

GEETANJALI COLLEGE - RAJKOT
B.SC. (IT) SEM – 3 (CMS Using JOOMLA)
Unit-1. OOPS & MVC Strucutre

Class:

- > Class is a programmer-defined data type, which includes local methods and local variables.
- > Class is a collection of objects. Object has properties and behavior.

Example:

```
<?php
class Books
{
    function name()
    {
        echo "Drupal book";
    }
    function price()
    {
        echo "900 Rs/-";
    }
}
//To create php object we have to use a new operator. Here php object is the object of the Books Class.
$obj = new Books();
$obj->name();
$obj->price();
?>
```

Object:

- > An individual instance of the data structure defined by a class. You define a class once and then make many objects that belong to it. Objects are also known as instance.

Example:

```
<?php
class demo
{
    function demo_obj()
    {
        echo "Object Created.";
    }
}
$a = new demo;
$a->demo_obj();
?>
```

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.

Example:

```
<?php
class SimpleClass
{
    // valid as of PHP 5.6.0:
    public $var1 = 'hello ' . 'world';
    // valid as of PHP 5.3.0:
    public $var2 = <<<EOD
hello world
EOD;
    // valid as of PHP 5.6.0:
    public $var3 = 1+2;
    // invalid property declarations:
    public $var4 = self::myStaticMethod;
```

```

public $var5 = $myVar;
// valid property declarations:
public $var6 = myConstant;
public $var7 = array(true, false);
// valid as of PHP 5.3.0:
public $var8 = <<<'EOD'
hello world
EOD;
}
?>

```

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*.

Example:

```

<?php
class MyClass {
    const CONSTANT = 'constant value';
    function showConstant() {
        echo self::CONSTANT . "\n";    }    }
echo MyClass::CONSTANT . "\n";
$classname = "MyClass";
echo $classname::CONSTANT . "\n"; // As of PHP 5.3.0
$class = new MyClass();
$class->showConstant();
echo $class::CONSTANT . "\n"; // As of PHP 5.3.0
?>

```

Namespace:

-> A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.

Example:

```

<?php
namespace my\name; // see "Defining Namespaces" section
class MyClass {}
function myfunction() {}
const MYCONST = 1;
$a = new MyClass;
$c = new \my\name\MyClass; // see "Global Space" section
$a = strlen('hi'); // see "Using namespaces: fallback to global
// function/constant" section
$d = namespace\MYCONST; // see "namespace operator and __NAMESPACE__
// constant" section
$d = __NAMESPACE__ . '\MYCONST';
echo constant($d); // see "Namespaces and dynamic language features" section
?>

```

Constructor:

-> It is refer to a special type of function which will be called automatically whenever there is an object formation from a class.

-> Constructor Functions are special type of functions which are called automatically whenever an object is created. So we take full advantage of this behaviour, by initializing many things through constructor functions.

-> PHP provides a special function called `__construct()` to define a constructor. You can pass as many as arguments you like into the constructor function.

Example:

```

<?php
class BaseClass {
    function __construct() {
        print "In BaseClass constructor\n";
    }
}

```

```

class SubClass extends BaseClass {
    function __construct() {
        parent::__construct();
        print "In SubClass 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();

```

?>

Destructor:

- > It is refer to a special type of function which will be called automatically whenever an object is deleted or goes out of scope.
- > 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.

Example:

```

<?php
class MyDestructableClass
{
    function __construct() {
        print "In constructor\n";
    }

    function __destruct() {
        print "Destroying " . __CLASS__ . "\n";
    }
}
$obj = new MyDestructableClass();

```

Visibility:

- > The visibility of a property, a method or 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.

Example:

```

<?php
/**
 * Define MyClass
 */
class MyClass
{
    public $public = 'Public';
    protected $protected = 'Protected';
    private $private = 'Private';
    function printHello()
    {
        echo $this->public;
        echo $this->protected;
        echo $this->private;
    }
}

```

```

$obj = new MyClass();
echo $obj->public; // Works
echo $obj->protected; // Fatal Error
echo $obj->private; // Fatal Error
$obj->printHello(); // Shows Public, Protected and Private
/**
 * Define MyClass2
 */
class MyClass2 extends MyClass
{
    // We can redeclare the public and protected properties, but not private
    public $public = 'Public2';
    protected $protected = 'Protected2';
    function printHello()
    {
        echo $this->public;
        echo $this->protected;
        echo $this->private;
    }
}
$obj2 = new MyClass2();
echo $obj2->public; // Works
echo $obj2->protected; // Fatal Error
echo $obj2->private; // Undefined
$obj2->printHello(); // Shows Public2, Protected2, Undefined

```

?>

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 reimplement all of the shared functionality.

Example:

```

<?php
class Foo
{
    public function printItem($string)
    {
        echo 'Foo: ' . $string . PHP_EOL;
    }
    public function printPHP()
    {
        echo 'PHP is great.' . PHP_EOL;
    }
}
class Bar extends Foo
{
    public function printItem($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 Paamayim Nekudotayim) 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.

Example:

-> from outside the class definition:

```
<?php
    class MyClass {
        const CONST_VALUE = 'A constant value';
    }
    $classname = 'MyClass';
    echo $classname::CONST_VALUE; // As of PHP 5.3.0
    echo MyClass::CONST_VALUE;

?>
```

-> from inside the class definition:

```
<?php
    class OtherClass extends MyClass
    {
        public static $my_static = 'static var';

        public static function doubleColon() {
            echo parent::CONST_VALUE . "\n";
            echo self::$my_static . "\n";
        }
    }
    $classname = 'OtherClass';
    $classname::doubleColon(); // As of PHP 5.3.0
    OtherClass::doubleColon();

?>
```

-> Calling a parent's method:

```
<?php
    class MyClass
    {
        protected function myFunc() {
            echo "MyClass::myFunc()\n";
        }
    }
    class OtherClass extends MyClass
    {
        // Override parent's definition
        public function myFunc()
        {
            // But still call the parent function
            parent::myFunc();
            echo "OtherClass::myFunc()\n";
        }
    }
    $class = new OtherClass();
    $class->myFunc();

?>
```

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).

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.

Example:

```
<?php
    class Foo{
        public static function aStaticMethod() {
            // ...
        }
    }
```

```

Foo::aStaticMethod();
$classname = 'Foo';
$classname::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.

Example:

```

<?php
class Foo
{
    public static $my_static = 'foo';
    public function staticValue() {
        return self::$my_static;
    }
}
class Bar extends Foo
{
    public function fooStatic() {
        return parent::$my_static;
    }
}
print Foo::$my_static . "\n";
$foo = new Foo();
print $foo->staticValue() . "\n";
print $foo->my_static . "\n"; // Undefined "Property" my_static
print $foo::$my_static . "\n";
$classname = 'Foo';
print $classname::$my_static . "\n"; // As of PHP 5.3.0
print Bar::$my_static . "\n";
$bar = new Bar();
print $bar->fooStatic() . "\n";

```

```
?>
```

Class Abstraction:

-> 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.

Example:

```

<?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() . "\n";
    }
}
class ConcreteClass1 extends AbstractClass
{
    protected function getValue() {
        return "ConcreteClass1";
    }
}

```

```

    public function prefixValue($prefix) {
        return "{$prefix}ConcreteClass1";
    }
}
class ConcreteClass2 extends AbstractClass
{
    public function getValue() {
        return "ConcreteClass2";
    }
    public function prefixValue($prefix) {
        return "{$prefix}ConcreteClass2";
    }
}
$class1 = new ConcreteClass1;
$class1->printOut();
echo $class1->prefixValue('FOO_') . "\n";
$class2 = new ConcreteClass2;
$class2->printOut();
echo $class2->prefixValue('FOO_') . "\n";
?>

```

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.

Example:

```

<?php
// Declare the interface 'iTemplate'
interface iTemplate
{
    public function setVariable($name, $var);
    public function getHtml($template);
}
// Implement the interface
// This will work
class Template implements iTemplate
{
    private $vars = array();
    public function setVariable($name, $var)
    {
        $this->vars[$name] = $var;
    }
    public function getHtml($template)
    {
        foreach($this->vars as $name => $value) {
            $template = str_replace('{ ' . $name . ' }', $value, $template);
        }
        return $template;
    }
}
// This will not work
// Fatal error: Class BadTemplate contains 1 abstract methods
// and must therefore be declared abstract (iTemplate::getHtml)
class BadTemplate implements iTemplate
{
    private $vars = array();
    public function setVariable($name, $var)
    {
        $this->vars[$name] = $var;
    }
}
?>

```

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.

Example:

```
<?php
// Declare the interface 'iTemplate'
interface iTemplate
{
    public function setVariable($name, $var);
    public function getHtml($template);
}
// Implement the interface
// This will work
class Template implements iTemplate
{
    private $vars = array();
    public function setVariable($name, $var)
    {
        $this->vars[$name] = $var;
    }
    public function getHtml($template)
    {
        foreach($this->vars as $name => $value) {
            $template = str_replace('{ ' . $name . ' }', $value, $template);
        }
        return $template;
    }
}
// This will not work
// Fatal error: Class BadTemplate contains 1 abstract methods
// and must therefore be declared abstract (iTemplate::getHtml)
class BadTemplate implements iTemplate
{
    private $vars = array();
    public function setVariable($name, $var)
    {
        $this->vars[$name] = $var;
    }
}
?>
```

Extendable Interfaces:

```
<?php
interface a
{
    public function foo();
}
interface b extends a
{
    public function baz(Baz $baz);
}
// This will work
class c implements b
{
    public function foo()
    {
    }
    public function baz(Baz $baz)
    {
    }
}
// This will not work and result in a fatal error
```



```

class d implements b
{
    public function foo()
    {
    }
    public function baz(Foo $foo)
    {
    }
}
?>

```

Traits:

-> Traits are a mechanism for code reuse in single inheritance languages such as PHP. A Trait is intended to reduce some limitations of single inheritance by enabling a developer to reuse sets of methods freely in several independent classes living in different class hierarchies. The semantics of the combination of Traits and classes is defined in a way which reduces complexity, and avoids the typical problems associated with multiple inheritance and Mixins.

-> A Trait is similar to a class, but only intended to group functionality in a fine-grained and consistent way. It is not possible to instantiate a Trait on its own. It is an addition to traditional inheritance and enables horizontal composition of behavior; that is, the application of class members without requiring inheritance.

Example:

```

<?php
trait ezReflectionReturnInfo {
    function getReturnType() { /*1*/ }
    function getReturnDescription() { /*2*/ }
}

class ezReflectionMethod extends ReflectionMethod {
    use ezReflectionReturnInfo;
    /* ... */
}

class ezReflectionFunction extends ReflectionFunction {
    use ezReflectionReturnInfo;
    /* ... */
}
?>

```

Anonymous Classes:

-> Anonymous classes can now be defined using new class. Anonymous class can be used in place of a full class definition.

Example:

```

<?php
interface Logger {
    public function log(string $msg);
}

class Application {
    private $logger;

    public function getLogger(): Logger {
        return $this->logger;
    }
    public function setLogger(Logger $logger) {
        $this->logger = $logger;
    }
}

$app = new Application;
$app->setLogger(new class implements Logger {
    public function log(string $msg) {
        print($msg);
    }
});

$app->getLogger()->log("My first Log Message");
?>

```

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.

-> All overloading methods must be defined as *public*.

Example:

```
<?php
class PropertyTest
{
    /** Location for overloaded data. */
    private $data = array();
    /** Overloading not used on declared properties. */
    public $declared = 1;
    /** Overloading only used on this when accessed outside the class. */
    private $hidden = 2;
    public function __set($name, $value)
    {
        echo "Setting '$name' to '$value'\n";
        $this->data[$name] = $value;
    }
    public function __get($name)
    {
        echo "Getting '$name'\n";
        if (array_key_exists($name, $this->data)) {
            return $this->data[$name];
        }
        $trace = debug_backtrace();
        trigger_error(
            'Undefined property via __get(): ' . $name .
            ' in ' . $trace[0]['file'] .
            ' on line ' . $trace[0]['line'],
            E_USER_NOTICE);
        return null;
    }
    /** As of PHP 5.1.0 */
    public function __isset($name)
    {
        echo "Is '$name' set?\n";
        return isset($this->data[$name]);
    }
    /** As of PHP 5.1.0 */
    public function __unset($name)
    {
        echo "Unsetting '$name'\n";
        unset($this->data[$name]);
    }
    /** Not a magic method, just here for example. */
    public function getHidden()
    {
        return $this->hidden;
    }
}
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));
echo "\n";
```

```

echo $obj->declared . "\n\n";
echo "Let's experiment with the private property named 'hidden':\n";
echo "Privates are visible inside the class, so __get() not used...\n";
echo $obj->getHidden() . "\n";
echo "Privates not visible outside of class, so __get() is used...\n";
echo $obj->hidden . "\n";
?>

```

Object Iteration:

-> It is provide 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.

Example:

```

<?php
class MyClass
{
    public $var1 = 'value 1';
    public $var2 = 'value 2';
    public $var3 = 'value 3';
    protected $protected = 'protected var';
    private $private = 'private var';
    function iterateVisible() {
        echo "MyClass::iterateVisible:\n";
        foreach ($this as $key => $value) {
            print "$key => $value\n";
        }
    }
}
$class = new MyClass();
foreach($class as $key => $value) {
    print "$key => $value\n";
}
echo "\n";
$class->iterateVisible();
?>

```

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.

sleep() and wakeup() :

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.

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

Example:

```

<?php
class Connection
{
    protected $link;
    private $dsn, $username, $password;
    public function __construct($dsn, $username, $password)
    {
        $this->dsn = $dsn;
        $this->username = $username;
        $this->password = $password;
        $this->connect();
    }
}

```

```

    }
    private function connect()
    {
        $this->link = new PDO($this->dsn, $this->username, $this->password);
    }
    public function __sleep()
    {
        return array('dsn', 'username', 'password');
    }
    public function __wakeup()
    {
        $this->connect();
    }
}??>

```

toString():

-> 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.

Example:

```

<?php
// Declare a simple class
class TestClass
{
    public $foo;
    public function __construct($foo)
    {
        $this->foo = $foo;
    }
    public function __toString()
    {
        return $this->foo;
    }
}
$class = new TestClass('Hello');
echo $class;
?>

```

Final Keyword:

-> It 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.

This keyword is used as a modifier, but only applicable for PHP classes and functions.

If the *final* modifier is applied to PHP classes, then, the class cannot be inherited by some other class. At the same time, if this keyword is applied for a function declaration, then, this function cannot be overridden.

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()
?>
```

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;
```

Example:

```
<?php
class SubObject
{
    static $instances = 0;
    public $instance;
    public function __construct() {
        $this->instance = ++self::$instances;
    }
    public function __clone() {
        $this->instance = ++self::$instances;
    }
}
class MyCloneable
{
    public $object1;
    public $object2;
    function __clone()
    {
        // Force a copy of this->object, otherwise
        // it will point to same object.
        $this->object1 = clone $this->object1;
    }
}
$obj = new MyCloneable();
$obj->object1 = new SubObject();
$obj->object2 = new SubObject();
$obj2 = clone $obj;
print("Original Object:\n");
print_r($obj);
print("Cloned Object:\n");
print_r($obj2);
?>
```

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.

An example will clarify these rules.

Example:

```
<?php
function bool2str($bool)
{
    if ($bool === false) {
        return 'FALSE';
    } else {
        return 'TRUE';
    }
}
```

```

function compareObjects(&$o1, &$o2)
{
    echo 'o1 == o2 : ' . bool2str($o1 == $o2) . "\n";
    echo 'o1 != o2 : ' . bool2str($o1 != $o2) . "\n";
    echo 'o1 === o2 : ' . bool2str($o1 === $o2) . "\n";
    echo 'o1 !== o2 : ' . bool2str($o1 !== $o2) . "\n";
}
class Flag
{
    public $flag;

    function __construct($flag = true) {
        $this->flag = $flag;
    }
}
class OtherFlag
{
    public $flag;

    function __construct($flag = true) {
        $this->flag = $flag;
    }
}
$o = new Flag();
$p = new Flag();
$q = $o;
$r = new OtherFlag();
echo "Two instances of the same class\n";
compareObjects($o, $p);
echo "\nTwo references to the same instance\n";
compareObjects($o, $q);
echo "\nInstances of two different classes\n";
compareObjects($o, $r);
?>

```

Type Hinting:

-> With Type hinting we can specify the expected data type (arrays, objects, interface, etc.) for an argument in a function declaration. This practice can be most advantageous because it results in better code organization and improved error messages.

This tutorial will start by explaining the subject of type hinting for arrays and objects which is supported in both PHP5 as well as PHP7.

It will also explain the subject of type hinting for basic data types (integers, floats, strings, and booleans) which is only supported in PHP7.

Example:

```

<?php
define('TYPEHINT_PCRE','/^Argument (\d)+ passed to (?:(\w+):)?(\w+)\(\\) must be an instance
of (\w+), (\w+) given/');
class Typehint
{
    private static $Typehints = array(
        'boolean' => 'is_bool',
        'integer' => 'is_int',
        'float' => 'is_float',
        'string' => 'is_string',
        'resrouce' => 'is_resource'
    );
    private function __Constrct() {}
    public static function initializeHandler()
    {
        set_error_handler('Typehint::handleTypehint');
        return TRUE;
    }
    private static function getTypehintedArgument($ThBackTrace, $ThFunction, $ThArgIndex,
    &$ThArgValue)

```

```

{
    foreach ($ThBackTrace as $ThTrace)
    {
        // Match the function; Note we could do more defensive error checking.
        if (isset($ThTrace['function']) && $ThTrace['function'] == $ThFunction)
        {
            $ThArgValue = $ThTrace['args'][$ThArgIndex - 1];
            return TRUE;
        }
    }
    return FALSE;
}
public static function handleTypehint($ErrLevel, $ErrorMessage)
{
    if ($ErrLevel == E_RECOVERABLE_ERROR)
    {
        if (preg_match(TYPEHINT_PCRE, $ErrorMessage, $ErrMatches))
        {
            list($ErrMatch, $ThArgIndex, $ThClass, $ThFunction, $ThHint, $ThType) = $ErrMatches;
            if (isset(self::$Typehints[$ThHint]))
            {
                $ThBacktrace = debug_backtrace();
                $ThArgValue = NULL;
                if
                (self::getTypehintedArgument($ThBacktrace, $ThFunction, $ThArgIndex, $ThArgValue))
                {
                    if (call_user_func(self::$Typehints[$ThHint], $ThArgValue))
                    {
                        return TRUE;
                    }
                }
            }
        }
    }
    return FALSE;
}
}
Typehint::initializeHandler();
?>

```

Late Static Bindings:

-> late static bindings work by storing the class named in the last "non-forwarding call". In case of static method calls, this is the class explicitly named (usually the one on the left of the `::` operator); in case of non static method calls, it is the class of the object. A "forwarding call" is a static one that is introduced by `self::`, `parent::`, `static::`, or, if going up in the class hierarchy, `forward_static_call()`. The function `get_called_class()` can be used to retrieve a string with the name of the called class and `static::` introduces its scope.

-> This feature was named "late static bindings" with an internal perspective in mind. "Late binding" comes from the fact that `static::` will not be resolved using the class where the method is defined but it will rather be computed using runtime information. It was also called a "static binding" as it can be used for (but is not limited to) static method calls.

Late Static Bindings' usage:

-> Late static bindings tries to solve that limitation by introducing a keyword that references the class that was initially called at runtime. Basically, a keyword that would allow you to reference `B` from `test()` in the previous example. It was decided not to introduce a new keyword but rather use `static` that was already reserved.

Example:

```

<?php
class A {
    public static function who() {
        echo __CLASS__;
    }
    public static function test() {
        static::who(); // Here comes Late Static Bindings
    }
}

```

```

}
class B extends A {
    public static function who() {
        echo __CLASS__;
    }
}
B::test();
?>

```

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 accessors 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.

Example:

```

<?php
class A {
    public $foo = 1;
}
$a = new A;
$b = $a;    // $a and $b are copies of the same identifier
            // ($a) = ($b) = <id>
$b->foo = 2;
echo $a->foo."\n";
$c = new A;
$d = &$c;    // $c and $d are references
            // ($c,$d) = <id>
$d->foo = 2;
echo $c->foo."\n";
$e = new A;
function foo($obj) {
    // ($obj) = ($e) = <id>
    $obj->foo = 2;
}
foo($e);
echo $e->foo."\n";
?>

```

Object Serialization:

-> serialize() returns a string containing a byte-stream representation of any value that can be stored in PHP. unserialize() can use this string to recreate the original variable values. Using serialize to save an object will save all variables in an object. The methods in an object will not be saved, only the name of the class.

-> In order to be able to unserialize() an object, the class of that object needs to be defined. That is, if you have an object of class A and serialize this, you'll get a string that refers to class A and contains all values of variables contained in it. If you want to be able to unserialize this in another file, an object of class A, the definition of class A must be present in that file first. This can be done for example by storing the class definition of class A in an include file and including this file or making use of the spl_autoload_register() function.

Example:

```

<?php
// classa.inc:
class A {
    public $one = 1;
    public function show_one() {
        echo $this->one;
    }
}

// page1.php:

```



```

include("classa.inc");
$a = new A;
$s = serialize($a);
// store $s somewhere where page2.php can find it.
file_put_contents('store', $s);
// page2.php:
// this is needed for the unserialize to work properly.
include("classa.inc");
$s = file_get_contents('store');
$a = unserialize($s);
// now use the function show_one() of the $a object.
$a->show_one();
?>

```

-> If an application is using sessions and uses session_register() to register objects, these objects are serialized automatically at the end of each PHP page, and are unserialized automatically on each of the following pages. This means that these objects can show up on any of the application's pages once they become part of the session. However, the session_register() is removed since PHP 5.4.0.

-> It is strongly recommended that if an application serializes objects, for use later in the application, that the application includes the class definition for that object throughout the application. Not doing so might result in an object being unserialized without a class definition, which will result in PHP giving the object a class of `__PHP_Incomplete_Class_Name`, which has no methods and would render the object useless.

-> So if in the example above `$a` became part of a session by running `session_register("a")`, you should include the file `classa.inc` on all of your pages, not only `page1.php` and `page2.php`.

Unit-2. Introduction About Joomla!

Joomla! is an award-winning content management system (CMS), which enables you to build Web sites and powerful online applications. Many aspects, including its ease-of-use and extensibility, have made Joomla! the most popular Web site software available. Best of all, Joomla! is an open source solution that is freely available to everyone.

What's a content management system (CMS)?

A content management system is software that keeps track of every piece of content on your Web site, much like your local public library keeps track of books and stores them. Content can be simple text, photos, music, video, documents, or just about anything you can think of. A major advantage of using a CMS is that it requires almost no technical skill or knowledge to manage. Since the CMS manages all your content, you don't have to.

- Corporate Web sites or portals
- Corporate intranets and extranets
- Online magazines, newspapers, and publications
- E-commerce and online reservations
- Government applications
- Small business Web sites
- Non-profit and organizational Web sites
- Community-based portals
- School and church Web sites
- Personal or family homepages

Who uses Joomla?

Millions of web sites are powered with Joomla.

Discover examples of companies using Joomla! in the official [Joomla! Showcase Directory](#).

I need to build a site for a client. How will Joomla! help me?

Joomla! is designed to be easy to install and set up even if you're not an advanced user. Many Web hosting services offer a single-click install, getting your new site up and running in just a few minutes. Since Joomla! is so easy to use, as a Web designer or developer, you can quickly build sites for your clients. Then, with a minimal amount of instruction, you can empower your clients to easily manage their own sites themselves.

If your clients need specialized functionality, Joomla! is highly extensible and thousands of extensions (most for free under the [GPL license](#)) are available in the [Joomla! Extensions Directory](#).

How can I be sure there will be Joomla! support in the future?

Joomla! is the most popular open source CMS currently available as evidenced by a vibrant and growing community of friendly users and talented developers. Joomla's roots go back to 2000 and, with over 200,000 community users and contributors, the future looks bright for the award-winning Joomla! Project.

I'm a developer. What are some advanced ways I can use Joomla?

Many companies and organizations have requirements that go beyond what is available in the basic Joomla! package. In those cases, Joomla's powerful application framework makes it easy for developers to create sophisticated add-ons that extend the power of Joomla into virtually unlimited directions.

The core [Joomla! Framework](#) enables developers to quickly and easily build:

- Inventory control systems
- Data reporting tools
- Application bridges
- Custom product catalogs
- Integrated e-commerce systems
- Complex business directories
- Reservation systems
- Communication tools

-> Since Joomla! is based on PHP and MySQL, you're building powerful applications on an open platform anyone can use, share, and support. To find out more information on leveraging the Joomla! Framework, visit the [Joomla! Developer Network](#).

-> Joomla! is one of the world's most popular software packages used to build, organize, manage and publish content for websites, blogs, Intranets and mobile applications. Owing to its scalable MVC architecture its also a great base to build web applications.

-> With more than 3 percent of the Web running on Joomla! and a CMS market share of more than 9 percent, Joomla! powers the web presence of hundreds of thousands of small businesses, governments, non-profits and large organizations worldwide. As an award winning CMS led by an international community of more than a half million active contributors, helping the most inexperienced user to seasoned web developer make their digital visions a reality.

Joomla! Core Features

User Management:

- > Joomla has a registration system that allows users to configure personal options. There are nine user groups with various types of permissions on what users are allowed to access, edit, publish and administrate.
- > Authentication is an important part of user management and Joomla support multiple protocols, including LDAP and even Gmail. This allows users to use their existing account information to streamline the registration process.
- > Granular access control allows you to have complete control over who can see and do what.

Media Manager:

- > The Media Manager is the tool for easily managing media files or folders and you can configure the MIME type settings to handle any type of file. The Media Manager is integrated into the Article Editor tool so you can grab images and other files at any time. Language Manager.
- > There is international support for many world languages and UTF-8 encoding. If you need your Web site in one language and the administrator panel in another, multiple languages are possible.

Banner Management:

- > It's easy to set up banners on your Web site using the Banner Manager, starting with creating a client profile. Once you add campaigns and as many banners as you need, you can set impression numbers, special URLs, and more.

Contact Management:

- > The Contact Manager helps your users to find the right person and their contact information. It also supports multiple contact forms going to specific individuals as well as groups.

Search:

- > Help navigate users to most popular search items and provide the admin with search statistics.

Web Link Management:

- > Providing link resources for site users is simple and you can sort them into categories, even count every click.

Content Management:

- > Joomla's flexible category system of articles makes organizing your content a snap. You can organize your content any way you want and not necessarily how it will be on your Web site. Your users can rate articles or e-mail them to a friend. Administrators can archive content for safekeeping, hiding it from site visitors.
- > On public Web sites, built-in e-mail cloaking protects email addresses from spambots. Creating content is simple with the WYSIWYG editor, giving even novice users the ability to combine text, images in an attractive way. Once you've created your articles, there are a number of pre-installed modules to show the most popular articles, latest new items, newsflashes, related articles, and more.

Syndication and Newsfeed Management:

- > With Joomla, it's easy to syndicate your site content, allowing your users to subscribe to new content in their favorite RSS reader. It's equally easy to integrate RSS feeds from other sources and aggregate them all on your site.
- > The Menu Manager allows you to create as many menus and menu items as you need. You can structure your menu hierarchy (and nested menu items) completely independent of your content structure. Put one menu in multiple places and in any style you want; use rollovers, dropdown, flyouts and just about any other navigation system you can think of. Also automatic breadcrumbs are generated to help navigate your site users.

Template Management:

-> Templates in Joomla are a powerful way to make your site look exactly the way you want and either use a single template for the entire site or a separate template for each site section. The level of visual control goes a step further with powerful template overrides, allowing you to customize each part of your pages.

Integrated Help System:

-> Joomla has a built-in help section to assist users with finding what they need. A glossary explains the terms in plain English, a version checker makes sure you're using the latest version, a system information tool helps you troubleshoot, and, if all else fails, links to a wealth of online resources for additional help and support.

System Features:

-> Speedy page loads are possible with page caching, granular-level module caching, and GZIP page compression. If your system administrator needs to troubleshoot an issue, debugging mode and error reporting are invaluable. The FTP Layer allows file operations (like installing Extensions) without having to make all the folders and files writable, making your site administrator's life easier and increasing the security of your site. Administrators quickly and efficiently communicate with users one-on-one through private messaging or all site users via the mass mailing system.

Web Services:

-> With Web services, you can use Remote Procedure Calls (via HTTP and XML). You can also integrate XML-RPC services with the Blogger and Joomla APIs.

Powerful Extensibility:

-> These are just some of the basic Joomla features and the real power is in the way you customize Joomla. Visit the Joomla Extensions Directory to see thousands of ways to enhance Joomla to suit your needs.

Advantages & Disadvantages of Joomla:

Advantages:

Easy to install:

-> Joomla! Is quite simple to install. It takes only about ten minutes from downloading to having a working script on a server. It is not as easy as Quick.Cms or WordPress, but is still much simpler than Drupal.

Plugins:

-> The script has several thousands of free plugins available at the homepage. WordPress may have even more, but to make it as functional as Joomla!, you have to instal dozen or so plugins to start with.

Support:

-> There is abundance of programmer's tools and tutorials available for users. There's also an extensive discussion board.

Navigation Management:

-> The script has a comprehensive navigation system, that can successfully manage several hierarchies. It allows to easily manage a site even with couple hundred subpages.

Good looking URLs:

-> Links generated by the script are very friendly and make for better SEO positioning.

Updates:

-> When the page design is ready, there will come a time to update the script to a newer version. You can do it from web browser.

Advanced Administration:

-> Administration panel provides many functions that can be intimidating in the beginning. In time, however, you can master most of them to use the full potential of the script.

Disadvantages:

Limited adjustment options:

-> Even though Joomla! Has many modules and templates, it is always missing something for the more advanced users. It's still better than in case of WordPress.

Server resources and efficiency:

-> Modularity and expendability often means bigger demands on server parameters. This certainly is the case. Still, if the website is not too large and there will not be thousands of visitors, there should be no problems, at least not in the beginning.

Paid plugins:

-> Some of plugins and modules for Joomla! are paid, unlike for e.g. WordPress or Drupal. It pays to spend some time to make sure you won't have to buy an addition that is free in some other script.

Plugins compatibility:

-> There may occur some frustrating compatibility issues between some of the plugins. It may turn out that it will be impossible to get some functionalities without some serious work on the PHP code.

First contact:

-> Many users, beginners especially, are terrified by multitude of possibilities and functions. So if the website is to be simple and the user or the client is just beginning, it would be wiser to use Quick.Cms or WordPress.

Technical Requirements**Requirements for Supported Software:****Requirements for Joomla! 3.x**

Software	Recommended	Minimum	More information
PHP ^[1] (Magic Quotes GPC, MB String Overload = off) (Zlib Compression Support, XML Support, INI Parser Support, JSON Support, Mcrypt Support, MB Language = Default)	5.6 or 7.0 +	5.3.10	https://secure.php.net

Supported Databases:

MySQL ^[2] (InnoDB support required)	5.5.3 +	5.1	https://www.mysql.com
SQL Server	10.50.1600.1 +	10.50.1600.1	https://www.microsoft.com/sql
PostgreSQL	9.1 +	8.3.18	https://www.postgresql.org/

Supported Web Servers:

Apache ^[3] (with mod_mysql, mod_xml, and mod_zlib)	2.4 +	2.0	https://www.apache.org
Nginx	1.8 +	1.0	https://www.nginx.com/resources/wiki/
Microsoft IIS ^[6]	7	7	https://www.iis.net

Configuration Options:

-> If installing on a local computer, there are a number of packages that will help you get set up quicker than individual installations:

LAMP (Linux) - Most Linux distributions come with a pre-configured LAMP server.

WAMP (Windows) - For more information, visit [the WampServer homepage](#)

MAMP (Apple OS) - For more information, visit [the MAMP homepage](#)

XAMPP (Multi-platform) - Not for live sites. For more information, visit [the XAMPP homepage](#)

GEETANJALI COLLEGE - RAJKOT

Installation & Configuration

Installing Joomla

Before you can begin using Joomla! you will need a working installation of Joomla! If you want your site to be available on the Internet, make sure that you have an account on a web server. For most people this means signing up with a hosting company and purchasing a domain that will serve as your site's main address.

- **Want to build a free Joomla! website?**

joomla.com is a newest Joomla! service that allows you to start, build and maintain a completely free website on a joomla.com subdomain for an unlimited time. The site building software features all of the Joomla! CMS core functionalities that make building a website easy and flexible.

- **If you want to test Joomla!** and you haven't purchased a domain yet, you can see and experiment with a working installation at demo.joomla.org. As an added bonus, the demo site provides links for creating and hosting a working Joomla! website with a 90 day trial. (Note that the Joomla! Project receives a royalty if you choose paid hosting from that site.)

- **Already have a hosting company?** Joomla! is offered by most hosting companies under "[One Click Installs](#)" (also called Auto Installers) for their customers. The "One Click Install" method offers an "instant" installation of Joomla! which is quick and easy. Follow the instructions your host provides.

- **Use the conventional method of installation.** This requires you to copy the Joomla! zip file to your hosting account, unzip, create a database, and then run the installation. Complete instructions can be found at [Installing Joomla](#). Further information can be found at [Joomla! Installation Resources](#).

- **Install Joomla! on your own computer** (without your site appearing on the Internet), you can install it using the [XAMPP package](#). Install XAMPP and then use the "conventional method" to get your Joomla! test site working.

Requirements

Hosting Requirements

-> Before we start installing Joomla!, there are a couple prerequisites that need to be met to install Joomla! 3.x successfully.

-> These apply whether you have a dedicated server, a shared hosting plan server, or are installing a copy on a local computer for testing or development.

-> You'll need to meet the following requirements below to install and use Joomla!.

Recommended PHP.ini Settings

-> There are some PHP settings that need to be sufficient for Joomla to install. The settings are usually in a "php.ini" or "user.ini".

-> Talk to your host about how to change these settings if it is possible to do so. If working on a localhost e.g. with XAMPP, you should not be restricted by these settings and VPS or dedicated hosting should also not be as restrictive.

The values for PHP.ini below are *suggested values* only.

- memory_limit - Minimum: 64M Recommended: 128M or better
- upload_max_filesize - Minimum: 20M
- post_max_size - Minimum: 20M
- max_execution_time: At Least 120 Recommended: 300

Prepare for Install

You will need to complete two tasks before you can install Joomla! on your server. First, you will need to download the Joomla! package files. Next, you will need to have a database for Joomla! use.

Downloading and Uploading Joomla! Package Files

1. Download the current release of [Joomla! 3.x](#)
2. Move the downloaded Joomla! installation package to the server. Use a FTP Client to transfer the Joomla! 3.x files to your server. There are several available for use, here is a detailed list of FTP Clients. Please make sure you are using a FTP client's official release.
Hint - This can be accomplished by simply moving the downloaded package to your server, then unpacking it. Or you can unpack the files on your local computer, then move the Joomla installation over to your server. Either way, the Joomla installation needs to be unpacked in the root of your site.

The "root" of your site is the public folder where all web page files are stored so that a user can view the site examples include public_html and htdocs. What your Host uses depends on them.

Your Server's "root" Folder

Normally you upload your web files to the root folder. This is typically named "public_html" but other variations include "htdocs" and this depends on what your host has the set up on the server. For Joomla purposes, you can load the files directly into "public_html" or a sub-folder within it.

Warning!

If you unpack the files on your own computer, then copy them to your server, be sure to move only the folders and files contained INSIDE the Joomla! package. If you unpack the folders and files into a folder, for example called, Joomla and then upload that folder, your site will have to be accessed at `yoursitename.com/Joomla` instead of `yoursitename.com`.

Database for Joomla! Installation

1. If you need to create a database, please read "Create a database for use with Joomla!" first or skip to step #2.
2. You will need to note basic database information needed when the actual Joomla! installation is started.
 - o Location of database, localhost? Or a specific host's server such as `dbserver1.yourhost.com`?
 - o The database name
 - o The database user's name
 - o The database user's password

Start Install

Main Configuration

With the above requirements met, a database created and the required Joomla! files in place, you are ready to install Joomla!. Start the Joomla! web installer by opening your favorite browser and browsing to the site's domain name. On host installation you will use `http://www.yoursitename.com`. If you are installing Joomla! locally, you will use `http://localhost/<path to Joomla files>`, and you should see the installation screen.

Joomla! is free software released under the GNU General Public License.

1 Configuration 2 Database 3 Overview

Select Language: English (United States) [Next]

Main Configuration

Site Name *
Enter the name of your Joomla! site.

Admin Email *
Enter an email address. This will be the email address of the Web site Super Administrator.

Description
Enter a description of the overall Web site that is to be used by search engines. Generally, a maximum of 20 words is optimal.

Admin Username *
You may change the default username **admin**.

Admin Password *
Set the password for your Super Administrator account and confirm it in the field below.

Confirm Admin Password *

Site Offline: ☒ No ☐ Yes
Set the site frontend offline when installation is completed. The site can be set online later on through the Global Configuration.

Joomla! will try to identify the Select Language field automatically from your browser's language. You can change this if needed.

Fill in the following information.

- **Site Name:** the name of your website — this can be changed at any point later in the Site Global Configuration page.

- **Description:** enter a description of the website. This is a global fallback meta description used on every page which will be used by search engines. Generally, a maximum of 20 to 25 words is optimal. Again, this can be changed on the Site Global Configuration page at any time. For more on metadata, see Global Metadata Settings and Entering search engine meta-data.
- **Admin Email Address:** the admin email address. Enter a valid email in case you forget your password. This is the email address where you'll receive a link to change the admin password.
- **Admin Username:** Joomla! uses a default "admin" as the username for the Super User. You can leave it as is, change it now (which a good Security measure) or use My Profile in the Administration interface to change it later.
- **Admin Password:** remember that super user has maximum control of the site (frontend & backend), so try to use a difficult password. Use My Profile in the Administration interface to change it later. Confirm the password in the Confirm Admin Password box.
- **Site Offline:** click the Yes or No box. Yes - this means when installation is complete, your Joomla! website will display the 'Site is offline' message when you browse to `yoursitename.com` to view the home page. No - this means the site is live when you browse to `yoursitename.com` to view the home page. You can use the Site Global Configuration in the Administration interface to change the Offline status at any time.

When everything on the first page is completed, click the next button to proceed:

Database Configuration Configuration Settings

You will need to enter the information about the database you will use for Joomla! now. It was suggested to write this information down under "Prepare for Install" tab. You may also read or review Creating a Database for Joomla!.

Joomla!® is free software released under the GNU General Public License.

1 Configuration 2 Database 3 Overview

Database Configuration Previous Next

Database Type * MySQLi
This is probably "MySQLi"

Host Name * localhost
This is usually "localhost"

Username *
Either something as "root" or a username given by the host

Password
For site security using a password for the database account is mandatory

Database Name *
Some hosts allow only a certain DB name per site. Use table prefix in this case for distinct Joomla! sites.

Table Prefix *
Choose a table prefix or use the **randomly generated**. Ideally, three or four characters long, contain only alphanumeric characters, and **MUST** end in an underscore. **Make sure that the prefix chosen is not used by other tables.**

Old Database Process * Backup Remove
Any existing backup tables from former Joomla! installations will be replaced

For simplification, these instructions are a reference to installing with a MySQLidatabase. The instructions on the installation page are self explanatory, but here they are again:

- **Database Type:** MySQLi is the common database used
- **Hostname** Where is your database located? Common is `localhost`, but some hosts use a specific database server such as `dbserver1.yourhost.com`
- **Username:** the username used to connect to the database
- **Password:** the password for the database's username
- **Database Name:** the name of the database

- **Table Prefix:** one is generated automatically, but you can change it. For example, `j0s3_` can be used. Just don't forget to put the underscore character (`_`) at the end of the prefix.
- **Old Database Process:** should the installer backup or delete existing tables during the installation of new tables? Click, Yes or No to select the choice.

All these choices can be edited on the Site Global Configuration page, under Server options after the installation is completed. Note, you will break your installation if you change these settings after installation unless you have a complete copy of the current database being used by the Joomla! installation. Common uses would be to update the username and password of the database or to complete a move of an existing installation to a new host with different parameters.

When all the information has been filled in, click the **next button** to proceed:

Finalise Overview

-> It is now time to finalise the Joomla! installation. The last page of the web browser installation contains all the information about the installation. This includes the options(at the top) for installing sample data and the installation's configurations(at the bottom).

Install Sample Data and Email Configurations

The first options are for automatically installing sample content to the website and emailing the configuration settings.

The screenshot shows the Joomla! installation interface. At the top, it states "Joomla!® is free software released under the GNU General Public License." Below this are three tabs: "1 Configuration", "2 Database", and "3 Overview", with "3 Overview" being the active tab. The "Finalisation" section includes a "Previous" button and an "Install" button. Under "Install Sample Data", there are radio button options: "None", "Blog English (GB) Sample Data", "Brochure English (GB) Sample Data", "Default English (GB) Sample Data" (which is selected), "Learn Joomla English (GB) Sample Data", and "Test English (GB) Sample Data". A note states: "Installing sample data is strongly recommended for beginners. This will install sample content that is included in the Joomla! installation package." The "Overview" section shows an "Email Configuration" section with "No" and "Yes" buttons, where "No" is selected. A note below says: "Send configuration settings to `email@gmail.org` by email after installation."

If you are new to Joomla! it would be beneficial to install some sample data to see how Joomla! works. You can at this time choose to have the configuration settings emailed to you. If the Email Configuration choice is selected, the Email Password choice will appear. The email password is off by default for security. You can choose to have the password included, just click Yes. Time to check the configurations of your install and the environment of the installation.

Configuration Check

Checking Your Configurations

If everything is in order, you will see the install at the top of the overview page. If not, this is the place to check and see what may be causing an issue.

Main Configuration

Site Name	My Website
Description	My test website
Site Offline	No
Admin Email	email@email.org
Admin Username	Administrator
Admin Password	*****

Database Configuration

Database Type	mysql
Host Name	localhost
Username	joomla_test
Password	*****
Database Name	joomla_test
Table Prefix	joomla_
Old Database Process	Backup

Pre-Installation Check

PHP Version >= 5.3.1	Yes
Magic Quotes GPC Off	Yes
Register Globals Off	Yes
Zlib Compression Support	Yes
XML Support	Yes
Database Support: (mysql, pdo, mysqli)	Yes
MB Language is Default	Yes
MB String Overload Off	Yes
INI Parser Support	Yes
JSON Support	Yes
configuration.php Writeable	Yes

Recommended settings:

These settings are recommended for PHP in order to ensure full compatibility with Joomla!. However, Joomla! will still operate if your settings do not quite match the recommended configuration.

Directive	Recommended	Actual
Safe Mode	Off	Off
Display Errors	Off	Off
File Uploads	On	On
Magic Quotes Runtime	Off	Off
Output Buffering	Off	On
Session Auto Start	Off	Off
Native ZIP support	On	On

The section is broken into 4 groups:

- **Main Configuration:** all the website specific information, such as the website name, description, admin username, etc.
- **Database Configuration:** contains the information about the database Joomla! will use.
- **Pre-Installation Check:** these requirements must all be shown as Yes, otherwise you will not be able to install Joomla! With the exception of the PHP Version, the rest are usually controlled in the [php.ini](#). You may need assistance from your host in correcting these settings or checking to see if it is possible to adjust them.
- **Typical PHP settings** that might cause the install to fail and may need adjusting include the following with suggested values: (1) `memory_limit` (64M), (2) `max_execution_time` (300), (3) `post_max_size` (30M), (4) `upload_max_filesize` (30M). For more information, see [PHP configuration file](#) file.
- **Recommended Settings:** these are settings are recommended in your PHP configuration, but will not prevent Joomla! from being installed. You can refer to the above instructions on how they may be changed.

-> If everything is correct and all checks are passed, you may now click the Install button in the top right corner of the Overview page. This will start the actual installation process.

After you click the Install button, you should see a progress bar with additional information of the installation. Once the installation completes, you should see the success page!

Finishing Up

Success and Finishing Up the Installation

-> Congratulations! Joomla! 3 is now installed. If you want to start using Joomla! right way without installing extra languages there is one last step to complete the installation. You must delete the Installation Folder. Click on Remove Installation folder and a success message will appear. Now you can navigate to the Administrator log in by clicking Administrator or go right to your site by clicking Site.

Congratulations! Joomla! is now installed.

PLEASE REMEMBER TO COMPLETELY REMOVE THE INSTALLATION FOLDER.

You will not be able to proceed beyond this point until the installation directory has been removed. This is a security feature of Joomla!.

Remove installation folder

Site

Administrator

Administration Login Details

Email

email@email.org

Username

Administrator

Joomla! in your own language?

Visit the Joomla! Community Site for language packs downloads.

Installing Extra Languages

Joomla! in your own language and/or automatic basic native multilingual site creation

Before removing the installation folder you can install extra languages. If you want to add extra languages to your Joomla! application click the following button.

→ Extra steps: Install languages

Note: you will need Internet access for Joomla! to download and install the new languages.

Some server configurations won't allow Joomla! to install the languages. If this is your case, don't worry, you will be able to install them later using the Joomla! administrator.

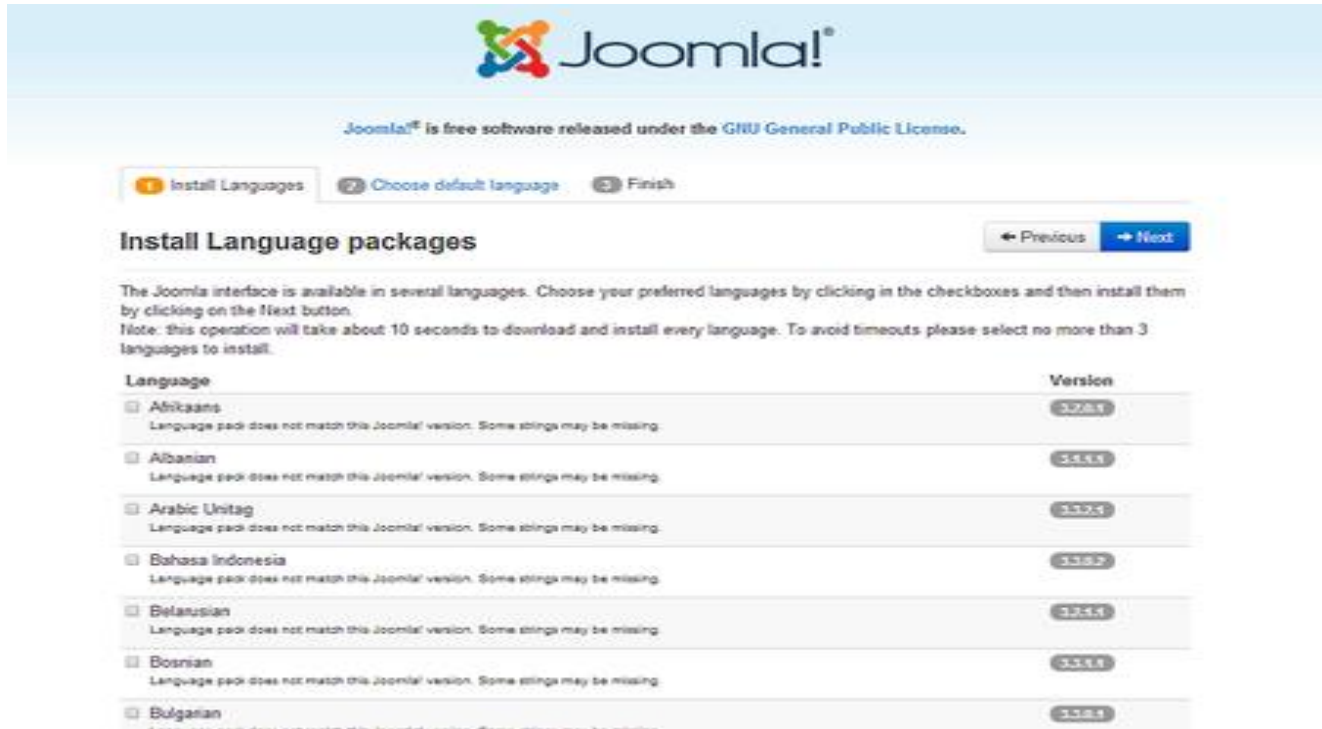
Before you complete your installation by deleting the Installation Folder, click on:

→ Extra steps: Install languages

This will continue the installation of Joomla! by taking you to a new installation page.

Install Languages

A list of language packs is displayed.



Joomla!® is free software released under the GNU General Public License.

1 Install Languages 2 Choose default language 3 Finish

Install Language packages

The Joomla! interface is available in several languages. Choose your preferred languages by clicking in the checkboxes and then install them by clicking on the Next button.
Note: this operation will take about 10 seconds to download and install every language. To avoid timeouts please select no more than 3 languages to install.

Language	Version
<input type="checkbox"/> Afrikaans Language pack does not match this Joomla! version. Some strings may be missing.	3.2.0.1
<input type="checkbox"/> Albanian Language pack does not match this Joomla! version. Some strings may be missing.	3.4.1.1
<input type="checkbox"/> Arabic (United Kingdom) Language pack does not match this Joomla! version. Some strings may be missing.	3.3.3.1
<input type="checkbox"/> Bahasa Indonesia Language pack does not match this Joomla! version. Some strings may be missing.	3.3.0.2
<input type="checkbox"/> Belarusian Language pack does not match this Joomla! version. Some strings may be missing.	3.3.1.1
<input type="checkbox"/> Bosnian Language pack does not match this Joomla! version. Some strings may be missing.	3.3.1.1
<input type="checkbox"/> Bulgarian Language pack does not match this Joomla! version. Some strings may be missing.	3.3.1.1

Check the language or language packs you wish to install. Remember the following:

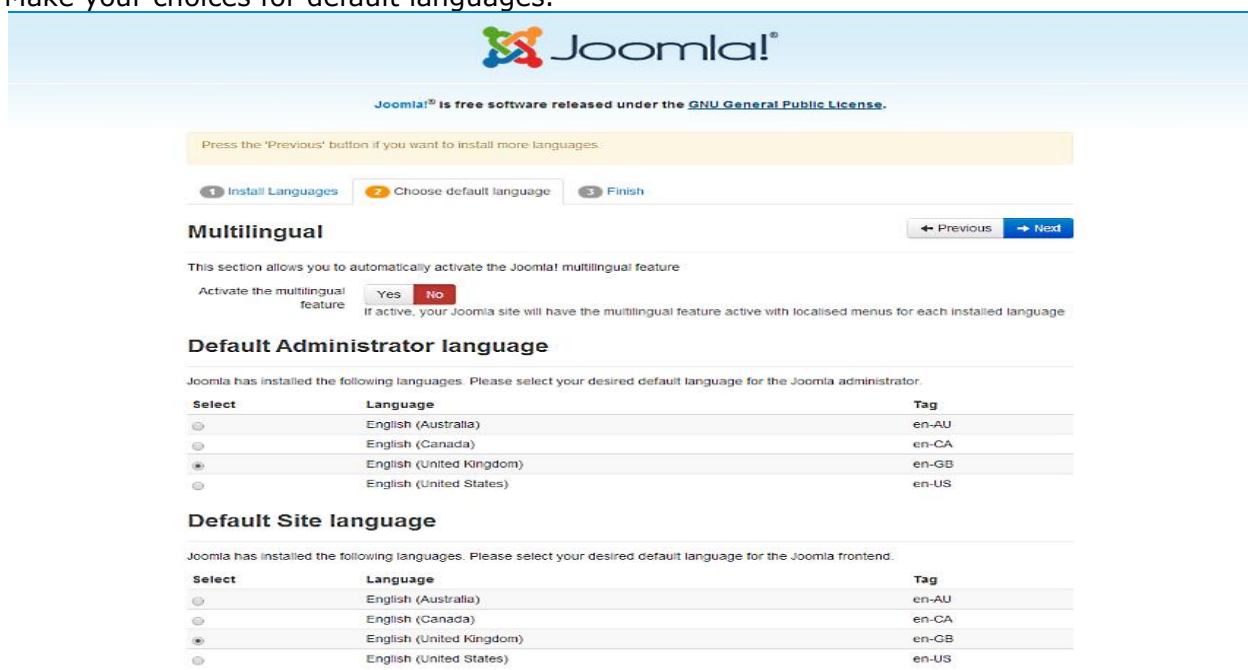
- Language packs included in custom distributions will not be listed at this stage as they are already installed.
- A version of the packs proposed will match the Joomla Major version (3.0.x, 3.1.x, etc.). The minor version of the pack may not correspond, e.g. you are installing version 3.3.3 and a 3.3.2 language pack is shown.
- Unmatched language packs in the above example may have untranslated strings.
- The unmatched language packs will be offered as an update when the packs are updated by the registered Translation teams. The available update will be shown in the Control panel as well as in Extensions Manager → Update. This behavior is similar to Extensions Manager → Install Languages.

Click Next and a progress bar will be display while the language pack or packs are downloaded.

Choose Default Language

When the download is complete you can choose the default language for the Site and the Administrator interface.

- Make your choices for default languages.



Joomla!® is free software released under the GNU General Public License.

Press the 'Previous' button if you want to install more languages.

1 Install Languages 2 Choose default language 3 Finish

Multilingual

This section allows you to automatically activate the Joomla! multilingual feature

Activate the multilingual feature ☐ Yes ☒ No
If active, your Joomla! site will have the multilingual feature active with localised menus for each installed language

Default Administrator language

Joomla has installed the following languages. Please select your desired default language for the Joomla administrator.

Select	Language	Tag
<input type="radio"/>	English (Australia)	en-AU
<input type="radio"/>	English (Canada)	en-CA
<input checked="" type="radio"/>	English (United Kingdom)	en-GB
<input type="radio"/>	English (United States)	en-US

Default Site language

Joomla has installed the following languages. Please select your desired default language for the Joomla frontend.

Select	Language	Tag
<input type="radio"/>	English (Australia)	en-AU
<input type="radio"/>	English (Canada)	en-CA
<input checked="" type="radio"/>	English (United Kingdom)	en-GB
<input type="radio"/>	English (United States)	en-US

You may also choose to activate the multilingual features of Joomla! at this time too.

- Click **Yes** next to **Activate the multilingual feature**
- Two additional choice will appear.

Multilingual

← Previous → Next

This section allows you to automatically activate the Joomla! multilingual feature

Activate the multilingual feature ☒ Yes ☐ No
If active, your Joomla site will have the multilingual feature active with localised menus for each installed language

Install localised content ☒ Yes ☐ No
If active, Joomla will automatically create one content category per each installed language. Also, one featured article containing dummy content will be created on each category

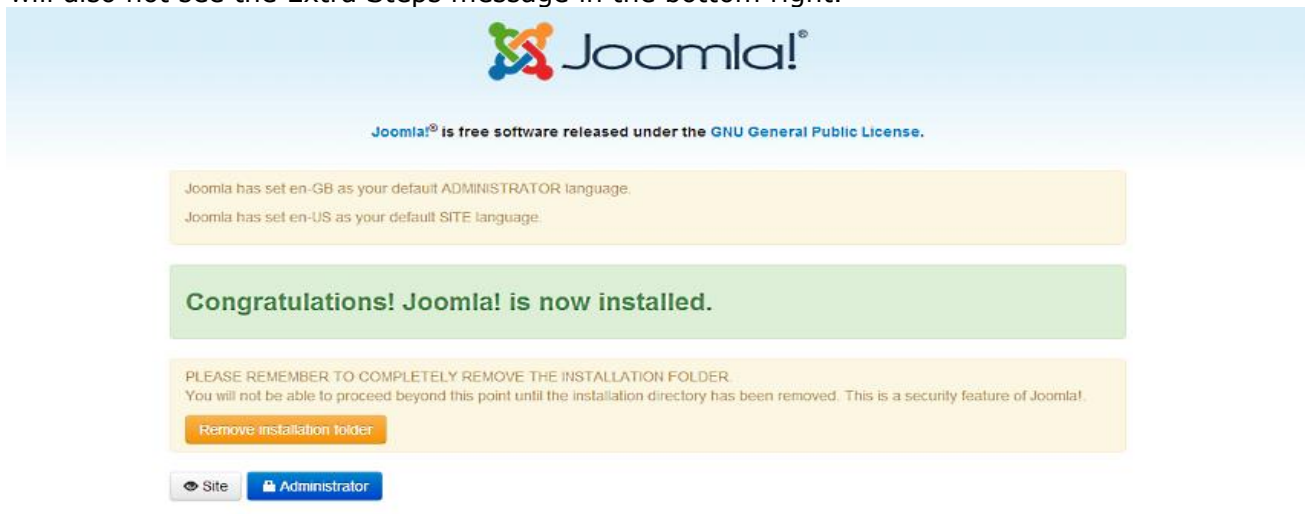
Enable the language code plugin ☐ Yes ☒ No
If enabled, the language code plugin will add the ability to change the language code in the generated HTML document to improve SEO.

- **Install localised content**, yes or no. This will automatically create one content category for each language installed and one featured article with dummy content for each content category installed.
- **Enable the language code plugin**, yes or no. If enabled, the plugin will add the ability to change the language code in the generated HTML document to improve SEO.

When you completed all the choices of language installation, click Next to proceed.

Finalise

You will now be presented with a very similar Congratulations! Joomla! is now installed. screen. The difference will be a notation of the default Administrator and Site language settings, if they were set. You will also not see the Extra Steps message in the bottom right.



Now you can delete the Installation Folder. Click on Remove Installation folder and a success message will appear. Now you can navigate to the Administratorlog in by clicking Administrator or go right to your site by clicking Site.

Unit-3

Understanding Joomla! templates

Typical Template Directory Structure

-> Joomla! CMS templates use a structure of directories and files but they can vary from template to template

- **Site** templates (templates that change what your website looks like) can be found in the /templates directory. For example, if your template is called "mytemplate", then it would be placed in the folder:
<path-to-Joomla!>/templates/mytemplate
- **Administrator** templates (templates that change what the administrator section of the site looks like) can be found in the /administrator/templates directory. For example, if your administrator template is called "myadmintemplate", then it would be placed in the folder:
<path-to-Joomla!>/administrator/templates/myadmintemplate

Template directories

-> A typical template for Joomla! will include the following directories:

- **css** - contains all the .css files
- **html** - contains template override files for core output and module chrome
- **images** - contains all images used by the template
- **language** - contains additional language files used by the template

-> Depending on the complexity and design of the template it may also contain:

- **javascript** - contains supporting JavaScript used by the template for added functionality

Example structure with files

-> Typical path of a template is <root>/public_html/domain-name/template/<name of your template> which will contain the following directories and files based on your template.

```

/css
/html
/images
/javascript
/language
component.php
error.php
favicon.ico
index.php
templateDetails.xml
template_preview.png
template_thumbnail.png

```

Template files

-> It is most common for a template to have at least the following files:

- **index.php**
Provides the logic for the display and positioning of modules and components.
- **component.php**
Provides the logic for the display of the printer friendly page, "E-mail this link to a friend." etc.
- **error.php**
Provides a method to handle errors such as 404, page not found error.
- **favicon.ico**
favicon icon file
- **template.css**
Handles the presentational aspects of the template including specifications for margins, fonts, headings, image borders, list formatting, etc. The .css files may also be located in the /css directory.
- **templateDetails.xml**
Holds meta-information related to the template and is used by the Installer and the Template Manager.
- **template_preview.ext** - replace .ext with the extension format of the image (.jpg, .png, .gif)
Generally a 600x400 pixel image that is shown when the cursor is clicked on the thumbnail image in Template Manager:Templates, not Template Manager:Styles. This gives the Administrator a pop up modal window of the template before applying it to the Site.

- **template_thumbnail.ext** - replace .ext with the extension format of the image (.jpg, .png, .gif)
Generally a 200x150 pixel thumbnail image that is shown when viewing the Template list in Template Manager:Templates, not Template Manager:Styles . This gives the Administrator a thumbnail view of the template before applying it to the Site.
- > The templateDetails.xml file is required for Joomla! templates and it can be found in the root template directory of any template inside the Joomla! templates directory. This XML file holds the basic meta-data that Joomla! needs in order to display and provide it as template option in the backend. It also contains a variety of other meta-data that is used to provide information about the template and the template authors and define files and folders that are used by the template. It also defines template language files, as well as parameters and settings the template will offer in the backend.
- > The templateDetails.xml file uses a fairly basic XML format and structure. The XML data in this file is separated into sections and specifically formatted in to render the various pieces and parameters. The XML data is read and parsed by powerful tools in the Joomla! core, then rendered to register the template and create the display seen in the Template Manager. This provides template developers a relatively simple method for creating assignments, settings and parameters for use in the template.
- > The implementation of templateDetails.xml will vary from template to template, it can be simple or complex depending on what features the template offers. The default Joomla! 3 Protostar template serves as an excellent example to demonstrate how this file is used. The various sections of this file are explained below.

XML Format

- > The top two lines of every templateDetails.xml file need to start with defining the XML format and doctype.

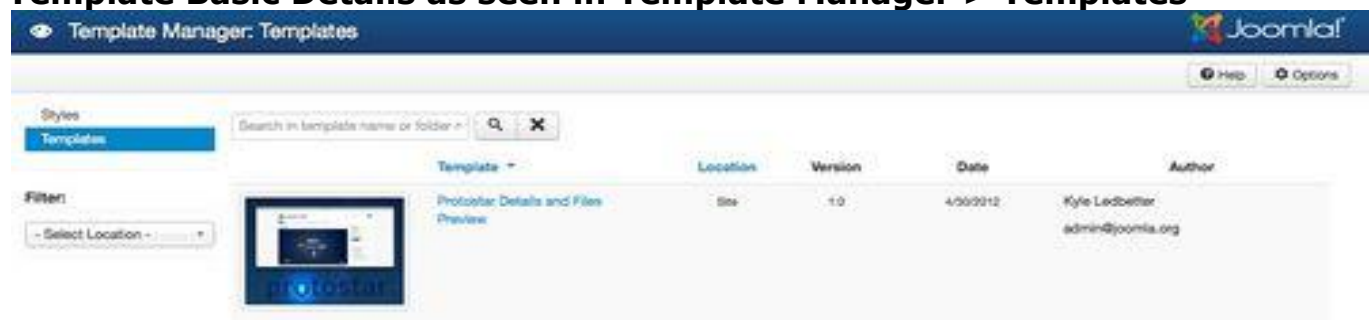
```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE install PUBLIC "-//Joomla! 2.5//DTD template 1.0//EN"
"http://www.joomla.org/xml/dtd/2.5/template-install.dtd">
```
- > The next line is also required as it tells Joomla! that the data in this file is to be used for an extension, in this case a template. All template data is contained within this <extension> tag, and the file concludes by closing this tag </extension>.

```
<extensionversion="3.1" type="template" client="site">
```

Basic Details

- > The first section is generally where template developers include information about the template. Information such as names, dates, contact information, copyrights, version number and a basic description are common. This data is used in the Template Manager and is shown in the list of available templates and can be displayed in the template editing screens as well.

Template Basic Details as seen in Template Manager > Templates



Template Details example

Protostar XML code for basic details:

```
<extension version="3.1" type="template" client="site">
  <name>protostar</name>
  <version>1.0</version>
  <creationDate>4/30/2012</creationDate>
  <author>Kyle Ledbetter</author>
  <authorEmail>admin@joomla.org</authorEmail>
  <copyright>Copyright (C) 2005 - 2014 Open Source Matters, Inc. All rights
reserved.</copyright>
  <description>TPL_PROTOSTAR_XML_DESCRIPTION</description>
```

-> You may notice that the final line of this code, the description, contains a language string and not the actual description. This string references a language file in the template where the actual description is defined and written. This is a preferred method in Joomla! for extensions that are distributed for public use to accommodate international language support, however one could simply type the description if translation is not a concern. The language files are also defined in the templateDetails.xml file, in the language section, which is covered in more detail later in this article.

Template Description seen in Protostar Editor



Template Details example

Folder Structure

-> All Folders related to the template installation are listed here. All files and folders located in the root of the template directory during installation need to be listed here. Any files contained inside a folder that is listed are automatically included. Each folder contains full path information starting at the template root. The Administrator's Installer uses this information when storing the files during installation.

-> Below is the file and folder structure from the default Protostar template. This is a fairly standard file structure for a basic Joomla! template.

<files>

```
<filename>component.php</filename>
<filename>error.php</filename>
<filename>favicon.ico</filename>
<filename>index.php</filename>
<filename>templateDetails.xml</filename>
<filename>template_preview.png</filename>
<filename>template_thumbnail.png</filename>
<folder>css</folder>
<folder>html</folder>
<folder>images</folder>
<folder>img</folder>
<folder>js</folder>
<folder>language</folder>
<folder>less</folder>
```

</files>

File directory view of J3 Protostar



Module Positions

-> The available Module Positions that can be used in the template are defined in this area.

-> These are the module positions defined in the default Protostar template:

```
<positions>
    <position>banner</position>
    <position>debug</position>
    <position>position-0</position>
    <position>position-1</position>
    <position>position-2</position>
    <position>position-3</position>
    <position>position-4</position>
    <position>position-5</position>
    <position>position-6</position>
    <position>position-7</position>
    <position>position-8</position>
    <position>position-9</position>
    <position>position-10</position>
    <position>position-11</position>
    <position>position-12</position>
    <position>position-13</position>
    <position>position-14</position>
    <position>footer</position>
</positions>
```

-> The list of positions are contained within the <positions> tag. Each <position> tag creates a module position that is available from the position list in the module manager and in other areas of Joomla! where module positions can be selected.

-> The simple format of the position list means it can be easily customized. For example, to add a new module position to the list, simply add a new <position> tag inside the <positions> tag with a unique name using all lowercase letters with no spaces. Keep in mind, this only adds the position to the backend and additional development in other template files is required to render the new position on the front end.

Languages

-> Some templates may include language files to allow translation of static text in the template. Notice that the language folder is defined and the two language files inside are also included. Even though the language folder was defined earlier, these files need their own definitions. This method tells Joomla! where the language files that contain strings used by the template reside.

-> The first file holds the language file for text that will be viewed by the User. The second file, with the .sys, or system, extension is for text that will be viewed in the Administrator area.

This is the language folder and file structure from the default Protostar template:

```
<languagesfolder="language">
    <languagegettag="en-GB">en-GB/en-GB.tpl_protostar.ini</language>
    <languagegettag="en-GB">en-GB/en-GB.tpl_protostar.sys.ini</language>
</languages>
```

-> Language strings are used in templates and throughout Joomla! for the purpose of utilizing the extensive international language support features in Joomla!. This method provides developers and users with a relatively simple method to translate any text that is used in templates and extension screens. Joomla! will check the language files for any language strings that are used and load the corresponding text in the language chosen by the user in place of the string. In this instance, there are only files for English (en-GB), any text translations in other languages must be provided by Users or Developers before they can be used by Joomla!.

More information about Language Files:

- [Creating a Language Definition File](#)
- [Loading extra language files](#)

Parameters

-> Protostar Advanced Parameters Screen

-> A template may offer display options and other parameters that can be chosen by the Administrator in the Template Manager. For instance, the default Protostar template allows the Administrator to change various colors, fonts and add a logo, these parameters are found under the Advanced tab, which is also defined and created by the XML parameters.

-> Template parameters are contained inside a `<config>` tag, which contains a `<fields>` tag with a **name attribute** of "params". Inside the `<fields>` tag is where the parameter groups and individual parameters are defined. The `<fieldset>` tag is used to create groups of parameters. Individual parameters are defined with the `<field>` tag.

-> Each `<fieldset>`, and each `<field>` parameter within a `<fieldset>`, require a unique name defined by the **name attribute**. This name defines the parameter itself and is used to pass settings to the front end files. Each parameter should also contain a **label attribute** and **description attribute**. The label text appears with the parameter in the settings screen to identify what the setting is used for and more detailed information can be included in the description.

-> A parameter field can be virtually any type of form input with corresponding options, this is selected by the **type attribute**. Any necessary options, such as radio button or select choices, are defined in `<option>` tags. CSS class names can be defined with the **class attribute**. and a default parameter setting can be defined using the **default attribute**.

-> Below are the parameter definitions in the default Protostar Template. In this example, all Labels, Descriptions and Options are using language string definitions from the Language files defined in the previous section, as well as some from the Joomla! core, so they can be translated into different languages as necessary.

-> The Protostar template illustrates a few different ways that XML can be used in a template interface, but there are many more possibilities. In this example, the `<fieldset name="advanced">` tag encloses all of the parameters and it uses the **name attribute** to create the "Advanced" tab in the interface. All that is necessary to create another tab in the interface is another `<fieldset>` tag with a different **name attribute**. With this in mind, it is relatively simple to create as many additional tabs and parameters as necessary in a template.

Additional Resources

- [More information about Creating a Joomla! Template](#)
- [Creating a Language Definition File](#)
- [Loading extra language files](#)
- [Defining a parameter in templateDetails.xml](#)

-> The index.php file is the skeleton of the website. Every page that Joomla! delivers is "index.php" fleshed out with a selection of content inserted from the database.

-> The index.php file for a template contains a mixture of code that will be delivered as it is, and php code, which will be modified before it is delivered. The code will be familiar to anyone who has designed a simple html webpage: there are 2 main sections - the <head> and <body>. Where index.php differs is the use of php code to insert information selected from a database.

-> Here is an example:

A tradition HTML head section:

```
<head>
<title>My Example Webpage</title>
<meta name="title" content="example" />
<link rel="stylesheet" href="www.example.com/css/css.css" type="text/css" />
</head>
```

And the same thing done the Joomla! way:

```
<head>
<jdoc:include type="head" />
<link rel="stylesheet" href="<?php echo $this->baseurl ?>templates/mytemplate/css/css.css"
type="text/css" />
</head>
```

-> So, instead of these header parts being defined on the index.php file, the header parts are looked up from the database by bits of php code. The clever part is that both these scripts will deliver the same code to a user. If you look at the code of a joomla website, all the <?php blah /> will have been replaced by regular html code.

-> Good template design

-> index.php should be as bare-boned as you can make it because it will be re-sent every time a new page is loaded. Elements such as styling should be delivered in css files that are saved in the users cache. The tutorials here will go through the technical aspects of creating your index.php.

-> Why index.php ?

-> Index.htm has historically been the name given to the home page of a website. Thus when a user navigates to www.example.org, the webserver delivers www.example.org/index.htm. Because Joomla! is written in PHP, index.php is the automatically served file. To further complicate things, when a user navigates to the joomla website, the index.php of the root directory redirects to the index.php of the current default template.

Model-View-Controller

-> Joomla makes extensive use of the Model-View-Controller design pattern.

-> When Joomla is started to process a request from a user, such as a GET for a particular page, or a POST containing form data, one of the first things that Joomla does is to analyse the URL to determine which component will be responsible for processing the request, and hand control over to that component.

-> If the component has been designed according to the MVC pattern, it will pass control to the controller. The controller is responsible for analysing the request and determining which model(s) will be needed to satisfy the request, and which view should be used to return the results back to the user.

-> The model encapsulates the data used by the component. In most cases this data will come from a database, either the Joomla database, or some external database, but it is also possible for the model to obtain data from other sources, such as via a web services API running on another server. The model is also responsible for updating the database where appropriate. The purpose of the model is to isolate the controller and view from the details of how data is obtained or amended.

-> The view is responsible for generating the output that gets sent to the browser by the component. It calls on the model for any information it needs and formats it appropriately. For example, a list of data items pulled from the model could be wrapped into an HTML table by the view.

-> Since Joomla is designed to be highly modular, the output from the component is generally only part of the complete web page that the user will ultimately see. Once the view has generated the output, the component hands control back to the Joomla framework which then loads and executes the template. The template combines the output from the component, and any modules that are active on the current page, so that it can be delivered to the browser as a single page.

-> To provide additional power and flexibility to web designers, who may only be concerned with creating new designs rather than manipulating the underlying code, Joomla splits the traditional view into a separate view and layout. The view pulls data from the model, as in a traditional MVC pattern, but then simply makes that data available to the layout, which is responsible for formatting

the data for presentation to the user. The advantage of having this split is that the Joomla template system provides a simple mechanism for layouts to be overridden in the template. These layout overrides (often called "template overrides" because they form part of the template, although actually it is the layout that is being overridden) are bundled with the template and give the template designer complete control over all the output from the Joomla core and any installed third-party extensions that comply with the MVC design pattern.

Module

-> Modules are lightweight and flexible extensions used for page rendering. These modules are often "boxes" arranged around a component on a typical page. A well-known example is the login module. Modules are assigned per menu item, so you can decide to show or hide (for example) the login module depending on which page (menu item) the user is currently on. Some modules are linked to components: the "latest news" module, for example, links to the content component (com_content) and displays links to the newest content items. However, modules do not need to be linked to components; they don't even need to be linked to anything and can be just static HTML or text.

-> Modules are managed in the Joomla! Administrator view by the Module Manager. More information about module management can be found on the appropriate version help screens.

- [Joomla! 2.5 Module Manager Help Screen](#)
- [Joomla! 3.4 Module Manager Help Screen](#)

Module Positions

-> A module position is a placeholder in a template. Placeholders identify one or several positions within the template and tell the Joomla! application where to place output from modules assigned to a particular position. The template designer has complete control over module positions, creating variations between templates and the respective Joomla! default positions assigned to modules in the installation sample data.

-> For example, the module position **Left** could be defined to be on the left side of the template to display a site navigation menu. So if a module is assigned the **Left** position, it will be displayed wherever the designer puts that **Left** module position - not necessarily the left side of the page.

Recommended Reading

-> Modules are one of the simplest parts of Joomla and a great entry point for people learning to use the system (the equivalent of widgets in wordpress). They can be displayed just about anywhere on a page (in all the positions a template allows as well as in the main content area using the loadmodule plugin for the com_content component).

Beginner

-> To understand how to install and use a module in Joomla it is recommended to read Module Manager. You can read also Module Manager Custom HTML for an example.

Beginner/Intermediate

-> Creating a simple module for Joomla is one of the simplest development steps you can do - and the creating a simple module tutorial is designed to take you through this. It starts with a simple module and then shows a few things you can then do with the module.

Advanced

-> Introduced into Joomla 3.2 is a hidden component that allows modules to create AJAX requests. You can find documentation about Using Joomla Ajax Interface to help you create even better modules.

Joomla! Default Modules

-> Joomla! is packaged with many modules, more are available on the JED([Joomla! Extension Directory](#)). Here are the standard modules available in a new Joomla! installation.

- Archived Articles → This Module shows a list of the calendar months containing Archived Articles.
- Articles - Newsflash → The Newsflash Module will display a fixed number of articles from a specific category.
- Articles - Related Articles → This Module displays other Articles that are related to the one currently being viewed....
- Articles Categories → This module displays a list of categories from one parent category.
- Articles Category → This module displays a list of articles from one or more categories.
- Banners → The Banner Module displays the active Banners from the Component.
- Breadcrumbs → This Module displays the Breadcrumbs
- Custom → This Module allows you to create your own HTML Module using a WYSIWYG editor.

- Feed Display → This module allows the displaying of a syndicated feed
- Footer → This module shows the Joomla! copyright information.
- Language Switcher → This module displays a list of available Content Languages (as defined and published in...)
- Latest News → This Module shows a list of the most recently published and current Articles. Some that...
- Latest Users → This module displays the latest registered users
- Login → This module displays a username and password login form. It also displays a link to...
- Menu → This module displays a menu on the frontend.
- Most Read Content → This Module shows a list of the currently published Articles which have the highest...
- Popular Tags → The Popular Tags Module displays the most commonly used tags, optionally within specific...
- Random Image → This Module displays a random image from your chosen directory.
- Search → This module will display a search box.
- Similar Tags → The Similar Tags Module displays links to other items with similar tags. The closeness...
- Smart Search Module → This is a search module for the Smart Search system.
- Statistics → The Statistics Module shows information about your server installation together with...
- Syndication Feeds → Smart Syndication Module that creates a Syndicated Feed for the page where the Module is...
- Weblinks → This modules displays Web Links from a category defined in the Weblinks component.
- Who's Online → The Who's Online Module displays the number of Anonymous Users (e.g. Guests) and...
- Wrapper → This Module shows an iframe window to specified location.

Unit-4 (Plugin)

-> A plugin is a kind of Joomla! extension. Plugins provide functions which are associated with trigger events. Joomla provides a set of core plugin events, but any extension can fire (custom) events. When a particular event occurs, all plugin functions of the type associated with the event are executed in sequence. This is a powerful way of extending the functionality of the Joomla! Platform. It also offers extension developers a way to allow other extensions to respond to their actions, making extensions extensible.

-> The Joomla! plugin architecture follows the Observer design pattern. The JPlugin class provides the means to register custom plugin code with core or custom events. The JDispatcher class (JEventDispatcher in Joomla 3.x) is an event handler which calls all plugins registered for a particular event, when that event is triggered.

Learning More

Beginner

-> To understand how to install and use a plugin in Joomla it is recommended to read Administration of a Plugin in Joomla

Intermediate

-> To understand plugins better, you should create a basic plugin for Joomla!.


-> If you have a basic content plugin in a custom module or component, we recommended reading Triggering content plugins in your extension.

Advanced

-> To understand the principles on which the Plugin system works Plugin Developer Overview. To then implement this in a component you have designed it is recommended to read Supporting plugins in your component.

Using Plugins


-> Plugins are grouped together depending on which event they run on. For developer reference there is a full list of plugins grouped by their event type. Please note, the names of a lot of events changed between the Joomla 1.5 and Joomla 2.5 versions. Here is a full list of the 1.5 to 2.5 plugin event name changes. There are also several simple tutorials on making some sample plugins running on some of these events to help running searches on extensions using both the search and smart search components:

- Creating a search plugin
- Creating a Smart Search plug-in ( 2.5 only)

-> There are further tutorials using the user triggers on how to create an authentication plugin to help users log into Joomla and creating a profile plugin for Joomla.

- Creating an Authentication Plugin for Joomla
- Creating a profile plugin

Developing an MVC Component:

-> This is a multiple-article series of tutorials on how to develop a Model-View-Controller Component for Joomla! Version .

-> Begin with the Introduction, and navigate the articles in this series by using the navigation button at the bottom or the box to the right (the Articles in this series).

Notes

- If you are new to Joomla!, please read Absolute Basics of How a Component Functions.
- This tutorial is adapted from Christophe Demko: Developing a Model-View-Controller Component/2.5
 - **WARNING:** this tutorial will not repeat the comments of Demko. To see them, please have a look to the original tutorial for Joomla! 2.5: Developing an MVC Component for Joomla! 2.5 - Introduction

Requirements

-> You need Joomla! 3.0 (with PHP, MySQL and Apache/Microsoft IIS) or greater for this tutorial. I gathered a lot of information and then I started to migrate the component of the new Joomla! 2.5 to 3.0. Below is some important information used for migration. Please see also all the information about migration Upgrading Versions. Use "display_errors On" to help with debugging errors.

Migrating Joomla! 2.5 to Joomla! 3.0:

-> Remember that you need to add Legacy at any place where you are directly extending JModel, JView or JController. If it is indirect (like through JModelList) you don't have to, it's already taken care of. Other than that and the fact that, as announced long ago, deprecated code has been removed (I'd guess that JParameter is the biggest impact), extensions should only need minor

changes ... Although you will want to look at the output changes that Kyle is working on. Of course, if you are building standalone platform applications, the new MVC and JApplicationWeb/JApplicationCLI are completely the way you should work and the nice thing about the way we have done this is that the new packages are already right there on your server having arrived with the CMS. (Elin in development list)

-> Samples: DS Since we've removed the DS constant in 3.0, we need to replace the uses of the constant in com_media. The most unobtrusive change is to simply replace it with PHP's DIRECTORY_SEPARATOR constant since DS is an alias to that. However, the recommended way to do it is to simply use the slash, i.e. 'components/com_example/models/example.php' instead of 'components'.DS.'com_example'.DS.'models'.DS.'example.php'. This is windows safe.

New MVC in Joomla! 3.0

-> "Version 12.1 of the platform introduced a new format for the model-view-controller paradigm. In principle, the classes JModel, JView and JController are now interfaces and the base abstract classes are now JModelBase, JViewBase and JControllerBase, respectively. In addition, all classes have been simplified, removing a lot of coupling with the Joomla! CMS, that was unnecessary for standalone Joomla! Platform applications." ... [Joomla! Platform Manual MVC](#)

joomla-extensions/boilerplate

-> Boilerplate files for Joomla! Extensions

component	removed category ACL	a month ago
module	cs	a year ago
plugin	Add a menu XML	10 months ago
plugins	Add search plugin boilerplate	2 months ago
template	Add initial version of template boilerplate	a year ago
.gitignore	Fixed missing replacement	a year ago
LICENSE	Initial commit	a year ago
README.md	Add missing line	a year ago

README.md

Boilerplate

Boilerplate files for Joomla! extensions.

Installation

-> The boilerplates can be installed as-is using the Extension Manager. However, the component, module and plugin will be called Foo :)


-> To create installable zip packages, you only need to zip the folder with the files and it is ready to be installed.

Customizing

-> To customize the boilerplates using your own name you need to take the following steps:

1. Do a **case-sensitive** replace on the following strings and replace them with your own name:
 - o foo
 - o Foo
 - o FOO
2. Do a **case-sensitive** replace on the following tags with their actual information:
 - o [DATE]
 - o [PROJECT_NAME]
 - o [AUTHOR]
 - o [AUTHOR_EMAIL]
 - o [COPYRIGHT]
 - o [PACKAGE_NAME]

Creating a simple module/Developing a Basic Module

This is a multiple article series on how to [create a module for Joomla!](#) Version . You can navigate the articles in this series by using the navigation drop down menu.

Begin with the [Introduction](#), and navigate the articles in this series by using the navigation button at the bottom or the box to the right (*Articles in this series*).

Modules are lightweight and flexible extensions. They are used for small bits of the page that are generally less complex and are able to be seen across different components.

You can see many examples of modules in the standard Joomla! install: - menus - Latest News - Login form - and many more.

This tutorial will explain how to go about creating a simple Hello World module. Through this tutorial you will learn the basic file structure of a module. This basic structure can then be expanded to produce more elaborate modules.

File Structure

There are four basic files that are used in the standard pattern of module development:

- `mod_helloworld.php` - This file is the main entry point for the module. It will perform any necessary initialization routines, call helper routines to collect any necessary data, and include the template which will display the module output.
- `mod_helloworld.xml` - This file contains information about the module. It defines the files that need to be installed by the Joomla! installer and specifies configuration parameters for the module.
- `helper.php` - This file contains the helper class which is used to do the actual work in retrieving the information to be displayed in the module (usually from the database or some other source).
- `tmpl/default.php` - This is the module template. This file will take the data collected by `mod_helloworld.php` and generate the HTML to be displayed on the page.

Creating `mod_helloworld.php`

The `mod_helloworld.php` file will perform three tasks:

- include the `helper.php` file which contains the class to be used to collect the necessary data
- invoke the appropriate helper class method to retrieve the data
- include the template to display the output.

The helper class is defined in our `helper.php` file. This file is included with a `require_once` statement:

```
require_once( __FILE__ ). '/helper.php';
```

`require_once` is used because our helper functions are defined within a class, and we only want the class defined once.

Our helper class has not been defined yet, but when it is, it will contain one method: `getHello()`. For our basic example, it is not really necessary to do this - the "Hello, World" message that this method returns could simply be included in the template. We use a helper class here to demonstrate this basic technique.

Our module currently does not use any parameters, but we will pass them to the helper method anyway so that it can be used later if we decide to expand the functionality of our module.

The helper class method is invoked in the following way:

```
$hello=modHelloWorldHelper::getHello($params);
```

Completed `mod_helloworld.php` file

The complete `mod_helloworld.php` file is as follows:

```
<?php
/**
 * Hello World! Module Entry Point
 *
 * @package   Joomla.Tutorials
 * @subpackage Modules
 * @license   GNU/GPL, see LICENSE.php
 * @link      http://docs.joomla.org/J3.x:Creating_a_simple_module/Developing_a_Basic_Module
 * mod_helloworld is free software. This version may have been modified pursuant
 * to the GNU General Public License, and as distributed it includes or
 * is derivative of works licensed under the GNU General Public License or
 * other free or open source software licenses.
 */

// No direct access
defined('_JEXEC') or die;
// Include the syndicate functions only once
require_once( __FILE__ ). '/helper.php';

$hello=modHelloWorldHelper::getHello($params);
requireJModuleHelper::getLayoutPath('mod_helloworld');
```


The one line that we haven't explained so far is the first line. This line checks to make sure that this file is being included from the Joomla! application. This is necessary to prevent variable injection and other potential security concerns.

Creating helper.php

The helper.php file contains the helper class that is used to retrieve the data to be displayed in the module output. As stated earlier, our helper class will have one method: getHello(). This method will return the 'Hello, World' message.

Here is the code for the helper.php file:

```
<?php
/**
 * Helper class for Hello World! module
 *
 * @package Joomla.Tutorials
 * @subpackage Modules
 * @link http://docs.joomla.org/J3.x:Creating_a_simple_module/Developing_a_Basic_Module
 * @license GNU/GPL, see LICENSE.php
 * mod_helloworld is free software. This version may have been modified pursuant
 * to the GNU General Public License, and as distributed it includes or
 * is derivative of works licensed under the GNU General Public License or
 * other free or open source software licenses.
 */
classModHelloWorldHelper
{
/**
 * Retrieves the hello message
 *
 * @param array $params An object containing the module parameters
 *
 * @access public
 */
publicstaticfunctiongetHello($params)
{
return'Hello, World!';
}
}
```

There is no rule stating that we must name our helper class as we have, but it is helpful to do this so that it is easily identifiable and locatable. Note that it is required to be in this format if you plan to use the com_ajax plugin.

More advanced modules might include database requests or other functionality in the helper class method.

Creating tmpl/default.php

The default.php file is the template which displays the module output.

The code for the default.php file is as follows:

```
<?php
// No direct access
defined('_JEXEC')or die;?>
<?php echo $hello;?>
```

An important point to note is that the template file has the same scope as the mod_helloworld.php file. What this means is that the variable \$hello can be defined in the mod_helloworld.php file and then used in the template file without any extra declarations or function calls.

Creating mod_helloworld.xml

The mod_helloworld.xml is used to specify which files the installer needs to copy and is used by the Module Manager to determine which parameters are used to configure the module. Other information about the module is also specified in this file.

The code for mod_helloworld.xml is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<extension type="module" version="3.1.0" client="site" method="upgrade">
<name>Hello, World!</name>
<author>John Doe</author>
```

```

<version>1.0.0</version>
<description>A simple Hello, World! module.</description>
<files>
<filename>mod_helloworld.xml</filename>
<filenamemodule="mod_helloworld">mod_helloworld.php</filename>
<filename>index.html</filename>
<filename>helper.php</filename>
<filename>tmpl/default.php</filename>
<filename>tmpl/index.html</filename>
</files>
<config>
</config>
</extension>

```

Manifest files explains the technical details of the elements used in the XML file.

You will notice that there are two additional files that we have not yet mentioned: index.html and tmpl/index.html. These files are included so that these directories cannot be browsed. If a user attempts to point their browser to these folders, the index.html file will be displayed. These files can be left empty or can contain the simple line:

```
<html><bodybgcolor="#FFFFFF"></body></html>
```

which will display an empty page.

Since our module does not use any form fields, the config section is empty.

Conclusion

Module development for Joomla! is a fairly simple, straightforward process. Using the techniques described in this tutorial, an endless variety of modules can be developed with little hassle.

Creating a Plugin for Joomla

The plugin structure for Joomla! 1.5 was very flexible and powerful. Not only can plugins be used to handle events triggered by the core application and extensions, but plugins can also be used to make third party extensions extensible and powerful. The main change from Joomla! 1.5 to the 2.5/3.x series was the change in the names of events.

This How-To should provide you with the basics of what you need to know to develop your own plugin. Most plugins consist of just a single code file but to correctly install the plugin code it must be packaged into an installation file which can be processed by the Joomla! installer.

Creating the Installation File

As with all extensions in Joomla, plugins are easily installed as a .zip file (.tar.gz is also supported) but a correctly formatted XML file must be included. As an example, here is the XML installation file for the categories search plugin.

```

<?xml version="1.0" encoding="utf-8"?>
<extensionversion="3.1"type="plugin"group="search">
  <name>plg_search_categories</name>
  <author>Joomla! Project</author>
  <creationDate>November 2005</creationDate>
  <copyright>Copyright (C) 2005 - 2013 Open Source Matters. All rights reserved.</copyright>
  <license>GNU General Public License version 2 or later; see LICENSE.txt</license>
  <authorEmail>admin@joomla.org</authorEmail>
  <authorUrl>www.joomla.org</authorUrl>
  <version>3.1.0</version>
  <description>PLG_SEARCH_CATEGORIES_XML_DESCRIPTION</description>
  <files>
    <filenameplugin="categories">categories.php</filename>
    <filename>index.html</filename>
  </files>
  <languages>
    <language tag="en-GB">en-GB.plg_search_categories.ini</language>
    <language tag="en-GB">en-GB.plg_search_categories.sys.ini</language>
  </languages>
  <config>
    <fieldsname="params">

```

```

<fieldsetname="basic">
    <fieldname="search_limit"type="text"
        default="50"
        description="JFIELD_PLG_SEARCH_SEARCHLIMIT_DESC"
        label="JFIELD_PLG_SEARCH_SEARCHLIMIT_LABEL"
        size="5"
    />

    <fieldname="search_content"type="radio"
        default="0"
        description="JFIELD_PLG_SEARCH_ALL_DESC"
        label="JFIELD_PLG_SEARCH_ALL_LABEL"
    >
        <optionvalue="0">JOFF</option>
        <optionvalue="1">JON</option>
    </field>

    <fieldname="search_archived"type="radio"
        default="0"
        description="JFIELD_PLG_SEARCH_ARCHIVED_DESC"
        label="JFIELD_PLG_SEARCH_ARCHIVED_LABEL"
    >
        <optionvalue="0">JOFF</option>
        <optionvalue="1">JON</option>
    </field>
</fieldset>

</fields>
</config>
</extension>

```

As you can see, the system is similar to other Joomla! XML installation files. You only have to look out for the group="xxx" entry in the <extension> tag and the extended information in the <filename> tag. This information tells Joomla! into which folder to copy the file and to which group the plugin should be added.



If you are creating a plugin that responds to existing core events, the group="xxx" attribute would be changed to reflect the name of existing plugin folder for the event type you wish to augment. e.g. group="authentication" or group="user". See [Plugin/Events](#) for a complete list of existing core event categories. In creating a new plugin to respond to core events it is important that your plugin's name is unique and does not conflict with any of the other plugins that may also be responding to the core event you wish to service as well.

If you are creating a plugin to respond to non-core system events your choice for the group="xxx" tag should be different than any of the existing core categories.

Tip If you add the attribute method="upgrade" to the tag extension, this plugin can be installed without uninstalling an earlier version. All existing files will be overwritten, but old files will not be deleted.

Creating the Plugin

The object-oriented way of writing plugins involves writing a subclass of `JPlugin`, a base class that implements the basic properties of plugins. In your methods, the following properties are available:

- `$this->params`: the [parameters](#) set for this plugin by the administrator
- `$this->_name`: the name of the plugin
- `$this->_type`: the group (type) of the plugin
- `$this->db`: the db object (since )
- `$this->app`: the application object (since )

Tip To use `$this->db` and `$this->app`, `JPlugin` tests if the property exists and is not private. If it is desired for the default objects to be used, create un-instantiated properties in the plugin class (i.e. `protected $db`; `protected $app`; in the same area as `protected $autoloadLanguage = true`;). The properties will not exist unless explicitly created.

In the following code example, <PluginGroup> represents the group (type) of the plugin, and <PluginName> represents its name. Note that class and function names in PHP are case-insensitive.

```

<?php
// no direct access
defined('_JEXEC')or die;

class plg<PluginGroup><PluginName> extends JPlugin
{
    /**
     * Load the language file on instantiation. Note this is only available in Joomla 3.1 and higher.
     * If you want to support 3.0 series you must override the constructor
     *
     * @var    boolean
     * @since  3.1
     */
    protected $autoloadLanguage = true;

    /**
     * Plugin method with the same name as the event will be called automatically.
     */
    function <EventName>()
    {
        /**
         * Plugin code goes here.
         * You can access database and application objects and parameters via $this->db,
         * $this->app and $this->params respectively
         */
        return true;
    }
}
?>

```

Using Plugins in Your Code

Now that you've created your plugin, you will probably want to call it in your code. You might not: the Joomla! core has a number of built-in events that you might want your plugin code to be registered to. In that case you don't need to do the following.

If you want to trigger an event then you use code like this:

```

$dispatcher = JDispatcher::getInstance();
$results = $dispatcher->trigger('<EventName>', <ParameterArray>);

```

It is important to note that the parameters have to be in an array. The plugin function itself will get the parameters as single values. The return value will consist of an array of return values from the different plugins (so it can also contain multilevel arrays).

If you are creating a plugin for a new, non-core event, remember to activate your plugin after you install it. Precede any reference to your new plugin with the `JPluginHelper::importPlugin()` command.

Creating a basic Joomla! template

Introduction

The purpose of this tutorial is to serve as an introduction to creating Joomla! templates. It will cover the essential files and code needed to create a basic template. The code is presented so it can be copy and pasted with very little modification needed.

Setting up a directory structure

To make the most basic template, **create a new folder** in the *templates* folder. Name this folder after your template i.e. *mynewtemplate*.

Using your favourite text editor create the files `index.php` and `templateDetails.xml`. To keep things organized, make **2 new folders** called *images* and *css*. Inside the *css* folder create a file called `template.css`.

Although it is fine to place all your CSS code directly in your `index.php` file to start, many web developers prefer to place their CSS code in a separate file that can be linked from multiple pages using the link tag. This may also shorten the loading time of your pages, since the separate file can be cached.

This is the most basic practical setup. Outline of folder and file structure:

```
* mynewtemplate/
```

```

** css/
*** template.css
** images/
** index.php
** templateDetails.xml

```

Creating a basic templateDetails.xml file

The templateDetails.xml file is essential. Without it, your template won't be seen by Joomla!. The file holds key metadata about the template.

The syntax of the file is different for each Joomla version.

So, as you can see, we have a set of information between markup tags (the <element>s). Your best approach is to copy and paste this into your templateDetails.xml file and change the relevant bits (such as <name> and <author>).

The <files> part should contain all the files that you use - you possibly don't know what they are called yet - don't worry, update it later. The <folder> element can be used to define an entire folder at once. Leave the positions as they are - these are a common set so you will be able to switch easily from the standard templates.

Creating a basic index.php file

The index.php file becomes the core of every page that Joomla! delivers. Essentially, you make a page (like any HTML page) but place PHP code where the content of your site should go. The template works by adding Joomla code into module positions and the component section in your template. Anything added to the template will appear on all pages unless it is added to one of these sections via the Joomla CMS (or customised code).

This page will show the bare-bones code ready for you to cut and paste into your own design.

Begin


A Joomla template begins with the following lines:

```

<?phpdefined('_JEXEC')ordie('Restricted access');?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
  xml:lang="<?phpecho$this->language;?>" lang="<?phpecho$this->language;?>" >

```

The first line stops naughty people looking at your coding and getting up to bad things.

The second line is the Document Type Declaration (DOCTYPE), which tells the browser (and web crawlers) which flavor of HTML the page is using. The doctype used above is HTML5, a newer version of HTML that is largely backwards compatible, but contains many new features. You should be aware that this will not work well in Internet Explorer 8 or earlier without a hack. You might want to investigate this situation and your clients' wishes before deciding on which doctype you want to use. However this is used in Joomla  3.0 and higher.

The third line begins our HTML document and describes what language the website is in. A html document is divided into two parts, head and body. The head will contain the information about the document and the body will contain the website code which controls the layout.

Head

```

<head>
<jdoc:include type="head" />
<link rel="stylesheet" href="<?phpecho$this->baseurl?>/templates/system/css/system.css"
type="text/css" />
<link rel="stylesheet" href="<?phpecho$this->baseurl?>/templates/system/css/general.css"
type="text/css" />
<link rel="stylesheet" href="<?phpecho$this->baseurl?>/templates/<?phpecho$this->
>template;?>/css/template.css" type="text/css" />
</head>

```

The first line gets Joomla to put the correct header information in. This includes the page title, meta information as well as system JavaScript. The rest creates links to two system style sheets and to your own style sheet (if it's named template.css and is located in the css folder of your template directory. So if your template is in http://www.mysite.com/templates/my_template/ then the css files will go in http://www.mysite.com/templates/my_template/css/).

Body Section

```

<body>
<jdoc:includetype="modules"name="top"/>

```

```
<jdoc:includetype="component"/>
<jdoc:includetype="modules"name="bottom"/>
</body>
```

Amazingly, this will suffice! Yes, it's a very basic layout, but it will do the job. Everything else will be done by Joomla!. These lines, usually called jdoc statements, tell Joomla to include output from certain parts of the Joomla system. Note: you will need to ensure your menu is set to go into the "top" module position.

Module Positions

Above, the line which says name="top" adds a module position called *top* and allows Joomla to place modules into this section of the template. The type="component" line contains all articles and main content (actually, the component) and is very important. It goes in the centre of the template.

Note: You can add your own module lines anywhere you want in the body, but you have to add a corresponding line to the templateDetails.xml file which sits alongside the index.php of your template.

End

Finish it off - one last bit:

```
</html>
```

Custom Images

If you want to add any images to the template you can do so like this:

```

```

Here the template variable will fill in the name of your template.

Custom CSS

You can add custom css like this:

```
<link rel="stylesheet" href="<?phpecho$this->baseurl?>/templates/<?phpecho$this->template;?>/css/styles.css" type="text/css" />
```

Every file which is added must have a line in the templateDetails.xml file for the template, unless it resides in a sub-folder (which can be included in a <folder> element).

This leaves a final file of:

```
<?phpdefined('_JEXEC')ordie('Restricted access');?>
<!DOCTYPE html>
<html xml:lang="<?phpecho$this->language;?>" lang="<?phpecho$this->language;?>" >
<head>
<jdoc:include type="head" />
<link rel="stylesheet" href="<?phpecho$this->baseurl?>/templates/<?phpecho$this->template;?>/css/template.css" type="text/css" />
</head>
<body>
<jdoc:include type="modules" name="top" />
<jdoc:include type="component" />
<jdoc:include type="modules" name="footer" />
</body>
</html>
```

Testing the template

Find the template in the Template Manager, select it and click **Default** to make it the default template.

1.5 In Joomla! 1.5, your new template will show up immediately in the Template Manager, accessible via Extensions -> Template Manager.

2.5 + In the Joomla! 2.5 series and later, you first need to tell Joomla! that you have created a new template. This feature is called *Discover Extensions* and can be accessed via Extensions -> Extension Manager -> Discover (i.e. the Discover *tab*). Click **Discover** (i.e. the Discover *button*) to discover your template, then select it and click **Install** to install it. Now your template should show up in the Template Manager (Styles), accessible via Extensions -> Template Manager.

Note you can create your template outside of Joomla and simply install it like any regular extension.

HINT: there are a couple of ways you can preview your index page as you put it together, either insert the styles into the head of the index page or directly link it to the style sheet you will be using temporarily. You can remove these links before packaging the file.

Packaging the template for installation

A directory with several loose files is not a convenient package for distribution. So the final step is to make a *package*. This is a compressed archive containing the directory structure and all the files. The

package can be in **ZIP** format (with a .zip extension), in **TAR-gzip** format (with a .tar.gz extension), or in **TAR-bz2** format (with a .tar.bz2 extension).

If your template is in a directory mytemplate/ then to make the package you can connect to that directory and use commands like:

- `tar cvvzf ../mytemplate.tar.gz *`
- `zip -r ../mytemplate.zip *.*`

Note to Mac OS X users

Note to template developers using Mac OS X systems: the Finder's "compress" menu item produces a usable ZIP format package, but with one catch. It stores the files in **AppleDouble** format, adding extra files with names beginning with "._". Thus it adds a file named "._templateDetails.xml", which Joomla 1.5.x can sometimes misinterpret. The symptom is an error message, "XML Parsing Error at 1:1. Error 4: Empty document". The workaround is to compress from the command line, and set a shell environment variable "COPYFILE_DISABLE" to "true" before using "compress" or "tar". See the [AppleDouble](#) article for more information.

To set an environment variable on a Mac, open a terminal window and type:

```
export COPYFILE_DISABLE=true
```

Then in the same terminal window, change directories into where your template files reside and issue the zip command. For instance, if your template files have been built in a folder in your personal directory called myTemplate, then you would do the following:

```
cd myTemplate
```

```
zip -r myTemplate.zip *
```

Conclusion

You should now have created a template that works. It won't look like much yet. The best thing to do now is start experimenting with the layout.

Unit-5

Advance Component Development

Adding Language Management in Joomla:

With your favourite file manager and editor, put a file `site/language/en-GB/en-GB.com_helloworld.ini`. This file will contain translation for the public part. For the moment, this file is empty.

`site/language/en-GB/en-GB.com_helloworld.ini`

For the moment, there are no translations strings in this file.

Adding language translation when managing the component

With your favourite file manager and editor, put a file `admin/language/en-GB/en-GB.com_helloworld.ini`. This file will contain translation for the backend part.

`admin/language/en-GB/en-GB.com_helloworld.ini`

```
; Joomla! Project
; Copyright (C) 2005 - 2018 Open Source Matters. All rights reserved.
; License GNU General Public License version 2 or later; see LICENSE.txt, see LICENSE.php
; Note : All ini files need to be saved as UTF-8
```

```
COM_HELLOWORLD_NUM="#"
COM_HELLOWORLD_PUBLISHED="Published"
COM_HELLOWORLD_HELLOWORLDS_NAME="Name"
COM_HELLOWORLD_ID="Id"
```

```
COM_HELLOWORLD_HELLOWORLD_FIELD_GREETING_DESC="This message will be displayed"
COM_HELLOWORLD_HELLOWORLD_FIELD_GREETING_LABEL="Message"
COM_HELLOWORLD_HELLOWORLD_HEADING_GREETING="Greeting"
COM_HELLOWORLD_HELLOWORLD_HEADING_ID="Id"
```

Adding language translation when managing the menus in the backend

With your favourite file manager and editor, put a file `admin/language/en-GB/en-GB.com_helloworld.sys.ini`. This file will contain translation for the backend part.

`admin/language/en-GB/en-GB.com_helloworld.sys.ini`

```
; Joomla! Project
; Copyright (C) 2005 - 2015 Open Source Matters. All rights reserved.
; License GNU General Public License version 2 or later; see LICENSE.txt, see LICENSE.php
; Note : All ini files need to be saved as UTF-8
```

```
COM_HELLOWORLD="Hello World!"
COM_HELLOWORLD_DESCRIPTION="This is the Hello World description"
COM_HELLOWORLD_HELLOWORLD_VIEW_DEFAULT_TITLE="Hello World"
COM_HELLOWORLD_HELLOWORLD_VIEW_DEFAULT_DESC="This view displays a selected message"
COM_HELLOWORLD_MENU="Hello World!"
```

Language File Location Options

Starting with version 1.7 there are 2 ways to install language files for an extension. One can use one or the other or a combination of both.

The 1.5 way will install the files in the CORE language folders

(`ROOT/administrator/language/` and `ROOT/language/`). Since version 1.6, include the files in a "language" folder installed at the root of the extension.

Therefore an extension can include a language folder with a `.sys.ini` different from the one installed in the Joomla core language folders. (This last one not being included in that language folder but in root or any other folder not installed.)

This lets us display 2 different descriptions: one from the `sys.ini` in the "language" folder is used as a message displayed when installation is finished, the other (from the `.ini`) is used for "normal" operation, that is, when the extension is edited in the back-end. This can be extremely useful when the installation also uses some scripts and requires a different value for the description.

Language file used by the installation script during the installation of a component(*the first install*, not an upgrade) obeys specific rules described in the article, [Specification of language files](#). During the first installation, only the language file included in the component folder `(/administrator/components/com_helloworld/language)` is used when present. If this file is only provided in the CORE language folder `(/administrator/language)`, no translation occurs. This also applies to KEYS used in the manifest file.

When upgrading or uninstalling the extension (not installing), it is the sys.ini file present in the extension root language folder which will display the result of the installation from the description key/value. Thereafter, if present, the sys.ini as well as the .ini installed in the CORE language folder will have priority over the files present in the root language folder of the extension.

Adding backend actions: Adding a toolbar

In Joomla, the administrator interacts generally with components through the use of a toolbar. In the file `admin/views/helloworlds/view.html.php` put this content. It will create a basic toolbar and a title for the component.

The args used in ex. `JToolBarHelper::addNew` is used to set a controller instance which will be used after button is clicked.

`admin/views/helloworlds/view.html.php`

```
<?php
// No direct access to this file
defined('_JEXEC') or die('Restricted access');

// import Joomla view library
jimport('joomla.application.component.view');

/**
 * HelloWorlds View
 */
class HelloWorldViewHelloWorlds extends JView
{
    /**
     * HelloWorlds view display method
     * @param string $tpl The name of the template file to parse; automatically searches
     through the template paths.
     *
     * @return mixed A string if successful, otherwise a JError object.
     */
    function display($tpl = null)
    {
        // Get data from the model
        $items = $this->get('Items');
        $pagination = $this->get('Pagination');

        // Check for errors.
        if (count($errors = $this->get('Errors')))
        {
            JError::raiseError(500, implode('<br />', $errors));
            return false;
        }

        // Assign data to the view
        $this->items = $items;
        $this->pagination = $pagination;

        // Set the toolbar
        $this->addToolBar();

        // Display the template
```

```

        parent::display($tpl);
    }

    /**
     * Setting the toolbar
     */
    protected function addToolBar()
    {
        JToolBarHelper::title(JText::_('COM_HELLOWORLD_MANAGER_HELLOWORLDS'));
        JToolBarHelper::deleteList('', 'helloworlds.delete');
        JToolBarHelper::editList('helloworld.edit');
        JToolBarHelper::addNew('helloworld.add');
    }
}

```

You can find other classic backend actions in the *administrator/includes/toolbar.php* file of your Joomla installation.

Since the view can perform some actions, we have to add some input data. With your favorite file manager and editor, put in the file *admin/views/helloworlds/tmpl/default.php*

```

<?php
// No direct access to this file
defined('_JEXEC') or die('Restricted Access');
// load tooltip behavior
JHtml::_('behavior.tooltip');
?>
<form action="<?php echo JRoute::_('index.php?option=com_helloworld'); ?>" method="post"
name="adminForm" id="adminForm">
    <table class="adminlist">
        <thead><?php echo $this->loadTemplate('head');?></thead>
        <tfoot><?php echo $this->loadTemplate('foot');?></tfoot>
        <tbody><?php echo $this->loadTemplate('body');?></tbody>
    </table>
    <div>
        <input type="hidden" name="task" value="" />
        <input type="hidden" name="boxchecked" value="0" />
        <?php echo JHtml::_('form.token'); ?>
    </div>
</form>

```

Adding specific controllers

Three actions have been added:

- *helloworlds.delete*
- *helloworld.edit*
- *helloworld.add*

read more about [subcontrollers...](#)

These are compound tasks (*controller.task*). So two new controllers *HelloWorldControllerHelloWorlds* and *HelloWorldControllerHelloWorld* have to be coded. *admin/controllers/helloworlds.php*

```

<?php
// No direct access to this file
defined('_JEXEC') or die('Restricted access');

// import Joomla controlleradmin library
jimport('joomla.application.component.controlleradmin');

/**
 * HelloWorlds Controller

```

```

*/
class HelloWorldControllerHelloWorlds extends JControllerAdmin
{
    /**
     * Proxy for getModel.
     * @since      2.5
     */
    public function getModel($name = 'HelloWorld', $prefix = 'HelloWorldModel')
    {
        $model = parent::getModel($name, $prefix, array('ignore_request' => true));
        return $model;
    }
}

```

admin/controllers/helloworld.php

```

<?php
// No direct access to this file
defined('_JEXEC') or die('Restricted access');

// import Joomla controllerform library
jimport('joomla.application.component.controllerform');

/**
 * HelloWorld Controller
 */
class HelloWorldControllerHelloWorld extends JControllerForm
{
}

```

Adding an editing view

With your favorite file manager and editor, put a file *admin/views/helloworld/view.html.php* containing:

```

<?php
// No direct access to this file
defined('_JEXEC') or die('Restricted access');

// import Joomla view library
jimport('joomla.application.component.view');

/**
 * HelloWorld View
 */
class HelloWorldViewHelloWorld extends JView
{
    /**
     * display method of Hello view
     * @return void
     */
    public function display($tpl = null)
    {
        // get the Data
        $form = $this->get('Form');
        $item = $this->get('Item');

        // Check for errors.
        if (count($errors = $this->get('Errors')))
        {
            JError::raiseError(500, implode('<br />', $errors));
        }
    }
}

```

```

        return false;
    }
    // Assign the Data
    $this->form = $form;
    $this->item = $item;

    // Set the toolbar
    $this->addToolBar();

    // Display the template
    parent::display($tpl);
}

/**
 * Setting the toolbar
 */
protected function addToolBar()
{
    $input = JFactory::getApplication()->input;
    $input->set('hidemainmenu', true);
    $isNew = ($this->item->id == 0);
    JToolBarHelper::title($isNew ?
JText::_('COM_HELLOWORLD_MANAGER_HELLOWORLD_NEW')
    :
JText::_('COM_HELLOWORLD_MANAGER_HELLOWORLD_EDIT'));
    JToolBarHelper::save('helloworld.save');
    JToolBarHelper::cancel('helloworld.cancel', $isNew ? 'JTOOLBAR_CANCEL'
    : 'JTOOLBAR_CLOSE');
}
}

```

This view will display data using a layout.
Put a file `admin/views/helloworld/tmpl/edit.php` containing
`admin/views/helloworld/tmpl/edit.php`

```

<?php
// No direct access
defined('_JEXEC') or die('Restricted access');
JHtml::_('behavior.tooltip');
?>
<form action="<?php echo JRoute::_('index.php?option=com_helloworld&layout=edit&id='.(int) $this->
item->id); ?>"
    method="post" name="adminForm" id="adminForm">
    <fieldset class="adminform">
        <legend><?php echo JText::_('COM_HELLOWORLD_HELLOWORLD_DETAILS' );
?></legend>
        <ul class="adminformlist">
<?php foreach($this->form->getFieldset() as $field): ?>
            <li><?php echo $field->label;echo $field->input;?></li>
<?php endforeach; ?>
        </ul>
    </fieldset>
    <div>
        <input type="hidden" name="task" value="helloworld.edit" />
        <?php echo JHtml::_('form.token'); ?>
    </div>
</form>

```

Adding a model and modifying the existing one

The *HelloWorldViewHelloWorld* view asks form and data from a model. This model has to provide a *getTable*, a *getForm* method and a *loadData* method (called from the *JModelAdmin* controller) *admin/models/helloworld.php*

```
<?php
// No direct access to this file
defined('_JEXEC') or die('Restricted access');

// import Joomla modelform library
jimport('joomla.application.component.modeladmin');

/**
 * HelloWorld Model
 */
class HelloWorldModelHelloWorld extends JModelAdmin
{
    /**
     * Returns a reference to the a Table object, always creating it.
     *
     * @param    type    The table type to instantiate
     * @param    string  A prefix for the table class name. Optional.
     * @param    array   Configuration array for model. Optional.
     * @return   JTable  A database object
     * @since    2.5
     */
    public function getTable($type = 'HelloWorld', $prefix = 'HelloWorldTable', $config = array())
    {
        return JTable::getInstance($type, $prefix, $config);
    }
    /**
     * Method to get the record form.
     *
     * @param    array    $data      Data for the form.
     * @param    boolean  $loadData  True if the form is to load its own data (default
case), false if not.
     * @return   mixed    A JForm object on success, false on failure
     * @since    2.5
     */
    public function getForm($data = array(), $loadData = true)
    {
        // Get the form.
        $form = $this->loadForm('com_helloworld.helloworld', 'helloworld',
            array('control' => 'jform', 'load_data' => $loadData));
        if (empty($form))
        {
            return false;
        }
        return $form;
    }
    /**
     * Method to get the data that should be injected in the form.
     *
     * @return   mixed    The data for the form.
     * @since    2.5
     */
    protected function loadFormData()
    {
        // Check the session for previously entered form data.
```

```

        $data = JFactory::getApplication()-
>getUserState('com_helloworld.edit.helloworld.data', array());
        if (empty($data))
        {
            $data = $this->getItem();
        }
        return $data;
    }
}

```

This model inherits from the *JModelAdmin* class and uses its *loadForm* method. This method searches for forms in the *forms* folder. With your favorite file manager and editor, put a file *admin/models/forms/helloworld.xml* containing:

Adding decorations to the backend:

Adding some icons

With your favourite file manager put a 16x16 image and a 48x48 image (I choose *Tux*) in a *media/images/* folder and add a *media* tag in your install file. Modify the *menu* tag in order to use the new icon.

Note that default admin template (isis) disables menu icons ! One possible workaround is to use *icoMoon* in your language string:

```
admin/language/en-GB/en-GB.com_helloworld.sys.ini
```

```

; Joomla! Project
; Copyright (C) 2005 - 2018 Open Source Matters. All rights reserved.
; License GNU General Public License version 2 or later; see LICENSE.txt, see LICENSE.php
; Note : All ini files need to be saved as UTF-8

```

```

COM_HELLOWORLD="Hello World!"
COM_HELLOWORLD_DESCRIPTION="This is the Hello World description"
COM_HELLOWORLD_HELLOWORLD_VIEW_DEFAULT_TITLE="Hello World"
COM_HELLOWORLD_HELLOWORLD_VIEW_DEFAULT_DESC="This view displays a selected message"
COM_HELLOWORLD_MENU=" <i class="icon-smiley-2"></i>Hello World!"

```

Icon list can be found here: [Standard IcoMoon Fonts](#).

Modifying the views

In the `admin/views/helloworlds/view.html.php` file put these lines:

```

admin/views/helloworlds/view.html.php

<?php
/**
 * @package   Joomla.Administrator
 * @subpackage com_helloworld
 *
 * @copyright Copyright (C) 2005 - 2018 Open Source Matters, Inc. All rights reserved.
 * @license   GNU General Public License version 2 or later; see LICENSE.txt
 */

// No direct access to this file
defined('_JEXEC') or die('Restricted access');

/**
 * HelloWorlds View
 *
 * @since 0.0.1
 */
class HelloWorldViewHelloWorlds extends JViewLegacy
{
    /**
     * Display the Hello World view
     *
     * @param string $tpl The name of the template file to parse; automatically searches through the template paths.
     *
     * @return void
     */
}

```

```

function display($tpl = null)
{
    // Get application
    $app = JFactory::getApplication();
    $context = "helloworld.list.admin.helloworld";
    // Get data from the model
    $this->items = $this->get('Items');
    $this->pagination = $this->get('Pagination');
    $this->state = $this->get('State');
    $this->filter_order = $app->getUserStateFromRequest($context.'filter_order', 'filter_order',
'greeting', 'cmd');
    $this->filter_order_Dir = $app->getUserStateFromRequest($context.'filter_order_Dir',
'filter_order_Dir', 'asc', 'cmd');
    $this->filterForm = $this->get('FilterForm');
    $this->activeFilters = $this->get('ActiveFilters');

    // Check for errors.
    if (count($errors = $this->get('Errors')))
    {
        JError::raiseError(500, implode('<br />', $errors));

        return false;
    }

    // Set the toolbar and number of found items
    $this->addToolBar();

    // Display the template
    parent::display($tpl);

    // Set the document
    $this->setDocument();
}

/**
 * Add the page title and toolbar.
 *
 * @return void
 *
 * @since 1.6
 */
protected function addToolBar()
{
    $title = JText::_('COM_HELLOWORLD_MANAGER_HELLOWORLDS');

    if ($this->pagination->total)
    {
        $title .= "<span style='font-size: 0.5em; vertical-align: middle;'>(" . $this->pagination-
>total . ")</span>";
    }

    JToolBarHelper::title($title, 'helloworld');
    JToolBarHelper::deleteList("", 'helloworlds.delete');
    JToolBarHelper::editList('helloworld.edit');
    JToolBarHelper::addNew('helloworld.add');
}

/**
 * Method to set up the document properties
 *
 * @return void
 */
protected function setDocument()
{
    $document = JFactory::getDocument();
    $document->setTitle(JText::_('COM_HELLOWORLD_ADMINISTRATION'));
}
}

```


`$this->filter_order` and `$this->filter_order_Dir` store the active sorting column and the sorting direction respectively. Those variables are retrieved from the app state variables.

See [How to use user state variables](#) for more information about the use of the state variables.

This view uses a second parameter for the `JToolBarHelper::title` function. It will be used to construct the css class for the title. The `_setDocument` method sets the browser title.

In the `admin/views/helloworlds/tmpl/default.php` file put these lines:

`admin/views/helloworlds/tmpl/default.php`

```
<?php
/**
 * @package Joomla.Administrator
 * @subpackage com_helloworld
 *
 * @copyright Copyright (C) 2005 - 2018 Open Source Matters, Inc. All rights reserved.
 * @license GNU General Public License version 2 or later; see LICENSE.txt
 */

// No direct access to this file
defined('_JEXEC') or die('Restricted Access');

JHtml::_('formbehavior.chosen', 'select');

$listOrder = $this->escape($this->filter_order);
$listDirn = $this->escape($this->filter_order_Dir);
?>
<form action="index.php?option=com_helloworld&view=helloworlds" method="post" id="adminForm" name="adminForm">
    <div class="row-fluid">
        <div class="span6">
            <?php echo JText::_('COM_HELLOWORLD_HELLOWORLDS_FILTER'); ?>
            <?php
                echo JLayoutHelper::render(
                    'joomla.searchtools.default',
                    array('view' => $this)
                );
            ?>
        </div>
    </div>
    <table class="table table-striped table-hover">
        <thead>
            <tr>
                <th width="1%"><?php echo JText::_('COM_HELLOWORLD_NUM'); ?></th>
                <th width="2%">
                    <?php echo JHtml::_('grid.checkall'); ?>
                </th>
                <th width="90%">
                    <?php echo JHtml::_('grid.sort', 'COM_HELLOWORLD_HELLOWORLDS_NAME',
'greeting', $listDirn, $listOrder); ?>
                </th>
                <th width="5%">
                    <?php echo JHtml::_('grid.sort', 'COM_HELLOWORLD_PUBLISHED', 'published',
$listDirn, $listOrder); ?>
                </th>
                <th width="2%">
                    <?php echo JHtml::_('grid.sort', 'COM_HELLOWORLD_ID', 'id', $listDirn,
$listOrder); ?>
                </th>
            </tr>
        </thead>
        <tfoot>
            <tr>
                <td colspan="5">
                    <?php echo $this->pagination->getListFooter(); ?>
                </td>
            </tr>
        </tfoot>
        <tbody>
            <?php if (!empty($this->items)) : ?>
                <?php foreach ($this->items as $i => $row) :
                    $link =
JRoute::_('index.php?option=com_helloworld&task=helloworld.edit&id=' . $row->id);
```

```

?>
        <tr>
            <td><?php echo $this->pagination->getRowOffset($i);
?></td>
            <td>
                <?php echo JHtml::_('grid.id', $i, $row->id); ?>
            </td>
            <td>
                <a href="<?php echo $link; ?>" title="<?php
echo JText::_('COM_HELLOWORLD_EDIT_HELLOWORLD'); ?>">
                    <?php echo $row->greeting; ?>
                </a>
            </td>
            <td align="center">
                <?php echo JHtml::_('jgrid.published', $row-
>published, $i, 'helloworlds.', true, 'cb'); ?>
            </td>
            <td align="center">
                <?php echo $row->id; ?>
            </td>
        </tr>
    <?php endforeach; ?>
<?php endif; ?>
</tbody>
</table>
<input type="hidden" name="task" value=""/>
<input type="hidden" name="checked" value="0"/>
<input type="hidden" name="filter_order" value="<?php echo $listOrder; ?>"/>
<input type="hidden" name="filter_order_Dir" value="<?php echo $listDirn; ?>"/>
<?php echo JHtml::_('form.token'); ?>
</form>

```

In `admin/views/helloworld/view.html.php`, put these lines:

```

admin/views/helloworld/view.html.php
<?php
/**
 * @package Joomla.Administrator
 * @subpackage com_helloworld
 *
 * @copyright Copyright (C) 2005 - 2018 Open Source Matters, Inc. All rights reserved.
 * @license GNU General Public License version 2 or later; see LICENSE.txt
 */

// No direct access to this file
defined('_JEXEC') or die('Restricted access');

/**
 * HelloWorld View
 *
 * @since 0.0.1
 */
class HelloWorldViewHelloWorld extends JViewLegacy
{
    /**
     * View form
     *
     * @var form
     */
    protected $form = null;

    /**
     * Display the Hello World view
     *
     * @param string $tpl The name of the template file to parse; automatically searches through the template paths.
     *
     * @return void
     */
    public function display($tpl = null)
    {
        // Get the Data
        $this->form = $this->get('Form');
    }
}

```

```

$this->item = $this->get('Item');

// Check for errors.
if (count($errors = $this->get('Errors')))
{
    JError::raiseError(500, implode('<br />', $errors));

    return false;
}

// Set the toolbar
$this->addToolBar();

// Display the template
parent::display($tpl);

// Set the document
$this->setDocument();
}

/**
 * Add the page title and toolbar.
 *
 * @return void
 *
 * @since 1.6
 */
protected function addToolBar()
{
    $input = JFactory::getApplication()->input;

    // Hide Joomla Administrator Main menu
    $input->set('hidemainmenu', true);

    $isNew = ($this->item->id == 0);

    if ($isNew)
    {
        $title = JText::_('COM_HELLOWORLD_MANAGER_HELLOWORLD_NEW');
    }
    else
    {
        $title = JText::_('COM_HELLOWORLD_MANAGER_HELLOWORLD_EDIT');
    }

    JToolBarHelper::title($title, 'helloworld');
    JToolBarHelper::save('helloworld.save');
    JToolBarHelper::cancel(
        'helloworld.cancel',
        $isNew ? 'JTOOLBAR_CANCEL' : 'JTOOLBAR_CLOSE'
    );
}

/**
 * Method to set up the document properties
 *
 * @return void
 */
protected function setDocument()
{
    $isNew = ($this->item->id < 1);
    $document = JFactory::getDocument();
    $document->setTitle($isNew ? JText::_('COM_HELLOWORLD_HELLOWORLD_CREATING') :
    JText::_('COM_HELLOWORLD_HELLOWORLD_EDITING'));
}
}

```

This view also uses the second parameter of the `JToolBarHelper::title` function and set the browser title.

Adding the filters

In `admin/models/helloworlds.php`, put these lines:

```
admin/models/helloworlds.php

<?php
/**
 * @package Joomla.Administrator
 * @subpackage com_helloworld
 *
 * @copyright Copyright (C) 2005 - 2018 Open Source Matters, Inc. All rights reserved.
 * @license GNU General Public License version 2 or later; see LICENSE.txt
 */
// No direct access to this file
defined('_JEXEC') or die('Restricted access');

/**
 * HelloWorldList Model
 *
 * @since 0.0.1
 */
class HelloWorldModelHelloWorlds extends JModelList
{
    /**
     * Constructor.
     *
     * @param array $config An optional associative array of configuration settings.
     *
     * @see JController
     * @since 1.6
     */
    public function __construct($config = array())
    {
        if (empty($config['filter_fields']))
        {
            $config['filter_fields'] = array(
                'id',
                'greeting',
                'published'
            );
        }

        parent::__construct($config);
    }

    /**
     * Method to build an SQL query to load the list data.
     *
     * @return string An SQL query
     */
    protected function getListQuery()
    {
        // Initialize variables.
        $db = JFactory::getDbo();
        $query = $db->getQuery(true);

        // Create the base select statement.
        $query->select('*')
            ->from($db->quoteName('#__helloworld'));

        // Filter: like / search
        $search = $this->getState('filter.search');

        if (!empty($search))
        {
            $like = $db->quote('%' . $search . '%');
            $query->where('greeting LIKE ' . $like);
        }

        // Filter by published state
        $published = $this->getState('filter.published');
```

```

        if (is_numeric($published))
        {
            $query->where('published = ' . (int) $published);
        }
        elseif ($published === '')
        {
            $query->where('(published IN (0, 1))');
        }

        // Add the list ordering clause.
        $orderCol = $this->state->get('list.ordering', 'greeting');
        $orderDirn = $this->state->get('list.direction', 'asc');

        $query->order($db->escape($orderCol) . ' ' . $db->escape($orderDirn));

        return $query;
    }
}

```

With your favourite file manager and editor put a file `admin/models/forms/filter_helloworlds.xml` containing `admin/models/forms/filter_helloworlds.xml`

```

<?xml version="1.0" encoding="utf-8"?>
<form>
    <fields name="filter">
        <field
            name="search"
            type="text"
            label="COM_BANNERS_SEARCH_IN_TITLE"
            hint="JSEARCH_FILTER"
            class="js-stools-search-string"
        />
        <field
            name="published"
            type="status"
            label="JOPTION_SELECT_PUBLISHED"
            description="JOPTION_SELECT_PUBLISHED_DESC"
            onchange="this.form.submit();"
        >
            <option value="">JOPTION_SELECT_PUBLISHED</option>
        </field>
    </fields>
    <fields name="list">
        <field
            name="limit"
            type="limitbox"
            class="input-mini"
            default="25"
            label="COM_CONTENT_LIST_LIMIT"
            description="COM_HELLOWORLD_LIST_LIMIT_DESC"
            onchange="this.form.submit();"
        />
    </fields>
</form>

```

Modifying the main entry file

In the `admin/helloworld.php` file, put these lines in order to use the 48x48 icon:

```

<?php
/**
 * @package Joomla.Administrator
 * @subpackage com_helloworld
 *
 * @copyright Copyright (C) 2005 - 2018 Open Source Matters, Inc. All rights reserved.
 * @license GNU General Public License version 2 or later; see LICENSE.txt
 */

// No direct access to this file

```

```
defined('_JEXEC') or die('Restricted access');
```

```
// Set some global property
```

```
$document = JFactory::getDocument();
```

```
$document->addStyleDeclaration('.icon-helloworld {background-image: url(../media/com_helloworld/images/Tux-16x16.png);}');
```

```
// Get an instance of the controller prefixed by HelloWorld
```

```
$controller = JControllerLegacy::getInstance('HelloWorld');
```

```
// Perform the Request task
```

```
$input = JFactory::getApplication()->input;
```

```
$controller->execute($input->getCmd('task'));
```

```
// Redirect if set by the controller
```

```
$controller->redirect();
```

Adding some strings in the language file

Modify the `admin/language/en-GB/en-GB.com_helloworld.ini` and put these lines

```
admin/language/en-GB/en-GB.com_helloworld.ini
```

```
; Joomla! Project
```

```
; Copyright (C) 2005 - 2018 Open Source Matters. All rights reserved.
```

```
; License GNU General Public License version 2 or later; see LICENSE.txt, see LICENSE.php
```

```
; Note : All ini files need to be saved as UTF-8
```

```
COM_HELLOWORLD_ADMINISTRATION="HelloWorld - Administration"
```

```
COM_HELLOWORLD_NUM="#"
```

```
COM_HELLOWORLD_HELLOWORLDS_FILTER="Filters"
```

```
COM_HELLOWORLD_PUBLISHED="Published"
```

```
COM_HELLOWORLD_HELLOWORLDS_NAME="Name"
```

```
COM_HELLOWORLD_ID="Id"
```

```
COM_HELLOWORLD_HELLOWORLD_CREATING="HelloWorld - Creating"
```

```
COM_HELLOWORLD_HELLOWORLD_DETAILS="Details"
```

```
COM_HELLOWORLD_HELLOWORLD_EDITING="HelloWorld - Editing"
```

```
COM_HELLOWORLD_HELLOWORLD_FIELD_GREETING_DESC="This message will be displayed"
```

```
COM_HELLOWORLD_HELLOWORLD_FIELD_GREETING_LABEL="Message"
```

```
COM_HELLOWORLD_HELLOWORLD_HEADING_GREETING="Greeting"
```

```
COM_HELLOWORLD_HELLOWORLD_HEADING_ID="Id"
```

```
COM_HELLOWORLD_MANAGER_HELLOWORLD_EDIT="HelloWorld manager: Edit Message"
```

```
COM_HELLOWORLD_MANAGER_HELLOWORLD_NEW="HelloWorld manager: New Message"
```

```
COM_HELLOWORLD_MANAGER_HELLOWORLDS="HelloWorld manager"
```

```
COM_HELLOWORLD_EDIT_HELLOWORLD="Edit message"
```

```
COM_HELLOWORLD_N_ITEMS_DELETED_1="One message deleted"
```

```
COM_HELLOWORLD_N_ITEMS_DELETED_MORE="%d messages deleted"
```

```
COM_HELLOWORLD_N_ITEMS_PUBLISHED="%d message(s) published"
```

```
COM_HELLOWORLD_N_ITEMS_UNPUBLISHED="%d message(s) unpublished"
```

```
COM_HELLOWORLD_HELLOWORLD_GREETING_LABEL="Greeting"
```

```
COM_HELLOWORLD_HELLOWORLD_GREETING_DESC="Add Hello World Greeting"
```

Packaging the component

Content of your code directory. Each file link below takes you to the step in the tutorial which has the latest version of that source code file.

- [helloworld.xml](#)
- [site/helloworld.php](#)
- [site/index.html](#)
- [site/controller.php](#)
- [site/views/index.html](#)
- [site/views/helloworld/index.html](#)
- [site/views/helloworld/view.html.php](#)
- [site/views/helloworld/tmpl/index.html](#)
- [site/views/helloworld/tmpl/default.xml](#)

- [site/views/helloworld/tmpl/default.php](#)
- [site/models/index.html](#)
- [site/models/helloworld.php](#)
- [site/language/index.html](#)
- [site/language/en-GB/index.html](#)
- [site/language/en-GB/en-GB.com helloworld.ini](#)
- [admin/index.html](#)
- [admin/helloworld.php](#)
- [admin/controller.php](#)
- [admin/sql/index.html](#)
- [admin/sql/install.mysql.utf8.sql](#)
- [admin/sql/uninstall.mysql.utf8.sql](#)
- [admin/sql/updates/index.html](#)
- [admin/sql/updates/mysql/index.html](#)
- [admin/sql/updates/mysql/0.0.1.sql](#)
- [admin/sql/updates/mysql/0.0.6.sql](#)
- [admin/models/index.html](#)
- [admin/models/fields/index.html](#)
- [admin/models/fields/helloworld.php](#)
- [admin/models/helloworlds.php](#)
- [admin/models/helloworld.php](#)
- [admin/models/forms/filter helloworlds.xml](#)
- [admin/models/forms/index.html](#)
- [admin/models/forms/helloworld.xml](#)
- [admin/controllers/helloworld.php](#)
- [admin/controllers/helloworlds.php](#)
- [admin/controllers/index.html](#)
- [admin/views/index.html](#)
- [admin/views/helloworld/index.html](#)
- [admin/views/helloworld/view.html.php](#)
- [admin/views/helloworld/tmpl/index.html](#)
- [admin/views/helloworld/tmpl/edit.php](#)
- [admin/views/helloworlds/index.html](#)
- [admin/views/helloworlds/view.html.php](#)
- [admin/views/helloworlds/tmpl/index.html](#)
- [admin/views/helloworlds/tmpl/default.php](#)
- [admin/tables/index.html](#)
- [admin/tables/helloworld.php](#)
- [admin/language/index.html](#)
- [admin/language/en-GB/index.html](#)
- [admin/language/en-GB/en-GB.com helloworld.ini](#)
- [admin/language/en-GB/en-GB.com helloworld.sys.ini](#)
- [media/index.html](#)
- [media/images/index.html](#)
- [media/images/Tux-16x16.png](#)
- [media/images/Tux-48x48.png](#)

Create a compressed file of this directory archive or directly download the archive and install it using the extension manager of Joomla. You can add a menu item of this component using the menu manager in the backend.

helloworld.xml

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<extension type="component" version="3.0" method="upgrade">
```

```
  <name>COM_HELLOWORLD</name>
```

```
  <!-- The following elements are optional and free of formatting constraints -->
```

```
  <creationDate>January 2018</creationDate>
```

```
  <author>John Doe</author>
```

```
  <authorEmail>john.doe@example.org</authorEmail>
```

```
  <authorUrl>http://www.example.org</authorUrl>
```

```
  <copyright>Copyright Info</copyright>
```

```
  <license>License Info</license>
```

```
  <!-- The version string is recorded in the components table -->
```



```

<version>0.0.10</version>
<!-- The description is optional and defaults to the name -->
<description>COM_HELLOWORLD_DESCRIPTION</description>

<install> <!-- Runs on install -->
    <sql>
        <file driver="mysql" charset="utf8">sql/install.mysql.utf8.sql</file>
    </sql>
</install>
<uninstall> <!-- Runs on uninstall -->
    <sql>
        <file driver="mysql" charset="utf8">sql/uninstall.mysql.utf8.sql</file>
    </sql>
</uninstall>
<update> <!-- Runs on update; New since J2.5 -->
    <schemas>
        <schemapath type="mysql">sql/updates/mysql</schemapath>
    </schemas>
</update>

<!-- Site Main File Copy Section -->
<!-- Note the folder attribute: This attribute describes the folder
to copy FROM in the package to install therefore files copied
in this section are copied from /site/ in the package -->
<files folder="site">
    <filename>index.html</filename>
    <filename>helloworld.php</filename>
    <filename>controller.php</filename>
    <folder>views</folder>
    <folder>models</folder>
</files>

<languages folder="site/language">
    <language tag="en-GB">en-GB/en-GB.com_helloworld.ini</language>
</languages>

```

```

<media destination="com_helloworld" folder="media">
    <filename>index.html</filename>
    <folder>images</folder>
</media>

```

```

<administration>
    <!-- Administration Menu Section -->
    <menu link='index.php?option=com_helloworld' img='../media/com_helloworld/images/Tux-
16x16.png'>COM_HELLOWORLD_MENU</menu>
    <!-- Administration Main File Copy Section -->
    <!-- Note the folder attribute: This attribute describes the folder
to copy FROM in the package to install therefore files copied
in this section are copied from /admin/ in the package -->
    <files folder="admin">
        <!-- Admin Main File Copy Section -->
        <filename>index.html</filename>
        <filename>helloworld.php</filename>
        <filename>controller.php</filename>
        <!-- SQL files section -->
        <folder>sql</folder>
        <!-- tables files section -->
        <folder>tables</folder>
        <!-- models files section -->
        <folder>models</folder>
        <!-- views files section -->
        <folder>views</folder>
        <!-- controllers files section -->
        <folder>controllers</folder>
    </files>
    <languages folder="admin/language">
        <language tag="en-GB">en-GB/en-GB.com_helloworld.ini</language>
        <language tag="en-GB">en-GB/en-GB.com_helloworld.sys.ini</language>
    </languages>
</administration>

```

</extension>

Adding Verifications in Joomla:

This tutorial is part of the [Developing an MVC Component for Joomla! 3.2](#) tutorial. You are encouraged to read the previous parts of the tutorial before reading this. See [Form validation](#) for some more general info on form validation (rules).

You can watch a video associated with this step at [Step 11, adding verifications](#). This and subsequent videos are recorded in HD; click on the YouTube settings wheel to view at best resolution.

Verifying the form (client side)

See [Client-side form validation](#) for further tutorial information relating to client-side verification. Forms can be verified on the client side using javascript code. In the [admin/views/helloworld/tmpl/edit.php](#) file, put these lines: `admin/views/helloworld/tmpl/edit.php`

```
<?php
/**
 * @package Joomla.Administrator
 * @subpackage com_helloworld
 *
 * @copyright Copyright (C) 2005 - 2018 Open Source Matters, Inc. All rights reserved.
 * @license GNU General Public License version 2 or later; see LICENSE.txt
 */

// No direct access
defined('_JEXEC') or die('Restricted access');
JHtml::_('behavior.formvalidator');
?>
<form action="<?php echo JRoute::_('index.php?option=com_helloworld&layout=edit&id=' . (int) $this->item->id); ?>"
method="post" name="adminForm" id="adminForm" class="form-validate">
  <div class="form-horizontal">
    <fieldset class="adminform">
      <legend><?php echo JText::_('COM_HELLOWORLD_HELLOWORLD_DETAILS'); ?></legend>
      <div class="row-fluid">
        <div class="span6">
          <?php foreach ($this->form->getFieldset() as $field): ?>
            <div class="control-group">
              <div class="control-label"><?php echo $field->label; ?></div>
              <div class="controls"><?php echo $field->input; ?></div>
            </div>
          <?php endforeach; ?>
        </div>
      </div>
    </fieldset>
    <input type="hidden" name="task" value="helloworld.edit" />
    <?php echo JHtml::_('form.token'); ?>
  </form>
```

You may have noted that the html form contained in the [admin/views/helloworld/tmpl/edit.php](#) file now has the `form-validate` css class. And that we added a `JHtml::_('behavior.formvalidator');` call to tell Joomla to use its javascript form validation.

Modify the `admin/models/forms/helloworld.xml` file to indicate that the `greeting` field has to be verified:

```
<?xml version="1.0" encoding="utf-8"?>
<form
  addrulepath="/administrator/components/com_helloworld/models/rules"
>
  <fieldset>
    <field
      name="id"
      type="hidden"
    />
    <field
      name="greeting"
      type="text"
      label="COM_HELLOWORLD_HELLOWORLD_GREETING_LABEL"
      description="COM_HELLOWORLD_HELLOWORLD_GREETING_DESC"
      size="40"
      class="inputbox validate-greeting"
    />
  </fieldset>
</form>
```

```

        validate="greeting"
        required="true"
        default=""
    />

```

```
</fieldset>
```

```
</form>
```

Note for the moment that the css class is now *"inputbox validate-greeting"* and that the attribute *required* is set to *true*. It means that this field is required and has to be verified by a handler of the form validator framework of Joomla.

With your favourite file manager and editor put a file `admin/models/forms/helloworld.js` containing:

```
admin/models/forms/helloworld.js
```

```

jQuery(function() {
    document.formvalidator.setHandler('greeting',
        function (value) {
            regex=/^[^0-9]+$;/
            return regex.test(value);
        });
});

```

It adds a handler to the form validator of Joomla for fields having the *"validate-greeting"* css class. Each time the *greeting* field is modified, the handler will be executed to verify its validity (no digits). The final step is to verify the form when the save button is clicked. With your favourite file manager and editor put a file `admin/views/helloworld/submitbutton.js` containing:

```
admin/views/helloworld/submitbutton.js
```

```

Joomla.submitbutton = function(task)
{
    if (task == "")
    {
        return false;
    }
    else
    {
        var isValid=true;
        var action = task.split('.');
        if (action[1] != 'cancel' && action[1] != 'close')
        {
            var forms = jQuery('form.form-validate');
            for (var i = 0; i < forms.length; i++)
            {
                if (!document.formvalidator.isValid(forms[i]))
                {
                    isValid = false;
                    break;
                }
            }
            if (isValid)
            {
                Joomla.submitform(task);
                return true;
            }
            else
            {
                alert(Joomla.JText._('COM_HELLOWORLD_HELLOWORLD_ERROR_UNACCEPTABLE',
                    'Some values are unacceptable'));
                return false;
            }
        }
    }
}

```

This function will verify that all forms which have the *"form-validate"* css class are valid. Note that it will display an alert message translated by the Joomla framework.

The *HelloWorldViewHelloWorld* view class has to be modified to use these javascript files:

```
admin/views/helloworld/view.html.php
```

```

<?php
/**
 * @package    Joomla.Administrator
 * @subpackage com_helloworld

```

```

*
* @copyright Copyright (C) 2005 - 2018 Open Source Matters, Inc. All rights reserved.
* @license GNU General Public License version 2 or later; see LICENSE.txt
*/

// No direct access to this file
defined('_JEXEC') or die('Restricted access');

/**
 * HelloWorld View
 *
 * @since 0.0.1
 */
class HelloWorldViewHelloWorld extends JViewLegacy
{
    /**
     * View form
     *
     * @var form
     */
    protected $form = null;

    /**
     * Display the Hello World view
     *
     * @param string $tpl The name of the template file to parse; automatically searches
through the template paths.
     *
     * @return void
     */
    public function display($tpl = null)
    {
        // Get the Data
        $this->form = $this->get('Form');
        $this->item = $this->get('Item');
        $this->script = $this->get('Script');

        // Check for errors.
        if (count($errors = $this->get('Errors')))
        {
            JError::raiseError(500, implode('<br />', $errors));

            return false;
        }

        // Set the toolbar
        $this->addToolBar();

        // Display the template
        parent::display($tpl);

        // Set the document
        $this->setDocument();
    }

    /**
     * Add the page title and toolbar.
     *
     * @return void
     *
     * @since 1.6
     */
    protected function addToolBar()
    {
        $input = JFactory::getApplication()->input;

        // Hide Joomla Administrator Main menu
        $input->set('hidemainmenu', true);
    }
}

```

```

        $isNew = ($this->item->id == 0);

        if ($isNew)
        {
            $title = JText::_('COM_HELLOWORLD_MANAGER_HELLOWORLD_NEW');
        }
        else
        {
            $title = JText::_('COM_HELLOWORLD_MANAGER_HELLOWORLD_EDIT');
        }

        JToolBarHelper::title($title, 'helloworld');
        JToolBarHelper::save('helloworld.save');
        JToolBarHelper::cancel(
            'helloworld.cancel',
            $isNew ? 'JTOOLBAR_CANCEL' : 'JTOOLBAR_CLOSE'
        );
    }
    /**
     * Method to set up the document properties
     *
     * @return void
     */
    protected function setDocument()
    {
        $isNew = ($this->item->id < 1);
        $document = JFactory::getDocument();
        $document->setTitle($isNew ? JText::_('COM_HELLOWORLD_HELLOWORLD_CREATING') :
        JText::_('COM_HELLOWORLD_HELLOWORLD_EDITING'));
        $document->addScript(JURI::root() . $this->script);
        $document->addScript(JURI::root() . "/administrator/components/com_helloworld
            ."/views/helloworld/submitbutton.js");
        JText::script('COM_HELLOWORLD_HELLOWORLD_ERROR_UNACCEPTABLE');
    }
}

```

This view now:

- verifies if the model has no error;
- adds two javascript files;
- injects javascript translation using the Joomla `JText::script` function.

The final step is to implement a `getScript` function in the *HelloWorldModelHelloWorld* model:

`admin/models/helloworld.php`

```

<?php
/**
 * @package   Joomla.Administrator
 * @subpackage com_helloworld
 *
 * @copyright  Copyright (C) 2005 - 2018 Open Source Matters, Inc. All rights reserved.
 * @license    GNU General Public License version 2 or later; see LICENSE.txt
 */

// No direct access to this file
defined('_JEXEC') or die('Restricted access');

/**
 * HelloWorld Model
 *
 * @since 0.0.1
 */
class HelloWorldModelHelloWorld extends JModelAdmin
{
    /**
     * Method to get a table object, load it if necessary.
     *
     * @param string $type The table name. Optional.
     * @param string $prefix The class prefix. Optional.
     */
}

```

```

* @param array $config Configuration array for model. Optional.
*
* @return JTable A JTable object
*
* @since 1.6
*/
public function getTable($type = 'HelloWorld', $prefix = 'HelloWorldTable', $config = array())
{
    return JTable::getInstance($type, $prefix, $config);
}

/**
 * Method to get the record form.
 *
 * @param array $data Data for the form.
 * @param boolean $loadData True if the form is to load its own data (default case), false if
not.
 *
 * @return mixed A JForm object on success, false on failure
 *
 * @since 1.6
 */
public function getForm($data = array(), $loadData = true)
{
    // Get the form.
    $form = $this->loadForm(
        'com_helloworld.helloworld',
        'helloworld',
        array(
            'control' => 'jform',
            'load_data' => $loadData
        )
    );

    if (empty($form))
    {
        return false;
    }

    return $form;
}

/**
 * Method to get the script that have to be included on the form
 *
 * @return string Script files
 */
public function getScript()
{
    return 'administrator/components/com_helloworld/models/forms/helloworld.js';
}

/**
 * Method to get the data that should be injected in the form.
 *
 * @return mixed The data for the form.
 *
 * @since 1.6
 */
protected function loadFormData()
{
    // Check the session for previously entered form data.
    $data = JFactory::getApplication()->getUserState(
        'com_helloworld.edit.helloworld.data',
        array()
    );

    if (empty($data))
    {

```

```

        $data = $this->getItem();
    }

    return $data;
}
}

```

Verifying the form (server side)

See [Server-side form validation](#) for further tutorial information relating to server-side verification. Verifying the form on the server side is done by inheritance of `JControllerForm` class. We have specified in the [admin/models/forms/helloworld.xml](#) file that the validate server function will use a `greeting.php` file.

With your favourite file manager and editor, put a `admin/models/rules/greeting.php` file containing:

```
admin/models/rules/greeting.php
```

```

<?php
/**
 * @package    Joomla.Administrator
 * @subpackage com_helloworld
 *
 * @copyright  Copyright (C) 2005 - 2018 Open Source Matters, Inc. All rights reserved.
 * @license    GNU General Public License version 2 or later; see LICENSE.txt
 */

// No direct access to this file
defined('_JEXEC') or die('Restricted access');

/**
 * Form Rule class for the Joomla Framework.
 */
class JFormRuleGreeting extends JFormRule
{
    /**
     * The regular expression.
     */
    * @access      protected
    * @var          string
    * @since        2.5
    */
    protected $regex = '^[^0-9]+$';
}

```

Adding Categories:

Modifying the SQL

In order to manage categories, we have to change the SQL tables. With your favourite editor, modify `admin/sql/install.mysql.utf8.sql` and put these lines:

```
admin/sql/install.mysql.utf8.sql
```

```

DROP TABLE IF EXISTS `#__helloworld`;

CREATE TABLE `#__helloworld` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `greeting` VARCHAR(25) NOT NULL,
  `published` tinyint(4) NOT NULL,
  `catid` int(11) NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`)
)
ENGINE = MyISAM
AUTO_INCREMENT = 0
DEFAULT CHARSET = utf8;

```



```
INSERT INTO `#__helloworld` (`greeting`) VALUES
```

```
('Hello World!'),
```

```
('Good bye World!');
```

```
admin/sql/updates/mysql/0.0.12.sql
```

```
ALTER TABLE `#__helloworld` ADD `catid` int(11) NOT NULL DEFAULT '0';
```

Modifying the form

A HelloWorld message can now belong to a category. We have to modify the editing form. In the `admin/models/forms/helloworld.xml` file, put these lines:

```
admin/models/forms/helloworld.xml
```

```
<?xml version="1.0" encoding="utf-8"?>
<form
    addrulepath="/administrator/components/com_helloworld/models/rules"
>
    <fieldset>
        <field
            name="id"
            type="hidden"
        />
        <field
            name="greeting"
            type="text"
            label="COM_HELLOWORLD_HELLOWORLD_GREETING_LABEL"
            description="COM_HELLOWORLD_HELLOWORLD_GREETING_DESC"
            size="40"
            class="inputbox validate-greeting"
            validate="greeting"
            required="true"
            default=""
        />
        <field
            name="catid"
            type="category"
            extension="com_helloworld"
            class="inputbox"
            default=""
            label="COM_HELLOWORLD_HELLOWORLD_FIELD_CATID_LABEL"
            description="COM_HELLOWORLD_HELLOWORLD_FIELD_CATID_DESC"
            required="true"
        />
        <option value="0">JOPTION_SELECT_CATEGORY</option>
    </fieldset>
</form>
```

Note that the category can be 0 (representing no category).

Modifying the menu type

The HelloWorld menu type displays a drop down list of all messages. If the message is categorized, we have to add the category in this display.

In the `admin/models/fields/helloworld.php` file, put these lines:

```
admin/models/fields/helloworld.php
```

```
<?php
/**
 * @package Joomla.Administrator
 * @subpackage com_helloworld
 *
 * @copyright Copyright (C) 2005 - 2015 Open Source Matters, Inc. All rights reserved.
 * @license GNU General Public License version 2 or later; see LICENSE.txt
 */

// No direct access to this file
defined('_JEXEC') or die('Restricted access');

JFormHelper::loadFieldClass('list');

/**
 * HelloWorld Form Field class for the HelloWorld component
 */
```

```

* @since 0.0.1
*/
class JFormFieldHelloWorld extends JFormFieldList
{
    /**
     * The field type.
     *
     * @var string
     */
    protected $type = 'HelloWorld';

    /**
     * Method to get a list of options for a list input.
     *
     * @return array An array of JHtml options.
     */
    protected function getOptions()
    {
        $db = JFactory::getDBO();
        $query = $db->getQuery(true);
        $query->select('#__helloworld.id as id,greeting,#__categories.title as category,catid');
        $query->from('#__helloworld');
        $query->leftJoin('#__categories on catid=#__categories.id');
        // Retrieve only published items
        $query->where('#__helloworld.published = 1');
        $db->setQuery((string) $query);
        $messages = $db->loadObjectList();
        $options = array();

        if ($messages)
        {
            foreach ($messages as $message)
            {
                $options[] = JHtml::_('select.option', $message->id, $message->greeting .
                    ($message->catid ? ' (' . $message->category . ')' : ''));
            }
        }

        $options = array_merge(parent::getOptions(), $options);

        return $options;
    }
}

```

It will now display the category between parenthesis. Note: We have added a where clause too to filter out unpublished items.

Managing the submenu

The `com_categories` component allows to set the submenu using a helper file. With your favourite file manager and editor, put a `admin/helpers/helloworld.php` file containing these lines:

```

admin/helpers/helloworld.php

<?php
/**
 * @package Joomla.Administrator
 * @subpackage com_helloworld
 *
 * @copyright Copyright (C) 2005 - 2015 Open Source Matters, Inc. All rights reserved.
 * @license GNU General Public License version 2 or later; see LICENSE.txt
 */

// No direct access to this file
defined('_JEXEC') or die('Restricted access');

/**
 * HelloWorld component helper.
 *
 * @param string $submenu The name of the active view.

```

```

*
* @return void
*
* @since 1.6
*/
abstract class HelloWorldHelper extends JHelperContent
{
    /**
     * Configure the Linkbar.
     *
     * @return Bool
     */

    public static function addSubmenu($submenu)
    {
        JHtmlSidebar::addEntry(
            JText::_('COM_HELLOWORLD_SUBMENU_MESSAGES'),
            'index.php?option=com_helloworld',
            $submenu == 'helloworlds'
        );

        JHtmlSidebar::addEntry(
            JText::_('COM_HELLOWORLD_SUBMENU_CATEGORIES'),
            'index.php?option=com_categories&view=categories&extension=com_helloworld',
            $submenu == 'categories'
        );

        // Set some global property
        $document = JFactory::getDocument();
        $document->addStyleDeclaration('.icon-48-helloworld ' .
                                                                    '{background-
image: url(..../media/com_helloworld/images/tux-48x48.png);}');
        if ($submenu == 'categories')
        {
            $document->setTitle(JText::_('COM_HELLOWORLD_ADMINISTRATION_CATEGORIES'));
        }
    }
}

```

NOTE: You MUST use your component name (without com_) for the helper file name, or else your submenus won't show in category view.

To import the helper class, put these lines in the `admin/helloworld.php` file:

```

admin/helloworld.php
<?php
/**
 * @package Joomla.Administrator
 * @subpackage com_helloworld
 *
 * @copyright Copyright (C) 2005 - 2015 Open Source Matters, Inc. All rights reserved.
 * @license GNU General Public License version 2 or later; see LICENSE.txt
 */

// No direct access to this file
defined('_JEXEC') or die('Restricted access');

// Set some global property
$document = JFactory::getDocument();
$document->addStyleDeclaration('.icon-helloworld {background-image: url(..../media/com_helloworld/images/tux-
16x16.png);}');

// Require helper file
JLoader::register('HelloWorldHelper', JPATH_COMPONENT . '/helpers/helloworld.php');

// Get an instance of the controller prefixed by HelloWorld
$controller = JControllerLegacy::getInstance('HelloWorld');

// Perform the Request task

```

```
$input = JFactory::getApplication()->input;
$controller->execute($input->getCmd('task'));
```

```
// Redirect if set by the controller
$controller->redirect();
```

This function will be automatically called by the *com_categories* component. Note that it will:

- change the submenu
- change some css properties (for displaying icons)
- change the browser title if the submenu is *categories*
- change the title and add a *preferences* button

The *.icon-48-helloworld* css class is now set in the *addSubmenu* function. We have now to call this function in the helloworlds view:

```
admin/views/helloworlds/view.html.php
```

```
<?php
/**
 * @package Joomla.Administrator
 * @subpackage com_helloworld
 *
 * @copyright Copyright (C) 2005 - 2015 Open Source Matters, Inc. All rights reserved.
 * @license GNU General Public License version 2 or later; see LICENSE.txt
 */

// No direct access to this file
defined('_JEXEC') or die('Restricted access');

/**
 * HelloWorlds View
 *
 * @since 0.0.1
 */
class HelloWorldViewHelloWorlds extends JViewLegacy
{
    /**
     * Display the Hello World view
     *
     * @param string $tpl The name of the template file to parse; automatically searches through the template paths.
     *
     * @return void
     */

    function display($tpl = null)
    {
        // Get application
        $app = JFactory::getApplication();
        $context = "helloworld.list.admin.helloworld";
        // Get data from the model
        $this->items = $this->get('Items');
        $this->pagination = $this->get('Pagination');
        $this->state = $this->get('State');
        $this->filter_order = $app->getUserStateFromRequest($context.'filter_order', 'filter_order', 'greeting',
'cmd');
        $this->filter_order_Dir = $app->getUserStateFromRequest($context.'filter_order_Dir', 'filter_order_Dir',
'asc', 'cmd');

        $this->filterForm = $this->get('FilterForm');
        $this->activeFilters = $this->get('ActiveFilters');

        // Check for errors.
        if (count($errors = $this->get('Errors')))
        {
            JError::raiseError(500, implode('<br />', $errors));

            return false;
        }

        // Set the submenu
```

```
HelloWorldHelper::addSubmenu('helloworlds');
```

```
// Set the toolbar and number of found items
$this->addToolBar();
```

```
// Display the template
parent::display($tpl);
```

```
// Set the document
$this->setDocument();
```

```
}
```

```
/**
```

```
 * Add the page title and toolbar.
```

```
 *
```

```
 * @return void
```

```
 *
```

```
 * @since 1.6
```

```
 */
```

```
protected function addToolBar()
```

```
{
```

```
    $title = JText::_('COM_HELLOWORLD_MANAGER_HELLOWORLDS');
```

```
    if ($this->pagination->total)
```

```
    {
```

```
        $title .= "<span style='font-size: 0.5em; vertical-align: middle;'>(" . $this->pagination->total .
```

```
)</span>";
```

```
    }
```

```
    JToolBarHelper::title($title, 'helloworld');
```

```
    JToolBarHelper::addNew('helloworld.add');
```

```
    JToolBarHelper::editList('helloworld.edit');
```

```
    JToolBarHelper::deleteList("", 'helloworlds.delete');
```

```
}
```

```
/**
```

```
 * Method to set up the document properties
```

```
 *
```

```
 * @return void
```

```
 */
```

```
protected function setDocument()
```

```
{
```

```
    $document = JFactory::getDocument();
```

```
    $document->setTitle(JText::_('COM_HELLOWORLD_ADMINISTRATION'));
```

```
}
```

```
}
```

Adding some ACL

```
admin/access.xml
```

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<access component="com_helloworld">
```

```
    <section name="component">
```

```
        <action name="core.admin" title="JACTION_ADMIN" description="JACTION_ADMIN_COMPONENT_DESC"
```

```
    />
```

```
        <action name="core.manage" title="JACTION_MANAGE"
```

```
description="JACTION_MANAGE_COMPONENT_DESC" />
```

```
        <action name="core.create" title="JACTION_CREATE"
```

```
description="JACTION_CREATE_COMPONENT_DESC" />
```

```
        <action name="core.delete" title="JACTION_DELETE"
```

```
description="JACTION_DELETE_COMPONENT_DESC" />
```

```
        <action name="core.edit" title="JACTION_EDIT" description="JACTION_EDIT_COMPONENT_DESC" />
```

```
    </section>
```

```
    <section name="message">
```

```
        <action name="core.delete" title="JACTION_DELETE"
```

```
description="COM_HELLOWORLD_ACCESS_DELETE_DESC" />
```

```
        <action name="core.edit" title="JACTION_EDIT" description="COM_HELLOWORLD_ACCESS_EDIT_DESC"
```

```
    />
```

```
    </section>
```

</access>

NOTE: If you don't add this file, buttons "New" "Edit" and ... don't show in category view. For more information, read section Adding ACL on top of the page.

Adding some translation strings

Some strings have to be translated. In the `admin/language/en-GB/en-GB.com_helloworld.ini` file, put these lines:

```
; Joomla! Project
; Copyright (C) 2005 - 2015 Open Source Matters. All rights reserved.
; License GNU General Public License version 2 or later; see LICENSE.txt, see LICENSE.php
; Note : All ini files need to be saved as UTF-8

COM_HELLOWORLD_ADMINISTRATION="HelloWorld - Administration"
COM_HELLOWORLD_ADMINISTRATION_CATEGORIES="HelloWorld - Categories"
COM_HELLOWORLD_NUM="#"
COM_HELLOWORLD_HELLOWORLDS_FILTER="Filters"
COM_HELLOWORLD_PUBLISHED="Published"
COM_HELLOWORLD_HELLOWORLDS_NAME="Name"
COM_HELLOWORLD_ID="Id"

COM_HELLOWORLD_HELLOWORLD_CREATING="HelloWorld - Creating"
COM_HELLOWORLD_HELLOWORLD_DETAILS="Details"
COM_HELLOWORLD_HELLOWORLD_EDITING="HelloWorld - Editing"
COM_HELLOWORLD_HELLOWORLD_ERROR_UNACCEPTABLE="Some values are unacceptable"
COM_HELLOWORLD_HELLOWORLD_FIELD_CATID_DESC="The category the messages belongs to"
COM_HELLOWORLD_HELLOWORLD_FIELD_CATID_LABEL="Category"
COM_HELLOWORLD_HELLOWORLD_FIELD_GREETING_DESC="This message will be displayed"
COM_HELLOWORLD_HELLOWORLD_FIELD_GREETING_LABEL="Message"
COM_HELLOWORLD_HELLOWORLD_HEADING_GREETING="Greeting"
COM_HELLOWORLD_HELLOWORLD_HEADING_ID="Id"
COM_HELLOWORLD_MANAGER_HELLOWORLD_EDIT="HelloWorld manager: Edit Message"
COM_HELLOWORLD_MANAGER_HELLOWORLD_NEW="HelloWorld manager: New Message"
COM_HELLOWORLD_MANAGER_HELLOWORLDS="HelloWorld manager"
COM_HELLOWORLD_EDIT_HELLOWORLD="Edit message"
COM_HELLOWORLD_N_ITEMS_DELETED_1="One message deleted"
COM_HELLOWORLD_N_ITEMS_DELETED_MORE="%d messages deleted"
COM_HELLOWORLD_N_ITEMS_PUBLISHED="%d message(s) published"
COM_HELLOWORLD_N_ITEMS_UNPUBLISHED="%d message(s) unpublished"
COM_HELLOWORLD_HELLOWORLD_GREETING_LABEL="Greeting"
COM_HELLOWORLD_HELLOWORLD_GREETING_DESC="Add Hello World Greeting"
COM_HELLOWORLD_SUBMENU_MESSAGES="Messages"
COM_HELLOWORLD_SUBMENU_CATEGORIES="Categories"
```

Adding Configuration:

Adding configuration parameters

The Joomla framework allows the use of parameters stored in each component. With your favourite file manager and editor, put a file `admin/config.xml` file containing these lines:

```
admin/config.xml

<?xml version="1.0" encoding="utf-8"?>
<config>
    <fieldset
        name="greetings"
        label="COM_HELLOWORLD_CONFIG_GREETING_SETTINGS_LABEL"
        description="COM_HELLOWORLD_CONFIG_GREETING_SETTINGS_DESC"
    >

        <field
            name="show_category"
            type="radio"
            label="COM_HELLOWORLD_HELLOWORLD_FIELD_SHOW_CATEGORY_LABEL"
            description="COM_HELLOWORLD_HELLOWORLD_FIELD_SHOW_CATEGORY_DESC"
            default="0"
        >
            <option value="0">JHIDE</option>
            <option value="1">JSHOW</option>
        </field>
    </fieldset>
```

</config>

This file will be read by the *com_config* component of the Joomla core. For the moment, we defined only one parameter: is the category title displayed or not in the frontend.

The best way to set the parameters is to put a *Preferences* button in a toolbar.

With your favourite editor, put these lines in `admin/views/helloworlds/view.html.php`.

`admin/views/helloworlds/view.html.php`

```
<?php
/**
 * @package      Joomla.Administrator
 * @subpackage    com_helloworld
 *
 * @copyright     Copyright (C) 2005 - 2015 Open Source Matters, Inc. All rights reserved.
 * @license       GNU General Public License version 2 or later; see LICENSE.txt
 */

// No direct access to this file
defined('_JEXEC') or die('Restricted access');

/**
 * HelloWorlds View
 *
 * @since 0.0.1
 */
class HelloWorldViewHelloWorlds extends JViewLegacy
{
    /**
     * Display the Hello World view
     *
     * @param string $tpl The name of the template file to parse; automatically searches through the template
     paths.
     *
     * @return void
     */
    function display($tpl = null)
    {
        // Get application
        $app = JFactory::getApplication();
        $context = "helloworld.list.admin.helloworld";
        // Get data from the model
        $this->items = $this->get('Items');
        $this->pagination = $this->get('Pagination');
        $this->state = $this->get('State');
        $this->filter_order = $app->getUserStateFromRequest($context.'filter_order', 'filter_order',
'greeting', 'cmd');
        $this->filter_order_Dir = $app->getUserStateFromRequest($context.'filter_order_Dir',
'filter_order_Dir', 'asc', 'cmd');
        $this->filterForm = $this->get('FilterForm');
        $this->activeFilters = $this->get('ActiveFilters');

        // Check for errors.
        if (count($errors = $this->get('Errors')))
        {
            JError::raiseError(500, implode('<br />', $errors));

            return false;
        }

        // Set the submenu
        HelloWorldHelper::addSubmenu('helloworlds');

        // Set the toolbar and number of found items
        $this->addToolBar();

        // Display the template
        parent::display($tpl);

        // Set the document
        $this->setDocument();
    }

    /**
     * Add the page title and toolbar.
     *
     * @return void
     */
}
```



```

*
* @since 1.6
*/
protected function addToolBar()
{
    $title = JText::_('COM_HELLOWORLD_MANAGER_HELLOWORLDS');

    if ($this->pagination->total)
    {
        $title .= "<span style='font-size: 0.5em; vertical-align: middle;'>(" . $this->
        pagination->total . ")</span>";
    }

    JToolBarHelper::title($title, 'helloworld');
    JToolBarHelper::addNew('helloworld.add');
    JToolBarHelper::editList('helloworld.edit');
    JToolBarHelper::deleteList('', 'helloworlds.delete');
    JToolBarHelper::preferences('com_helloworld');
}
/**
 * Method to set up the document properties
 *
 * @return void
 */
protected function setDocument()
{
    $document = JFactory::getDocument();
    $document->setTitle(JText::_('COM_HELLOWORLD_ADMINISTRATION'));
}
}

```

Using configuration parameters as default value

We want to define this parameter individually on all HelloWorld data. With your favourite editor, put these lines into the `admin/models/forms/helloworld.xml`

`admin/models/forms/helloworld.xml`

```

<?xml version="1.0" encoding="utf-8"?>
<form

    addrulepath="/administrator/components/com_helloworld/models/rules"
>
    <fieldset
        name="details"
        label="COM_HELLOWORLD_HELLOWORLD_DETAILS"
    >
        <field
            name="id"
            type="hidden"
        />
        <field
            name="greeting"
            type="text"
            label="COM_HELLOWORLD_HELLOWORLD_GREETING_LABEL"
            description="COM_HELLOWORLD_HELLOWORLD_GREETING_DESC"
            size="40"
            class="inputbox validate-greeting"
            validate="greeting"
            required="true"
            default=""
        />
        <field
            name="catid"
            type="category"
            extension="com_helloworld"
            class="inputbox"
            default=""
            label="COM_HELLOWORLD_HELLOWORLD_FIELD_CATID_LABEL"
            description="COM_HELLOWORLD_HELLOWORLD_FIELD_CATID_DESC"
            required="true"
        >
            <option value="0">JOPTION_SELECT_CATEGORY</option>

```

```

        </field>
    </fieldset>
    <fields name="params">
        <fieldset
            name="params"
            label="JGLOBAL_FIELDSET_DISPLAY_OPTIONS"
        >
            <field
                name="show_category"
                type="list"

                label="COM_HELLOWORLD_HELLOWORLD_FIELD_SHOW_CATEGORY_LABEL"

                description="COM_HELLOWORLD_HELLOWORLD_FIELD_SHOW_CATEGORY_DESC"
                default=""
            >
                <option value="">JGLOBAL_USE_GLOBAL</option>
                <option value="0">JHIDE</option>
                <option value="1">JSHOW</option>
            </field>
        </fieldset>
    </fields>
</form>

```

We define the same parameter for each message with an additional value: *Use global*.

Note: The *details* fieldset now has a label too (the same that was set in the *edit* view before).

Modifying the SQL

Data now contains a new parameter: *params*. The SQL structure has to be modified. With your favourite editor, put these lines into `admin/sql/install.mysql.utf8.sql`:

```

admin/sql/install.mysql.utf8.sql

DROP TABLE IF EXISTS `#__helloworld`;

CREATE TABLE `#__helloworld` (
    `id` INT(11) NOT NULL AUTO_INCREMENT,
    `greeting` VARCHAR(25) NOT NULL,
    `published` tinyint(4) NOT NULL,
    `catid` int(11) NOT NULL DEFAULT '0',
    `params` VARCHAR(1024) NOT NULL DEFAULT '',
    PRIMARY KEY (`id`)
)
ENGINE =MyISAM
AUTO_INCREMENT =0
DEFAULT CHARSET =utf8;

INSERT INTO `#__helloworld` (`greeting`) VALUES
('Hello World!'),
('Good bye World!');

```

With your favourite editor, put these lines into `admin/sql/updates/mysql/0.0.13.sql`:

```

admin/sql/updates/mysql/0.0.13.sql

ALTER TABLE `#__helloworld` ADD `params` VARCHAR(1024) NOT NULL DEFAULT '';

```

The `TableHelloWorld` has to be modified in order to deal with these parameters: they will be stored in a JSON format and get in a `JParameter` class. We have to overload the *bind* and the *load* method. With your favourite editor, put these lines into `admin/tables/helloworld.php`.

```

admin/tables/helloworld.php

<?php
/**
 * @package      Joomla.Administrator
 * @subpackage   com_helloworld
 *
 * @copyright     Copyright (C) 2005 - 2015 Open Source Matters, Inc. All rights reserved.
 * @license      GNU General Public License version 2 or later; see LICENSE.txt
 */
// No direct access
defined('_JEXEC') or die('Restricted access');

/**
 * Hello Table class
 *
 * @since 0.0.1

```

```

*/
class HelloWorldTableHelloWorld extends JTable
{
    /**
     * Constructor
     *
     * @param JDatabaseDriver &$db A database connector object
     */
    function __construct(&$db)
    {
        parent::__construct('#__helloworld', 'id', $db);
    }

    /**
     * Overloaded bind function
     *
     * @param array $array named array
     * @return null|string null is operation was satisfactory, otherwise returns an error
     * @see JTable:bind
     * @since 1.5
     */
    public function bind($array, $ignore = '')
    {
        if (isset($array['params']) && is_array($array['params']))
        {
            // Convert the params field to a string.
            $parameter = new JRegistry;
            $parameter->loadArray($array['params']);
            $array['params'] = (string)$parameter;
        }
        return parent::bind($array, $ignore);
    }

    /**
     * Overloaded load function
     *
     * @param int $pk primary key
     * @param boolean $reset reset data
     * @return boolean
     * @see JTable:load
     */
    public function load($pk = null, $reset = true)
    {
        if (parent::load($pk, $reset))
        {
            // Convert the params field to a registry.
            $params = new JRegistry;
            $params->loadString($this->params, 'JSON');

            $this->params = $params;
            return true;
        }
        else
        {
            return false;
        }
    }
}

```

Modifying the backend

The backend edit view has to display the options to the administrator. With your favourite editor, put these lines into the `admin/views/helloworld/tmpl/edit.php` file:

`admin/views/helloworld/tmpl/edit.php`

```

<?php
/**
 * @package Joomla.Administrator
 * @subpackage com_helloworld
 *
 * @copyright Copyright (C) 2005 - 2015 Open Source Matters, Inc. All rights reserved.
 * @license GNU General Public License version 2 or later; see LICENSE.txt
 */

// No direct access
defined('_JEXEC') or die('Restricted access');
JHtml::_('behavior.formvalidation');
?>
<form action="<?php echo JRoute::_('index.php?option=com_helloworld&layout=edit&id=' . (int) $this->item->id); ?>"

```

GEETANJALI COLLEGE - RAJKOT

```
method="post" name="adminForm" id="adminForm" class="form-validate">
  <div class="form-horizontal">
    <?php foreach ($this->form->getFieldsets() as $name => $fieldset): ?>
      <fieldset class="adminform">
        <legend><?php echo JText::_($fieldset->label); ?></legend>
        <div class="row-fluid">
          <div class="span6">
            <?php foreach ($this->form->getFieldset($name) as
$fieldset): ?>
              <div class="control-group">
                <div class="control-label"><?php
echo $fieldset->label; ?></div>
                <div class="controls"><?php echo
$fieldset->input; ?></div>
              </div>
            <?php endforeach; ?>
          </div>
        </div>
      </fieldset>
    <?php endforeach; ?>
  </div>
  <input type="hidden" name="task" value="helloworld.edit" />
  <?php echo JText::_('form.token'); ?>
</form>
```

Note: The outer *foreach* retrieves all of the form fieldsets while the inner one gets the selected fieldset fields. This is the result of the `$this->form->getFieldsets()` command:

```
Array
(
    [details] => stdClass Object
        (
            [name] => details
            [label] => COM_HELLOWORLD_HELLOWORLD_DETAILS
            [description] =>
        )
    [params] => stdClass Object
        (
            [name] => params
            [label] => JGLOBAL_FIELDSET_DISPLAY_OPTIONS
            [description] =>
        )
)
```

Modifying the frontend

The frontend has to be modified according to the new *show_category* parameter.

We have to modify the model:

- it has to merge global parameters and individual parameters
- it has to provide the category

With your favourite editor, put these lines into the `site/models/helloworld.php` file:

```
site/models/helloworld.php

<?php
/**
 * @package Joomla.Administrator
 * @subpackage com_helloworld
 *
 * @copyright Copyright (C) 2005 - 2015 Open Source Matters, Inc. ALL rights reserved.
 * @license GNU General Public License version 2 or later; see LICENSE.txt
 */

// No direct access to this file
defined('_JEXEC') or die('Restricted access');

/**
 * HelloWorld Model
 *
 * @since 0.0.1
 */
class HelloWorldModelHelloWorld extends JModelItem
{
    /**
     * @var object item
     */
}
```

```
protected $item;
```

```
/**
 * Method to auto-populate the model state.
 *
 * This method should only be called once per instantiation and is designed
 * to be called on the first call to the getState() method unless the model
 * configuration flag to ignore the request is set.
 *
 * Note. Calling getState in this method will result in recursion.
 *
 * @return void
 * @since 2.5
 */
protected function populateState()
{
    // Get the message id
    $jinput = JFactory::getApplication()->input;
    $id      = $jinput->get('id', 1, 'INT');
    $this->setState('message.id', $id);

    // Load the parameters.
    $this->setState('params', JFactory::getApplication()->getParams());
    parent::populateState();
}

/**
 * Method to get a table object, load it if necessary.
 *
 * @param string $type The table name. Optional.
 * @param string $prefix The class prefix. Optional.
 * @param array $config Configuration array for model. Optional.
 *
 * @return JTable A JTable object
 *
 * @since 1.6
 */
public function getTable($type = 'HelloWorld', $prefix = 'HelloWorldTable', $config = array())
{
    return JTable::getInstance($type, $prefix, $config);
}

/**
 * Get the message
 * @return object The message to be displayed to the user
 */
public function getItem()
{
    if (!isset($this->item))
    {
        $id      = $this->getState('message.id');
        $db      = JFactory::getDbo();
        $query = $db->getQuery(true);
        $query->select('h.greeting, h.params, c.title as category')
            ->from('#__helloworld as h')
            ->leftJoin('#__categories as c ON h.catid=c.id')
            ->where('h.id=' . (int)$id);
        $db->setQuery((string)$query);

        if ($this->item = $db->loadObject())
        {
            // Load the JSON string
            $params = new JRegistry;
            $params->loadString($this->item->params, 'JSON');
            $this->item->params = $params;

            // Merge global params with item params
            $params = clone $this->getState('params');
            $params->merge($this->item->params);
            $this->item->params = $params;
        }
        return $this->item;
    }
}
```

The view has to ask the model for the category. With your favourite editor, put these lines into the `site/views/helloworld/view.html.php`

`site/views/helloworld/view.html.php`

```
<?php
/**
 * @package      Joomla.Administrator
 * @subpackage    com_helloworld
 *
 * @copyright     Copyright (C) 2005 - 2015 Open Source Matters, Inc. All rights reserved.
 * @license       GNU General Public License version 2 or later; see LICENSE.txt
 */

// No direct access to this file
defined('_JEXEC') or die('Restricted access');

/**
 * HTML View class for the HelloWorld Component
 *
 * @since 0.0.1
 */
class HelloWorldViewHelloWorld extends JViewLegacy
{
    /**
     * Display the Hello World view
     *
     * @param string $tpl The name of the template file to parse; automatically searches through the template
     paths.
     *
     * @return void
     */
    function display($tpl = null)
    {
        // Assign data to the view
        $this->item = $this->get('Item');

        // Check for errors.
        if (count($errors = $this->get('Errors')))
        {
            JLog::add(implode('<br />', $errors), JLog::WARNING, 'jerror');

            return false;
        }

        // Display the view
        parent::display($tpl);
    }
}
```

The layout can now display correctly the category or not. With your favourite editor, put these lines into `site/views/helloworld/tmpl/default.php`

`site/views/helloworld/tmpl/default.php`

```
<?php
/**
 * @package      Joomla.Administrator
 * @subpackage    com_helloworld
 *
 * @copyright     Copyright (C) 2005 - 2015 Open Source Matters, Inc. All rights reserved.
 * @license       GNU General Public License version 2 or later; see LICENSE.txt
 */

// No direct access to this file
defined('_JEXEC') or die('Restricted access');
?>
<h1><?php echo $this->item->greeting.((($this->item->category and $this->item->params->get('show_category'))
? (' ( '.$this->item->category.' ) : '' ); ?>
</h1>
```

Access Control List (ACL):

Access Control List or ACL is according to the [Wikipedia definition](#), "...ACL specifies which users or system processes are granted access to objects, as well as what operations are allowed to be performed on given objects." In the case of Joomla there are two separate aspects to its Access Control List which site administrators can control:

- **Which users can gain access to what parts of the website?** For example, will a given menu choice be visible for a given user? A registered user can view, but the public at large cannot. Perhaps the menu choice is hidden from all except an Editor user and higher.
- **What operations (or actions) can a user perform on any given object?** For example, can a user listed as an "Editor" submit an article or only edit an existing article. The ACL settings could allow submitting and editing, or allow a change an article's category, add tags or any combination.

The implementation of ACL in Joomla! 2.5 was substantially changed in the Joomla! 2.5 series which allowed for more flexibility in groups and permissions.

Adding ACL supporting plugins in your component

Action permissions for components in Joomla! 2.5 can be defined at up to 3 levels:

1. **Global Configuration:** determines the default permissions for each action and group.
2. **Component Options->Permissions:** can override the default permissions for our component
3. **Record Options->Permissions:** this is what we discuss in this tutorial.

The other level that you will see is **Category** which can override the default permissions for objects assigned to category. We can use this if we use category in our component. I separate this one from others as it's parent is **Global Configuration**.

Modify own database table

The own database table needs to have a column with an asset-ID. If this kind of column is missing in your own database table structure, then add something like this:

```
ALTER TABLE `#__asterman_units` ADD `asset_id` INT(255) UNSIGNED NOT NULL DEFAULT '0' AFTER `id`;
```

Using JTable To Handle Action Control Saving and Loading

The JTable class has some built-in support for ACL, to use this you have to add one integer field in your table named 'asset_id'. This field will be used to perform action control for your object. Do not confuse this with the 'access' field, which is only for access permission (read side) as Joomla! separates other actions from read access.

Normally, we have to override the __construct() method in our descendant class. We may also override the check() method to validate our data before saving to the database. For ACL action support we have to override these methods:-

- `public function bind($array, $ignore =)` to let JTable save rule data to Joomla's assets table.
- `protected function _getAssetName()` to give an asset name to our asset.
- `protected function _getAssetParentId($table = null, $id = null)` to give a parent id to our asset. Normally Joomla! uses root (1) as a parent. Generally we will change parent of node to our component. Doing all of these makes JTable handle storing and loading action control settings for us.

Add Rules Field to Your Form

Next, we have to add two fields to our form so we will have action control setting input in the view. The first one is 'asset_id' and it's type is hidden. This is same name as we provided in our table. The second one is 'rules' and it's type is rules. We have to provide this name as it will be processed by JTable's descendant class. Please see the JTable class for more detail.

For the rules field type, we have to provide the **component attribute** as our component name and **section attribute** according to the section defined in our access.xml. Please see the example below.

You will see that I have also created a new action for my component. The "asterman.edit.basic" action controls whether the user has rights to edit some part of the record or not.

Put Action Control In Action

Using action control in your code is easier than you think. We use JUser's method `authorise()` which requires two parameters; one is action and the second is asset name. Unlike our global component access control, here we refer to asset name per record, please see the JTable descendant class above.

```
JFactory::getUser()->authorise($action, $assetName);
```


Background

Joomla! Observer Implementation

Joomla! implements the Observer pattern at a global level through the JPlugin (Observer) and JEventDispatcher (Observable) classes. Anyone wanting to receive notification of events will create a plug-in that extends the JPlugin class. Subclasses of JPlugin will automatically register themselves to the global JEventDispatcher class when their plug-in category has been loaded (more on that later). The JEventDispatcher class is used as a dispatching mechanism that receives events from the communicators and forwards them on to the listeners that have been loaded. For a full explanation of this it is recommended to read the Plugin Developer Overview

Why Become a Communicator?

There may be times in your component's lifecycle when it would be nice to notify others that some action has taken place. For example, let's say you have a compact disk (CD) library component. You may decide that you would like to let others know when a new CD has been added to the library. In this case, you could document a well known event (onCdAddedToLibrary for example) and at the appropriate time, *trigger* the event passing in information about the new CD that was added to the library. All plug-ins that have implemented that event will be notified with the information and can handle it as they see fit. Instant communication!

Implementation

How to Become a Communicator

Since all the dispatching is handled by the Joomla! Core, it is easy to become a communicator. In fact, it's really just a matter of loading a certain set of plug-ins and calling the trigger method of the JEventDispatcher class.

You may wonder how to know which set of plug-ins to load. That's up to you. Plug-ins are managed at a group level that is defined in the plug-in's XML deployment file. There are eight predefined plug-in groups and each plug-in group is meant to handle a different set of tasks. For example, there is a search plug-in group that is meant to handle searching and a user plug-in group meant to handle user specific functions such as adding and removing a user from the Joomla! system. These plug-in groups are only loaded when they are needed, namely by the communicator. So you can create your own plug-in group and call it whatever you want. Because your component is well-defined, all listeners will know exactly which plug-in group and events they should be listening for.

How to Trigger Events

This leads us to the meat of this article: how to trigger events so implementing plug-ins can act on them. The first thing you need to do is to load your plug-in group. This is done via the following code: `JPluginHelper::importPlugin('myplugingroup');`

This will load all enabled plug-ins that have defined themselves as part of your group. The next thing you need to do is get an instance of the `JEventDispatcher` class:

```
$dispatcher=JEventDispatcher::getInstance();
```

Notice two things here. First, we are using the `getInstance()` method, not `new`, to create a new instance. That is because we need to get the global singleton instance of the `JEventDispatcher` object that contains a list of all the plug-ins available.

Next, we need to trigger our custom event:

```
$results=$dispatcher->trigger('onCdAddedToLibrary',array(&$artist,&$title));
```

Here we have triggered the event `onCdAddedToLibrary` and passed in the artist name and title of the track. Which and how many parameters you pass are up to you. Passing parameters by reference (using an `&`) allows the plug-ins to change the variables passed in. All plug-ins will receive these parameters, process them and optionally pass back information. The calling code can access the returned values via the array `$results` (each element corresponds to the result of one plugin).

Caveats

You Are Defining An API:

By offering events for plugins to respond to, you are effectively creating an API. Proper planning is extremely important – once you release your code, other developers will start to depend on your events' names and parameters. Changing them later will break compatibility with all the plugins that use them.

Load the Right Plugin Group

One thing to notice about the trigger method is that there is nothing defining which group of plug-ins should be notified. In actuality, all plug-ins that have been loaded are notified regardless of the group they are in. It's important to make sure you have an event name that does not conflict with any other plug-in group's event name. Most of the time this is not an issue because your component is the one that is loading the plug-in group, so you know which ones are loaded. However be aware that the system plug-in group is loaded very close to the beginning of the request. Make sure you don't have any event naming conflicts with the system events.

What is the Joomla! Framework ?

The Joomla! Framework™ is a new PHP framework (a collection of software libraries/packages) for writing web and command line applications in PHP, without the features and corresponding overhead found in the Joomla! Content Management System (CMS). It provides a structurally sound foundation, which is easy to adapt and easy to extend.

The Joomla! Framework is free and open source software, distributed under the GNU General Public License version 2 or later; and is comprised of code originally developed for the Joomla! CMS™.

The Joomla! Framework should not be confused with the hugely popular Joomla! CMS. It is important to remember that you do not need to install the Joomla! Framework to use the CMS, nor do you need to install the Joomla! CMS to use the Framework.

The new Joomla! Framework is now available to install via Composer and you can find the list of packages on [Packagist.org](https://packagist.org). There are plenty of avenues you can explore to get started on working with the Framework.

For the official docs, check out the README.md file found in each package. You can also review the [Joomla! Framework organization on GitHub](#).

Get the Sample Application

One of the easiest ways to get to know the Joomla! Framework is to start with looking at a sample application. This site is powered by the Joomla! Framework and serves as a great example of using the Framework.

1. Install Composer.
2. Download this [website application](#) repository from GitHub.
3. Run composer install.

4. View in your browser.

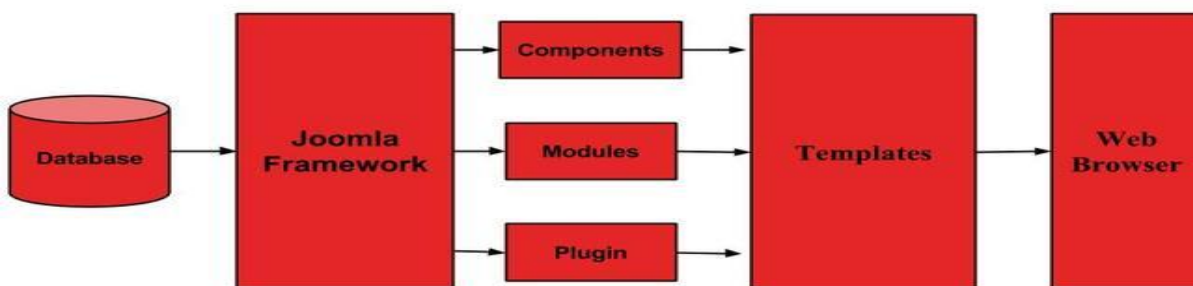
Why build the Joomla! Framework?

Separating the framework from the CMS was a big deal, and in retrospect, a smart decision. By separating the two code bases Joomla!® can now offer the stability the CMS requires while still taking advantage of current and modern trends in PHP level

What is the Framework good for?

- Building a RESTful web services platform
- Building both simple and complex command line tools
- Building next generation web applications

Framework Architecture



What does the Joomla! Framework mean for you?

Whether you are a Joomla! Extension developer looking to spread your wings & delve into developing standalone applications or a PHP coder looking for a stable lightweight framework for your next app, the Framework provides benefits for everyone.

I am already developing Extensions for the Joomla! CMS. Why should I consider using the Joomla! Framework?

You have invested significant time & effort in learning the Joomla! way of doing things & writing extensions. You can now apply this same knowledge within a non-CMS environment because the Joomla! Framework maintains a similar set of function, class, and method names.

It's important to keep in mind other situations, such as what happens when you land a project that needs a different type of application. Or when something doesn't really fit into the website or CMS box, or the CMS is simply too much overhead. There may be times when you need a simple tool to migrate data from one Business Intelligence system to another or a lightweight RESTful service to talk to a mobile application. These are all cases when a framework-based application will prove much more beneficial.

These are only a few of the many situations where a framework based solution would work far better than attempting to shoehorn a solution into the Joomla! CMS. The Joomla! Framework allows you to leverage all that Joomla! knowledge to build apps without the overhead of the CMS.

I know PHP already. Why should I use this framework?

If you are a strong coder looking for a lightweight framework that is easy to adapt and extend, you are in the right place.

The Joomla! Framework is available using Composer. This allows you to build projects from the many packages included in the Joomla! Framework as well as packages that best fit your needs from any of the other PHP frameworks that use Composer.

As a Joomla CMS User. How will I be affected?

The Joomla! CMS and the Joomla! Framework are currently developed independently and, thus, the Framework's launch will not have an immediate impact for you. However, there are already parts of the Framework integrated into the CMS (e.g. the Dependency Injection package, added in Joomla! 3.2), and more is coming. Expect to see some great extensions made available to you, built using the Framework!

The Joomla! Framework aims at getting the latest in PHP developments and features into Joomla! at the framework level. This allows the CMS to better focus its aims on providing the best features for its end users and staying ahead of the game.

Bottom line, you can rest assured that your favorite CMS will continue to have a strong, up-to-date and robust base that can evolve with the web.

Get Involved

Support

The Joomla! Framework incorporated support in a simple step-by-step process. You can quickly determine the best way to get help by following one of the options below:

First Option: Just fix it

Developers are encouraged to take an active approach with the Framework, including future updates and development. If you find a problem or bug and can fix the issue directly, then you can fix the issue and submit a pull request against the repository.

Not sure how to get started with submitting a pull request? You can view [pull request tutorials](#) in the Joomla! Documentation wiki.

Second Option: Report it

If you find an issue in a Framework package and cannot fix the issue directly, then the next step is to report the problem. Issues can be reported on the [Joomla! Framework organization](#) under the appropriate package. Submitting an issue here will help everyone track issues in a central location. Please be available to add comments and respond to questions related to your issue as other developers begin assisting in writing code to fix the issue.

More information regarding [reporting issues](#) can be found here.

Other Option: Learn it

Perhaps as you're working with the Framework you find a feature or an issue that you are simply unsure about how to use it. In this case, it would not be appropriate to attempt either of the steps listed above. You can take advantage of the [Joomla! Framework mailing list](#) where you can get help from other developers.

Be respectful and courteous in the mailing list and you will receive quick helpful feedback to help you better utilize the Framework to its full potential.

Joomla! 4 working group

The next major version for Joomla! has been on the radar for some time but lacking follow through, for reasons both organisational and technical. At JandBeyond, May/June 2015 in Prague the community took matters into their own hand and set up a "make it happen" session to discuss Joomla 4.

Requirements were discussed firstly from a user perspective and later from a technical perspective, drawing heavily on a series of [blog posts](#) by Nicholas Dionysopoulos.

Brainstorming, the requirements were divided into "must have" and "would like". The goal being a "must have list" that results in a Minimal Viable Product (MVP) for Joomla!. The least number of issues with the biggest impact, Items touching upon backward compatibility (BC), would go in the "must have" column. Items building upon new BC items would be moved to the "would like" column.

With any major release, backwards compatibility is and has been much discussed. For Joomla 4 we are aiming for a one-click migration. The one-click migration is a challenging objective for the Joomla core, but certainly for the extensions. We will need to put effort in communicating and training to make all of this happen.

The result of this community driven approach will need to be discussed and refined further, as we also need more information on architecture, UX/UI, and last but not least verification with the Marketing

Team. With the Marketing Team's help we want to ensure we make decisions on data/facts and not on feelings of a few individuals.

It was amazing to see how much can be accomplished in a few hours of face to face discussion regarding the perceived requirements.

As for the way forward, we are taking a quote out of Einstein's quote book

"Insanity: doing the same thing over and over again and expecting different results", we opt for a different more guided approach. We will use an agile way of working, having a planned list of features to work on during sprints (fixed periods of time) should allow us to have something to show for every couple of weeks, and report back to the community.

Having taken these results back to PLT it was decided to expand the "make it happen" initiative into a formal Joomla 4 working group <http://volunteers.joomla.org/working-groups/joomla-4-working-group> with a Glip team-room "Joomla 4 Working Group" to discuss.

We aim to be a collective of participants, we are no "academic debating group". Anybody willing and able to contribute is welcome to join, there are no secrets here, only doers. [Contact me](#) and we will take it from there.

As this is a large undertaking, parts of the may be split of in satellite groups. Groups that have/are been formed focus around Architecture (<http://volunteers.joomla.org/working-groups/joomla-4-architecture>) and UX/UI.

Why the Transformation in Joomla 4?

Though Joomla has been improving through the time, Joomla however is based on 10 years old technologies. So, it can't respond to many modern technologies. It's like an old-fashioned Joomla. You may defend that there are many people still using it, but please take note that there are not so many new users and new Joomla users from the market hasn't been expanding.

Furthermore, the outdated technologies prevent developers from doing many things to build great projects with it. So, Joomla must transform to catch up with the modern technology and be more responsive to the need of its users.

Improvements in Joomla Admin Workflow

The development team has been working on a new and useful Joomla admin dashboard. It's the very first thing users see after logging in the admin area so they are trying to make it better for users with useful, relevant information right in the dashboard.

Also there will be new onboarding process to help new Joomla users become familiar with Joomla much easier. Plus, Parameter Defaults Review is also going to be in the development team's consideration.

An Evolution in Joomla Code

Coding improvements in Joomla 4 will make you dance on your feet, my fellow developers! Code in Joomla core will be more testable (so less bugs, of course) and easier to maintain.

I can list out some improvements in coding you can expect in this awesome version: a new rewritten plugin system, Dependency Injection Containers to replace JFactory, Joomla framework to be used in more components (so less code to maintain later), Code Namespacing and Depreciation.

Orthogonal Component Structure

Orth...orthogono..what? What a long name to remember! Let's understand it this way instead:

Joomla 4 structure is the combination of vertical components like content or users and horizontal components like workflow, tagging, versioning. This means while we have the development in Joomla core, at the same time all the new features regarding functionality like publishing, tagging and versioning, etc. will be added automatically in Joomla and all are configurable.

You can learn more about this new structure [here](#).

Strict MVC Implementation

Decision:

The proposed command structure will be implemented. (see references; naming according to GOF Design Patterns)

A command decides which model(s) to use. It is responsible for making input available to the model and to add the output to the visitable output object graph. Thus, the model can be literally any object with public members - in theory even a J1.5 MVC triad or a non-Joomla solution.

The output graph is transformed into a streamable format (according to content negotiation), and - if appropriate - wrapped into a PSR-7 response object.

Reason:

The command/controller approach gives most possible flexibility for the implementation of models, so it is possible to integrate existing software. it allows to do proper CQRS, if wanted, by letting read ('Query') and write ('Command') commands use their own model.

The renderer approach allows to server any output channel (JSON, XML, HTML, PDF, ePub,) without the model or controller even knowing about that.