$azione_pref* ogni volta che questa view *welcome* viene servita?

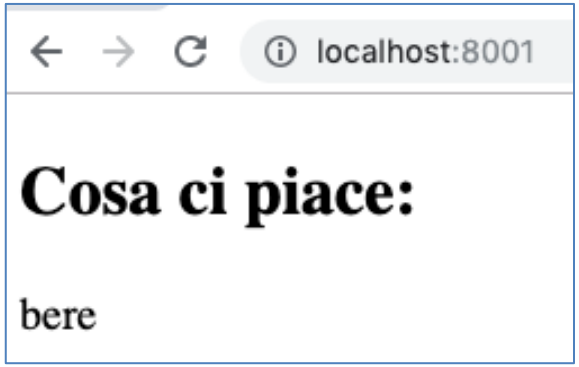
```
{{-- welcome.blade.php --}}

@extends('layout')

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

View parametriche: meccanismo

- Dunque: come riceve un valore il parametro *$azione_pref* ogni volta che la view *welcome* (a destra) viene servita?
- Ricordando che la view *welcome* veniva attivata da una route che invoca la funzione *view('welcome')*, occorre che questa chiamata abbia, per 2° argomento, un array hash con chiave *'azione_pref'*
- Il valore corrispondente alla chiave (qui *'bere'*) verrà assunto da *$azione_pref* nella view da servire
- Ecco che cambia la view servita!
- Se il valore di *$azione_pref* potesse dipendere, p.es., da parti della URL di richiesta, avremmo view ancor più che *parametriche*, cioè davvero *dinamiche,* ovvero dipendenti dalla richiesta HTTP!

```
{{-- 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']);
});
```



View parametriche: array

- Il valore passato dalla route al parametro può essere anche un array, sul quale la view potrà 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' =>    // $azioni_pref è chiave e parametro view
      ['bere','mangiare']// valore passato per $azioni_pref alla
   ]);                    // view welcome
});
```

- 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 ma più concise ed eleganti dei tag `<?php... ?>`
- Si confrontino, p.es., 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 dei tag blade `@foreach` e simili avviene solo la prima volta che la pagina è servita (per maggiore efficienza)
- Si può ottenere ancora più concisione rimpiazzando, come nell'esempio sopra a destra, l'espressione `<?= ... ?>` con `{{...}}` (senza '`;`' nel PHP!)

Altre facility sintattiche per il parametro

- Nel file delle rotte il valore del parametro (come l'array `['bere','mangiare']` valore di `$azioni_pref`), se complesso, si può specificare in modo più chiaro assegnandolo *prima* a una variabile locale alla closure associata alla 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*
  - ciò consente al routing che invoca la view *welcome* di rendere anche il titolo parametrico

```
{{-- 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
```

- Nella gestione delle rotte in *web.php*, nella closure, la funzione *view()* ha ora, rispetto a prima, un 3° argomento: la coppia hash con chiave *'titol'* e valore *'Benvenuti'*, corrispondente quindi al 2° parametro *$titol* della view *welcome*

```
<!-- 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*
- Non lo si può però istanziare dando a *view()*, nella rotta in *web.php*, un 4° argomento (causa 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 un numero qualsiasi di parametri!

Parametri della URL

- Il valore da assegnare a una variabile come *$quando*, nella gestione della route in *web.php* può anche essere estratto dalla URL, rendendo così l'HTML generato dalla view veramente dinamico
- Per questo scopo si usa la funzione *request('id')*, che va a cercare un 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');
   ]);
});
```

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

@extends('layout')

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

{{...}} - particolarità

- {{...}} non equivale esattamente a <?= ... ?> come si vede qui a fianco
- La variabile in ... potrebbe contenere una stringa con codice javascript!
- {{...}} protegge, in quanto non interpreta il codice!
- Invece <?= ... ?> interpreta il codice javascript in ...

Così, in *welcome.blade.php*

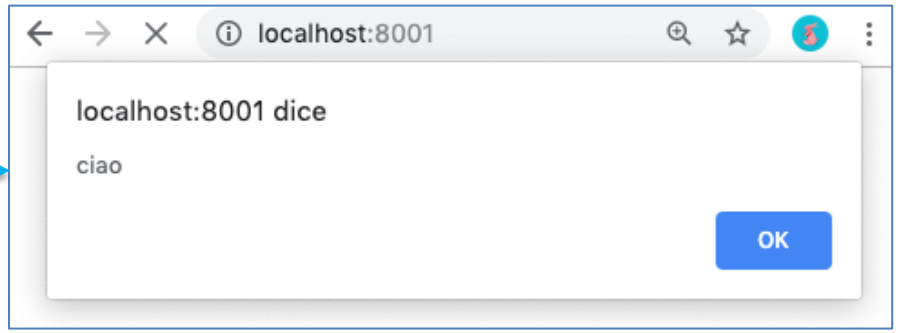
```
<h2>Cosa ci piace <?= $quando ?>:</h2>
```

produce

perché l'*alert()* viene interpretato

```
<!-- 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 {!! ... !!}

- Come detto, `{{...}}` protegge il proprio argomento (espressione) perché potrebbe essere 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 ... ?>`
  - di norma dovrebbe essere PHP legale, ma potrebbe non esserlo se proviene in parte da parametri GET (`?XXX=YYY`) o da form inviati con POST
  - in questi casi `{{...}}` dà maggiore protezione (si dice che "quota" o "protegge" il proprio argomento)
  - ciò non accade, invece, con il costrutto blade `{!!...!!}` che presenta quindi dei rischi come `<?php ... ?>`
  - si dice che il codice in `...` "va considerato colpevole finché non lo si dimostra innocente"!

Facility per variabili nelle view: ->*with*

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

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

- Lo stesso effetto si può ottenere con l'operatore ->*with**X**yz(V)*, come qui, nel 2° riquadro, equivalente al 1°
- NB: è bene che il nome della chiave (p.es. *xyz*) contenga solo minuscole

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

- Altra forma equivalente nel 3° riquadro: unico *with* applicato a tutto un hash-array che associa più chiavi ciascuna a un valore

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

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'));
});
```