

## What is PHP

PHP is an open-source, interpreted, and object-oriented scripting language that can be executed at the server-side. PHP is well suited for web development. Therefore, it is used to develop web applications (an application that executes on the server and generates the dynamic page.).

PHP was created by Rasmus Lerdorf in 1994 but appeared in the market in 1995. PHP 8 is the latest version of PHP. Some important points need to be noticed about PHP are as followed:

- PHP stands for Hypertext Preprocessor.
- PHP is an interpreted language, i.e., there is no need for compilation.
- PHP is faster than other scripting languages, for example, ASP and JSP.
- PHP is a server-side scripting language, which is used to manage the dynamic content of the website.
- PHP can be embedded into HTML.
- PHP is an object-oriented language.
- PHP is an open-source scripting language.
- PHP is simple and easy to learn language.



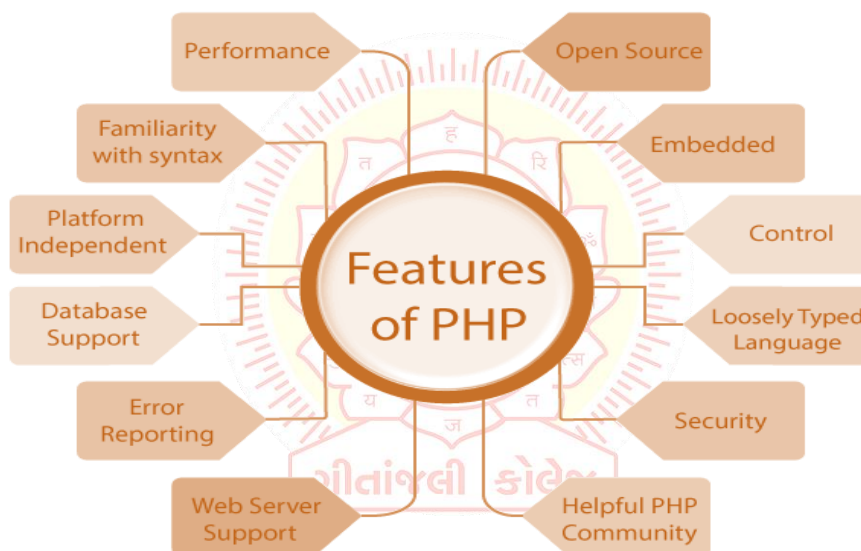
## Why use PHP

PHP is a server-side scripting language, which is used to design the dynamic web applications with MySQL database.

- It handles dynamic content, database as well as session tracking for the website.
- You can create sessions in PHP.
- It can access cookies variable and also set cookies.
- It helps to encrypt the data and apply validation.
- PHP supports several protocols such as **HTTP, POP3, SNMP, LDAP, IMAP**, and many more.
- Using PHP language, you can control the user to access some pages of your website.
- As PHP is easy to install and set up, this is the main reason why PHP is the best language to learn.

## PHP Features

PHP is very popular language because of its simplicity and open source. There are some important features of PHP given below:



### Performance:

PHP script is executed much faster than those scripts which are written in other languages such as JSP and ASP. PHP uses its own memory, so the server workload and loading time is automatically reduced, which results in faster processing speed and better performance.

### Open Source:

PHP source code and software are freely available on the web. You can develop all the versions of PHP according to your requirement without paying any cost. All its components are free to download and use.

### Familiarity with syntax:

PHP has easily understandable syntax. Programmers are comfortable coding with it.

### Embedded:

PHP code can be easily embedded within HTML tags and script.

**Platform Independent:**

PHP is available for WINDOWS, MAC, LINUX & UNIX operating system. A PHP application developed in one OS can be easily executed in another OS also.

**Database Support:**

PHP supports all the leading databases such as MySQL, SQLite, ODBC, etc.

**Error Reporting -**

PHP has predefined error reporting constants to generate an error notice or warning at runtime. E.g., E\_ERROR, E\_WARNING, E\_STRICT, E\_PARSE.

**Loosely Typed Language:**

PHP allows us to use a variable without declaring its datatype. It will be taken automatically at the time of execution based on the type of data it contains on its value.

**Web servers Support:**

PHP is compatible with almost all local servers used today like Apache, Netscape, Microsoft IIS, etc.

**Security:**

PHP is a secure language to develop the website. It consists of multiple layers of security to prevent threads and malicious attacks.

**Control:**

Different programming languages require long script or code, whereas PHP can do the same work in a few lines of code. It has maximum control over the websites like you can make changes easily whenever you want.

**A Helpful PHP Community:**

It has a large community of developers who regularly updates documentation, tutorials, online help, and FAQs. Learning PHP from the communities is one of the significant benefits.

**Web Development**

PHP is widely used in web development nowadays. PHP can develop dynamic websites easily. But you must have the basic the knowledge of following technologies for web development as well.

- HTML, CSS, JavaScript, Ajax, XML and JSON, jQuery
- Prerequisite

Before learning PHP, you must have the basic knowledge of HTML, CSS, and JavaScript. So, learn these technologies for better implementation of PHP.

**HTML** - HTML is used to design static webpage.

**CSS** - CSS helps to make the webpage content more effective and attractive.

**JavaScript** - JavaScript is used to design an interactive website.

## PHP WITH OOPS CONCEPT

Object-oriented programming is a programming model organized around Object rather than the actions and data rather than logic.

### Class

A class is an entity that determines how an object will behave and what the object will contain. In other words, it is a blueprint or a set of instruction to build a specific type of object. In PHP, declare a class using the class keyword, followed by the name of the class and a set of curly braces ({}).

#### Syntax to Create Class in PHP

```
<?php
class MyClass {
    // Class properties and methods go here
}
?>
```

A class defines an individual instance of the data structure. We define a class once and then make many objects that belong to it. Objects are also known as an instance.

An object is something that can perform a set of related activities.

#### Syntax:

```
<?php
class MyClass {
    // Class properties and methods go here
}
$obj = new MyClass;
var_dump($obj);
?>
```

#### Example of class and object:

```
<?php
class demo {
    private $a= "Hello Geetanjali";
    public function display(){
        echo $this->a;
    }
}
$obj = new demo();
$obj->display();
?>
```

#### Output:

Hello Geetanjali

## Properties

Class member variables are called "properties". You may also see them referred to using other terms such as "**attributes**" or "**fields**", but for the purposes of this reference we will use "properties". They are defined by using one of the keywords **public**, **protected**, or **private**, followed by a normal variable declaration. This declaration may include an initialization, but this initialization must be a constant value--that is, it must be able to be evaluated at compile time and must not depend on run-time information in order to be evaluated.

<pre>&lt;?php class SimpleClass{ // valid as of PHP 5.6.0: public \$var1='hello '. 'geetanjali'; // valid as of PHP 5.3.0: public \$var2=&lt;&lt;&lt;&lt;college hello Geetanjali college; // valid as of PHP 5.6.0: public \$var3=1+2; public \$var7=array(true,false); // valid as of PHP 5.3.0: public \$var8=&lt;&lt;&lt;&lt;'geet' hello Geetanjali College geet; } \$smpClass= new SimpleClass(); echo \$smpClass-&gt;var1."&lt;BR&gt;"; echo \$smpClass-&gt;var2."&lt;BR&gt;"; echo \$smpClass-&gt;var7[0]."&lt;BR&gt;"; echo \$smpClass-&gt;var8."&lt;BR&gt;"; ?&gt;</pre>	<pre>OP hello geetanjali hello Geetanjali 1 hello Geetanjali College</pre>
--	--

## Class Constants

It is possible to define constant values on a per-class basis remaining the same and unchangeable. Constants differ from normal variables in that you don't use the \$ symbol to declare or use them. The default visibility of class constants is **public**. It's also possible for interfaces to have **constants**.

```
<?php
class MyClass{
    const geet='constant value';
    function showConstant(){
        echo self::geet;
    }
}
echo MyClass::geet;
$classname="MyClass";
echo $classname::geet;// As of PHP 5.3.0

$class = new MyClass();
$class->showConstant();
echo $class::geet;// As of PHP 5.3.0
?>
```

OP: constant valueconstantvalueconstantvalueconstant value

## Autoloading Classes

Many developers writing object-oriented applications create one PHP source file per class definition. One of the biggest annoyances is having to write a long list of needed includes at the beginning of each script (one for each class).

```
<?php
spl_autoload_register( function ($class_name) {
    include $class_name . '.php';
});
$obj = new MyClass1();
$obj2 = new MyClass2();
?>
```

In PHP 5, this is no longer necessary. The **spl\_autoload\_register()** function registers any number of autoloaders, enabling for classes and interfaces to be automatically loaded if they are currently not defined. By registering autoloaders, PHP is given a last chance to load the class or interface before it fails with an error.

## Constructors and Destructors

### Constructor

**void \_\_construct ([ mixed \$args = "" [, \$... ] ] )**

PHP 5 allows developers to declare constructor methods for classes. Classes which have a constructor method call this method on each newly-created object, so it is suitable for any initialization that the object may need before it is used.

For backwards compatibility with PHP 3 and 4, if PHP cannot find a **\_\_construct()** function for a given class, it will search for the old-style constructor function, by the name of the class. Effectively, it means that the only case that would have compatibility issues is if the class had a method named **\_\_construct()** which was used for different semantics.

Unlike with other methods, PHP will not generate an E\_STRICT level error message when **\_\_construct()** is overridden with different parameters than the parent **\_\_construct()** method has.

<pre>&lt;?php class BaseClass {     function __construct() {         print "InBaseClass constructor\n";     } } class SubClass extends BaseClass {     function __construct() {         parent::__construct();         print "InSubClass constructor\n";     } } class OtherSubClass extends BaseClass {     // inherits BaseClass 's constructor } // In BaseClass constructor \$obj = new BaseClass ();  // In BaseClass constructor // In SubClass constructor \$obj = new SubClass ();  // In BaseClass constructor \$obj = new OtherSubClass (); ?&gt;</pre>	<p>OP:</p> <ul style="list-style-type: none"> <li>In BaseClass constructor</li> <li>In BaseClass constructor</li> <li>In SubClass constructor</li> <li>In BaseClass constructor</li> </ul>
---	--



## Destructor

### void \_\_destruct ( void )

PHP 5 introduces a destructor concept similar to that of other object-oriented languages, such as C++. The destructor method will be called as soon as there are no other references to a particular object, or in any order during the shutdown sequence.

Like constructors, parent destructors will not be called implicitly by the engine. In order to run a parent destructor, one would have to explicitly call **parent::\_\_destruct()** in the destructor body. Also like constructors, a child class may inherit the parent's destructor if it does not implement one itself.

The destructor will be called even if script execution is stopped using **exit()**. Calling **exit()** in a destructor will prevent the remaining shutdown routines from executing.

```
<?php
class new Account {
function happy(){
echo "<br>hello<br>";
}
function __construct(){
echo "Creating new Savings Account<BR>";
print"Adding MIN Balance : 5000<BR>";
print"-----<BR>";
print"-----<BR>";
$this->name="Mr. ABC";
}
function __destruct(){
print"Auto Generating Complete. <BR>Removing Traces for
".$this->name;
}
}
$obj= new new Account();
$obj->happy();
?>
```

### Output

```
Creating new Savings Account
Adding MIN Balance : 5000
-----
-----
hello
Auto Generating Complete.
Removing Traces for Mr. ABC
```



## Visibility

The visibility of a property, a method or (as of PHP 7.1.0) a constant can be defined by prefixing the declaration with the keywords **public**, **protected** or **private**. Class members declared public can be accessed everywhere. Members declared protected can be accessed only within the class itself and by inheriting and parent classes. Members declared as private may only be accessed by the class that defines the member.

### Property Visibility

Class properties must be defined as **public**, **private**, or **protected**. If declared using var, the property will be defined as public.

```
<?php
class MyClass{
public $a='Public ';
protected $b='Protected ';
private $c='Private ';


function printHello() {
echo $this->a;
echo $this->b;
echo $this->c;
}
}

$obj=new MyClass();
echo $obj->a; // Works
echo $obj->b; // Fatal Error
echo $obj->c; // Fatal Error
$obj->printHello(); // Shows Public, Protected and Private

class MyClass2 extends MyClass{
// We can redeclare the public and protected properties, but not private
public $a='Public 2';
protected $b='Protected 2';

function printHello() {
echo $this->a;
echo $this->b;
echo $this->c;
}
}

$obj2=new MyClass2();
echo $obj2->a; // Works
echo $obj2->b; // Fatal Error
echo $obj2->c; // Undefined
$obj2->printHello(); // Shows Public 2, Protected 2, Undefined
?>
```



## Method Visibility

Class methods may be defined as **public**, **private**, or **protected**. Methods declared without any explicit visibility keyword are defined as public.

```
<?php

class MyClass{
// Declare a public constructor
public function __construct() {}

// Declare a public method
public function MyPublic() {}

// Declare a protected method
protected function MyProtected() {}

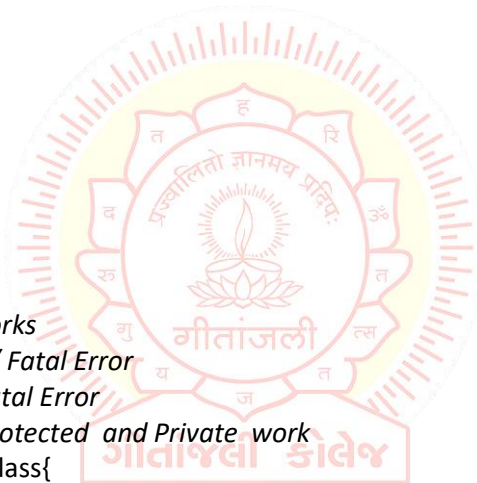
// Declare a private method
private function MyPrivate() {}

// This is public
function Foo() {
$this->MyPublic();
$this->MyProtected();
$this->MyPrivate();
}

$myclass= new MyClass;
$myclass->MyPublic(); // Works
$myclass->MyProtected(); // Fatal Error
$myclass->MyPrivate(); // Fatal Error
$myclass->Foo(); // Public, Protected and Private work
class MyClass2 extends MyClass{
// This is public
function Foo2()
{
$this->MyPublic();
$this->MyProtected();
$this->MyPrivate(); // Fatal Error
}
}

$myclass2=new MyClass2;
$myclass2->MyPublic(); // Works
$myclass2->Foo2(); // Public and Protected work, not Private

class Bar {
public function test() {
$this->testPrivate();
$this->testPublic();
}
}
```



```

public function testPublic() {
    echo"Bar::testPublic\n";
}

private function testPrivate() {
    echo"Bar::testPrivate\n";
}
}

class Foo extends Bar{
    public function testPublic() {
        echo"Foo::testPublic\n";
    }

    private function testPrivate() {
        echo"Foo::testPrivate\n";
    }
}

$myFoo= new Foo();
$myFoo->test(); // Bar::testPrivate
// Foo::testPublic
?>

```

### Constant Visibility

As of PHP 7.1.0, class constants may be defined as public, private, or protected. Constants declared without any explicit visibility keyword are defined as public.

```

<?php
class MyClass
{
    // Declare a public constant
    public const MY_PUBLIC='public';

    // Declare a protected constant
    protected const MY_PROTECTED='protected';

    // Declare a private constant
    private const MY_PRIVATE='private';

    public function foo()
    {
        echo self::MY_PUBLIC;
        echo self::MY_PROTECTED;
        echo self::MY_PRIVATE;
    }
}

```

```

$myclass= new MyClass();
MyClass::MY_PUBLIC ; // Works
MyClass::MY_PROTECTED ; // Fatal Error
MyClass::MY_PRIVATE ; // Fatal Error
$myclass->foo(); // Public,Protected and Private work
class MyClass2 extends MyClass
{
// This is public
function foo2()
{
echo self::MY_PUBLIC ;
echo self::MY_PROTECTED ;
echo self::MY_PRIVATE ; // Fatal Error
}
}

$myclass2=new MyClass2;
echo MyClass2::MY_PUBLIC ; // Works
$myclass2->foo2(); // Public and Protected work, not Private
?>

```

### Visibility from other objects

Objects of the same type will have access to each other's private and protected members even though they are not the same instances. This is because the implementation specific details are already known when inside those objects.

```

<?php
class Test{
private $foo;
public function __construct($foo){
$this->foo=$foo;
}

private function bar(){
echo'Accessed the private method.';
}

public function baz(Test $other){
// We can change the private property:
$other->foo='hello';
var_dump($other->foo);

// We can also call the private method:
$other->bar();
}
}
$test= new Test('test');
$test->baz( new Test('other'));
?>

```

**Output:** string(5) "hello"Accessed the private method.

## Object Inheritance

Inheritance is a well-established programming principle, and PHP makes use of this principle in its object model. This principle will affect the way many classes and objects relate to one another.

**For example**, when you extend a class, the subclass inherits all of the public and protected methods from the parent class. Unless a class overrides those methods, they will retain their original functionality.

This is useful for defining and abstracting functionality, and permits the implementation of additional functionality in similar objects without the need to re-implement all of the shared functionality.

```
<?php
class Foo{
public function printItem($string){
echo'Foo: '.$string. PHP_EOL;
}

public function printPHP(){
echo'PHP is great.'. PHP_EOL;
}
}

classBar extends Foo{
public function printItem($string)
{
echo'Bar: '.$string. PHP_EOL;
}
}

$foo= new Foo();
$bar= new Bar();
$foo->printItem('baz');// Output: 'Foo: baz'
$foo->printPHP(); // Output: 'PHP is great'
$bar->printItem('baz');// Output: 'Bar: baz'
$bar->printPHP(); // Output: 'PHP is great'
?>
```



## Scope Resolution Operator (::)

The Scope Resolution Operator (also called **PaamayimNekudotayim**) or in simpler terms, the double colon, is a token that allows access to static, constant, and overridden properties or methods of a class.

When referencing these items from outside the class definition, use the name of the class.

**PaamayimNekudotayim** would, at first, seem like a strange choice for naming a double-colon. However, while writing the Zend Engine 0.5 (which powers PHP 3), that's what the Zend team decided to call it. It actually does mean double-colon - in Hebrew!

```

<?php
class MyClass {
const CONST_VALUE='A constant value';
}

$class name='MyClass';
echo $class name::CONST_VALUE; // As of PHP 5.3.0

echo MyClass::CONST_VALUE;
?>
Output: A constant valueA constant value

```

## Static Keyword

Declaring class properties or methods as static makes them accessible without needing an instantiation of the class. A property declared as static cannot be accessed with an instantiated class object (though a static method can).

For compatibility with PHP 4, if no visibility declaration is used, then the property or method will be treated as if it was declared as public.

## Static methods

Because static methods are callable without an instance of the object created, the pseudo-variable `$this` is not available inside the method declared as static.

```

<?php
class Foo {
public static function aStaticMethod() {
// ...
}
}
Foo::aStaticMethod();
$class name='Foo';
$class name::aStaticMethod(); // As of PHP 5.3.0
?>

```

## Static properties

Static properties cannot be accessed through the object using the arrow operator `->`.

Like any other PHP static variable, static properties may only be initialized using a literal or constant before PHP 5.6; expressions are not allowed. In PHP 5.6 and later, the same rules apply as `const` expressions: some limited expressions are possible, provided they can be evaluated at compile time.

As of PHP 5.3.0, it's possible to reference the class using a variable. The variable's value cannot be a keyword (e.g. `self`, `parent` and `static`).

```

<?php
class Foo{
public static$my_static='foo';

public function staticValue() {
return self::$my_static;
}
}
class Bar extends Foo{
public function fooStatic() {
returnparent::$my_static;
}
}
print Foo::$my_static."\n";

$foo= new Foo();
print$foo->staticValue() ."\n";
print$foo->my_static."\n"; // Undefined "Property" my_static

print$foo::$my_static."\n";
$class name='Foo';
print$class name::$my_static."\n"; // As of PHP 5.3.0

print Bar::$my_static."\n";
$bar= new Bar();
print$bar->fooStatic() ."\n";
?>

```

## Predefined Variables

**Superglobals** — Built-in variables that are always available in all scopes

**\$GLOBALS** — References all variables available in global scope

**\$\_SERVER** — Server and execution environment information

**\$\_GET** — HTTP GET variables

**\$\_POST** — HTTP POST variables

**\$\_FILES** — HTTP File Upload variables

**\$\_REQUEST** — HTTP Request variables

**\$\_SESSION** — Session variables

**\$\_ENV** — Environment variables

**\$\_COOKIE** — HTTP Cookies

**\$php\_errormsg** — The previous error message

**\$http\_response\_header** — HTTP response headers

**\$argc** — The number of arguments passed to script

**\$argv** — Array of arguments passed to script



## Exceptions

PHP try and catch are the blocks with the feature of exception handling, which contain the code to handle exceptions. They play an important role in exception handling. There is one more important keyword used with the try-catch block is **throw**. The **throw** is a keyword that is used to throw an exception.

Each try block must have at least one catch block. On the other hand, a try block can also have multiple catch blocks to handle various classes of exception.

### try

The try block contains the code that may contain an exception. An exception raised in try block during runtime is caught by the catch block. Therefore, each try block must have at least one catch block. It consists of the block of code in which an exception can occur.

Following points needs to be noted about the try:

- The try block must be followed by a catch or finally block.
- A try block must have at least one catch block.
- There can be multiple catch blocks with one try block.

### catch

The catch block catches the exception raised in the try block. It contains the code to catch the exception, which is thrown by throw keyword in the try block. The catch block executes when a specific exception is thrown. PHP looks for the matching catch block and assigns the exception object to a variable.

Following points to be noted about the catch:

- There can be more than one catch block with a try.
- The thrown exception is caught and resolved by one or more catch.
- The catch block is always used with a try block. It cannot be used alone.
- It comes just after the try block.

### throw

It is a keyword, which is used to throw an exception. Note that one throw at least has one "catch block" to catch the exception.

It lists the exceptions thrown by function, which cannot be handled by the function itself.

### finally

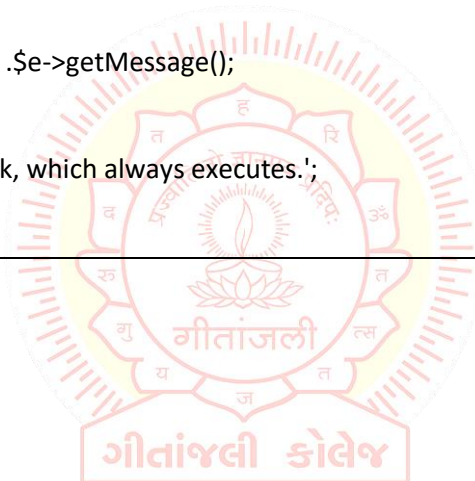
It is a block that contains the essential code of the program to execute. The finally block is also used for clean-up activity in PHP. It is similar to the catch block, which is used to handle exception. The only difference is that it always executes whether an exception is handled or not.

The finally block can be specified after or in place of catch block. It always executes just after the try and catch block whether an exception has been thrown or not, and before the normal execution restarts. It is useful in the following scenarios - Closing of database connection, stream.

```
<?php
//user-defined function with an exception
function checkNumber($num) {
    if($num>=1) {
        //throw an exception
        throw new Exception("Value must be less than 1");
    }
    return true;
}

//trigger an exception in a "try" block
try {
    checkNumber(5);
    //If the exception throws, below text will not be display
    echo 'If you see this text, the passed value is less than 1';
}

//catch exception
catch (Exception $e) {
    echo 'Exception Message: ' . $e->getMessage();
}
finally {
    echo '<br> It is finally block, which always executes.';
}
?>
```



## Class Abstraction

PHP 5 introduces abstract classes and methods. Classes defined as abstract may not be instantiated, and any class that contains at least one abstract method must also be abstract. Methods defined as abstract simply declare the method's signature - they cannot define the implementation.

When inheriting from an abstract class, all methods marked abstract in the parent's class declaration must be defined by the child; additionally, these methods must be defined with the same (or a less restricted) visibility. For example, if the abstract method is defined as protected, the function implementation must be defined as either protected or public, but not private. Furthermore the signatures of the methods must match, i.e. the type hints and the number of required arguments must be the same. For example, if the child class defines an optional argument, where the abstract method's signature does not, there is no conflict in the signature. This also applies to constructors as of PHP 5.4. Before 5.4 constructor signatures could differ.

```
<?php
abstract class AbstractClass{

// Force Extending class to define this method
abstract protected function getValue();
abstract protected function prefixValue($prefix);

// Common method
public function printOut() {
print $this->getValue() . "<br>"; }
}
class ConcreteClass1 extends AbstractClass {
protected function getValue() {
return "ConcreteClass 1"; }
public function prefixValue($prefix) {
return "{$prefix}ConcreteClass 1";
}
}
class ConcreteClass2 extends AbstractClass {
public function getValue() {
return "ConcreteClass 2";
}
public function prefixValue($prefix) {
return "{$prefix}ConcreteClass 2";
}
}
$class 1= new ConcreteClass 1;
$class 1->printOut();
echo $class 1->prefixValue('FOO_') . "<br>";

$class 2= new ConcreteClass 2;
$class 2->printOut();
echo $class 2->prefixValue('FOO_') . "<br>";
?>
```

The above example will output:

```
ConcreteClass 1
FOO_ConcreteClass 1
ConcreteClass 2
FOO_ConcreteClass 2
```

## Object Interfaces

Object interfaces allow you to create code which specifies which methods a class must implement, without having to define how these methods are implemented.

Interfaces are defined in the same way as a class, but with the interface keyword replacing the class keyword and without any of the methods having their contents defined. All methods declared in an interface must be public; this is the nature of an interface.

Note that it is possible to declare a constructor in an interface, what can be useful in some contexts, e.g. for use by factories.

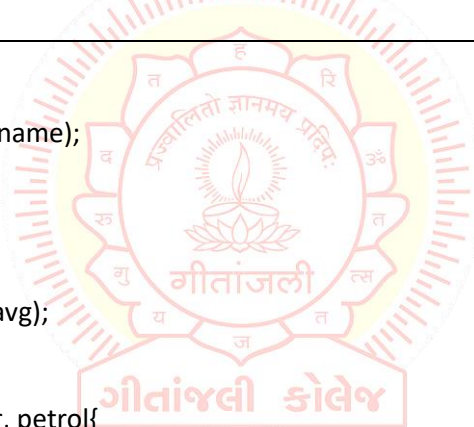
### implements

To implement an interface, the implements operator is used. All methods in the interface must be implemented within a class; failure to do so will result in a fatal error. Classes may implement more than one interface if desired by separating each interface with a comma.

### Constants

It's possible for interfaces to have constants. Interface constants work exactly like class constants except they cannot be overridden by a class /interface that inherits them.

```
<?php
interface car{
    function setModel($name);
    function getModel();
    const abc='CONST';
}
interface petrol{
    function setPetrol($avg);
    function getPetrol();
}
class minicar implements car, petrol{
    public $model;
    public $average;
    function setModel($name){
        $this->model=$name;
    }
    function getModel(){
        return $this->model;
    }
    function setPetrol($avg){
        $this->average=$avg;
    }
    function getPetrol(){
        return $this->average;
    }
}
$test=new minicar();
$test->setModel('Tata Nano');
$test->setPetrol('15');
```



```
echo car::abc;
echo $test->getModel();
echo $test->getPetrol();
?>
OUTPUT : CONSTTata Nano15
```

## Anonymous classes

Support for anonymous classes was added in PHP 7. Anonymous classes are useful when simple, one-off objects need to be created.

```
<?php

$Anonymous=new class{
    private $readOnly = 'Anonymous Class';

    public function printOnly(){
        return $this->readOnly;
    }
};
echo $Anonymous->printOnly();

?>
OUTPUT : Anonymous Class
```



## Overloading

Overloading in PHP provides means to dynamically "create" properties and methods. These dynamic entities are processed via magic methods one can establish in a class for various action types.

The overloading methods are invoked when interacting with properties or methods that have not been declared or are not visible in the current scope. The rest of this section will use the terms "inaccessible properties" and "inaccessible methods" to refer to this combination of declaration and visibility.

**Note : All overloading methods must be defined as public.**

### Property overloading

```
public void __set ( string $name, mixed $value )
```

```
public mixed __get ( string $name )
```

```
public bool __isset ( string $name )
```

```
public void __unset ( string $name )
```

**\_\_set()** is run when writing data to inaccessible properties.

**\_\_get()** is utilized for reading data from inaccessible properties.

**\_\_isset()** is triggered by calling **isset()** or **empty()** on inaccessible properties.

**\_\_unset()** is invoked when **unset()** is used on inaccessible properties.

The **\$name** argument is the name of the property being interacted with.

The **\_\_set()** method's **\$value** argument specifies value the **\$name**'ed property should set to.

Property overloading only works in object context. These magic methods will not be triggered in static context. Therefore these methods should not be declared static. As of PHP 5.3.0, a warning is issued if one of the magic overloading methods is declared static.

<pre>&lt;?php class PropertyTest { private \$data=array(); public function __set(\$name, \$value) { echo "Setting '\$name' to '\$value'\n"; \$this-&gt;data[\$name] =\$value; }  public function __get(\$name) { echo "Getting '\$name'\n"; return \$this-&gt;data[\$name]; }  public function __isset(\$name) { echo "Is '\$name' set?\n"; return isset(\$this-&gt;data[\$name]); }  public function __unset(\$name) { echo "U nsetting '\$name'\n";</pre>	<p>OutPut</p> <pre>Setting 'a' to '1' Getting 'a' 1 Is 'a' set? bool(true) Unsetting 'a' Is 'a' set? bool(false)</pre>
---	--

```

unset($this->data[$name]);
}
}

echo"<pre>\n";
$obj=new PropertyTest;
$obj->a=1;
echo$obj->a."\n\n";

var_dump(isset($obj->a));
unset($obj->a);
var_dump(isset($obj->a));

?>

```

### Method overloading

**public mixed \_\_call ( string \$name, array \$arguments )**

**public static mixed \_\_callStatic ( string \$name, array \$arguments )**

**\_\_call()** is triggered when invoking inaccessible methods in an object context.

**\_\_callStatic()** is triggered when invoking inaccessible methods in a static context.

The **\$name** argument is the name of the method being called. The **\$arguments** argument is an enumerated array containing the parameters passed to the **\$name**'ed method.

<pre> &lt;?php class Overloading{ function __call(\$name,\$args) {     echo "\$name does not exists.     echo"&lt;pre&gt;";     print_r (\$args);     echo"&lt;/pre&gt;"; } static function __callStatic(\$name,\$args) {     echo"\$name does not exists.";      echo"&lt;pre&gt;";     print_r (\$args);     echo"&lt;/pre&gt;"; } } \$test=new Overloading; \$test-&gt;anything('123',123); \$test::anything('123',123); ?&gt; </pre>	<pre> OutPut anything does not exists. Array (     [0] =&gt; 123     [1] =&gt; 123 ) anything does not exists. Array (     [0] =&gt; 123     [1] =&gt; 123 ) </pre>
--	---



## Object Iteration

PHP 5 provides a way for objects to be defined so it is possible to iterate through a list of items, with, for example a foreach statement. By default, all visible properties will be used for the iteration.

<pre>&lt;?php class iterat{     public \$abc="Hello";     public \$def="Hello";     public \$hij="Hello";     private \$klm="Hello";     protected \$nop="hello";      function iterate(){         foreach(\$this as \$key=&gt;\$value)             print "\$key =&gt;\$value&lt;br&gt;";     } } \$test=new iterat; echo"&lt;pre&gt;"; \$test-&gt;iterate(); /*foreach(\$test as \$key =&gt; \$value){     echo "\$key =&gt; \$value &lt;br&gt;"; }*/ //var_export(get_object_vars(\$test)); ?&gt;</pre>	<p>Output</p> <pre>abc =&gt; Hello def =&gt; Hello hij =&gt; Hello klm =&gt; Hello nop =&gt; hello</pre>
---	--

### Example Interfaceliterator

<pre>&lt;?php class Iterator implements Iterator {     private \$myArray;      public function __construct( \$givenArray ) {         \$this-&gt;myArray=\$givenArray;     }     function rewind() {         var_dump(__METHOD__);         return reset(\$this-&gt;myArray);     }     function current() {         var_dump(__METHOD__);         return current(\$this-&gt;myArray);     }     function key() {         var_dump(__METHOD__);         return key(\$this-&gt;myArray);     }      function next() {         var_dump(__METHOD__);         return next(\$this-&gt;myArray);     } }</pre>	<p>OUTPUT</p> <pre>string(17) "titerator::rewind" string(16) "titerator::valid" string(18) "titerator::current" string(14) "titerator::key" 0 =&gt; Hello string(15) "titerator::next" string(16) "titerator::valid" string(18) "titerator::current" string(14) "titerator::key" 1 =&gt; I string(15) "titerator::next" string(16) "titerator::valid" string(18) "titerator::current" string(14) "titerator::key" 2 =&gt; AM string(15) "titerator::next" string(16) "titerator::valid" string(18) "titerator::current" string(14) "titerator::key" 3 =&gt; STUDENT string(15) "titerator::next" string(16) "titerator::valid"</pre>
---	--

```

}
function valid() {
    var_dump(__METHOD__);
    return key($this->myArray) !==null;
}
}
}
$it= new titerator(['Hello','I','AM','STUDENT']);
echo"<pre>";
foreach($itas$key=>$value) {
echo"$key =>$value";
echo"<br>";
}
?>

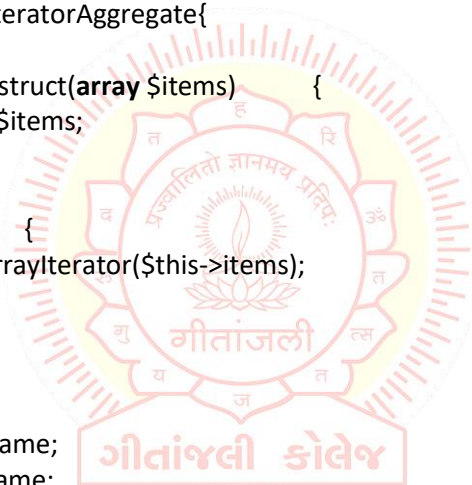
```

### Example: IteratorAggregate

```

<?php
class Collection implements IteratorAggregate{
    protected $items;
    public function __construct(array $items) {
        $this->items=$items;
    }
    function getIterator() {
        return new ArrayIterator($this->items);
    }
}
class User{
    public $FirstName;
    public $LastName;
    public $email;
}
$user1=new User;
$user2=new User;
$user1->email='abc@gmail.com';
$user1->FirstName="ABC";
$user1->LastName="XYZ";
$user2->email='def@gmail.com';
$user2->FirstName="DEF";
$user2->LastName="XYZ";
$users=newCollection([$user1,$user2]);
foreach($usersas$user)
{

```



```

    echo"$user->email";
    echo"$user->FirstName";
    echo"$user->LastName<br>";
}

?>
OUTPUT
abc@gmail.comABCXYZ
def@gmail.comDEFXYZ

```

## Magic Methods

The function names

**\_\_construct()**, **\_\_destruct()**, **\_\_call()**, **\_\_callStatic()**, **\_\_get()**, **\_\_set()**, **\_\_isset()**, **\_\_unset()**, **\_\_sleep()**, **\_\_wakeup()**, **\_\_toString()**, **\_\_invoke()**, **\_\_set\_state()**, **\_\_clone()** and **\_\_debugInfo()** are magical in PHP classes. You cannot have functions with these names in any of your classes unless you want the magic functionality associated with them.

### **\_\_toString()**

**public string \_\_toString ( void )**

The **\_\_toString()** method allows a class to decide how it will react when it is treated like a string. For example, what `echo $obj;` will print. This method must return a string, as otherwise a fatal `E_RECOVERABLE_ERROR` level error is emitted.

It is worth noting that before PHP 5.2.0 the **\_\_toString()** method was only called when it was directly combined with `echo` or `print`. Since PHP 5.2.0, it is called in any string context (e.g. in **printf()** with `%s` modifier) but not in other types contexts (e.g. with `%d` modifier). Since PHP 5.2.0, converting objects without **\_\_toString()** method to string would cause `E_RECOVERABLE_ERROR`.

### **\_\_invoke()**

**mixed \_\_invoke ([ \$... ] )**

The **\_\_invoke()** method is called when a script tries to call an object as a function .

### **\_\_set\_state()**

**static object \_\_set\_state ( array \$properties )**

This static method is called for classes exported by **var\_export()** since PHP 5.1.0. The only parameter of this method is an array containing exported properties in the form `array('property' => value, ...)`.

### **\_\_debugInfo()**

**array \_\_debugInfo ( void )**

This method is called by **var\_dump()** when dumping an object to get the properties that should be shown. If the method isn't defined on an object, then all public, protected and private properties will be shown. This feature was added in PHP 5.6.0.

```

<?php
class MagicMethods{

```

```

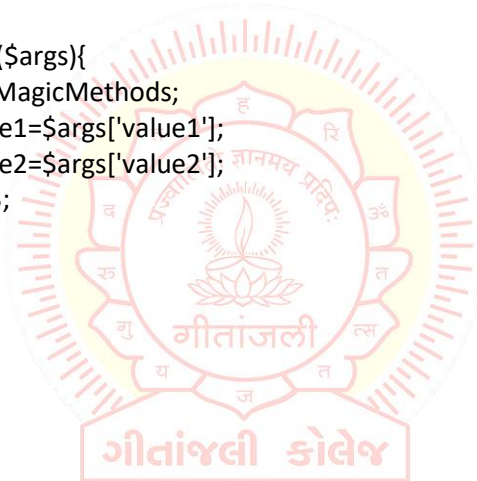
public $value1;
public $value2;

/* __construct() __destruct() __get() __set() __isset() __unset() __call() __callStatic() __invoke()
__clone() __toString() __sleep() __wakeup() __set_state() __debugInfo() */

function __invoke(){
    echo"<Br>This is completely out of Bounds.";
}
function __clone(){
    echo'<br>I M John\'s Copy';
}
function __toString(){
    return'<br>Here goes notihng';
}
function __debugInfo(){
    echo"<pre>";
    return ["Hello", 'sdsdsd', "I am new "];
    echo"</pre>";
}
function __set_state($args){
    $test3=new MagicMethods;
    $test3->value1=$args['value1'];
    $test3->value2=$args['value2'];
    return $test3;
}
}
$test=new MagicMethods;
$test->value1=15;
$test->value2='sdsdsd';
$str=var_export($test, true);
eval("$str.");
var_dump($str);
var_dump(new MagicMethods);
//echo serialize($test);
$test();
$test1=clone $test;
echo $test1;

?>
OUTPUT
string(79) "MagicMethods::__set_state(array( 'value1' => 15, 'value2' => 'sdsdsd', ))"
object(MagicMethods)#2 (3) {
    [0]=>
    string(5) "Hello"
    [1]=>
    string(6) "sdsdsd"
    [2]=>
    string(9) "I am new "
}

```



This is completely out of Bounds.  
I M John's Copy  
Here goes nothng

### **\_\_sleep() and \_\_wakeup()**

```
public array __sleep ( void )
void __wakeup ( void )
```

**serialize()** checks if your class has a function with the magic name `__sleep()`. If so, that function is executed prior to any serialization. It can clean up the object and is supposed to return an array with the names of all variables of that object that should be serialized. If the method doesn't return anything then NULL is serialized and `E_NOTICE` is issued.

The intended use of **\_\_sleep()** is to commit pending data or perform similar cleanup tasks. Also, the function is useful if you have very large objects which do not need to be saved completely.

Conversely, **unserialize()** checks for the presence of a function with the magic name `__wakeup()`. If present, this function can reconstruct any resources that the object may have.

The intended use of **\_\_wakeup()** is to reestablish any database connections that may have been lost during serialization and perform other reinitialization tasks.

```
<?php
class demoSleepWakeup {

public $arrayM=array(1, 2, 3, 4);

public function __sleep() {
return array('arrayM');
}

public function __wakeup() {
echo"array restarted ";
}
}

$obj=new demoSleepWakeup();
$serializedStr=serialize($obj);
echo"<pre>";
var_dump($serializedStr);
var_dump(unserialize($serializedStr));
echo"</pre>";
?>
```

```
OutPut
string(78)
"O:15:"demoSleepWakeup":1:{s:6:"arrayM";a:4:{i:0
;i:1;i:1;i:2;i:2;i:3;i:3;i:4;}}"
array restarted object(demoSleepWakeup)#2 (1) {
["arrayM"]=>
array(4) {
[0]=>
int(1)
[1]=>
int(2)
[2]=>
int(3)
[3]=>
int(4)
}
}
```

## Final Keyword

PHP 5 introduces the final keyword, which prevents child classes from overriding a method by prefixing the definition with final. If the class itself is being defined final then it cannot be extended.

### Example # Final methods example

```
<?php
class BaseClass {
    public function test() {
        echo "BaseClass::test() called\n";
    }
    final public function moreTesting() {
        echo "BaseClass::moreTesting() called\n";
    }
}
class ChildClass extends BaseClass {
    public function moreTesting() {
        echo "ChildClass::moreTesting() called\n";
    }
}
// Results in Fatal error: Cannot override final method BaseClass::moreTesting()
?>
```

### Example #2 Final class example

```
<?php
final class BaseClass {
    public function test() {
        echo "BaseClass::test() called\n";
    }
    // Here it doesn't matter if you specify the function as final or not
    final public function moreTesting() {
        echo "BaseClass::moreTesting() called\n";
    }
}
class ChildClass extends BaseClass {
}
// Results in Fatal error: Class ChildClass may not inherit from final class (BaseClass)
?>
```

## Object Cloning

Creating a copy of an object with fully replicated properties is not always the wanted behavior. A good example of the need for copy constructors, is if you have an object which represents a GTK window and the object holds the resource of this GTK window, when you create a duplicate you might want to create a new window with the same properties and have the new object hold the resource of the new window. Another example is if your object holds a reference to another object which it uses and when you replicate the parent object you want to create a new instance of this other object so that the replica has its own separate copy.

An object copy is created by using the clone keyword (which calls the object's **\_\_clone()** method if possible). An object's **\_\_clone()** method cannot be called directly.

### \$copy\_of\_object = clone \$Object;

When an object is cloned, PHP 5 will perform a shallow copy of all of the object's properties. Any properties that are references to other variables will remain references.

### void \_\_clone ( void )

Once the cloning is complete, if a **\_\_clone()** method is defined, then the newly created object's **\_\_clone()** method will be called, to allow any necessary properties that need to be changed.

```
<?php
class BOX {
    public $name="Hello";
}

$box=new BOX();
//$box->name= "Hello";

$box_ref=$box;
//DUPLICATE
//$box_ref->name="Sorry";

$box_clone=clone $box;
//$box_clone->name="NO";

$box_change=clone $box;
//$box_change->name="Hello";

$another_box=new BOX();
//$another_box->name="New";

//Comparison
echo $box==$box_ref?'true<br>':'false<br>';           //true
echo $box==$box_clone?'true<br>':'false<br>';         //true
echo $box==$box_change?'true<br>':'false<br>';         //true
echo $box==$another_box?'true<br>':'false<br>';         //true
//Identity Comparison
echo $box=== $box_ref?'true<br>':'false<br>';         //true
echo $box=== $box_clone?'true<br>':'false<br>';         //false
echo $box=== $box_change?'true<br>':'false<br>';         //false
echo $box=== $another_box?'true<br>':'false<br>';         //false
?>
```



## Comparing Objects

When using the comparison operator (**==**), object variables are compared in a simple manner, namely: Two object instances are equal if they have the same attributes and values (values are compared with **==**), and are instances of the same class.



When using the identity operator (**===**), object variables are identical if and only if they refer to the same instance of the same class.

## Type Hinting

### Type declarations

Note: Type declarations were also known as type hints in PHP 5.

Type declarations allow functions to require that parameters are of a certain type at call time. If the given value is of the incorrect type, then an error is generated: in PHP 5, this will be a recoverable fatal error, while PHP 7 will throw a `TypeError` exception.

To specify a type declaration, the type name should be added before the parameter name. The declaration can be made to accept NULL values if the default value of the parameter is set to NULL.

### Valid types

Type	Description	Min. PHP version
Class /interface name	The parameter must be an instance of the given class or interface name.	PHP 5.0.0
self	The parameter must be an instance of the same class as the one the method is defined on. This can only be used on class and instance methods.	PHP 5.0.0
array	The parameter must be an array.	PHP 5.1.0
callable	The parameter must be a valid callable.	PHP 5.4.0
bool	The parameter must be a boolean value.	PHP 7.0.0
float	The parameter must be a floating point number.	PHP 7.0.0
int	The parameter must be an integer.	PHP 7.0.0
string	The parameter must be a string.	PHP 7.0.0
Iterable	The parameter must be either an array or an instance of Traversable.	PHP 7.1.0

### Strict typing

By default, PHP will coerce values of the wrong type into the expected scalar type if possible. For example, a function that is given an integer for a parameter that expects a string will get a variable of type string.

It is possible to enable strict mode on a per-file basis. In strict mode, only a variable of exact type of the type declaration will be accepted, or a **TypeError** will be thrown. The only exception to this rule is that an integer may be given to a function expecting a float. Function calls from within internal functions will not be affected by the `strict_types` declaration.

### Type declarations

```
<?php
class Song{
    public $title;
    public $lyrics;
```

```

}
class Data{
    function sing(Song $song){
        $this->d_song=$song;
        echo"Best song of the year :".$this->d_song->title."<br>";
        echo"<p>".$this->d_song->lyrics." </p>";
    }
}
$data=new Data();
$hit=new Song();
$hit->title="VandeMatartam";
$hit->lyrics="Sujalamsuphalammalayajasheetalam";
$data->sing($hit);

//sing($hit);
?>

```

OUTPUT

Best song of the year :VandeMatartam  
Sujalamsuphalammalayajasheetalam

**Late Static Bindings**

Late static bindings tries to solve that limitation by introducing a keyword that references the class that was initially called at runtime. It was decided not to introduce a new keyword but rather use static that was already reserved.

**Example # static:: simple usage**

```

<?php
class Department{
    protected static $x=10;
    public function myFunction(){
        echo static::$x;
    }
}
class HR extends Department{
    protected static $x=20;
}
class Marketing extends Department{
    protected static $x=30;
}
$test=new Marketing;
$test->myFunction();
?>

```

The above example will output: 30

## Objects and References

One of the key-points of PHP 5 OOP that is often mentioned is that "objects are passed by references by default". This is not completely true. This section rectifies that general thought using some examples.

A PHP reference is an alias, which allows two different variables to write to the same value. As of PHP 5, an object variable doesn't contain the object itself as value anymore. It only contains an object identifier which allows object assessors to find the actual object. When an object is sent by argument, returned or assigned to another variable, the different variables are not aliases: they hold a copy of the identifier, which points to the same object.

<pre> &lt;?php class objRef{     public \$hi=5; } \$a=new objRef();  \$b=\$a; // Copy. //\$b=&amp;\$a; // Reference.  echo "a=".\$a-&gt;hi."&lt;br&gt;"; echo "b=".\$b-&gt;hi."&lt;br&gt;";  echo "a=".\$a-&gt;hi=3."&lt;br&gt;"; echo "b=".\$b-&gt;hi=10."&lt;br&gt;";  \$a=null;  echo "a=".\$a-&gt;hi."&lt;br&gt;"; echo "b=".\$b-&gt;hi."&lt;br&gt;";  ?&gt; </pre>	<p>Output:</p> <pre> a=5 b=5 a=3 b=10 </pre> <p><b>Notice:</b> Trying to get property of non-object</p> <pre> a= b=10 </pre>
---	--

## BOOTSTRAP

### What is Bootstrap?

Bootstrap is an Open Source product from **Mark Otto** and **Jacob Thornton** who, when initially released were both employees at **Twitter**. There was a need to standardize the front-end toolsets of engineers across the company. In the launch blog post, Mark Otto introduces the project like this:

*In the earlier days of Twitter, engineers used almost any library they were familiar with to meet front-end requirements. Inconsistencies among the individual applications made it difficult to scale and maintain them. Bootstrap began as an answer to these challenges and quickly accelerated during Twitter's first Hack week. By the end of Hackweek, we had reached a stable version that engineers could use across the company.*

— Mark Otto

<https://dev.twitter.com/blog/bootstrap-twitter>

Since Bootstrap launched in August, 2011 it has taken off in popularity. It has evolved away from being an entirely CSS driven project to include a host of JavaScript plugins, and icons that go hand in hand with forms and buttons. At its base, it allows for responsive web design, and features a robust 12 column, 940px wide grid. One of the highlights is the build tool on <http://getbootstrap.com> website where you can customize the build to suit your needs, choosing what CSS and JavaScript features that you want to include on your site. All of this, allows front-end web development to be catapulted forward, building on a stable foundation of forward-looking design, and development. Getting started with Bootstrap is as simple as dropping some CSS and JavaScript into the root of your site.

Starting a project new, Bootstrap comes with a handful of useful elements to get you started. Normally, when I start a project, I start with tools like Eric Meyer's CSS reset, and get going on my web project. With Bootstrap, you just need to include the `bootstrap.css` CSS file, and optionally the `bootstrap.js` JavaScript file into your website and you are ready to go.

### File Structure

```
bootstrap/
├── css/
│   ├── bootstrap.css
│   └── bootstrap.min.css
├── js/
│   ├── bootstrap.js
│   └── bootstrap.min.js
├── img/
│   ├── glyphicons-halflings.png
│   └── glyphicons-halflings-white.png
└── README.md
```

The Bootstrap download includes three folders: `css`, `js` and `img`. For simplicity, add these to the root of your project. Included are also minified versions of the CSS and JavaScript. Both the uncompressed and the minified versions do not need to be included. For the sake of brevity, I use the uncompressed during development, and then switch to the compressed version in production.

## Basic HTML Template

Normally, a web project looks something like this:

### Basic HTML Layout.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bootstrap 101 Template</title>
  </head>
  <body>
    <h1>Hello, world!</h1>
  </body>
</html>
```

With Bootstrap, we simply include the link to the CSS stylesheet, and the Javascript.

### Basic Bootstrap Template.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bootstrap 101 Template</title>
    <link href="css/bootstrap.min.css" rel="stylesheet">
  </head>
  <body>
    <h1>Hello, world!</h1>
    <script src="js/bootstrap.min.js"></script>
  </body>
</html>
```

## Global Styles

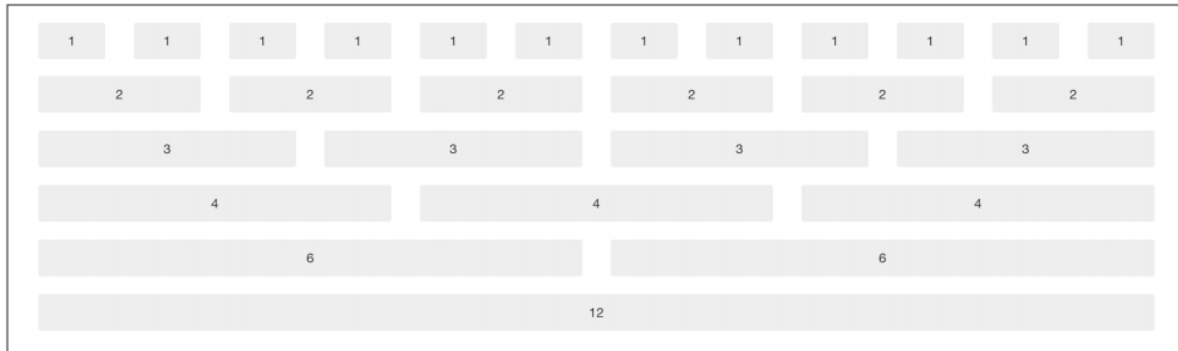
With Bootstrap, a number of items come prebuilt. Instead of using the old reset block that was part of the Bootstrap 1.0 tree, Bootstrap 2.0 uses Normalize.css, a project from Nicolas Gallagher that is part of the HTML5 Boilerplate. This is included in the Bootstrap.css file.

In particular, these default styles give special treatment to typography and links.

- margin has been removed from the body, and content will snug up to the edges of the browser window.
- background-color: white; is applied to the body
- Bootstrap is using the @baseFontFamily, @baseFontSize, and @baseLine Height attributes as our typographic base. This allows the height of headings, and other content around the site to maintain a similar line height.
- Bootstrap sets the global link color via @linkColor and applies link underlines only on :hover

## Default Grid System

The default Bootstrap grid system utilizes 12 columns, making for a 940px wide container without responsive features enabled. With the responsive CSS file added, the grid adapts to be 724px and 1170px wide depending on your viewport. Below 767px view-ports, for example, on tablets and smaller devices the columns become fluid and stack vertically. At the default width, each column is 60 pixels wide, offset 20 pixels to the left.



## Basic Grid HTML

To create a simple layout, create a container with a div that has a class of `.row`, and add the appropriate amount of `.span*` columns. Since we have 12-column grid, we just need to have the amount of `.span*` columns add up to twelve. We could use a 3-6-3 layout, 4-8, 3-5-4, 2-8-2, we could go on and on, but I think you get the gist.

### Basic Grid Layout.

```
<div class="row">
  <div class="span8">...</div>
  <div class="span4">...</div>
</div>
```

In the above example, we have `.span8`, and a `.span4` adding up to 12

## Offsetting Columns

You can move columns to the right using the `.offset*` class. Each class moves the span over that width. So an `.offset2` would move a `.span7` over two columns.

### Offsetting Columns.

```
<div class="row">
  <div class="span2">...</div>
  <div class="span7 offset2">...</div>
</div>
```



## Nesting Columns

To nest your content with the default grid, inside of a `.span*`, simply add a new `.row` with enough `.span*` that add up the number of spans of the parent container. So, let's say that you have a two columns layout, with a `span8`, and a `span4`, and you want to embed a two column layout inside of the layout, what spans would you use? For a four column layout?

Create a table that looks like this:

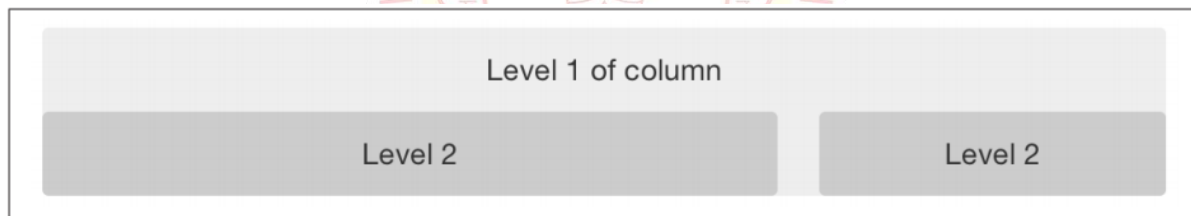
First	Last
Sanders	Kleinfeld
Karen	Tripp
Adam	Zaremba

Your markup should look something like this:

```
[options="header"]
|=====
|First | Last
|Sanders | Kleinfeld
|Karen | Tripp
|Adam | Zaremba
|=====
```

## Nesting Columns.

```
<div class="row">
  <div class="span9">
    Level 1 column
    <div class="row">
      <div class="span6">Level 2</div>
      <div class="span3">Level 2</div>
    </div>
  </div>
</div>
```



## Fluid Grid System

**Fluid Grid System** The fluid grid system uses percent instead of pixels for column widths. It has the same responsive capabilities as our fixed grid system, ensuring proper proportions for key screen resolutions and devices. You can make any row "fluid" by changing `.row` to `.rowfluid`. The column classes stay the exact same, making it easy to flip between fixed and fluid grids. To offset, you operate in the same way as the fixed grid system works by adding `.offset*` to any column to shift by your desired number of columns.

### Fluid Column Layout.

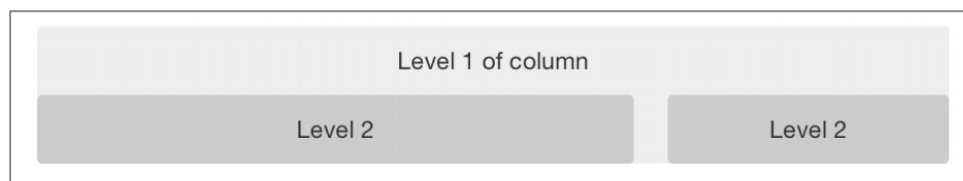
```
<div class="row-fluid">
  <div class="span4">...</div>
  <div class="span8">...</div>
</div>
<div class="row-fluid">
  <div class="span4">...</div>
  <div class="span4 offset2">...</div>
</div>
```



Nesting a fluid grid is a little different. Since we are using percentages, each `.row` resets the column count to 12. For example, If you were inside a `.span8`, instead of two `.span4` elements to divide the content in half, you would use two `.span6` divs. This is the case for responsive content, as we want the content to fill 100% of the container.

### Nesting Fluid Column Layout.

```
<div class="row-fluid">
  <div class="span8">
    <div class="row">
      <div class="span6">...</div>
      <div class="span6">...</div>
    </div>
  </div>
</div>
```



## Container Layouts

To add a fixed width, centered layout to your page, simply wrap the content in `<div class="container">...</div>`. If you would like to use a fluid layout, but want to wrap everything in a container, use the following: `<div class="container-fluid">...</div>`. Using a fluid layout is great when you are building applications, administration screens and other related projects.

## Responsive Design

Responsive Design to turn on the responsive features of Bootstrap, you need to add a meta tag to the of your webpage. If you haven't downloaded the compiled source, you will also need to add the responsive CSS file. An example of required files looks like this:

### Responsive Meta Tag and CSS.

```
<!DOCTYPE html>
<html>
<head>
  <title>My amazing Bootstrap site!</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link href="/css/bootstrap.css" rel="stylesheet">
  <link href="/css/bootstrap-responsive.css" rel="stylesheet">
</head>
```

## What Is Responsive Design?

Responsive design is a method for taking all of the existing content that is on the page, and optimizing it for the device that is viewing it. For example, the desktop not only gets the normal version of the website, but might get also get a widescreen layout, optimized for the larger displays that many people have attached to their computers. Tablets get an optimized layout, taking advantage of the portrait or landscape layouts of those devices. And then with phones, you can target the much narrower width of phones. To target these different widths, Bootstrap uses CSS media queries to measure the width of the browser viewport, and then using conditionals, change which parts of the stylesheets are loaded. Using the width of the browser viewport, Bootstrap can then optimize the content using a combination of ratios, widths, but mostly falls on minwidth and max-width properties.

At the core, Bootstrap supports five different layouts, each relying on CSS media queries. The largest layout has columns that are 70 pixels wide, contrasting the 60 pixels of the normal layout. The tablet layout brings the columns to 42 pixels wide, and when narrower than that, each column goes fluid, meaning the columns are stacked vertically and each column is the full width of the device.

Label	Layout width	Column width	Gutter width
Large display	1200px and up	70px	30px
Default	980px and up	60px	20px
Portrait Tablets	768px and above	42px	20px
Phones to Tablets	767px and below	Fluid columns, no fixed widths	
Phones	480px and below	Fluid columns, no fixed widths	

To add custom CSS based on the media query, you can either include all rules in one CSS file, via the media queries below, or use entirely different CSS files. CSS media queries in the CSS stylesheet.

```
/* Large desktop */
```

```
@media (min-width: 1200px) { ... }
```

```
/* Portrait tablet to landscape and desktop */
```

```
@media (min-width: 768px) and (max-width: 979px) { ... }
```

```
/* Landscape phone to portrait tablet */
```

```
@media (max-width: 767px) { ... }
```

```
/* Landscape phones and down */
```

```
@media (max-width: 480px) { ... }
```

For a larger site, you might want to separate them into separate files. In the HTML file, you can call them with the link tag in the head of your document. This is useful for keeping file sizes smaller, but does potentially increase the HTTP requests if being responsive.

### CSS media queries via the link tag in the HTML <head>.

```
<link rel="stylesheet" href="base.css" />
```

```
<link rel="stylesheet" media="(min-width:1200px)" href="large.css" />
```

```
<link rel="stylesheet" media="(min-width:768px) and (max-width: 979px)" href="tablet.css" />
```

```
<link rel="stylesheet" media="(max-width: 767px)" href="tablet.css" />
```

```
<link rel="stylesheet" media="(max-width: 480px)" href="phone.css" />
```

## Tables

One of my favorite parts of Bootstrap is the nice way that tables are handled. I do a lot of work looking at and building tables, and the clean layout is great feature that's included in Bootstrap right off the bat. Bootstrap supports the following elements:

Tag	Description
<table>	Wrapping element for displaying data in a tabular format
<thead>	Container element for table header rows (<tr>) to label table columns
<tbody>	Container element for table rows (<tr>) in the body of the table
<tr>	Container element for a set of table cells (<td> or <th>) that appears on a single row
<td>	Default table cell
<th>	Special table cell for column (or row, depending on scope and placement) labels. Must be used within a <thead>
<caption>	Description or summary of what the table holds, especially useful for screen readers

If you want a nice basic table style with just some light padding and horizontal dividers only, add the base class of **.table** to any table. The basic layout has a top border on all of the <td> elements.

Name	Phone Number	Rank
Kyle West	707-827-7001	Eagle
Davey Preston	707-827-7003	Eagle
Taylor Lemmon	707-827-7005	Eagle

### Optional Table Classes

With the base table markup, and adding the **.table** class, there are few additional classes that you can add to style the markup. There are three classes, **.table-striped**, **.table-bordered**, **.table-hover**, and **.table-condensed**.

### Striped Table

By adding the **.table-striped** class, you will get stripes on rows within the <tbody>. This is done via the CSS:nth-child selector which is not available on Internet Explorer7-8.

Name	Phone Number	Rank
Kyle West	707-827-7001	Eagle
Davey Preston	707-827-7003	Eagle
Taylor Lemmon	707-827-7005	Eagle

### Bordered Table

If you add the **.table-bordered** class, you will get borders surrounding every element, and rounded corners around the entire table.

Name	Phone Number	Rank
Kyle West	707-827-7001	Eagle
Davey Preston	707-827-7003	Eagle
Taylor Lemmon	707-827-7005	Eagle

### Hover Table

If you add the **.table-hover** class, when you hover over a row, a light grey background will be added to rows while the user hovers over them.

Name	Phone Number	Rank
Kyle West	707-827-7001	Eagle
Davey Preston	707-827-7003	Eagle
Taylor Lemmon	707-827-7005	Eagle

### Condensed Table

If you add the **.table-condensed** class, padding is cut in half on rows to condense the table. Useful if you want denser information.

Name	Phone Number	Rank
Kyle West	707-827-7001	Eagle
Davey Preston	707-827-7003	Eagle
Taylor Lemmon	707-827-7005	Eagle

### Table Row Classes

If you want to style the table rows, you could add the following classes to change the background color.

Class	Description	Background Color
.success	Indicates a successful or positive action.	Green
.error	Indicates a dangerous or potentially negative action.	Red
.warning	Indicates a warning that might need attention.	Yellow
.info	Used as an alternative to the default styles.	Blue

#	Product	Payment Taken	Status
1	TB - Monthly	01/04/2012	Approved
2	TB - Monthly	02/04/2012	Declined
3	TB - Monthly	03/04/2012	Pending
4	TB - Monthly	04/04/2012	Call in to confirm

## Forms

Another one of the highlights of using Bootstrap is the attention that is paid to forms. As a web developer, one of my least favorite things to do is style forms. Bootstrap makes it easy to do with the simple HTML markup and extended classes for different styles of forms.

The basic form structure comes styled in Bootstrap, without needing to add any extrahelper classes. If you use the placeholder, it is only supported in newer browsers. In older ones, no text will be displayed.

### Basic Form Structure.

```
<form>
  <fieldset>
    <legend>Legend</legend>
    <label for="name">Label name</label>
    <input type="text" id="name" placeholder="Type something...">
      <span class="help-block">Example block-level help text here.</span>
    <label class="checkbox" for="checkbox">
      <input type="checkbox" id="checkbox">Check me out
    </label>
    <button type="submit" class="btn">Submit</button>
  </fieldset>
</form>
```

### Optional Form Layouts

With a few helper classes, you can dynamically update the layout of your form. Bootstrap comes with a few preset styles you can use.

### Search Form

Add **.form-search** to the form tag, and then **.search-query** to the `<input>` for an input box with rounded corners, and an inline submit button.

### Basic Form Structure.

```
<form class="form-search">
  <input type="text" class="input-medium search-query">
  <button type="submit" class="btn">Search</button>
</form>
```

## Inline Form

To create a form where all of the elements are inline, and labels are along side, add the class **.form-inline** to the form tag. To have the label and the input on the same line, use the horizontal form below.

### Inline Form Code.

```
<form class="form-inline">
  <input type="text" class="input-sm" placeholder="Email">
  <input type="password" class="input-sm" placeholder="Password">
  <label class="checkbox">
    <input type="checkbox"> Remember me
  </label>
  <button type="submit" class="btn">Sign in</button>
</form>
```

## Horizontal Form

Bootstrap also comes with a pre-baked horizontal form; this one stands apart from the others not only in the amount of markup, but also in the presentation of the form. Traditionally you'd use a table to get a form layout like this, but Bootstrap manages to do it without. Even better, if you're using the responsive CSS, the horizontal form will automatically adapt to smaller layouts by stacking the controls vertically.

### To create a form that uses the horizontal layout, do the following:

- Add a class of form-horizontal to the parent <form> element
- Wrap labels and controls in a div with class control-group
- Add a class of control-label to the labels
- Wrap any associated controls in a div with class controls for proper alignment

### Horizontal Form Code.

```
<form class="form-horizontal">
  <div class="control-group">
    <label class="control-label" for="inputEmail">Email</label>
    <div class="controls">
      <input type="text" id="inputEmail" placeholder="Email">
```

```

        </div>
    </div>
    <div class="control-group">
        <label class="control-label" for="inputPassword">Password</label>
        <div class="controls">
            <input type="password" id="inputPassword" placeholder="Password">
        </div>
    </div>
    <div class="control-group">
    <div class="controls">
        <label class="checkbox">
            <input type="checkbox"> Remember me
        </label>
        <button type="submit" class="btn">Sign in</button>
    </div>
</div>
</form>

```

## Supported Form Controls

Bootstrap natively supports the most common form controls. Chief among them, input, text area, checkbox and radio, and select.

## Inputs

The most common form text field is the input—this is where users will enter most of the essential form data. Bootstrap offers support for all native HTML5 input types: text, password, date time, date time-local, date, month, time, week, number, email, url, search, tel and color.



## Input Code.

```
<input type="text" placeholder="Text input">
```



## Textarea

The textarea is used when you need multiple lines of input. You'll find you mainly modify the rows attribute, changing it to the number of rows that you need to support (fewer rows = smaller box, more rows = bigger box).



## Textarea Example.

```
<textarea rows="3"></textarea>
```



## Checkboxes and radios

Checkboxes and radio buttons are great for when you want users to be able to choose from a list of preset options. When building a form, use checkbox if you want the user to select any number of options from a list, and radio if you want to limit them to just one selection.

- Option one is this and that—be sure to include why it's great
- Option one is this and that—be sure to include why it's great
- Option two can be something else and selecting it will deselect option one

### Radio and Checkbox Code Example.

```
<label class="checkbox">
<input type="checkbox" value="">
  Option one is this and that—be sure to include why it's great
</label>
<label class="radio">
<input type="radio" name="optionsRadios" id="optionsRadios1" value="option1" checked>
  Option one is this and that—be sure to include why it's great
</label>
<label class="radio">
<input type="radio" name="optionsRadios" id="optionsRadios2" value="option2">
  Option two can be something else and selecting it will deselect option one
</label>
```

- Option one is this and that—be sure to include why it's great
- Option one is this and that—be sure to include why it's great
- Option two can be something else and selecting it will deselect option one

If you want multiple checkboxes to appear on the same line together, simply add the **.inline** class to a series of checkboxes or radios.

```
<label for="option1" class="checkbox inline">
<input id="option1" type="checkbox" id="inlineCheckbox1" value="option1"> 1
</label>
<label for="option2" class="checkbox inline">
<input id="option2" type="checkbox" id="inlineCheckbox2" value="option2"> 2
</label>
<label for="option3" class="checkbox inline">
<input id="option3" type="checkbox" id="inlineCheckbox3" value="option3"> 3
</label>
```

## Selects

A select is used when you want to allow the user to pick from multiple options, but by default it only allows one. It's best to use `<select>` for list options of which the user is familiar such as states or numbers. Use `multiple="multiple"` to allow the user to select more than one option. If you only want the user to choose one option, use `type="radio"`.

### Select Code Example.

```
<select>
  <option>1</option>
  <option>2</option>
  <option>3</option>
  <option>4</option>
  <option>5</option>
</select>
<select multiple="multiple">
  <option>1</option>
  <option>2</option>
  <option>3</option>
  <option>4</option>
  <option>5</option>
</select>
```

### Extending Form Controls

In addition to the basic form controls listed in the previous section, Bootstrap offers few other form components to complement the standard HTML form elements; for example, it lets you easily prepend and append content to inputs.

### Prepended and Appended Inputs

By adding prepended and appended content to an input field, you can add common elements to the text users input, like the dollar symbol, the @ for a Twitter username or anything else that might be common for your application interface. To use, wrap the input in a div with class `input-prepend` (to add the extra content before the user input) or `input-append` (to add it after). Then, within that same `<div>`, place your extra content inside a `<span>` with an add-on class, and place the `<span>` either before or after the `<input>` element.

**Prepend and Append Code Example.**

```
<div class="input-prepend">
<span class="add-on">@</span>
<input class="span2" id="prependedInput" type="text" placeholder="Username">
</div>
<div class="input-append">
<input class="span2" id="appendedInput" type="text">
<span class="add-on">.00</span>
</div>
```

If you combine both of them, you simply need to add both the `.input-prepend` and `.input-append` classes to the parent `<div>`.

**Append and Prepend Code Example.**

```
<div class="input-prepend input-append">
<span class="add-on">$</span>
<input class="span2" id="appendedPrependedInput" type="text">
<span class="add-on">.00</span>
</div>
```

Rather than using a `<span>`, you can instead use `<button>` with a class of `btn` to attach (surprise!) a button or two to the input.

**Attach Multiple Buttons Code Example.**

```
<div class="input-append">
<input class="span2" id="appendedInputButtons" type="text">
<button class="btn" type="button">Search</button>
<button class="btn" type="button">Options</button>
</div>
```

If you are appending a button to a search form, you will get the same nice rounded corners that you would expect.

```
<form class="form-search">
<div class="input-append">
<input type="text" class="span2 search-query">
<button type="submit" class="btn">Search</button>
</div>
<div class="input-prepend">
<button type="submit" class="btn">Search</button>
</div>
```

```
<input type="text" class="span2 search-query">
</div>
</form>
```

### Form Control Sizing

With the default grid system that is inherent in Bootstrap, you can use the .span\* system for sizing form controls. In addition to the span column-sizing method, you can also use a handful of classes that take a relative approach to sizing. If you want the input to act as a block level element, you can add .input-block-level and it will be the full width of the container element.



```
<input class="input-block-level" type="text" placeholder=".input-block-level">
```

### Relative Input Controls









```
<input class="input-mini" type="text" placeholder=".input-mini">
<input class="input-small" type="text" placeholder=".input-small">
<input class="input-medium" type="text" placeholder=".input-medium">
<input class="input-large" type="text" placeholder=".input-large">
<input class="input-xlarge" type="text" placeholder=".input-xlarge">
<input class="input-xxlarge" type="text" placeholder=".input-xxlarge">
```

### Grid Sizing

You can use any .span from .span1 to .span12 for form control sizing.



```

<input class="span1" type="text" placeholder=".span1">
<input class="span2" type="text" placeholder=".span2">
<input class="span3" type="text" placeholder=".span3">
<select class="span1">
...
</select>
<select class="span2">
...
</select>
<select class="span3">
...
</select>

```

If you want to use multiple inputs on a line, simply use the `.controls-row` modifier class to apply the proper spacing. It floats the inputs to collapse the white space, and set the correct margins, and like the `.row` class, it also clears the float.

```

<div class="controls">
  <input class="span5" type="text" placeholder=".span5">
</div>
<div class="controls controls-row">
  <input class="span4" type="text" placeholder=".span4">
  <input class="span1" type="text" placeholder=".span1">
</div>

```

### Uneditable Text

If you want to present a form control, but not have it editable, simply add the class `.uneditable-input`.

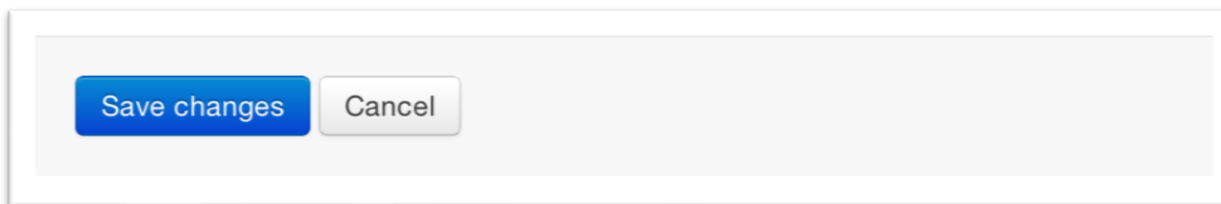
```

<span class="input-xlarge uneditable-input">Some value here</span>

```

### Form Actions

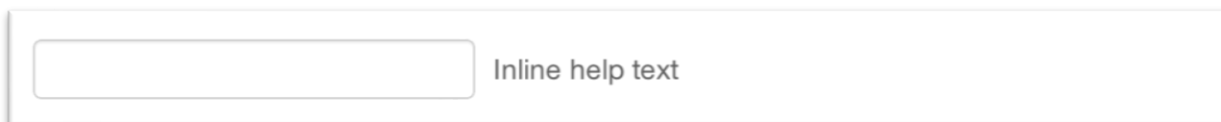
At the bottom of a horizontal-form you can place the form actions. Then inputs will correctly line up with the floated form controls.



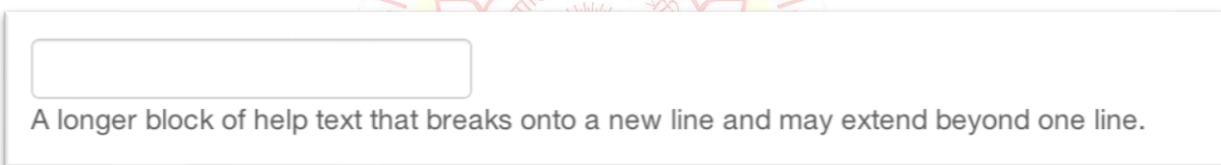
```
<div class="form-actions">
  <button type="submit" class="btn btn-primary">Save changes</button>
  <button type="button" class="btn">Cancel</button>
</div>
```

### Help Text

Bootstrap form controls can have either block or inline text that flows with the inputs.



```
<input type="text"><span class="help-inline">Inline help text</span>
```



```
<input type="text">
<span class="help-block">A longer block of help text that breaks onto a new line.</span>
```

### Form Control States

In addition to the: focus state, Bootstrap offers styling for disabled inputs, and classes for form validation.

### Input Focus

When an input receives: focus, that is to say, a user clicks into the input, or tabs into it, the outline of the input is removed, and a box-shadow is applied. I remember the first time that I saw this on Twitter's site, it blew me away, and I had to dig into the code to see how they did it. In WebKit, this accomplished in the following manner:

```
input {
  -webkit-box-shadow: inset 0 1px 1px rgba(0, 0, 0, 0.075);
  -webkit-transition: box-shadow linear 0.2s;
}
input:focus {
  -webkit-box-shadow: inset 0 1px 1px rgba(0, 0, 0, 0.075), 0 0 8px rgba(82, 168, 236, 0.6);
}
```

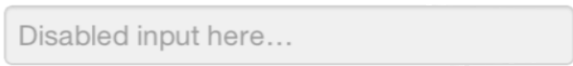
The `<input>` has a small inset box-shadow, this gives the appearance that the input sits lower than the page. When `:focus` is applied, an 8px light blue code is applied. The webkit-transition tells the browser to apply the effect in a linear manner over 0.2seconds. Nice and subtle, a great effect.



```
<input class="input-xlarge" id="focusedInput" type="text" value="This is focused...">
```

### Disabled Input

If you need to disable an input, simply add the `disabled` attribute to not only disable it, but change the styling, and the mouse cursor when it hover over the element.



```
<input class="input-xlarge" id="disabledInput" type="text" placeholder="Disabled input here..." disabled>
```

### Validation States

Bootstrap includes validation styles for error, warning, info, and success messages. Touse, simply add the appropriate class to the surrounding `.control-group`.



```
<div class="control-group warning">
<label class="control-label" for="inputWarning">Input with warning</label>
<div class="controls">
<input type="text" id="inputWarning">
<span class="help-inline">Something may have gone wrong</span>
</div>
</div>
<div class="control-group error">
<label class="control-label" for="inputError">Input with error</label>
<div class="controls">
<input type="text" id="inputError">
<span class="help-inline">Please correct the error</span>
```



```

</div>
</div>
<div class="control-group success">
<label class="control-label" for="inputSuccess">Input with success</label>
<div class="controls">
<input type="text" id="inputSuccess">
<span class="help-inline">Woohoo!</span>
</div>
</div>

```









## Buttons

One of my favorite features of Bootstrap is the way that buttons are styled. Dave Winer, inventor of RSS, and big fan of Bootstrap has this to say about it:

That this is needed, desperately needed, is indicated by the incredible uptake of Bootstrap. I use it in all the server software I'm working on. And it shows through in the templating-language I'm developing, so everyone who uses it will find it's "just there" and works, any-time you want to do a Bootstrap technique. Nothing to do, no libraries to include. It's as if it were part of the hardware. Same approach that Apple took with the Mac OS in 1984.

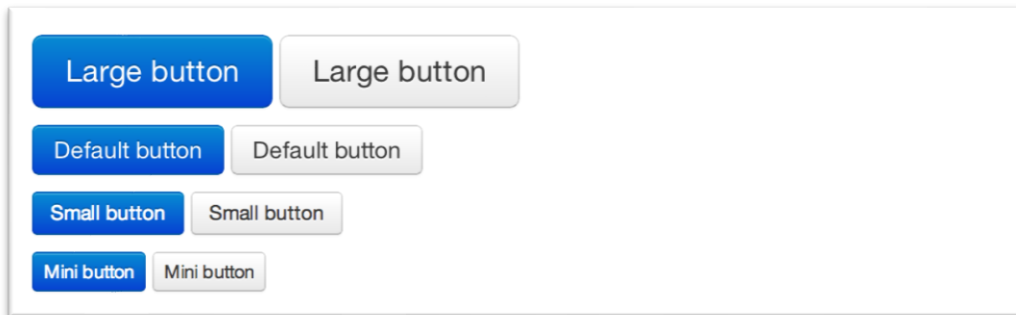
— Dave Winer  
scripting.com

I like to think that Bootstrap is doing that, unifying the web, and allowing a unified experience of what an interface can look like across the web. With the advent of Bootstrap, you can spot the sites that have taken it up usually first by the buttons that they use. A grid layout, and many of the other features fade into the background, but buttons, forms and other unifying elements are a key part of Bootstrap. Maybe I'm the only person that does this, but when I come across a site that is using Bootstrap, I want to give a high five to whomever answers the webmaster email at that domain, as they probably just get it. It reminds me of a few years ago I would do the same thing when I would see content in the HTML of sites that I would visit. Now, buttons, and links can all look alike with Bootstrap, anything that is given that class of btn will inherit the default look of a grey button with rounded corners. Adding extra classes will add colors to the buttons.

Buttons	Class	Description
	btn	Standard gray button with gradient
	btn btn-primary	Provides extra visual weight and identifies the primary action in a set of buttons
	btn btn-info	Used as an alternative to the default styles
	btn btn-success	Indicates a successful or positive action
	btn btn-warning	Standard gray button with gradient
	btn btn-danger	Indicates a dangerous or potentially negative action
	btn btn-inverse	Alternate dark gray button, not tied to a semantic action or use
	btn btn-link	Deemphasize a button by making it look like a link while maintaining button behavior

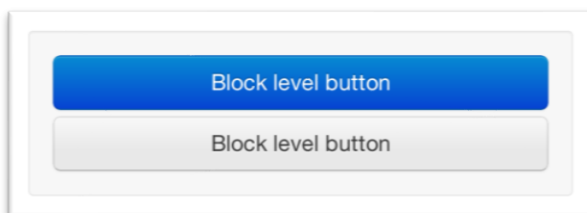
### Button Sizes

If you need larger or smaller buttons, simply add `.btn-large`, `.btn-small`, or `.btn-mini` to links or buttons.



```
<p>
<button class="btn btn-large btn-primary" type="button">Large button</button>
<button class="btn btn-large" type="button">Large button</button>
</p>
<p>
<button class="btn btn-primary" type="button">Default button</button>
<button class="btn" type="button">Default button</button>
</p>
<p>
<button class="btn btn-small btn-primary" type="button">Small button</button>
<button class="btn btn-small" type="button">Small button</button>
</p>
<p>
<button class="btn btn-mini btn-primary" type="button">Mini button</button>
<button class="btn btn-mini" type="button">Mini button</button>
</p>
```

If you want to create buttons that display like a block level element, simply add the `btnblockclass`. These buttons will display at 100% width.



```
<button class="btn btn-large btn-block btn-primary" type="button">Block level buton</button>
<button class="btn btn-large btn-block" type="button">Block level button</button>
```

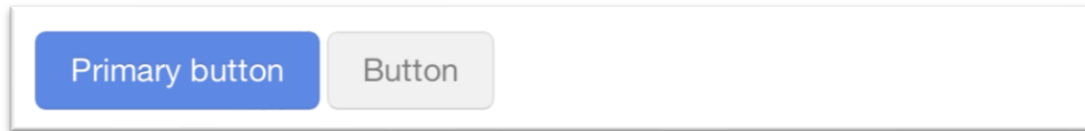
### Disabled Button Styling

For anchor elements, simply add the class of `.disabled` to the tag, and the link will drop back in color, and will lose the gradient.



```
<a href="#" class="btn btn-large btn-primary disabled">Primary link</a>
<a href="#" class="btn btn-large disabled">Link</a>
```

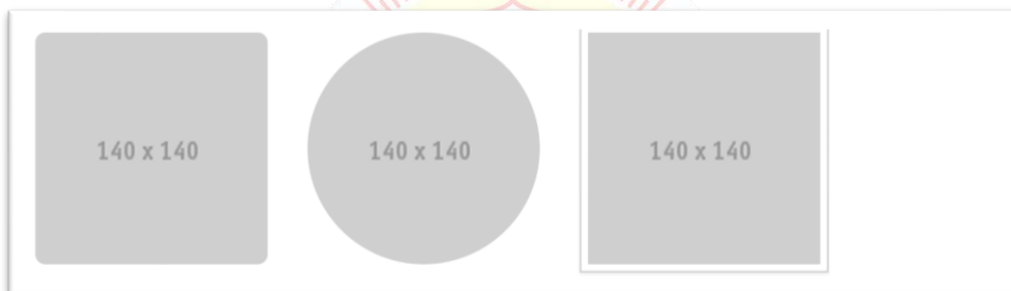
For a button, simply add the disabled attribute to the button. This will actually disable the button, so javascript is not directly needed.



```
<button type="button" class="btn btn-large btn-primary disabled" disabled="disabled">Primary button</button>
<button type="button" class="btn btn-large" disabled>Button</button>
```

## Images

Images have three classes to apply some simple styles. They are .img-rounded that adds border-radius:6px to give the image rounded corners, .img-circle that adds makes the entire image a circle by adding border-radius:500px making the image round, and lastly, img-polaroid, that adds a bit of padding and a grey border.



```



```

## Icons

Bootstrap bundles 140 icons into one sprite that can be used with buttons, links, navigation, and form fields. The icons are provided by Glyphicons.

 icon-glass	 icon-music	 icon-search	 icon-envelope
 icon-heart	 icon-star	 icon-star-empty	 icon-user
 icon-film	 icon-th-large	 icon-th	 icon-th-list
 icon-ok	 icon-remove	 icon-zoom-in	 icon-zoom-out
 icon-off	 icon-signal	 icon-cog	 icon-trash
 icon-home	 icon-file	 icon-time	 icon-road
 icon-download-alt	 icon-download	 icon-upload	 icon-inbox
 icon-play-circle	 icon-repeat	 icon-refresh	 icon-list-alt
 icon-lock	 icon-flag	 icon-headphones	 icon-volume-off
 icon-volume-down	 icon-volume-up	 icon-qr-code	 icon-barcode
 icon-tag	 icon-tags	 icon-book	 icon-bookmark
 icon-print	 icon-camera	 icon-font	 icon-bold
 icon-italic	 icon-text-height	 icon-text-width	 icon-align-left
 icon-align-center	 icon-align-right	 icon-align-justify	 icon-list
 icon-indent-left	 icon-indent-right	 icon-facetime-video	 icon-picture
 icon-pencil	 icon-map-marker	 icon-adjust	 icon-tint
 icon-edit	 icon-share	 icon-check	 icon-move
 icon-step-backward	 icon-fast-backward	 icon-backward	 icon-play
 icon-pause	 icon-stop	 icon-forward	 icon-fast-forward
 icon-step-forward	 icon-eject	 icon-chevron-left	 icon-chevron-right
 icon-plus-sign	 icon-minus-sign	 icon-remove-sign	 icon-ok-sign
 icon-question-sign	 icon-info-sign	 icon-screenshot	 icon-remove-circle
 icon-ok-circle	 icon-ban-circle	 icon-arrow-left	 icon-arrow-right
 icon-arrow-up	 icon-arrow-down	 icon-share-alt	 icon-resize-full
 icon-resize-small	 icon-plus	 icon-minus	 icon-asterisk
 icon-exclamation-sign	 icon-gift	 icon-leaf	 icon-fire
 icon-eye-open	 icon-eye-close	 icon-warning-sign	 icon-plane
 icon-calendar	 icon-random	 icon-comment	 icon-magnet
 icon-chevron-up	 icon-chevron-down	 icon-retweet	 icon-shopping-cart
 icon-folder-close	 icon-folder-open	 icon-resize-vertical	 icon-resize-horizontal
 icon-hdd	 icon-bullhorn	 icon-bell	 icon-certificate
 icon-thumbs-up	 icon-thumbs-down	 icon-hand-right	 icon-hand-left
 icon-hand-up	 icon-hand-down	 icon-circle-arrow-right	 icon-circle-arrow-left
 icon-circle-arrow-up	 icon-circle-arrow-down	 icon-globe	 icon-wrench
 icon-tasks	 icon-filter	 icon-briefcase	 icon-fullscreen

### Glyphicon Attribution

Users of Bootstrap are fortunate to use the Glyphicons free of use on Bootstrap projects. The developers have asked that you use a link back to Glyphicons when practical.

**Usage**

To use the icons, simply use an `<i>` tag with the name spaced `.icon-` class. For example, if you wanted to use the edit icon, you would simply add the `.icon-edit` class to the `<i>` tag.

```
<i class="icon-edit"></i>
```

If you want to use the white icon, simply add the `.icon-white` class to the tag.

```
<i class="icon-edit icon-white"></i>
```

**Button Groups**

Using button groups, combined with icons, you can create nice interface elements with minimal markup.



```
<div class="btn-toolbar">
  <div class="btn-group">
    <a class="btn" href="#"><i class="icon-align-left"></i></a>
    <a class="btn" href="#"><i class="icon-align-center"></i></a>
    <a class="btn" href="#"><i class="icon-align-right"></i></a>
    <a class="btn" href="#"><i class="icon-align-justify"></i></a>
  </div>
</div>
```

**Pagination**

Bootstrap handles pagination like a lot of interface elements, an unordered list, with wrapper `<div>` that has a specific class that identifies the element. In the basic form, adding `.pagination` to the parent `<div>` creates a row of bordered links. Each of the list items can be additionally styled by using the `.disabled` or `.active` class.

**Basic Pagination Code Example.**

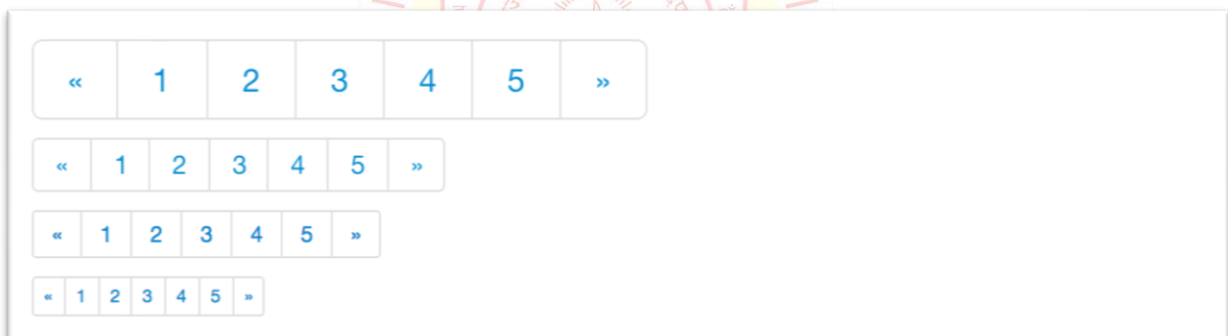
```
<div class="pagination">
<ul>
<li><a href="#">Prev</a></li>
<li><a href="#">1</a></li>
<li><a href="#">2</a></li>
<li><a href="#">3</a></li>
<li><a href="#">4</a></li>
<li><a href="#">Next</a></li>
</ul>
</div>
```



### Pagination with helper classes code examples.

```
<div class="pagination pagination-centered">
<ul>
<li class="disabled"><a href="#"><</a></li>
<li class="active"><a href="#">1</a></li>
<li><a href="#">2</a></li>
<li><a href="#">3</a></li>
<li><a href="#">4</a></li>
<li><a href="#">5</a></li>
<li><a href="#">></a></li>
</ul>
</div>
```

In addition to the `.active` and `.disabled` classes for list items, you can also add `.pagination-centered` to the parent `<div>`. This will center the contents of the `div`. If you want the items right aligned in the `<div>` add `.pagination-right`. For sizing, in addition to the normal size, there are three other sizes, applied by adding a class to the wrapper `<div>`. They are: `.pagination-large`, `pagination-small` and `paginationmini`.



### Pagination Code Example

```
<div class="pagination pagination-large">
<ul>
...
</ul>
</div>
<div class="pagination">
<ul>
...
</ul>
</div>
<div class="pagination pagination-small">
<ul>
...
</ul>
```



```

</div>
<div class="pagination pagination-mini">
<ul>
...
</ul>
</div>

```

### Pager

If you need to create simple pagination links that go beyond text, the pager can work quite well. Like the pagination links, the markup is an unordered list that sheds the wrapper <div>. By default, the links are centered.



### Basic Pager Code Example.

```

<ul class="pager">
<li><a href="#">Previous</a></li>
<li><a href="#">Next</a></li>
</ul>

```

To left/right align the different links, you just need to add the .previous and .next class to the list-items. Also, like .pagination above, you can add the disabled class for a muted look.



### Aligned Page Links Code Example.

```

<ul class="pager">
<li class="previous">
<li>
<li class="next">
<a href="#">Newer &rarr;</a>
</li>
</ul>

```

### Labels

Labels and Badges are great for offering counts, tips, or other markup for pages. Another one of my favorite little Bootstrap touches.



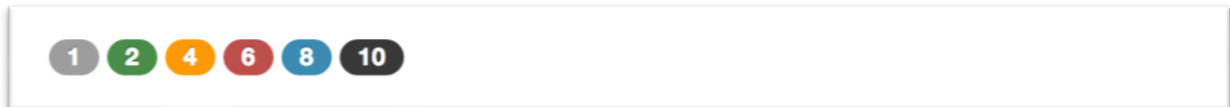


**Label Markup.**

```
<span class="label">Default</span>
<span class="label label-success">Success</span>
<span class="label label-warning">Warning</span>
<span class="label label-important">Important</span>
<span class="label label-info">Info</span>
<span class="label label-inverse">Inverse</span>
```

**Badges**

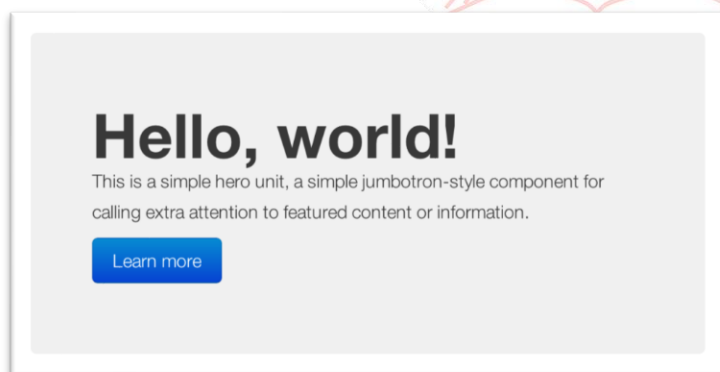
Badges are similar to labels, the primary difference is that they have more rounded corners. The colors of badges reflect the same classes as labels.

**Badges Code Example.**

```
<span class="badge">1</span>
<span class="badge badge-success">2</span>
<span class="badge badge-warning">4</span>
<span class="badge badge-important">6</span>
<span class="badge badge-info">8</span>
<span class="badge badge-inverse">10</span>
```

**Typographic Elements**

In addition to buttons, labels, forms, tables and tabs, Bootstrap has a few more elements for basic page layout. The hero unit is a large, content area that increased the size of headings, and adds a lot of margin for landing page content. To use, simply create a container `<div>` with the class of `.hero-unit`. In addition to a larger `<h1>`, all the fontweight is reduced to 200.

**Hero Unit Code Example.**

```
<div class="hero-unit">
<h1>Heading</h1>
<p>Tagline</p>
<p>
<a class="btn btn-primary btn-large">Learn more</a>
</p>
</div>
```

### Page Header

The page header is nice little feature to add appropriate spacing around the headings on a page. This is particularly helpful on a blog archive page where you may have several post titles, and need a way to add distinction to each of them. To use, wrap your heading in a `<div>` with a class of `.page-header`.

**Example page header** Subtext for header

### Page Header Code Example.

```
<div class="page-header">
<h1>Example page header <small>Subtext for header</small></h1>
</div>
```

### Thumbnails

A lot of sites need a way to layout images in a grid, and Bootstrap has an easy way to do this. At the simplest, you add an `<a>` tag with the class of `.thumbnail` around an image. This adds four pixels of padding, and a grey border. On hover, an animated glow is added around the image.

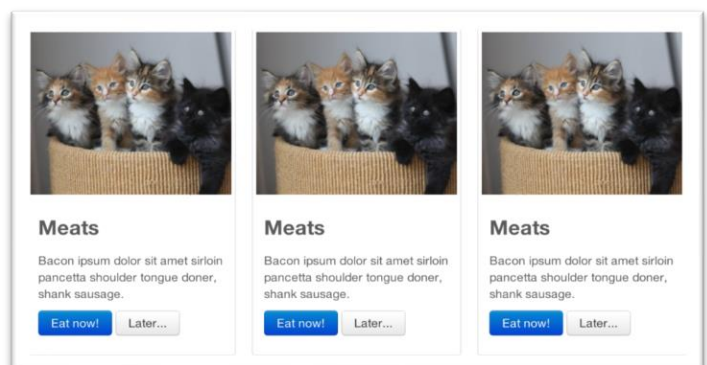


### Thumbnail Code Example

```
<a href="#" class="thumbnail">

</a>
```

To add more content to the markup, as an example, you could add headings, buttons and more, swap the `<a>` tag that has a class of `.thumbnail` to be a `<div>`. Inside of that `<div>`, you can add anything you need. Since this is a `<div>` we can use the default span based naming convention for sizing. If you want to group multiple images, place them in an unordered list, and each list item will be floated to left.



**Customizable Code Example.**

```

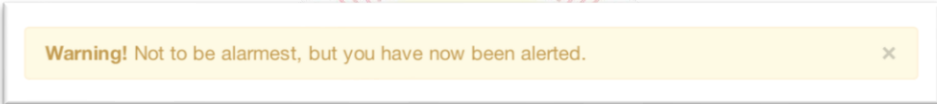
<ul class="thumbnails">
<li class="span4">
<div class="thumbnail">

<div class="caption">
<h3>Meats</h3>
<p>Bacon ipsum dolor sit amet sirloin pancetta shoulder tongue doner, shank sausage.</p>
<p><a href="#" class="btn btn-primary">Eat now!</a><a href="#" class="btn">Later...</a></p></div>
</li>
<li class="span4">
...
</li>
</ul>

```

**Alerts**

Alerts provide a way to style messages to the user. The default alert is created by creating a wrapper <div> and adding a class of .alert.



Warning! Not to be alarmest, but you have now been alerted.

**Basic Alert Code Example.**

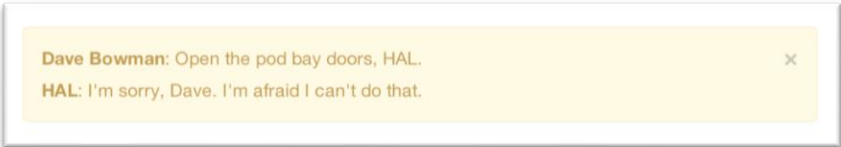
```

<div class="alert">
<a href="#" class="close" data-dismiss="alert">&times;</a>
<strong>Warning!</strong> Not to be alarmist, but you have now been alerted.
</div>

```

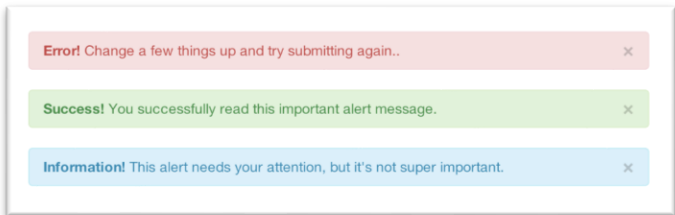
The alert uses the alerts jquery plugin that is covered in chapter 4. To close the alert, you can use a button that contains the data-dismiss="alert" attribute. Mobile Safari, and Mobile Opera browsers require an href="#" to close.

If you have a longer message in your alert, you can use the .alert-block class. This provides a little more padding above and below the content contained in the alert, particularly useful for multi-page lines of content.



Dave Bowman: Open the pod bay doors, HAL.  
HAL: I'm sorry, Dave. I'm afraid I can't do that.

There are also three other color options, to help provide a more semantic method for the alert. They are added by adding either .alert-error, .alert-success, or alertinfo.



Error! Change a few things up and try submitting again..

Success! You successfully read this important alert message.

Information! This alert needs your attention, but it's not super important.

## What is Laravel?

### The need for frameworks

Of all the server-side programming languages, PHP undoubtedly has the lowest entry barriers. It is almost always installed by default on even the cheapest web hosts, and it is also extremely easy to set up on any personal computer. For newcomers who have some experience with authoring web pages in HTML and CSS, the concepts of variables, inline conditions, and include statements are easy to grasp. PHP also provides many commonly used functions that one might need when developing a dynamic website. All of this contributes to what some refer to as the immediacy of PHP. However, this instant gratification comes at a cost. It gives a false sense of productivity to beginners, who almost inevitably end up with convoluted spaghetti code as they add more features and functionality to their site. This is mainly because PHP, out of the box, does not do much to encourage the separation of concerns.

### The limitations of homemade tools

If you already have a few PHP projects under your belt, but have not used a web application framework before, then you will probably have amassed a personal collection of commonly used functions and classes that you can use on new projects. These homegrown utilities might help you with common tasks, such as sanitizing data, authenticating users, and including pages dynamically. You might also have a predefined directory structure where these classes and the rest of your application code reside. However, all of this will exist in complete isolation; you will be solely responsible for the maintenance, inclusion of new features, and documentation. For a lone developer or an agency with ever-changing staff, this can be a tedious and time-consuming task, not to mention that if you were to collaborate with other developers on the project, they would first have to get acquainted with the way in which you build applications.

### Laravel to the rescue

This is exactly where a web application framework such as Laravel comes to the rescue. Laravel reuses and assembles existing components to provide you with a cohesive layer upon which you can build your web applications in a more structured and pragmatic way. Drawing inspiration from popular frameworks written not just in PHP but other programming languages too, Laravel offers a robust set of tools and an application architecture that incorporates many of the best features of frameworks like CodeIgniter, Yii, ASP.NET MVC, Ruby on Rails, Sinatra, and others. Most of these frameworks use the **Model-View-Controller (MVC)** paradigm or design pattern. If you have used one of the aforementioned tools or the MVC pattern, then you will find it quite easy to get started with Laravel 5.

## Features

So, what do you get out of the box with Laravel 5? Let's take a look and see how the following features can help boost your productivity:

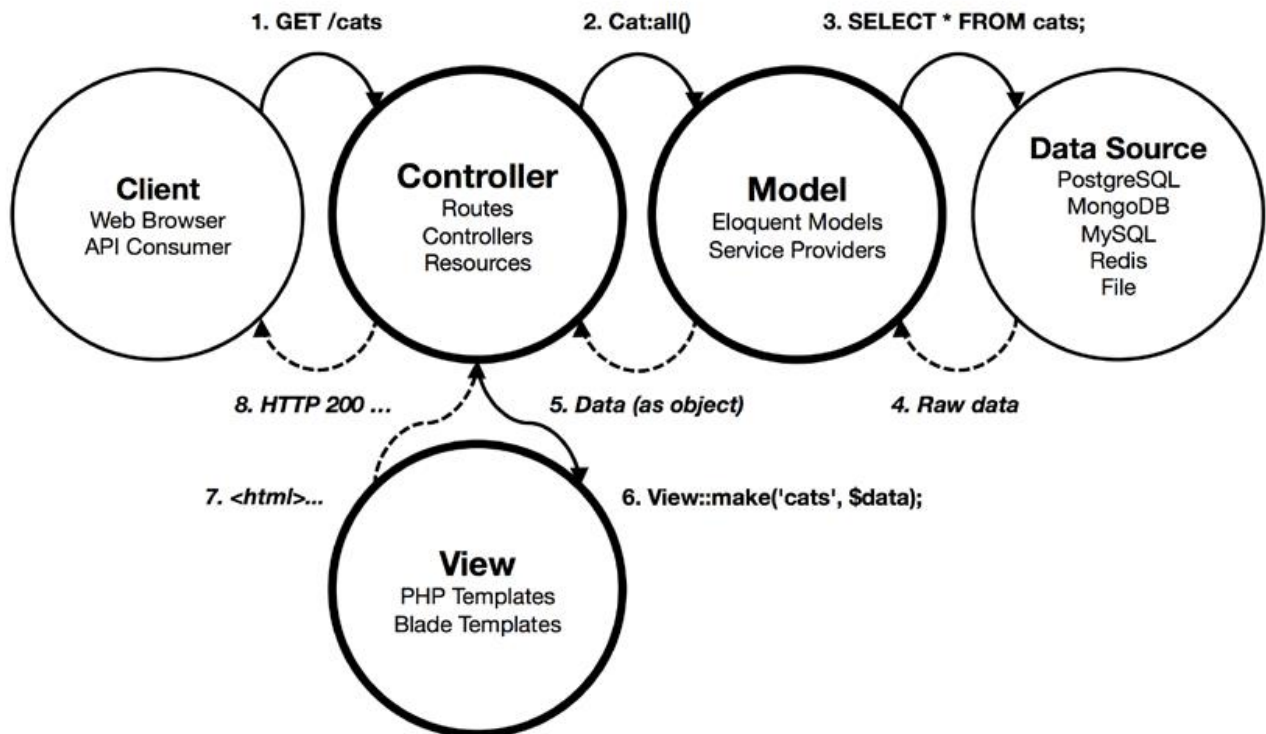
- **Modularity:** Laravel was built on top of over 20 different libraries and is itself split into individual modules. Tightly integrated with **Composer** dependency manager, these components can be updated with ease.
- **Testability:** Built from the ground up to ease testing, Laravel ships with several helpers that let you visit routes from your tests, crawl the resulting HTML, ensure that methods are called on certain classes, and even impersonate authenticated users in order to make sure the right code is run at the right time.
- **Routing:** Laravel gives you a lot of flexibility when you define the routes of your application. For example, you could manually bind a simple anonymous function to a route with an HTTP and HTTPS verb, such as GET, POST, PUT, or DELETE. This feature is inspired by micro-frameworks, such as **Sinatra**(Ruby) and **Silex** (PHP).
- **Configuration management:** More often than not, your application will be running in different environments, which means that the database ore-mail server credential's settings or the displaying of error messages will be different when your app is running on a local development server to when it is running on a production server. Laravel has a consistent approach to handle configuration settings, and different settings can be applied indifferent environments via the use of an .env file, containing settings unique for that environment.
- **Query builder and ORM:** Laravel ships with a fluent query builder, which lets you issue database queries with a PHP syntax, where you simply chain methods instead of writing SQL. In addition to this, it provides you with an **Object Relational Mapper (ORM)** and **Active Record** implementation, called **Eloquent**, which is similar to what you will find in Ruby on Rails, to help you define interconnected models. Both the query builder and the ORM are compatible with different databases, such as PostgreSQL, SQLite, MySQL, and SQL Server.
- **Schema builder, migrations, and seeding:** Also inspired by Rails, these features allow you to define your database schema in PHP code and keep track of any changes with the help of database migrations. A migration is a simple way of describing a schema change and how to revert to it. Seeding allows you to populate the selected tables of your database, for example, after running a migration.
- **Template engine:** Partly inspired by the **Razor** template language in ASP.NET MVC, Laravel ships with **Blade**, a lightweight template language with which you can create hierarchical layouts with predefined blocks in which dynamic content is injected.
- **E-mailing:** With its Mail class, which wraps the popular **SwiftMailer** library, Laravel makes it very easy to send an e-mail, even with rich content and attachments from your application. Laravel also comes with drivers for popular e-mail sending services such as **SendGrid**, **Mailgun**, and **Mandrill**.

- **Authentication:** Since user authentication is such a common feature in web applications, out of the box Laravel comes with a default implementation to register, authenticate, and even send password reminders to users.
- **Redis:** This is an in-memory key-value store that has a reputation for being extremely fast. If you give Laravel a Redis instance that it can connect to, it can use it as a session and general-purpose cache, and also give you the possibility to interact with it directly.
- **Queues:** Laravel integrates with several queue services, such as Amazon SQS, Beanstalkd, and IronMQ, to allow you to delay resource-intensive tasks, such as the e-mailing of a large number of users, and run them in the background, rather than keep the user waiting for the task to complete.
- **Event and command bus:** Although not new in version 5, Laravel has brought a command bus to the forefront in which it's easy to dispatch events(a class that represents something that's happened in your application),handle commands (another class that represents something that should happen in your application), and act upon these at different points in your application's lifecycle.





## MVC architecture



- **Models:** Models represent *resources* in your application. More often than not, they correspond to records in a data store, most commonly a database table. In this respect, you can think of models as representing *entities* in your application, be that a user, a news article, or an event, among others. In Laravel, models are classes that usually extend Eloquent's base Model class and are named in **CamelCase** (that is, News Article). This will correspond to a database table with the same name, but in **snake\_case** and plural (that is, news\_articles). By default, Eloquent also expects a primary key name `id`, and will also look for—and automatically update—the `created_at` and `updated_at` columns. Models can also describe the relationships they have with other models. For example, a News Article model might be associated with a User model, as a User model might be able to author a News Article model.

- **Controllers or routes:** Controllers, at their simplest, take a request, do something, and then send an appropriate response. Controllers are where the actual processing of data goes, whether that is retrieving data from a database, or handling a form submission, and saving data back to a database. Although you are not forced to adhere to any rules when it comes to creating controller classes in Laravel, it does offer you two sane approaches: RESTful controllers and resource controllers. A RESTful controller allows you to define your own actions and what HTTP methods they should respond to. Resource controllers are based around an entity and allow you to perform common operations on that entity, based on the HTTP method used.

- **Views or Templates:** Views are responsible for displaying the response returned from a controller in a suitable format, usually as an HTML webpage. They can be conveniently built by using the Blade template language or by simply using standard PHP. The file extension of the view, either **.blade.php** or simply **.php**, determines whether or not Laravel treats your view as a Blade template or not.



## Basic requirements for Laravel

Version	PHP (*)	Release	Bug Fixes Until	Security Fixes Until
6 (LTS)	7.2 - 8.0	September 3rd, 2019	January 25th, 2022	September 6th, 2022
7	7.2 - 8.0	March 3rd, 2020	October 6th, 2020	March 3rd, 2021
8	7.3 - 8.1	September 8th, 2020	July 26th, 2022	January 24th, 2023
9	8.0 - 8.1	February 8th, 2022	August 8th, 2023	February 8th, 2024
10	8.1	February 7th, 2023	August 7th, 2024	February 7th, 2025

## Using Laravel Installer

Laravel utilizes Composer to manage its dependencies. So, before using Laravel, make sure you have Composer installed on your machine.

**[composer global require "laravel/installer"]**

Make sure to place the `~/composer/vendor/bin` directory (or the equivalent directory for your OS) in your PATH so the laravel executable can be located by your system.

Once installed, the laravel new command will create a fresh Laravel installation in the directory you specify. For instance, `laravel new blog` will create a directory named `blog` containing a fresh Laravel installation with all of Laravel's dependencies already installed. This method of installation is much faster than installing via Composer:

**[laravel new blog]**

## Using Composer

Strongly inspired by popular dependency managers in other languages, such as Ruby's Bundler or Node.js's **Node Package Manager (npm)**, Composer brings these features to PHP and has quickly become the de-facto dependency manager in PHP.

### How does Composer work?

A few years ago, you may have used **PHP Extension and Application Repository(PEAR)** to download libraries. PEAR differs from Composer, in that PEAR would install packages on a system-level basis, whereas a dependency manager, such as Composer, installs them on a project-level basis. With PEAR, you could only have one version of a package installed on a system. Composer allows you to use different versions of the same package in different applications, even if they reside on the same system.

## Installation

### Linux

#### Locally

To install Composer locally, run the installer in your project directory. The installer will check a few PHP settings and then download composer.phar to your working directory. This file is the Composer binary. It is a PHAR (PHP archive), which is an archive format for PHP which can be run on the command line, amongst other things.

Now run php composer.phar in order to run Composer.

You can install Composer to a specific directory by using the --install-dir option and additionally (re)name it as well using the --filename option. When running the installer when following [the Download page instructions](#) add the following parameters:

```
php composer-setup.php --install-dir=bin --filename=composer
```

Now run php bin/composer in order to run Composer.

#### Globally

You can place the Composer PHAR anywhere you wish. If you put it in a directory that is part of your PATH, you can access it globally. On UNIX systems you can even make it executable and invoke it without directly using the php interpreter.

After running the installer following [the Download page instructions](#) you can run this to move composer.phar to a directory that is in your path:

```
mv composer.phar /usr/local/bin/composer
```

If you like to install it only for your user and avoid requiring root permissions, you can use ~/.local/bin instead which is available by default on some Linux distributions.

### Windows

#### Using the Installer

This is the easiest way to get Composer set up on your machine. Download and run [Composer-Setup.exe](#). It will install the latest Composer version and set up your PATH so that you can call composer from any directory in your command line.

#### Finding and installing new packages

Via Composer Create-Project

Alternatively, you may also install Laravel by issuing the Composer create-project command in your terminal:

```
composer create-project --prefer-dist laravel/laravel blog "9.*"
```

## Introduction

All of the configuration files for the Laravel framework are stored in the config directory. Each option is documented, so feel free to look through the files and get familiar with the options available to you.

### Environment configuration

It is often helpful to have different configuration values based on the environment the application is running in. For example, you may wish to use a different cache driver locally than you do on your production server. It's easy using environment-based configuration.

To make this a cinch, Laravel utilizes the [DotEnv](#) PHP library by Vance Lucas. In a fresh Laravel installation, the root directory of your application will contain a `.env.example` file. If you install Laravel via Composer, this file will automatically be renamed to `.env`. Otherwise, you should rename the file manually.

All of the variables listed in this file will be loaded into the `$_ENV` PHP super-global when your application receives a request. However, you may use the `env` helper to retrieve values from these variables in your configuration files. In fact, if you review the Laravel configuration files, you will notice several of the options already using this helper:

```
'debug' => env('APP_DEBUG', false),
```

The second value passed to the `env` function is the "default value". This value will be used if no environment variable exists for the given key.

Your `.env` file should not be committed to your application's source control, since each developer / server using your application could require a different environment configuration.

If you are developing with a team, you may wish to continue including a `.env.example` file with your application. By putting place-holder values in the example configuration file, other developers on your team can clearly see which environment variables are needed to run your application.

### Protecting Sensitive Configuration

For "real" applications, it is advisable to keep all of your sensitive configuration out of your configuration files. Things such as database passwords, Stripe API keys, and encryption keys should be kept out of your configuration files whenever possible. So, where should we place them? Thankfully, Laravel provides a very simple solution to protecting these types of configuration items using "dot" files.

First, configure your application to recognize your machine as being in the local environment. Next, create a `.env.local.php` file within the root of your project,

which is usually the same directory that contains your **composer.json** file. The **.env.local.php** should return an array of key-value pairs, much like a typical Laravel configuration file:

```
<?php
return array(
    'TEST_STRIPE_KEY' => 'super-secret-sauce',
);
```

All of the key-value pairs returned by this file will automatically be available via the `$_ENV` and `$_SERVER` PHP "superglobals". You may now reference these globals from within your configuration files:

```
'key' => $_ENV['TEST_STRIPE_KEY']
```

Be sure to add the `.env.local.php` file to your `.gitignore` file. This will allow other developers on your team to create their own local environment configuration, as well as hide your sensitive configuration items from source control.

Now, on your production server, create a `.env.php` file in your project root that contains the corresponding values for your production environment. Like the `.env.local.php` file, the production `.env.php` file should never be included in source control.

## Maintenance mode

When your application is in maintenance mode, a custom view will be displayed for all routes into your application. This makes it easy to "disable" your application while it is updating or when you are performing maintenance. A call to the `App::down` method is already present in your `app/start/global.php` file. The response from this method will be sent to users when your application is in maintenance mode.

To enable maintenance mode, simply execute the down Artisan command:

```
php artisan down
```

To disable maintenance mode, use the up command:

```
php artisan up
```

To show a custom view when your application is in maintenance mode, you may add something like the following to your application's `app/start/global.php` file:

```
App::down(function()
{
    return Response::view('maintenance', array(), 503);
});
```

If the Closure passed to the `down` method returns `NULL`, maintenance mode will be ignored for that request.

## Database configuration

Almost every modern web application interacts with a database. Laravel makes interacting with databases extremely simple across a variety of supported databases using raw SQL, a fluent query builder, and the Eloquent ORM. Currently, Laravel provides first-party support for five databases:

- MariaDB 10.3+
- MySQL 5.7+
- PostgreSQL 10.0+
- SQLite 3.8.8+
- SQL Server 2017+

### Configuration

The configuration for Laravel's database services is located in your application's **config/database.php** configuration file. In this file, you may define all of your database connections, as well as specify which connection should be used by default. Most of the configuration options within this file are driven by the values of your application's environment variables. Examples for most of Laravel's supported database systems are provided in this file.

### SQL Server Configuration

Laravel supports SQL Server out of the box; however, you will need to add the connection configuration for the database:

```
'sqlsrv' => [  
    'driver' => 'sqlsrv',  
    'host' => env('DB_HOST', 'localhost'),  
    'database' => env('DB_DATABASE', 'forge'),  
    'username' => env('DB_USERNAME', 'forge'),  
    'password' => env('DB_PASSWORD', ''),  
    'charset' => 'utf8',  
    'prefix' => '',  
],
```

### Configuration Using URLs

Typically, database connections are configured using multiple configuration values such as host, database, username, password, etc. Each of these configuration values has its own corresponding environment variable. This means that when configuring your database connection information on a production server, you need to manage several environment variables.

Some managed database providers such as AWS and Heroku provide a single database "URL" that contains all of the connection information for the database in a single string. An example database URL may look something like the following:

```
mysql://root:password@127.0.0.1/forge?charset=UTF-8
```

These URLs typically follow a standard schema convention:

```
driver://username:password@host:port/database?options
```

For convenience, Laravel supports these URLs as an alternative to configuring your database with multiple configuration options. If the url (or corresponding DATABASE\_URL environment variable) configuration option is present, it will be used to extract the database connection and credential information.

### Read / Write Connections

Sometimes you may wish to use one database connection for SELECT statements, and another for INSERT, UPDATE, and DELETE statements. Laravel makes this a breeze, and the proper connections will always be used whether you are using raw queries, the query builder, or the Eloquent ORM.

To see how read / write connections should be configured, let's look at this example:

```
'mysql' => [
  'read' => [
    'host' => [
      '192.168.1.1',
      '196.168.1.2',
    ],
  ],
  'write' => [
    'host' => [
      '196.168.1.3',
    ],
  ],
  'sticky' => true,
  'driver' => 'mysql',
  'database' => 'database',
  'username' => 'root',
  'password' => '',
  'charset' => 'utf8mb4',
  'collation' => 'utf8mb4_unicode_ci',
  'prefix' => '',
],
```



Note that two keys have been added to the configuration array: read and write. Both of these keys have array values containing a single key: host. The rest of the database options for the read and write connections will be merged from the main mysql array.

So, we only need to place items in the read and write arrays if we wish to override the values in the main array. So, in this case, 192.168.1.1 will be used as the "read" connection, while 192.168.1.2 will be used as the "write" connection. The database credentials, prefix, character set, and all other options in the main mysql array will be shared across both connections.



## Introduction

The default Laravel application structure is intended to provide a great starting point for both large and small applications. Of course, you are free to organize your application however you like. Laravel imposes almost no restrictions on where any given class is located - as long as Composer can autoload the class.

### The Root Directory

The **root** directory of a fresh Laravel installation contains a variety of directories:

The **app** directory, as you might expect, contains the core code of your application. We'll explore this directory in more detail soon.

The **bootstrap** directory contains a few files that bootstrap the framework and configure autoloading, as well as a cache directory that contains a few framework generated files for bootstrap performance optimization.

The **config** directory, as the name implies, contains all of your application's configuration files.

The **database** directory contains your database migration and seeds. If you wish, you may also use this directory to hold an SQLite database.

The **public** directory contains the front controller and your assets (images, JavaScript, CSS, etc.).

The **resources** directory contains your views, raw assets (LESS, SASS, CoffeeScript), and localization files.

The **storage** directory contains compiled Blade templates, file based sessions, file caches, and other files generated by the framework. This directory is segregated into app, framework, and logs directories. The app directory may be used to store any files utilized by your application. The framework directory is used to store framework generated files and caches. Finally, the logs directory contains your application's log files.

The **tests** directory contains your automated tests. An example PHPUnit is provided out of the box.

The **vendor** directory contains your Composer dependencies.



## The App Directory

The **"meat"** of your application lives in the app directory. By default, this directory is namespaced under App and is autoloaded by Composer using the PSR-4 autoloading standard.

The **app** directory ships with a variety of additional directories such as Console, Http, and Providers. Think of the Console and Http directories as providing an API into the "core" of your application. The HTTP protocol and CLI are both mechanisms to interact with your application, but do not actually contain application logic. In other words, they are simply two ways of issuing commands to your application. The Console directory contains all of your Artisan commands, while the Http directory contains your controllers, middleware, and requests.

The **Events** directory, as you might expect, houses event classes. Events may be used to alert other parts of your application that a given action has occurred, providing a great deal of flexibility and decoupling.

The **Exceptions** directory contains your application's exception handler and is also a good place to stick any exceptions thrown by your application.

The **Jobs** directory, of course, houses the queueable jobs for your application. Jobs may be queued by your application or run synchronously within the current request lifecycle.

The **Listeners** directory contains the handler classes for your events. Handlers receive an event and perform logic in response to the event being fired. For example, a UserRegistered event might be handled by a SendWelcomeEmail listener.

The **Policies** directory contains the authorization policy classes for your application. Policies are used to determine if a user can perform a given action against a resource. For more information check out the authorization documentation.

**Note:** Many of the classes in the app directory can be generated by Artisan via commands. To review the available commands, run the `php artisan list make` command in your terminal.

## Artisan Command Line Tool

### Introduction

Artisan is the name of the command-line interface included with Laravel. It provides a number of helpful commands for your use while developing your application. It is driven by the powerful Symfony Console component.

### Usage

#### Listing All Available Commands

To view a list of all available Artisan commands, you may use the list command:

```
php artisan list
```

#### Viewing The Help Screen For A Command

Every command also includes a "help" screen which displays and describes the command's available arguments and options. To view a help screen, simply precede the name of the command with help:

```
php artisan help migrate
```

## Writing Commands

In addition to the commands provided with Artisan, you may also build your own custom commands for working with your application. You may store your custom commands in the `app/Console/Commands` directory; however, you are free to choose your own storage location as long as your commands can be autoloaded based on your `composer.json` settings. To create a new command, you may use the `make:console` Artisan command, which will generate a command stub to help you get started:

```
php artisan make:console SendEmails
```

The command above would generate a class at `app/Console/Commands/SendEmails.php`. When creating the command, the `--command` option may be used to assign the terminal command name:

```
php artisan make:console SendEmails --command=emails:send
```

## Command Structure

Once your command is generated, you should fill out the signature and description properties of the class, which will be used when displaying your command on the list screen.

The `handle` method will be called when your command is executed. You may place any command logic in this method. Let's take a look at an example command.

Note that we are able to inject any dependencies we need into the command's constructor. The Laravel service container will automatically inject all dependencies type-hinted in the constructor. For greater code reusability, it is good practice to keep your console commands light and let them defer to application services to accomplish their tasks.

```
<?php
namespace App\Console\Commands;
use App\User;
use App\DripEmailer;
use Illuminate\Console\Command;

class SendEmails extends Command
{
    protected $signature = 'email:send {user}';

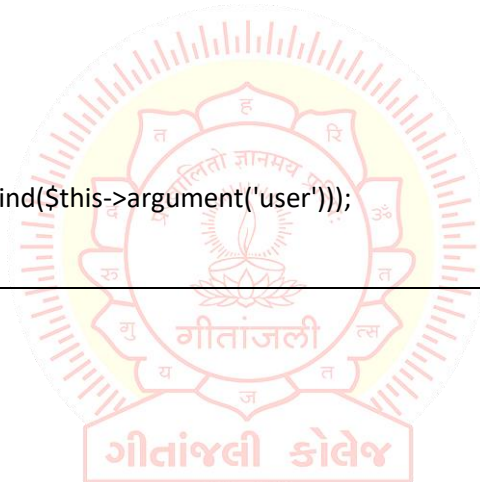
    protected $description = 'Send drip e-mails to a user';

    protected $drip;

    public function __construct(DripEmailer $drip)
    {
        parent::__construct();

        $this->drip = $drip;
    }

    public function handle()
    {
        $this->drip->send(User::find($this->argument('user')));
    }
}
```



## Types of Route Files

All Laravel routes are defined in your route files, which are located in the routes directory. These files are automatically loaded by your application's App\Providers\RouteServiceProvider. The routes/web.php file defines routes that are for your web interface. These routes are assigned the web middleware group, which provides features like session state and CSRF protection. The routes in routes/api.php are stateless and are assigned the api middleware group.

For most applications, you will begin by defining routes in your routes/web.php file. The routes defined in routes/web.php may be accessed by entering the defined route's URL in your browser. For example, you may access the following route by navigating to http://example.com/user in your browser:

```
use App\Http\Controllers\UserController;

Route::get('/user', [UserController::class, 'index']);
```

Routes defined in the routes/api.php file are nested within a route group by the RouteServiceProvider. Within this group, the /api URI prefix is automatically applied so you do not need to manually apply it to every route in the file. You may modify the prefix and other route group options by modifying your RouteServiceProvider class.

## Basic Routing

The most basic Laravel routes accept a URI and a closure, providing a very simple and expressive method of defining routes and behavior without complicated routing configuration files:

```
use Illuminate\Support\Facades\Route;

Route::get('/greeting', function () {
    return 'Hello World';
});
```

## Route Parameters

### Required Parameters

Sometimes you will need to capture segments of the URI within your route. For example, you may need to capture a user's ID from the URL. You may do so by defining route parameters:

```
Route::get('/user/{id}', function ($id) {
    return 'User ' . $id;
});
```

You may define as many route parameters as required by your route:

```
Route::get('/posts/{post}/comments/{comment}', function ($postId, $commentId) {
    //
});
```

Route parameters are always encased within {} braces and should consist of alphabetic characters. Underscores (\_) are also acceptable within route parameter names. Route parameters are injected into route callbacks / controllers based on their order - the names of the route callback / controller arguments do not matter.

### Parameters & Dependency Injection

If your route has dependencies that you would like the Laravel service container to automatically inject into your route's callback, you should list your route parameters after your dependencies:

```
use Illuminate\Http\Request;

Route::get('/user/{id}', function (Request $request, $id) {
    return 'User '.$id;
});
```

### Named Routes

Named routes allow the convenient generation of URLs or redirects for specific routes. You may specify a name for a route by chaining the name method onto the route definition:

```
Route::get('/user/profile', function () {
    //
})->name('profile');
```

You may also specify route names for controller actions:

```
Route::get(
    '/user/profile',
    [UserProfileController::class, 'show']
)->name('profile');
```

**Note : Route names should always be unique.**

### Generating URLs To Named Routes

Once you have assigned a name to a given route, you may use the route's name when generating URLs or redirects via Laravel's route and redirect helper functions:

```
// Generating URLs...
```

```
$url = route('profile');
```

```
// Generating Redirects...
```

```
return redirect()->route('profile');
```

```
return to_route('profile');
```

If the named route defines parameters, you may pass the parameters as the second argument to the route function. The given parameters will automatically be inserted into the generated URL in their correct positions:

```
Route::get('/user/{id}/profile', function ($id) {
    //
})->name('profile');

$url = route('profile', ['id' => 1]);
```

If you pass additional parameters in the array, those key / value pairs will automatically be added to the generated URL's query string:

```
Route::get('/user/{id}/profile', function ($id) {
    //
})->name('profile');

$url = route('profile', ['id' => 1, 'photos' => 'yes']);

// /user/1/profile?photos=yes
```

## Route Groups

Route groups allow you to share route attributes, such as middleware, across a large number of routes without needing to define those attributes on each individual route.

Nested groups attempt to intelligently "merge" attributes with their parent group. Middleware and where conditions are merged while names and prefixes are appended. Namespace delimiters and slashes in URI prefixes are automatically added where appropriate.

### Middleware

To assign middleware to all routes within a group, you may use the middleware method before defining the group. Middleware are executed in the order they are listed in the array:

```
Route::middleware(['first', 'second'])->group(function () {
    Route::get('/', function () {
        // Uses first & second middleware...
    });

    Route::get('/user/profile', function () {
        // Uses first & second middleware...
    });
});
```

## Controllers

If a group of routes all utilize the same controller, you may use the controller method to define the common controller for all of the routes within the group. Then, when defining the routes, you only need to provide the controller method that they invoke:

```
use App\Http\Controllers\OrderController;

Route::controller(OrderController::class)->group(function () {
    Route::get('/orders/{id}', 'show');
    Route::post('/orders', 'store');
});
```

## Route Model Binding

When injecting a model ID to a route or controller action, you will often query the database to retrieve the model that corresponds to that ID. Laravel route model binding provides a convenient way to automatically inject the model instances directly into your routes. For example, instead of injecting a user's ID, you can inject the entire User model instance that matches the given ID.

### Implicit Binding

Laravel automatically resolves Eloquent models defined in routes or controller actions whose type-hinted variable names match a route segment name. For example:

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    return $user->email;
});
```

Since the `$user` variable is type-hinted as the `App\Models\User` Eloquent model and the variable name matches the `{user}` URI segment, Laravel will automatically inject the model instance that has an ID matching the corresponding value from the request URI. If a matching model instance is not found in the database, a 404 HTTP response will automatically be generated.

Of course, implicit binding is also possible when using controller methods. Again, note the `{user}` URI segment matches the `$user` variable in the controller which contains an `App\Models\User` type-hint:

```
use App\Http\Controllers\UserController;
use App\Models\User;

// Route definition...
Route::get('/users/{user}', [UserController::class, 'show']);

// Controller method definition...
public function show(User $user)
{
    return view('user.profile', ['user' => $user]);
}
```



## Rate Limiting

### Defining Rate Limiters

Laravel includes powerful and customizable rate limiting services that you may utilize to restrict the amount of traffic for a given route or group of routes. To get started, you should define rate limiter configurations that meet your application's needs. Typically, this should be done within the configure RateLimiting method of your application's App\Providers\RouteServiceProvider class.

Rate limiters are defined using the RateLimiter facade's for method. The for method accepts a rate limiter name and a closure that returns the limit configuration that should apply to routes that are assigned to the rate limiter. Limit configuration are instances of the Illuminate\Cache\RateLimiting\Limit class. This class contains helpful "builder" methods so that you can quickly define your limit. The rate limiter name may be any string you wish:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\RateLimiter;

/**
 * Configure the rate limiters for the application.
 *
 * @return void
 */
protected function configureRateLimiting()
{
    RateLimiter::for('global', function (Request $request) {
        return Limit::perMinute(1000);
    });
}
```

If the incoming request exceeds the specified rate limit, a response with a 429 HTTP status code will automatically be returned by Laravel. If you would like to define your own response that should be returned by a rate limit, you may use the response method:

```
RateLimiter::for('global', function (Request $request) {
    return Limit::perMinute(1000)->response(function (Request $request, array $headers) {
        return response('Custom response...', 429, $headers);
    });
});
```

Since rate limiter callbacks receive the incoming HTTP request instance, you may build the appropriate rate limit dynamically based on the incoming request or authenticated user:

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()->vipCustomer()
        ? Limit::none()
        : Limit::perMinute(100);
});
```

## Accessing the current route

You may use the `current`, `currentRouteName`, and `currentRouteAction` methods on the Route facade to access information about the route handling the incoming request:

```
use Illuminate\Support\Facades\Route;

$route = Route::current(); // Illuminate\Routing\Route
$name = Route::currentRouteName(); // string
$action = Route::currentRouteAction(); // string
```

You may refer to the API documentation for both the underlying class of the Route facade and Route instance to review all of the methods that are available on the router and route classes.

## Routing Controllers

Routing controllers allow you to create the controller classes with methods used to handle the requests.

Now, we will understand the routing controllers through an example.

**Step 1:** First, we need to create a controller. We already created the controller named as 'PostController' in the previous topic.

**Step 2:** Open the web.php file and write the following code:

```
Route::get('/post', 'PostController@index');
```

In the above code, '/post' is the URL that we want to access, and PostController is the name of the controller. The 'index' is the name of the method available in the PostController.php file, and @index indicates that the index() method should be hit when we access the '/post' url.

**Step 3:** Add the code which is shown below as highlighted:

```
<?php

namespace App\Http\Controllers;
use Illuminate\Http\Request;

class PostController extends Controller {
    public function index() {
        return "Hello Geetanjali";
    }
    public function create() {
        //
    }
    public function store(Request $request) {
        //
    }
    public function show($id) {
        //
    }
}
```

```
public function edit($id) {  
    //  
}  
  
public function update(Request $request, $id) {  
    //  
}  
  
public function destroy($id) {  
    //  
}  
}
```

**Step 4:** Enter the URL in the browser, i.e., **localhost/laravelproject/public/host**, then the output would be shown as below:

Till now, we have observed how we can access the Controller. Now, we will see that how to pass the data to the Controller class.

## Passing data to the Controller

Let's understand through an example of how we can pass the data to the Controller:

**Step 1:** Open the **web.php** file and add the following code:

```
Route::get('/post/{id}', 'PostController@index');
```

The above code contains the 'id' parameter in the '/post' url.

**Step 2:** Edit the PostController.php file.

```
public function index($id)  
{  
    return "ID is :". $id;  
}
```

In the above case, we have updated the index() method. We have passed the 'id' parameter in the index() method.

**Step 3:** Enter the URL '**localhost/laravelproject/post/100**' into the web browser,

## Controllers and Namespaces

When we specify the controller class in **Route::get()** method, then we do not need to specify the full controller namespace. As **RouteServiceProvider** loads all the route files that contain the namespace, we just need to specify the class name that comes after the **App/Http/Controllers** portion of the namespace.

If the full controller class is **App/Http/Controllers/Post/PostController**, then we can register the routes of the Controller as given below:

```
Route::get('\post','Post\PostController@index');
```

### Single Action Controllers

If we want to use the single method in a controller, then we can use the **single \_\_invoke()** method on the controller.

When we create the controller by using the command `php artisan:make controller PostController` then the structure of the **PostController** file would be:

```
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class PostController extends Controller
{
    //
}
```

Now, we add the code of **\_\_invoke()** function in a **PostController** class:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class PostController extends Controller
{
    //
    public function __invoke($id)
    {
        return "id is : ". $id;
    }
}
```

In the end, we add the code in the **web.php** file, which is responsible for handling the actions.

```
route::get('/post/{id}','PostController');
```

The above code hits the **\_\_invoke()** method of a **PostController** class. This concludes that we do not need to write the **@invoke** method for accessing the single action controllers.

If no action is specified, i.e., we forget to write the **\_\_invoke()** method, then the **UnexpectedValueExpression** is thrown.

## Introduction

Instead of defining all of your request handling logic as closures in your route files, you may wish to organize this behavior using "controller" classes. Controllers can group related request handling logic into a single class. For example, a UserController class might handle all incoming requests related to users, including showing, creating, updating, and deleting users. By default, controllers are stored in the app/Http/Controllers directory.

### Basic Controllers

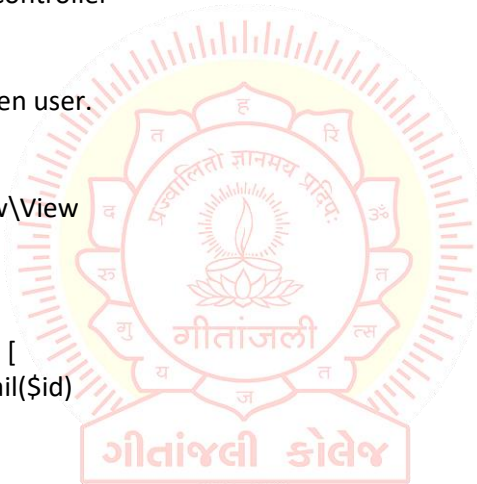
Let's take a look at an example of a basic controller. Note that the controller extends the base controller class included with Laravel: App\Http\Controllers\Controller:

```
<?php

namespace App\Http\Controllers;

use App\Models\User;

class UserController extends Controller
{
    /**
     * Show the profile for a given user.
     *
     * @param int $id
     * @return \Illuminate\View\View
     */
    public function show($id)
    {
        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```



You can define a route to this controller method like so:

```
use App\Http\Controllers\UserController;

Route::get('/user/{id}', [UserController::class, 'show']);
```

When an incoming request matches the specified route URI, the show method on the App\Http\Controllers\UserController class will be invoked and the route parameters will be passed to the method.

Controllers are not required to extend a base class. However, you will not have access to convenient features such as the middleware and authorize methods.

## Controller Middleware

Middleware may be assigned to the controller's routes in your route files:

```
Route::get('profile', [UserController::class, 'show']->middleware('auth'));
```

Or, you may find it convenient to specify middleware within your controller's constructor. Using the middleware method within your controller's constructor, you can assign middleware to the controller's actions:

```
class UserController extends Controller
{
    /**
     * Instantiate a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');
        $this->middleware('log')->only('index');
        $this->middleware('subscribed')->except('store');
    }
}
```

Controllers also allow you to register middleware using a closure. This provides a convenient way to define an inline middleware for a single controller without defining an entire middleware class:

```
$this->middleware(function ($request, $next) {
    return $next($request);
});
```

## Introduction

Blade is the simple, yet powerful templating engine that is included with Laravel. Unlike some PHP templating engines, Blade does not restrict you from using plain PHP code in your templates. In fact, all Blade templates are compiled into plain PHP code and cached until they are modified, meaning Blade adds essentially zero overhead to your application. Blade template files use the `.blade.php` file extension and are typically stored in the `resources/views` directory.

Blade views may be returned from routes or controllers using the global view helper. Of course, as mentioned in the documentation on views, data may be passed to the Blade view using the view helper's second argument:

```
Route::get('/', function () {
    return view('greeting', ['name' => 'Finn']);
});
```

## Displaying Data

You may display data that is passed to your Blade views by wrapping the variable in curly braces. For example, given the following route:

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

You may display the contents of the name variable like so:

```
Hello, {{ $name }}.
```

Blade's `{{ }}` echo statements are automatically sent through PHP's `htmlspecialchars` function to prevent XSS attacks.

You are not limited to displaying the contents of the variables passed to the view. You may also echo the results of any PHP function. In fact, you can put any PHP code you wish inside of a Blade echo statement:

```
The current UNIX timestamp is {{ time() }}.
```

## Components

Components and slots provide similar benefits to sections, layouts, and includes; however, some may find the mental model of components and slots easier to understand. There are two approaches to writing components: class based components and anonymous components.

To create a class based component, you may use the `make:component` Artisan command. To illustrate how to use components, we will create a simple Alert component. The `make:component` command will place the component in the `app/View/Components` directory:

```
php artisan make:component Alert
```



The **make:component** command will also create a view template for the component. The view will be placed in the `resources/views/components` directory. When writing components for your own application, components are automatically discovered within the `app/View/Components` directory and `resources/views/components` directory, so no further component registration is typically required.

## Slots

You will often need to pass additional content to your component via "slots". Component slots are rendered by echoing the `$slot` variable. To explore this concept, let's imagine that an alert component has the following markup:

```
<!-- /resources/views/components/alert.blade.php -->

<div class="alert alert-danger">
  {{ $slot }}
</div>
```

We may pass content to the slot by injecting content into the component:

```
<x-alert>
  <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

Sometimes a component may need to render multiple different slots in different locations within the component. Let's modify our alert component to allow for the injection of a "title" slot:

```
<!-- /resources/views/components/alert.blade.php -->

<span class="alert-title">{{ $title }}</span>

<div class="alert alert-danger">
  {{ $slot }}
</div>
```

You may define the content of the named slot using the `x-slot` tag. Any content not within an explicit `x-slot` tag will be passed to the component in the `$slot` variable:

```
<x-alert>
  <x-slot:title>
    Server Error
  </x-slot>

  <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

## Control Structures

In addition to template inheritance and displaying data, Blade also provides convenient short-cuts for common PHP control structures, such as conditional statements and loops. These short-cuts provide a very clean, terse way of working with PHP control structures, while also remaining familiar to their PHP counterparts.

### If Statements

You may construct if statements using the `@if`, `@elseif`, `@else`, and `@endif` directives. These directives function identically to their PHP counterparts:

```
@if (count($records) === 1)
    I have one record!
}elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

For convenience, Blade also provides an `@unless` directive:

```
@unless (Auth::check())
    You are not signed in.
@endunless
```

You may also determine if a given layout section has any content using the `@hasSection` directive:

```
<title>
    @hasSection ('title')
        @yield('title') - App Name
    @else
        App Name
    @endif
</title>
```

### Loops

In addition to conditional statements, Blade provides simple directives for working with PHP's supported loop structures. Again, each of these directives functions identically to their PHP counterparts:

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor
```

```
@foreach ($users as $user)
  <p>This is user {{ $user->id }}</p>
@endforeach
```

```
@foreach ($users as $user)
  <li>{{ $user->name }}</li>
@empty
  <p>No users</p>
@endforeach
```

```
@while (true)
  <p>'m looping forever.</p>
@endwhile
```

When using loops you might need to end the loop or skip the current iteration:

```
@foreach ($users as $user)
  @if ($user->type == 1)
    @continue
  @endif

  <li>{{ $user->name }}</li>

  @if ($user->number == 5)
    @break
  @endif
@endforeach
```

You may also include the condition with the directive declaration in one line:

```
@foreach ($users as $user)
  @continue($user->type == 1)

  <li>{{ $user->name }}</li>

  @break($user->number == 5)
@endforeach
```

### Including Sub-Views

Blade's `@include` directive, allows you to easily include a Blade view from within an existing view. All variables that are available to the parent view will be made available to the included view:

```
<div>
  @include('shared.errors')

  <form>
    <!-- Form Contents -->
  </form>
</div>
```

Even though the included view will inherit all data available in the parent view, you may also pass an array of extra data to the included view:

```
@include('view.name', ['some' => 'data'])
```

**Note:** You should avoid using the `__DIR__` and `__FILE__` constants in your Blade views, since they will refer to the location of the cached view.

### Rendering Views For Collections

You may combine loops and includes into one line with Blade's `@each` directive:

```
@each('view.name', $jobs, 'job')
```

The first argument is the view partial to render for each element in the array or collection. The second argument is the array or collection you wish to iterate over, while the third argument is the variable name that will be assigned to the current iteration within the view. So, for example, if you are iterating over an array of jobs, typically you will want to access each job as a `job` variable within your view partial. The key for the current iteration will be available as the `key` variable within your view partial.

You may also pass a fourth argument to the `@each` directive. This argument determines the view that will be rendered if the given array is empty.

```
@each('view.name', $jobs, 'job', 'view.empty')
```

### Comments

Blade also allows you to define comments in your views. However, unlike HTML comments, Blade comments are not included in the HTML returned by your application:

```
{{!-- This comment will not be present in the rendered HTML --}}
```

## Defining A Layout

Two of the primary benefits of using Blade are template inheritance and sections. To get started, let's take a look at a simple example. First, we will examine a "master" page layout. Since most web applications maintain the same general layout across various pages, it's convenient to define this layout as a single Blade view:

```
<!-- Stored in resources/views/layouts/master.blade.php -->

<html>
  <head>
    <title>App Name - @yield('title')</title>
  </head>
  <body>
    @section('sidebar')
      This is the master sidebar.
    @show
```

```
<div class="container">
  @yield('content')
</div>
</body>
</html>
```

As you can see, this file contains typical HTML mark-up. However, take note of the `@section` and `@yield` directives. The `@section` directive, as the name implies, defines a section of content, while the `@yield` directive is used to display the contents of a given section.

Now that we have defined a layout for our application, let's define a child page that inherits the layout.

## Extending A Layout

When defining a child page, you may use the Blade `@extends` directive to specify which layout the child page should "inherit". Views which `@extends` a Blade layout may inject content into the layout's sections using `@section` directives. Remember, as seen in the example above, the contents of these sections will be displayed in the layout using `@yield`:

```
<!-- Stored in resources/views/child.blade.php -->
```

```
@extends('layouts.master')
```

```
@section('title', 'Page Title')
```

```
@section('sidebar')
```

```
  @parent
```

```
  <p>This is appended to the master sidebar.</p>
```

```
@endsection
```

```
@section('content')
```

```
  <p>This is my body content.</p>
```

```
@endsection
```



In this example, the sidebar section is utilizing the `@parent` directive to append (rather than overwriting) content to the layout's sidebar. The `@parent` directive will be replaced by the content of the layout when the view is rendered.

Of course, just like plain PHP views, Blade views may be returned from routes using the global view helper function:

```
Route::get('blade', function () {
    return view('child');
});
```

## Including Subviews

Blade's `@include` directive allows you to include a Blade view from within another view. All variables that are available to the parent view will be made available to the included view:

```
<div>
  @include('shared.errors')

  <form>
    <!-- Form Contents -->
  </form>
</div>
```

Even though the included view will inherit all data available in the parent view, you may also pass an array of extra data to the included view:

```
@include('view.name', ['some' => 'data'])
```

If you attempt to `@include` a view which does not exist, Laravel will throw an error. If you would like to include a view that may or may not be present, you should use the `@includeIf` directive:

```
@includeIf('view.name', ['some' => 'data'])
```

If you would like to `@include` a view if a given boolean expression evaluates to true, you may use the `@includeWhen` directive:

```
@includeWhen($boolean, 'view.name', ['some' => 'data'])
```

If you would like to `@include` a view if a given boolean expression evaluates to false, you may use the `@includeUnless` directive:

```
@includeUnless($boolean, 'view.name', ['some' => 'data'])
```

To include the first view that exists from a given array of views, you may use the `includeFirst` directive:

```
@includeFirst(['custom.admin', 'admin'], ['some' => 'data'])
```

## Stacks

Blade allows you to push to named stacks which can be rendered somewhere else in another view or layout. This can be particularly useful for specifying any JavaScript libraries required by your child views:

```
@push('scripts')
  <script src="/example.js"></script>
@endpush
```

You may push to a stack as many times as needed. To render the complete stack contents, pass the name of the stack to the @stack directive:

```
<head>
  <!-- Head Contents -->

  @stack('scripts')
</head>
```

If you would like to prepend content onto the beginning of a stack, you should use the @prepend directive:

```
@push('scripts')
  This will be second...
@endpush
```

```
@prepend('scripts')
  This will be first...
@endprepend
```

## Service Injection

The @inject directive may be used to retrieve a service from the Laravel service container. The first argument passed to @inject is the name of the variable the service will be placed into, while the second argument is the class or interface name of the service you wish to resolve:

```
@inject('metrics', 'App\Services\MetricsService')

<div>
  Monthly Revenue: {{ $metrics->monthlyRevenue() }}.
</div>
```

### Extending Blade

Blade allows you to define your own custom directives using the directive method. When the Blade compiler encounters the custom directive, it will call the provided callback with the expression that the directive contains.



The following example creates a `@datetime($var)` directive which formats a given `$var`, which should be an instance of `DateTime`:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Blade::directive('datetime', function ($expression) {
            return "<?php echo ($expression)->format('m/d/Y H:i'); ?>";
        });
    }
}
```

As you can see, we will chain the `format` method onto whatever expression is passed into the directive. So, in this example, the final PHP generated by this directive will be:

```
<?php echo ($var)->format('m/d/Y H:i'); ?>
```

### Blade Operators

## Laravel Collective Forms & HTML

### Installation

Begin by installing this package through Composer. Edit your project's composer.json file to require laravelcollective/html.

```
$ composer require laravelcollective/html
```

Looking to install this package in Lumen? First of all, making this package compatible with Lumen will require some core changes to Lumen, which we believe would dampen the effectiveness of having Lumen in the first place. Secondly, it is our belief that if you need this package in your application, then you should be using Laravel anyway

### Opening A Form

```
{!! Form::open(['url' => 'foo/bar']) !!}  
  //  
{!! Form::close() !!}
```

By default, a POST method will be assumed; however, you are free to specify another method:

```
echo Form::open(['url' => 'foo/bar', 'method' => 'put'])
```

Note: Since HTML forms only support POST and GET, PUT and DELETE methods will be spoofed by automatically adding a `_method` hidden field to your form.

You may also open forms that point to named routes or controller actions:

```
echo Form::open(['route' => 'route.name'])  
  
echo Form::open(['action' => 'Controller@method'])
```

You may pass in route parameters as well:

```
echo Form::open(['route' => ['route.name', $user->id]])  
  
echo Form::open(['action' => ['Controller@method', $user->id]])
```

If your form is going to accept file uploads, add a `files` option to your array:

```
echo Form::open(['url' => 'foo/bar', 'files' => true])
```

## Labels

### Generating A Label Element

```
echo Form::label('email', 'E-Mail Address');
```

### Specifying Extra HTML Attributes

```
echo Form::label('email', 'E-Mail Address', ['class' => 'awesome']);
```

**Note:** After creating a label, any form element you create with a name matching the label name will automatically receive an ID matching the label name as well.

## Generating A Text Input

```
echo Form::text('username');
```

### Specifying A Default Value

```
echo Form::text('email', 'example@gmail.com');
```

Note: The hidden and textarea methods have the same signature as the text method.

## Generating A Password Input

```
echo Form::password('password', ['class' => 'awesome']);
```

## Generating Other Inputs

```
echo Form::email($name, $value = null, $attributes = []);
```

```
echo Form::file($name, $attributes = []);
```

## Checkboxes and Radio Buttons

### Generating A Checkbox Or Radio Input

```
echo Form::checkbox('name', 'value');
```

```
echo Form::radio('name', 'value');
```

### Generating A Checkbox Or Radio Input That Is Checked

```
echo Form::checkbox('name', 'value', true);
```

```
echo Form::radio('name', 'value', true);
```

## Number

### Generating A Number Input

```
echo Form::number('name', 'value');
```

## Date

### Generating A Date Input

```
echo Form::date('name', \Carbon\Carbon::now());
```

## File Input

### Generating A File Input

```
echo Form::file('image');
```

Note: The form must have been opened with the files option set to true.

**Drop-Down Lists**

## Generating A Drop-Down List

```
echo Form::select('size', ['L' => 'Large', 'S' => 'Small']);
```

## Generating A Drop-Down List With Selected Default

```
echo Form::select('size', ['L' => 'Large', 'S' => 'Small'], 'S');
```

## Generating a Drop-Down List With an Empty Placeholder

This will create an <option> element with no value as the very first option of your drop-down.

```
echo Form::select('size', ['L' => 'Large', 'S' => 'Small'], null, ['placeholder' => 'Pick a size...']);
```

## Generating A Grouped List

```
echo Form::select('animal', [
    'Cats' => ['leopard' => 'Leopard'],
    'Dogs' => ['spaniel' => 'Spaniel'],
]);
```

## Generating A Drop-Down List With A Range

```
echo Form::selectRange('number', 10, 20);
```

## Generating A List With Month Names

```
echo Form::selectMonth('month');
```

**Buttons**

## Generating A Submit Button

```
echo Form::submit('Click Me!');
```

Note: Need to create a button element? Try the button method. It has the same signature as submit.

**Custom Macros**

## Registering A Form Macro

It's easy to define your own custom Form class helpers called "macros". Here's how it works. First, simply register the macro with a given name and a Closure:

```
Form::macro('myField', function()
{
    return '<input type="awesome">';
});
```

Now you can call your macro using its name:

## Calling A Custom Form Macro

```
echo Form::myField();
```

### CSRF Protection

#### Adding The CSRF Token To A Form

Laravel provides an easy method of protecting your application from cross-site request forgeries. First, a random token is placed in your user's session. If you use the `Form::open` method with POST, PUT or DELETE the CSRF token will be added to your forms as a hidden field automatically. Alternatively, if you wish to generate the HTML for the hidden CSRF field, you may use the `token` method:

```
echo Form::token();
```

#### Attaching The CSRF Filter To A Route

```
Route::post('profile',  
  [  
    'before' => 'csrf',  
    function()  
    {  
      //  
    }  
  ]  
);
```



## Defining The Routes

First, let's assume we have the following routes defined in our app/Http/routes.php file:

```
// Display a form to create a blog post...
Route::get('post/create', 'PostController@create');
```

```
// Store a new blog post...
Route::post('post', 'PostController@store');
```

Of course, the GET route will display a form for the user to create a new blog post, while the POST route will store the new blog post in the database.

## Creating The Controller

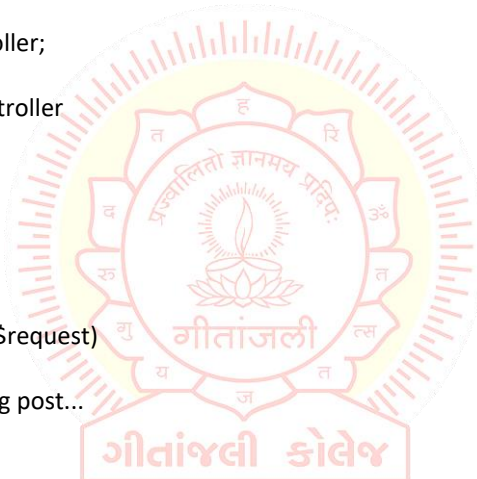
Next, let's take a look at a simple controller that handles these routes. We'll leave the store method empty for now:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
    public function create()
    {
        return view('post.create');
    }

    public function store(Request $request)
    {
        // Validate and store the blog post...
    }
}
```



## Writing the Validation Logic

Now we are ready to fill in our store method with the logic to validate the new blog post. If you examine your application's base controller (App\Http\Controllers\Controller) class, you will see that the class uses a ValidatesRequests trait. This trait provides a convenient validate method in all of your controllers.

The validate method accepts an incoming HTTP request and a set of validation rules. If the validation rules pass, your code will keep executing normally; however, if validation fails, an exception will be thrown and the proper error response will automatically be sent back to the user. In the case of a traditional HTTP request, a redirect response will be generated, while a JSON response will be sent for AJAX requests.

To get a better understanding of the validate method, let's jump back into the store method:

```
public function store(Request $request)
{
    $this->validate($request, [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // The blog post is valid, store in database...
}
```

As you can see, we simply pass the incoming HTTP request and desired validation rules into the validate method. Again, if the validation fails, the proper response will automatically be generated. If the validation passes, our controller will continue executing normally.

### Stopping On First Validation Failure

Sometimes you may wish to stop running validation rules on an attribute after the first validation failure. To do so, assign the bail rule to the attribute:

```
$this->validate($request, [
    'title' => 'bail|required|unique:posts|max:255',
    'body' => 'required',
]);
```

In this example, if the required rule on the title attribute fails, the unique rule will not be checked. Rules will be validated in the order they are assigned.

### A Note On Nested Attributes

If your HTTP request contains "nested" parameters, you may specify them in your validation rules using "dot" syntax:

```
$this->validate($request, [
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

### Displaying The Validation Errors

So, what if the incoming request parameters do not pass the given validation rules? As mentioned previously, Laravel will automatically redirect the user back to their previous location. In addition, all of the validation errors will automatically be flashed to the session.

Again, notice that we did not have to explicitly bind the error messages to the view in our GET route. This is because Laravel will check for errors in the session data, and automatically bind them to the view if they are available. The \$errors variable will be an instance of Illuminate\Support\MessageBag. For more information on working with this object, check out its documentation.



Note: The `$errors` variable is bound to the view by the `Illuminate\View\Middleware\ShareErrorsFromSession` middleware, which is provided by the web middleware group. When this middleware is applied an `$errors` variable will always be available in your views, allowing you to conveniently assume the `$errors` variable is always defined and can be safely used.

So, in our example, the user will be redirected to our controller's create method when validation fails, allowing us to display the error messages in the view:

```
<!-- /resources/views/post/create.blade.php -->
```

```
<h1>Create Post</h1>
@if (count($errors) > 0)
  <div class="alert alert-danger">
    <ul>
      @foreach ($errors->all() as $error)
        <li>{{ $error }}</li>
      @endforeach
    </ul>
  </div>
@endif
```

## Validating Arrays

Validating array form input fields doesn't have to be a pain. For example, to validate that each e-mail in a given array input field is unique, you may do the following:

```
$validator = Validator::make($request->all(), [
  'person.*.email' => 'email|unique:users',
  'person.*.first_name' => 'required_with:person.*.last_name',
]);
```

Likewise, you may use the `*` character when specifying your validation messages in your language files, making it a breeze to use a single validation message for array based fields:

```
'custom' => [
  'person.*.email' => [
    'unique' => 'Each person must have a unique e-mail address',
  ]
],
```

## Customizing The Error Messages

You may customize the error messages used by the form request by overriding the `messages` method. This method should return an array of attribute / rule pairs and their corresponding error messages:

```
public function messages(){
  return [
    'title.required' => 'A title is required',
    'body.required' => 'A message is required',
  ];
}
```

**accepted**

The field under validation must be yes, on, 1, or true. This is useful for validating "Terms of Service" acceptance.

**after:date**

The field under validation must be a value after a given date. The dates will be passed into the strtotime PHP function:

```
'start_date' => 'required|date|after:tomorrow'
```

Instead of passing a date string to be evaluated by strtotime, you may specify another field to compare against the date:

```
'finish_date' => 'required|date|after:start_date'
```

**alpha**

The field under validation must be entirely alphabetic characters.

**alpha\_dash**

The field under validation may have alpha-numeric characters, as well as dashes and underscores.

**alpha\_num**

The field under validation must be entirely alpha-numeric characters.

**array**

The field under validation must be a PHP array.

**before:date**

The field under validation must be a value preceding the given date. The dates will be passed into the PHP strtotime function.

**between:min,max**

The field under validation must have a size between the given min and max. Strings, numerics, and files are evaluated in the same fashion as the size rule.

**boolean**

The field under validation must be able to be cast as a boolean. Accepted input are true, false, 1, 0, "1", and "0".

**date**

The field under validation must be a valid date according to the strtotime PHP function.

**date\_format:format**

The field under validation must match the given format. The format will be evaluated using the PHP date\_parse\_from\_format function. You should use either date or date\_format when validating a field, not both.

**different:field**

The field under validation must have a different value than field.

**digits:value**

The field under validation must be numeric and must have an exact length of value.

**digits\_between:min,max**

The field under validation must have a length between the given min and max.

**email**

The field under validation must be formatted as an e-mail address.

**exists:table,column**

The field under validation must exist on a given database table.

**Basic Usage Of Exists Rule**

```
'state' => 'exists:states'
```

Specifying A Custom Column Name

```
'state' => 'exists:states,abbreviation'
```

You may also specify more conditions that will be added as "where" clauses to the query:

```
'email' => 'exists:staff,email,account_id,1'
```

These conditions may be negated using the ! sign:

```
'email' => 'exists:staff,email,role,!admin'
```

You may also pass NULL or NOT\_NULL to the "where" clause:

```
'email' => 'exists:staff,email,deleted_at,NULL'
```

```
'email' => 'exists:staff,email,deleted_at,NOT_NULL'
```

Occasionally, you may need to specify a specific database connection to be used for the exists query. You can accomplish this by prepending the connection name to the table name using "dot" syntax:

```
'email' => 'exists:connection.staff,email'
```

**file**

The field under validation must be a successfully uploaded file.

**in:foo,bar,...**

The field under validation must be included in the given list of values.

**integer**

The field under validation must be an integer.

**max:value**

The field under validation must be less than or equal to a maximum value. Strings, numerics, and files are evaluated in the same fashion as the size rule

**min:value**

The field under validation must have a minimum value. Strings, numerics, and files are evaluated in the same fashion as the size rule.

**not\_in:foo,bar,...**

The field under validation must not be included in the given list of values.

**numeric**

The field under validation must be numeric.

**regex:pattern**

The field under validation must match the given regular expression.

**Note:** When using the regex pattern, it may be necessary to specify rules in an array instead of using pipe delimiters, especially if the regular expression contains a pipe character.

**required**

The field under validation must be present in the input data and not empty. A field is considered "empty" if one of the following conditions are true:

The value is null.

The value is an empty string.

The value is an empty array or empty Countable object.

The value is an uploaded file with no path.

**Custom Validation Rules**

Laravel provides a variety of helpful validation rules; however, you may wish to specify some of your own. One method of registering custom validation rules is using the extend method on the Validator facade. Let's use this method within a service provider to register a custom validation rule:

```
namespace App\Providers;

use Validator;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider{
    public function boot() {
        Validator::extend('foo', function($attribute, $value, $parameters, $validator) {
            return $value == 'foo';
        });
    }

    function register() {
        //
    }
}
```

The custom validator Closure receives four arguments: the name of the \$attribute being validated, the \$value of the attribute, an array of \$parameters passed to the rule, and the Validator instance.

## Generating Migrations

To create a migration, use the `make:migration` Artisan command:

```
php artisan make:migration create_users_table
```

The new migration will be placed in your `database/migrations` directory. Each migration file name contains a timestamp which allows Laravel to determine the order of the migrations.

The `--table` and `--create` options may also be used to indicate the name of the table and whether the migration will be creating a new table. These options simply pre-fill the generated migration stub file with the specified table:

```
php artisan make:migration add_votes_to_users_table --table=users  
php artisan make:migration create_users_table --create=users
```

If you would like to specify a custom output path for the generated migration, you may use the `--path` option when executing the `make:migration` command. The provided path should be relative to your application's base path.

### Migration Structure

A migration class contains two **methods: up and down**. The up method is used to add new tables, columns, or indexes to your database, while the down method should simply reverse the operations performed by the up method.

Within both of these methods you may use the Laravel schema builder to expressively create and modify tables. To learn about all of the methods available on the Schema builder, check out its documentation. For example, let's look at a sample migration that creates a flights table:

```
<?php  
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Database\Migrations\Migration;  
class CreateFlightsTable extends Migration{  
    public function up() {  
        Schema::create('flights', function (Blueprint $table) {  
            $table->increments('id');  
            $table->string('name');  
            $table->string('airline');  
            $table->timestamps();  
        });  
    }  
    public function down() {  
        Schema::drop('flights');  
    }  
}
```

## Creation Migration

To run all outstanding migrations for your application, use migrate Artisan command. If you are using the Homestead virtual machine, you should run this command from within your VM:

```
php artisan migrate
```

If you receive a "class not found" error when running migrations, try running the composer dump-autoload command and re-issuing the migrate command.

## Rolling Back Migrations

To rollback the latest migration "operation", you may use the rollback command. Note that this rolls back the last "batch" of migrations that ran, which may include multiple migration files:

```
php artisan migrate:rollback
```

## Writing Migrations

### Creating Tables

To create a new database table, use the create method on the Schema facade. The create method accepts two arguments. The first is the name of the table, while the second is a Closure which receives a Blueprint object used to define the new table:

```
Schema::create('users', function (Blueprint $table) {
    $table->increments('id');
});
```

Of course, when creating the table, you may use any of the schema builder's column methods to define the table's columns.

### Checking For Table / Column Existence

You may easily check for the existence of a table or column using the **hasTable** and **hasColumn** methods:

```
if (Schema::hasTable('users')) {
    //
}
if (Schema::hasColumn('users', 'email')) {
    //
}
```

### Renaming / Dropping Tables

To rename an existing database table, use the rename method:

```
Schema::rename($from, $to);
```

To drop an existing table, you may use the drop or dropIfExists methods:

```
Schema::drop('users');
Schema::dropIfExists('users');
```

### Renaming Tables with Foreign Keys

Before renaming a table, you should verify that any foreign key constraints on the table have an explicit name in your migration files instead of letting Laravel assign a convention based name. Otherwise, the foreign key constraint name will refer to the old table name.

### Creating Columns

To update an existing table, we will use the table method on the Schema facade. Like the create method, the table method accepts two arguments: the name of the table and a Closure that receives a Blueprint instance we can use to add columns to the table:

```
Schema::table('users', function ($table) {
    $table->string('email');
});
```

### Available Column Types

Of course, the schema builder contains a variety of column types that you may use when building your tables:

Command	Description
<code>\$table-&gt;bigIncrements('id');</code>	Incrementing ID (primary key) using a "UNSIGNED BIG INTEGER" equivalent.
<code>\$table-&gt;bigInteger('votes');</code>	BIGINT equivalent for the database.
<code>\$table-&gt;binary('data');</code>	BLOB equivalent for the database.
<code>\$table-&gt;boolean('confirmed');</code>	BOOLEAN equivalent for the database.
<code>\$table-&gt;char('name', 4);</code>	CHAR equivalent with a length.
<code>\$table-&gt;date('created_at');</code>	DATE equivalent for the database.
<code>\$table-&gt;dateTime('created_at');</code>	DATETIME equivalent for the database.
<code>\$table-&gt;decimal('amount', 5, 2);</code>	DECIMAL equivalent with a precision and scale.
<code>\$table-&gt;double('column', 15, 8);</code>	DOUBLE equivalent with precision, 15 digits in total and 8 after the decimal point.
<code>\$table-&gt;enum('choices', ['foo', 'bar']);</code>	ENUM equivalent for the database.
<code>\$table-&gt;float('amount');</code>	FLOAT equivalent for the database.
<code>\$table-&gt;increments('id');</code>	Incrementing ID (primary key) using a "UNSIGNED INTEGER" equivalent.
<code>\$table-&gt;integer('votes');</code>	INTEGER equivalent for the database.
<code>\$table-&gt;ipAddress('visitor');</code>	IP address equivalent for the database.
<code>\$table-&gt;json('options');</code>	JSON equivalent for the database.
<code>\$table-&gt;longText('description');</code>	LONGTEXT equivalent for the database.
<code>\$table-&gt;macAddress('device');</code>	MAC address equivalent for the database.
<code>\$table-&gt;mediumInteger('numbers');</code>	MEDIUMINT equivalent for the database.
<code>\$table-&gt;mediumText('description');</code>	MEDIUMTEXT equivalent for the database.
<code>\$table-&gt;rememberToken();</code>	Adds remember_token as VARCHAR(100) NULL.
<code>\$table-&gt;smallInteger('votes');</code>	SMALLINT equivalent for the database.
<code>\$table-&gt;string('email');</code>	VARCHAR equivalent column.
<code>\$table-&gt;string('name', 100);</code>	VARCHAR equivalent with a length.
<code>\$table-&gt;text('description');</code>	TEXT equivalent for the database.
<code>\$table-&gt;time('sunrise');</code>	TIME equivalent for the database.



Command	Description
<code>\$table-&gt;tinyInteger('numbers');</code>	TINYINT equivalent for the database.
<code>\$table-&gt;timestamp('added_on');</code>	TIMESTAMP equivalent for the database.
<code>\$table-&gt;timestamps();</code>	Adds created_at and updated_at columns.

## Column Modifiers

In addition to the column types listed above, there are several other column "modifiers" which you may use while adding the column. For example, to make the column "nullable", you may use the nullable method:

```
Schema::table('users', function ($table) {
    $table->string('email')->nullable();
});
```

Below is a list of all the available column modifiers. This list does not include the index modifiers:

Modifier	Description
<code>-&gt;first()</code>	Place the column "first" in the table (MySQL Only)
<code>-&gt;after('column')</code>	Place the column "after" another column (MySQL Only)
<code>-&gt;nullable()</code>	Allow NULL values to be inserted into the column
<code>-&gt;default(\$value)</code>	Specify a "default" value for the column
<code>-&gt;unsigned()</code>	Set integer columns to UNSIGNED
<code>-&gt;comment('my comment')</code>	Add a comment to a column

## Database seeding

### Introduction

Laravel includes a simple method of seeding your database with test data using seed classes. All seed classes are stored in database/seeds. Seed classes may have any name you wish, but probably should follow some sensible convention, such as UsersTableSeeder, etc. By default, a DatabaseSeeder class is defined for you. From this class, you may use the call method to run other seed classes, allowing you to control the seeding order.

### Writing Seeders

To generate a seeder, you may issue the `make:seeder` Artisan command. All seeders generated by the framework will be placed in the database/seeds directory:

```
php artisan make:seeder UsersTableSeeder
```

A seeder class only contains one method by default: `run`. This method is called when the `db:seed` Artisan command is executed. Within the `run` method, you may insert data into your database however you wish. You may use the query builder to manually insert data or you may use Eloquent model factories.

As an example, let's modify the DatabaseSeeder class which is included with a default installation of Laravel. Let's add a database insert statement to the `run` method:

```
<?php
```

```
use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;
class DatabaseSeeder extends Seeder
{
    public function run()
    {
        DB::table('users')->insert([
            'name' => str_random(10),
            'email' => str_random(10).'@gmail.com',
            'password' => bcrypt('secret'),
        ]);
    }
}
```

### Running Seeders

Once you have written your seeder classes, you may use the `db:seed` Artisan command to seed your database. By default, the `db:seed` command runs the `DatabaseSeeder` class, which may be used to call other seed classes. However, you may use the `--class` option to specify a specific seeder class to run individually:

```
php artisan db:seed
php artisan db:seed --class=UsersTableSeeder
```

You may also seed your database using the `migrate:refresh` command, which will also roll-back and re-run all of your migrations. This command is useful for completely re-building your database:

```
php artisan migrate:refresh --seed
```

## Running Raw SQL Queries

Once you have configured your database connection, you may run queries using the DB facade. The DB facade provides methods for each type of query: select, update, insert, delete, and statement.

### Running A Select Query

To run a basic query, we can use the select method on the DB facade:

```
<?php
namespace App\Http\Controllers;

use DB;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    public function index()
    {
        $users = DB::select('select * from users where active = ?', [1]);
        return view('user.index', ['users' => $users]);
    }
}
```

The first argument passed to the select method is the raw SQL query, while the second argument is any parameter bindings that need to be bound to the query. Typically, these are the values of the where clause constraints. Parameter binding provides protection against SQL injection.

The select method will always return an array of results. Each result within the array will be a PHP stdClass object, allowing you to access the values of the results:

```
foreach ($users as $user) {
    echo $user->name;
}
```

### Using Named Bindings

Instead of using ? to represent your parameter bindings, you may execute a query using named bindings:

```
$results = DB::select('select * from users where id = :id', ['id' => 1]);
```

### Running An Insert Statement

To execute an insert statement, you may use the insert method on the DB facade. Like select, this method takes the raw SQL query as its first argument, and bindings as the second argument:

```
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Dayle']);
```

### Running An Update Statement

The update method should be used to update existing records in the database. The number of rows affected by the statement will be returned by the method:

```
$affected = DB::update('update users set votes = 100 where name = ?', ['John']);
```

### Running A Delete Statement

The delete method should be used to delete records from the database. Like update, the number of rows deleted will be returned:

```
$deleted = DB::delete('delete from users');
```

### Running A General Statement

Some database statements should not return any value. For these types of operations, you may use the statement method on the DB facade:

```
DB::statement('drop table users');
```

### Database Transactions

To run a set of operations within a database transaction, you may use the transaction method on the DB facade. If an exception is thrown within the transaction Closure, the transaction will automatically be rolled back. If the Closure executes successfully, the transaction will automatically be committed. You don't need to worry about manually rolling back or committing while using the transaction method:

```
DB::transaction(function () {  
    DB::table('users')->update(['votes' => 1]);  
  
    DB::table('posts')->delete();  
});
```

### Manually Using Transactions

If you would like to begin a transaction manually and have complete control over rollbacks and commits, you may use the beginTransaction method on the DB facade:

```
DB::beginTransaction();
```

You can rollback the transaction via the rollBack method:

```
DB::rollBack();
```

Lastly, you can commit a transaction via the commit method:

```
DB::commit();
```

**Note:** Using the DB facade's transaction methods also controls transactions for the query builder and Eloquent ORM.

## Introduction

The database query builder provides a convenient, fluent interface to creating and running database queries. It can be used to perform most database operations in your application, and works on all supported database systems.

**Note:** The Laravel query builder uses PDO parameter binding to protect your application against SQL injection attacks. There is no need to clean strings being passed as bindings.

### Retrieving Results

#### Retrieving All Rows From A Table

To begin a fluent query, use the table method on the DB facade. The table method returns a fluent query builder instance for the given table, allowing you to chain more constraints onto the query and then finally get the results. In this example, let's just get all records from a table:

```
<?php
namespace App\Http\Controllers;

use DB;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    public function index()
    {
        $users = DB::table('users')->get();

        return view('user.index', ['users' => $users]);
    }
}
```

Like raw queries, the get method returns an array of results where each result is an instance of the PHP StdClass object. You may access each column's value by accessing the column as a property of the object:

```
foreach ($users as $user) {
    echo $user->name;
}
```

#### Retrieving A Single Row / Column From A Table

If you just need to retrieve a single row from the database table, you may use the first method. This method will return a single StdClass object:

```
$user = DB::table('users')->where('name', 'John')->first();

echo $user->name;
```

If you don't even need an entire row, you may extract a single value from a record using the `value` method. This method will return the value of the column directly:

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

### Chunking Results From A Table

If you need to work with thousands of database records, consider using the `chunk` method. This method retrieves a small "chunk" of the results at a time, and feeds each chunk into a Closure for processing. This method is very useful for writing Artisan commands that process thousands of records. For example, let's work with the entire `users` table in chunks of 100 records at a time:

```
DB::table('users')->orderBy('id')->chunk(100, function($users) {
    foreach ($users as $user) {
        //
    }
});
```

You may stop further chunks from being processed by returning `false` from the Closure:

```
DB::table('users')->orderBy('id')->chunk(100, function($users) {
    // Process the records...

    return false;
});
```

### Retrieving A List Of Column Values

If you would like to retrieve an array containing the values of a single column, you may use the `pluck` method. In this example, we'll retrieve an array of role titles:

```
$titles = DB::table('roles')->pluck('title');

foreach ($titles as $title) {
    echo $title;
}
```

You may also specify a custom key column for the returned array:

```
$roles = DB::table('roles')->pluck('title', 'name');

foreach ($roles as $name => $title) {
    echo $title;
}
```

### Aggregates

The query builder also provides a variety of aggregate methods, such as `count`, `max`, `min`, `avg`, and `sum`. You may call any of these methods after constructing your query:

```
$users = DB::table('users')->count();
```

```
$price = DB::table('orders')->max('price');
```

Of course, you may combine these methods with other clauses to build your query:

```
$price = DB::table('orders')
    ->where('finalized', 1)
    ->avg('price');
```

## Selects

### Specifying A Select Clause

Of course, you may not always want to select all columns from a database table. Using the select method, you can specify a custom select clause for the query:

```
$users = DB::table('users')->select('name', 'email as user_email')->get();
```

The distinct method allows you to force the query to return distinct results:

```
$users = DB::table('users')->distinct()->get();
```

If you already have a query builder instance and you wish to add a column to its existing select clause, you may use the addSelect method:

```
$query = DB::table('users')->select('name');
```

```
$users = $query->addSelect('age')->get();
```

### Raw Expressions

Sometimes you may need to use a raw expression in a query. These expressions will be injected into the query as strings, so be careful not to create any SQL injection points! To create a raw expression, you may use the DB::raw method:

```
$users = DB::table('users')
    ->select(DB::raw('count(*) as user_count, status'))
    ->where('status', '<>', 1)
    ->groupBy('status')
    ->get();
```

## Joins

### Inner Join Statement

The query builder may also be used to write join statements. To perform a basic SQL "inner join", you may use the join method on a query builder instance. The first argument passed to the join method is the name of the table you need to join to, while the remaining arguments specify the column constraints for the join. Of course, as you can see, you can join to multiple tables in a single query:

```
$users = DB::table('users')
    ->join('contacts', 'users.id', '=', 'contacts.user_id')
    ->join('orders', 'users.id', '=', 'orders.user_id')
    ->select('users.*', 'contacts.phone', 'orders.price')
    ->get();
```



**Left Join Statement**

If you would like to perform a "left join" instead of an "inner join", use the leftJoin method. The leftJoin method has the same signature as the join method:

```
$users = DB::table('users')
    ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
    ->get();
```

**Cross Join Statement**

To perform a "cross join" use the crossJoin method with the name of the table you wish to cross join to. Cross joins generate a cartesian product between the first table and the joined table:

```
$users = DB::table('sizes')
    ->crossJoin('colours')
    ->get();
```

**Advanced Join Statements**

You may also specify more advanced join clauses. To get started, pass a Closure as the second argument into the join method. The Closure will receive a JoinClause object which allows you to specify constraints on the join clause:

```
DB::table('users')
    ->join('contacts', function ($join) {
        $join->on('users.id', '=', 'contacts.user_id')->orOn(...);
    })
    ->get();
```

If you would like to use a "where" style clause on your joins, you may use the where and orWhere methods on a join. Instead of comparing two columns, these methods will compare the column against a value:

```
DB::table('users')
    ->join('contacts', function ($join) {
        $join->on('users.id', '=', 'contacts.user_id')
            ->where('contacts.user_id', '>', 5);
    })
    ->get();
```

**Where Clauses****Simple Where Clauses**

To add where clauses to the query, use the where method on a query builder instance. The most basic call to where requires three arguments. The first argument is the name of the column. The second argument is an operator, which can be any of the database's supported operators. The third argument is the value to evaluate against the column.

For example, here is a query that verifies the value of the "votes" column is equal to 100:

```
$users = DB::table('users')->where('votes', '=', 100)->get();
```

For convenience, if you simply want to verify that a column is equal to a given value, you may pass the value directly as the second argument to the where method:

```
$users = DB::table('users')->where('votes', 100)->get();
```

Of course, you may use a variety of other operators when writing a where clause:

```
$users = DB::table('users')
    ->where('votes', '>=', 100)
    ->get();
```

```
$users = DB::table('users')
    ->where('votes', '<>', 100)
    ->get();
```

```
$users = DB::table('users')
    ->where('name', 'like', 'T%')
    ->get();
```

You may also pass an array of conditions to the where function:

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

### Or Statements

You may chain where constraints together, as well as add or clauses to the query. The or-Where method accepts the same arguments as the where method:

```
$users = DB::table('users')
    ->where('votes', '>', 100)
    ->orWhere('name', 'John')
    ->get();
```

### Additional Where Clauses

#### whereBetween

The whereBetween method verifies that a column's value is between two values:

```
$users = DB::table('users')
    ->whereBetween('votes', [1, 100])->get();
```

**whereNotBetween**

The whereNotBetween method verifies that a column's value lies outside of two values:

```
$users = DB::table('users')
    ->whereNotBetween('votes', [1, 100])
    ->get();
```

**whereIn / whereNotIn**

The whereIn method verifies that a given column's value is contained within the given array:

```
$users = DB::table('users')
    ->whereIn('id', [1, 2, 3])
    ->get();
```

The **whereNotIn** method verifies that the given column's value is not contained in the given array:

```
$users = DB::table('users')
    ->whereNotIn('id', [1, 2, 3])
    ->get();
```

**whereNull / whereNotNull**

The **whereNull** method verifies that the value of the given column is NULL:

```
$users = DB::table('users')
    ->whereNull('updated_at')
    ->get();
```

The **whereNotNull** method verifies that the column's value is not NULL:

```
$users = DB::table('users')
    ->whereNotNull('updated_at')
    ->get();
```

**whereColumn**

The whereColumn method may be used to verify that two columns are equal:

```
$users = DB::table('users')
    ->whereColumn('first_name', 'last_name');
```

You may also pass a comparison operator to the method:

The whereColumn method can also be passed an array of multiple conditions. These conditions will be joined using the and operator:

```
$users = DB::table('users')
    ->whereColumn([
        ['first_name', 'last_name'],
        ['updated_at', '>', 'created_at']
    ]);
```

## Defining Models

To get started, let's create an Eloquent model. Models typically live in the app directory, but you are free to place them anywhere that can be auto-loaded according to your composer.json file. All Eloquent models extend Illuminate\Database\Eloquent\Model class.

The easiest way to create a model instance is using the make:model Artisan command:

```
php artisan make:model User
```

If you would like to generate a database migration when you generate the model, you may use the **--migration** or **-m** option:

```
php artisan make:model User --migration
```

```
php artisan make:model User -m
```

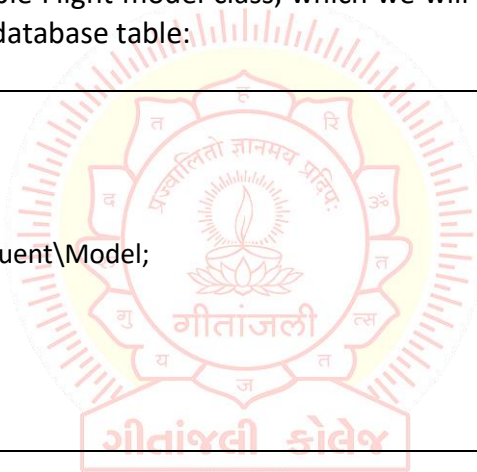
### Eloquent Model Conventions

Now, let's look at an example Flight model class, which we will use to retrieve and store information from our flights database table:

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    //
}
```



### Table Names

Note that we did not tell Eloquent which table to use for our Flight model. The "snake case", plural name of the class will be used as the table name unless another name is explicitly specified. So, in this case, Eloquent will assume the Flight model stores records in the flights table. You may specify a custom table by defining a table property on your model:

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    protected $table = 'my_flights';
}
```

### Primary Keys

Eloquent will also assume that each table has a primary key column named id. You may define a `$primaryKey` property to override this convention.

In addition, Eloquent assumes that the primary key is an incrementing integer value, which means that by default the primary key will be cast to an int automatically. If you wish to use a non-incrementing or a non-numeric primary key you must set the public `$incrementing` property on your model to false.

### Timestamps

By default, Eloquent expects `created_at` and `updated_at` columns to exist on your tables. If you do not wish to have these columns automatically managed by Eloquent, set the `$timestamps` property on your model to false:

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    public $timestamps = false;
}
```

If you need to customize the format of your timestamps, set the `$dateFormat` property on your model. This property determines how date attributes are stored in the database, as well as their format when the model is serialized to an array or JSON:

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The storage format of the model's date columns.
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

**Database Connection**

By default, all Eloquent models will use the default database connection configured for your application. If you would like to specify a different connection for the model, use the `$connection` property:

```
<?php
namespace App;
```

**Retrieving Multiple Models**

Once you have created a model and its associated database table, you are ready to start retrieving data from your database. Think of each Eloquent model as a powerful query builder allowing you to fluently query the database table associated with the model. For example:

```
<?php
namespace App\Http\Controllers;

use App\Flight;
use App\Http\Controllers\Controller;

class FlightController extends Controller
{
    /**
     * Show a list of all available flights.
     *
     * @return Response
     */
    public function index()
    {
        $flights = Flight::all();

        return view('flight.index', ['flights' => $flights]);
    }
}
```

**Accessing Column Values**

If you have an Eloquent model instance, you may access the column values of the model by accessing the corresponding property. For example, let's loop through each Flight instance returned by our query and echo the value of the name column:

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

**Adding Additional Constraints**

The Eloquent all method will return all of the results in the model's table. Since each Eloquent model serves as a query builder, you may also add constraints to queries, and then use the get method to retrieve the results:

```
$flights = App\Flight::where('active', 1)
    ->orderBy('name', 'desc')
    ->take(10)
    ->get();
```

**Note:** Since Eloquent models are query builders, you should review all of the methods available on the query builder. You may use any of these methods in your Eloquent queries.

**Collections**

For Eloquent methods like all and get which retrieve multiple results, an instance of Illuminate\Database\Eloquent\Collection will be returned. The Collection class provides a variety of helpful methods for working with your Eloquent results. Of course, you may simply loop over this collection like an array:

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

**Chunking Results**

If you need to process thousands of Eloquent records, use the chunk command. The chunk method will retrieve a "chunk" of Eloquent models, feeding them to a given Closure for processing. Using the chunk method will conserve memory when working with large result sets:

```
Flight::chunk(200, function ($flights) {
    foreach ($flights as $flight) {
        //
    }
});
```

The first argument passed to the method is the number of records you wish to receive per "chunk". The Closure passed as the second argument will be called for each chunk that is retrieved from the database.

**Note:** The database query is re-executed for each chunk.

**Retrieving Single Models / Aggregates**

Of course, in addition to retrieving all of the records for a given table, you may also retrieve single records using find and first. Instead of returning a collection of models, these methods return a single model instance:

```
// Retrieve a model by its primary key...
```

```
$flight = App\Flight::find(1);
```



```
// Retrieve the first model matching the query constraints...
```

```
$flight = App\Flight::where('active', 1)->first();
```

You may also call the find method with an array of primary keys, which will return a collection of the matching records:

```
$flights = App\Flight::find([1, 2, 3]);
```

### Not Found Exceptions

Sometimes you may wish to throw an exception if a model is not found. This is particularly useful in routes or controllers. The findOrFail and firstOrFail methods will retrieve the first result of the query. However, if no result is found, an Illuminate\Database\Eloquent\ModelNotFoundException will be thrown:

```
$model = App\Flight::findOrFail(1);
```

```
$model = App\Flight::where('legs', '>', 100)->firstOrFail();
```

If the exception is not caught, a 404 HTTP response is automatically sent back to the user, so it is not necessary to write explicit checks to return 404 responses when using these methods:

```
Route::get('/api/flights/{id}', function ($id) {
    return App\Flight::findOrFail($id);
});
```

### Retrieving Aggregates

Of course, you may also use count, sum, max, and other aggregate functions provided by the query builder. These methods return the appropriate scalar value instead of a full model instance:

```
$count = App\Flight::where('active', 1)->count();
```

```
$max = App\Flight::where('active', 1)->max('price');
```

### Inserting & Updating Models

#### Basic Inserts

To create a new record in the database, simply create a new model instance, set attributes on the model, then call the save method:

```
<?php

namespace App\Http\Controllers;

use App\Flight;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
```

```

class FlightController extends Controller
{
    public function store(Request $request)
    {
        // Validate the request...

        $flight = new Flight;

        $flight->name = $request->name;

        $flight->save();
    }
}

```

In this example, we simply assign the name parameter from the incoming HTTP request to the name attribute of the App\Flight model instance. When we call the save method, a record will be inserted into the database. The created\_at and updated\_at timestamps will automatically be set when the save method is called, so there is no need to set them manually.

### Basic Updates

The save method may also be used to update models that already exist in the database. To update a model, you should retrieve it, set any attributes you wish to update, and then call the save method. Again, the updated\_at timestamp will automatically be updated, so there is no need to manually set its value:

```

$flight = App\Flight::find(1);

$flight->name = 'New Flight Name';

$flight->save();

```

Updates can also be performed against any number of models that match a given query. In this example, all flights that are active and have a destination of San Diego will be marked as delayed:

```

App\Flight::where('active', 1)
    ->where('destination', 'San Diego')
    ->update(['delayed' => 1]);

```

The update method expects an array of column and value pairs representing the columns that should be updated.

### Mass Assignment

You may also use the create method to save a new model in a single line. The inserted model instance will be returned to you from the method. However, before doing so, you will need to specify either a fillable or guarded attribute on the model, as all Eloquent models protect against mass-assignment.

A mass-assignment vulnerability occurs when a user passes an unexpected HTTP parameter through a request, and that parameter changes a column in your database you did not expect. For example, a malicious user might send an `is_admin` parameter through an HTTP request, which is then mapped onto your model's `create` method, allowing the user to escalate themselves to an administrator.

So, to get started, you should define which model attributes you want to make mass assignable. You may do this using the `$fillable` property on the model. For example, let's make the `name` attribute of our `Flight` model mass assignable:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

Once we have made the attributes mass assignable, we can use the `create` method to insert a new record in the database. The `create` method returns the saved model instance:

```
$flight = App\Flight::create(['name' => 'Flight 10']);
```

While `$fillable` serves as a "white list" of attributes that should be mass assignable, you may also choose to use `$guarded`. The `$guarded` property should contain an array of attributes that you do not want to be mass assignable. All other attributes not in the array will be mass assignable. So, `$guarded` functions like a "black list". Of course, you should use either `$fillable` or `$guarded` - not both:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The attributes that aren't mass assignable.
     *
     * @var array
     */
}
```

```

*/
protected $guarded = ['price'];
}

```

In the example above, all attributes except for price will be mass assignable.

### Other Creation Methods

There are two other methods you may use to create models by mass assigning attributes: `firstOrCreate` and `firstOrCreate`. The `firstOrCreate` method will attempt to locate a database record using the given column / value pairs. If the model can not be found in the database, a record will be inserted with the given attributes.

The `firstOrCreate` method, like `firstOrCreate` will attempt to locate a record in the database matching the given attributes. However, if a model is not found, a new model instance will be returned. Note that the model returned by `firstOrCreate` has not yet been persisted to the database. You will need to call `save` manually to persist it:

```
// Retrieve the flight by the attributes, or create it if it doesn't exist...
```

```
$flight = App\Flight::firstOrCreate(['name' => 'Flight 10']);
```

```
// Retrieve the flight by the attributes, or instantiate a new instance...
```

```
$flight = App\Flight::firstOrCreate(['name' => 'Flight 10']);
```

### Deleting Models

To delete a model, call the `delete` method on a model instance:

```
$flight = App\Flight::find(1);
```

```
$flight->delete();
```

### Deleting An Existing Model By Key

In the example above, we are retrieving the model from the database before calling the `delete` method. However, if you know the primary key of the model, you may delete the model without retrieving it. To do so, call the `destroy` method:

```
App\Flight::destroy(1);
```

```
App\Flight::destroy([1, 2, 3]);
```

```
App\Flight::destroy(1, 2, 3);
```

### Deleting Models By Query

Of course, you may also run a delete query on a set of models. In this example, we will delete all flights that are marked as inactive:

```
$deletedRows = App\Flight::where('active', 0)->delete();
```

## Model Relationships

Database tables are often related to one another. For example, a blog post may have many comments, or an order could be related to the user who placed it. Eloquent makes managing and working with these relationships easy, and supports several different types of relationships:

**One To One**  
**One To Many**  
**Many To Many**  
**Has Many Through**  
**Polymorphic Relations**  
**Many To Many Polymorphic Relations**

## Defining Relationships

Eloquent relationships are defined as functions on your Eloquent model classes. Since, like Eloquent models themselves, relationships also serve as powerful query builders, defining relationships as functions provides powerful method chaining and querying capabilities. For example:

```
$user->posts()->where('active', 1)->get();
```

## One To One

A one-to-one relationship is a very basic relation. For example, a User model might be associated with one Phone. To define this relationship, we place a phone method on the User model. The phone method should return the results of the hasOne method on the base Eloquent model class:

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    public function phone()
    {
        return $this->hasOne('App\Phone');
    }
}
```

The first argument passed to the hasOne method is the name of the related model. Once the relationship is defined, we may retrieve the related record using Eloquent's dynamic properties. Dynamic properties allow you to access relationship functions as if they were properties defined on the model:

```
$phone = User::find(1)->phone;
```

Eloquent assumes the foreign key of the relationship based on the model name. In this case, the Phone model is automatically assumed to have a user\_id foreign key. If you wish to override this convention, you may pass a second argument to the hasOne method:

```
return $this->hasOne('App\Phone', 'foreign_key');
```

Additionally, Eloquent assumes that the foreign key should have a value matching the id (or the custom \$primaryKey) column of the parent. In other words, Eloquent will look for the value of the user's id column in the user\_id column of the Phone record. If you would like the relationship to use a value other than id, you may pass a third argument to the hasOne method specifying your custom key:

```
return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

### One To Many

A "one-to-many" relationship is used to define relationships where a single model owns any amount of other models. For example, a blog post may have an infinite number of comments. Like all other Eloquent relationships, one-to-many relationships are defined by placing a function on your Eloquent model:

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

Remember, Eloquent will automatically determine the proper foreign key column on the Comment model. By convention, Eloquent will take the "snake case" name of the owning model and suffix it with \_id. So, for this example, Eloquent will assume the foreign key on the Comment model is post\_id.

Once the relationship has been defined, we can access the collection of comments by accessing the comments property. Remember, since Eloquent provides "dynamic properties", we can access relationship functions as if they were defined as properties on the model:

```
$comments = App\Post::find(1)->comments;

foreach ($comments as $comment) {
    //
}
```

Of course, since all relationships also serve as query builders, you can add further constraints to which comments are retrieved by calling the comments method and continuing to chain conditions onto the query:

```
$comments = App\Post::find(1)->comments()->where('title', 'foo')->first();
```

Like the hasOne method, you may also override the foreign and local keys by passing additional arguments to the hasMany method:

```
return $this->hasMany('App\Comment', 'foreign_key');

return $this->hasMany('App\Comment', 'foreign_key', 'local_key');
```

### Many To Many

Many-to-many relations are slightly more complicated than hasOne and hasMany relationships. An example of such a relationship is a user with many roles, where the roles are also shared by other users. For example, many users may have the role of "Admin". To define this relationship, three database tables are needed: users, roles, and role\_user. The role\_user table is derived from the alphabetical order of the related model names, and contains the user\_id and role\_id columns.

Many-to-many relationships are defined by writing a method that calls the belongsToMany method on the base Eloquent class. For example, let's define the roles method on our User model:

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The roles that belong to the user.
     */
    public function roles()
    {
        return $this->belongsToMany('App\Role');
    }
}
```

Once the relationship is defined, you may access the user's roles using the roles dynamic property:

```
$user = App\User::find(1);

foreach ($user->roles as $role) {
    //
}
```



Of course, like all other relationship types, you may call the roles method to continue chaining query constraints onto the relationship:

```
$roles = App\User::find(1)->roles()->orderBy('name')->get();
```

As mentioned previously, to determine the table name of the relationship's joining table, Eloquent will join the two related model names in alphabetical order. However, you are free to override this convention. You may do so by passing a second argument to the belongsToMany method:

```
return $this->belongsToMany('App\Role', 'role_user');
```

In addition to customizing the name of the joining table, you may also customize the column names of the keys on the table by passing additional arguments to the belongsToMany method. The third argument is the foreign key name of the model on which you are defining the relationship, while the fourth argument is the foreign key name of the model that you are joining to:

```
return $this->belongsToMany('App\Role', 'role_user', 'user_id', 'role_id');
```

### Collections

All multi-result sets returned by Eloquent are instances of the Illuminate\Database\Eloquent\Collection object, including results retrieved via the get method or accessed via a relationship. The Eloquent collection object extends the Laravel base collection, so it naturally inherits dozens of methods used to fluently work with the underlying array of Eloquent models.

Of course, all collections also serve as iterators, allowing you to loop over them as if they were simple PHP arrays:

```
$users = App\User::where('active', 1)->get();

foreach ($users as $user) {
    echo $user->name;
}
```

However, collections are much more powerful than arrays and expose a variety of map / reduce operations that may be chained using an intuitive interface. For example, let's remove all inactive models and gather the first name for each remaining user:

```
$users = App\User::where('active', 1)->get();

$names = $users->reject(function ($user) {
    return $user->active === false;
})
->map(function ($user) {
    return $user->name;
});
```

### The Base Collection

All Eloquent collections extend the base Laravel collection object; therefore, they inherit all of the powerful methods provided by the base collection class:

**all, average, avg, chunk, collapse, combine, contains, count, diff, diffKeys, each, every, except, filter, first, flatMap, flatten, flip, forget, forPage, get, groupBy, has, implode, intersect, isEmpty, keyBy, keys, last, map, max, median, merge, min, mode, only, pipe, pluck, pop, prepend, pull, push, put, random, reduce, reject, reverse, search, shift, shuffle, slice, sort, sortBy, sortByDesc, splice, sum, take, toArray, toJson, transform, union, unique, values, where, whereLoose, whereIn, whereInLoose, zip,**

### Defining A Mutator

To define a mutator, define a setFooAttribute method on your model where Foo is the "studly" cased name of the column you wish to access. So, again, let's define a mutator for the first\_name attribute. This mutator will be automatically called when we attempt to set the value of the first\_name attribute on the model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model{
    public function setFirstNameAttribute($value) {
        $this->attributes['first_name'] = strtolower($value);
    }
}
```

The mutator will receive the value that is being set on the attribute, allowing you to manipulate the value and set the manipulated value on the Eloquent model's internal \$attributes property. So, for example, if we attempt to set the first\_name attribute to Sally:

```
$user = App\User::find(1);

$user->first_name = 'Sally';
```

In this example, the setFirstNameAttribute function will be called with the value Sally. The mutator will then apply the strtolower function to the name and set its value in the internal \$attributes array.

## API Resources Introduction

When building an API, you may need a transformation layer that sits between your Eloquent models and the JSON responses that are actually returned to your application's users. Laravel's resource classes allow you to expressively and easily transform your models and model collections into JSON.

## Generating Resources

To generate a resource class, you may use the `make:resource` Artisan command. By default, resources will be placed in the `app/Http/Resources` directory of your application. Resources extend the `Illuminate\Http\Resources\Json\JsonResource` class:

```
php artisan make:resource User
```

## Resource Collections

In addition to generating resources that transform individual models, you may generate resources that are responsible for transforming collections of models. This allows your response to include links and other meta information that is relevant to an entire collection of a given resource.

To create a resource collection, you should use the `--collection` flag when creating the resource. Or, including the word `Collection` in the resource name will indicate to Laravel that it should create a collection resource. Collection resources extend the `Illuminate\Http\Resources\Json\ResourceCollection` class:

```
php artisan make:resource Users --collection
```

```
php artisan make:resource UserCollection
```

## Writing Resources

In essence, resources are simple. They only need to transform a given model into an array. So, each resource contains a `toArray` method which translates your model's attributes into an API friendly array that can be returned to your users:

```
<?php
namespace App\Http\Resources;

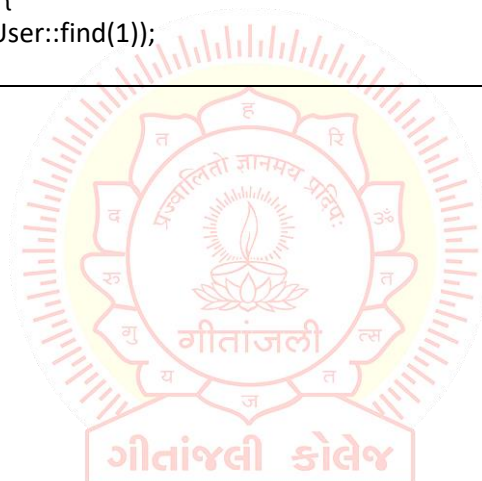
use Illuminate\Http\Resources\Json\JsonResource;

class User extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @param \Illuminate\Http\Request $request
     * @return array
     */
}
```

```
*/  
public function toArray($request)  
{  
    return [  
        'id' => $this->id,  
        'name' => $this->name,  
        'email' => $this->email,  
        'created_at' => $this->created_at,  
        'updated_at' => $this->updated_at,  
    ];  
}  
}
```

Once a resource has been defined, it may be returned directly from a route or controller:

```
use App\Http\Resources\User as UserResource;  
use App\User;  
  
Route::get('/user', function () {  
    return new UserResource(User::find(1));  
});
```



## Introduction

Laravel Passport provides a full OAuth2 server implementation for your Laravel application in a matter of minutes. Passport is built on top of the League OAuth2 server that is maintained by Andy Millington and Simon Hamp.

### Passport Or Sanctum?

Before getting started, you may wish to determine if your application would be better served by Laravel Passport or Laravel Sanctum. If your application absolutely needs to support OAuth2, then you should use Laravel Passport.

However, if you are attempting to authenticate a single-page application, mobile application, or issue API tokens, you should use Laravel Sanctum. Laravel Sanctum does not support OAuth2; however, it provides a much simpler API authentication development experience.

### Installation

To get started, install Passport via the Composer package manager:

```
composer require laravel/passport
```

Passport's service provider registers its own database migration directory, so you should migrate your database after installing the package. The Passport migrations will create the tables your application needs to store OAuth2 clients and access tokens:

```
php artisan migrate
```

Next, you should execute the `passport:install` Artisan command. This command will create the encryption keys needed to generate secure access tokens. In addition, the command will create "personal access" and "password grant" clients which will be used to generate access tokens:

```
php artisan passport:install
```

After running the `passport:install` command, add the `Laravel\Passport\HasApiTokens` trait to your `App\Models\User` model. This trait will provide a few helper methods to your model which allow you to inspect the authenticated user's token and scopes. If your model is already using the `Laravel\Sanctum\HasApiTokens` trait, you may remove that trait:

```
<?php
namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
// use Laravel\Sanctum\HasApiTokens; // Comment this
use Laravel\Passport\HasApiTokens; // Add this

class User extends Authenticatable{
    use HasApiTokens, HasFactory, Notifiable;
}
```

Finally, in your application's config/auth.php configuration file, you should define an api authentication guard and set the driver option to passport. This will instruct your application to use Passport's TokenGuard when authenticating incoming API requests:

```
'guards' => [  
  'web' => [  
    'driver' => 'session',  
    'provider' => 'users',  
  ],  
  
  // Add Below Code  
  
  'api' => [  
    'driver' => 'passport',  
    'provider' => 'users',  
  ],  
],
```

