Exception Not Found

ASP.NET MVC DEMYSTIFIED ASP.NET MVC MODELSTATE SAMPLE PROJECT DEBUGGING

Explaining the ModelState in ASP.NET MVC

Last updated 2 days ago by Matthew Jones
· · · · · 6 min read

Buy me a coffee

Ever wondered just what the ModelState was that keeps popping up in your ASP.NET MVC controllers? Have you seen the property ModelState.IsValid and didn't know what it did?

So have I. Let's break down what the ModelState is and why we use it.

What is ModelState?

ModelState is a property of a Controller, and can be accessed from those classes that inherit from System.Web.Mvc.Controller.

The ModelState represents a collection of name and value pairs that were submitted to the server during a POST. It also contains a collection of error messages for each value submitted. Despite its name, it doesn't actually know anything about any model classes, it only has names, values, and errors.

ModelState has two purposes: to store the value submitted to the server, and to store the validation errors associated with those values.

But that's the boring explanation. Show me the code!

The Sample Project

Check out my (very simple) sample project hosted over on GitHub!

exceptionnotfound/ModelStateDemo

Contribute to exceptionnotfound/ModelStateDemo development by creating an account on GitHub.

exceptionnotfound • GitHub

The Setup

Now, let's get started writing the code for this demo. First, we have the `AddUserVM` view model:

ViewModels/Home/AddUserVM.cs

```
public class AddUserVM
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string EmailAddress { get; set; }
}
```

Next, we have a simple view:

Views/Home/Add.cshtml

```
@model ModelStateDemo.ViewModels.Home.AddUserVM
<h2>Add</h2>

@using(Html.BeginForm())
```

```
    {
        <div>
            <div>
                @Html.TextBoxFor(x => x.FirstName)
            </div>
            <div>
                @Html.TextBoxFor(x => x.LastName)
            </div>
            <div>
                @Html.TextBoxFor(x => x.EmailAddress)
            </div>
            <div>
                <input type="submit" value="Save" />
            </div>
        </div>
    }
```

Finally, we have the controller actions:

Controllers/HomeController.cs

```
    ...
    [HttpGet]
    public ActionResult Add()
    {
        AddUserVM model = new AddUserVM();
        return View(model);
    }

    [HttpPost]
    public ActionResult Add(AddUserVM model)
    {
        if(!ModelState.IsValid)
        {
            return View(model);
        }
        return RedirectToAction("Index");
    }
```

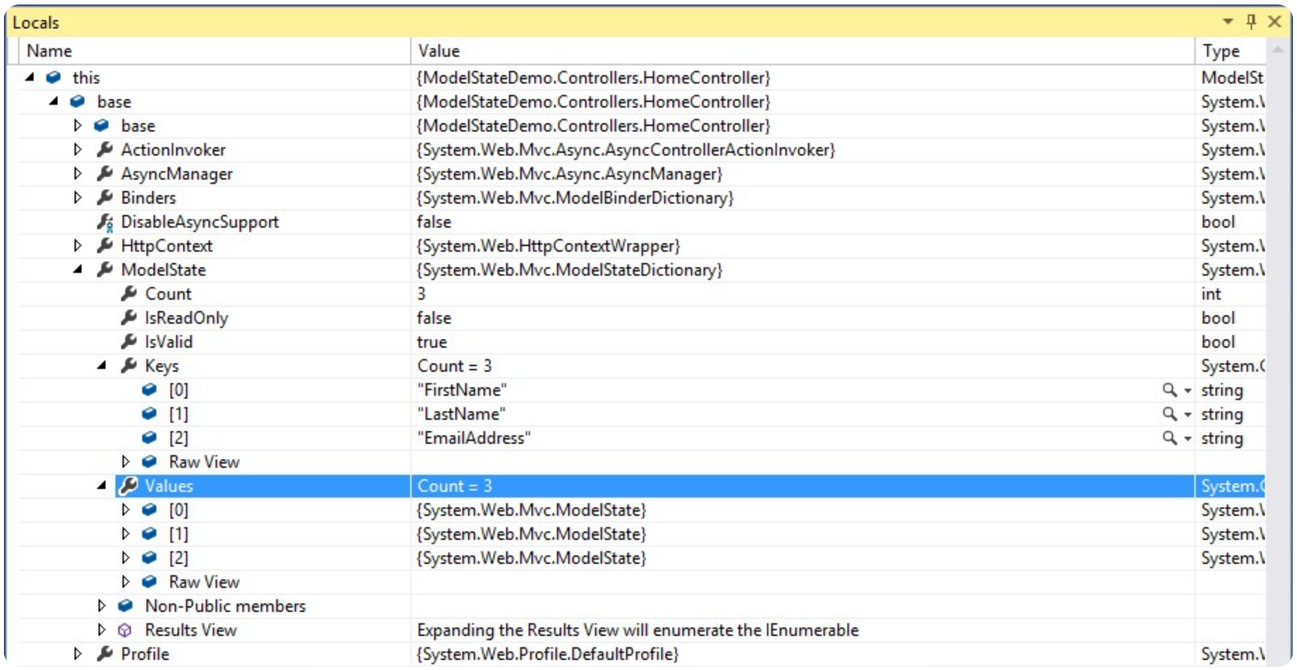
When we submit the form to the POST action, all of the values we entered will show up in the AddUserVM instance. But how did they get there?

The ModelStateDictionary Class

Let's look at the rendered HTML form for the Add page:

```
<form action="/Home/Add" method="post">
    <div>
        <div>
            <label for="FirstName">First Name:</label>
            <input id="FirstName" name="FirstName" type="text" value="">
        </div>
        <div>
            <label for="LastName">Last Name:</label>
            <input id="LastName" name="LastName" type="text" value="">
        </div>
        <div>
            <label for="EmailAddress">Email Address:</label>
            <input id="EmailAddress" name="EmailAddress" type="text" value="">
        </div>
        <div>
            <input type="submit" value="Save">
        </div>
    </div>
</form>
```

In a POST, all values in `<input>` tags are submitted to the server as key-value pairs. When MVC receives a POST, it takes all of the post parameters and adds them to a ModelStateDictionary instance. When debugging the controller POST action in Visual Studio, we can use the Locals window to investigate this dictionary:

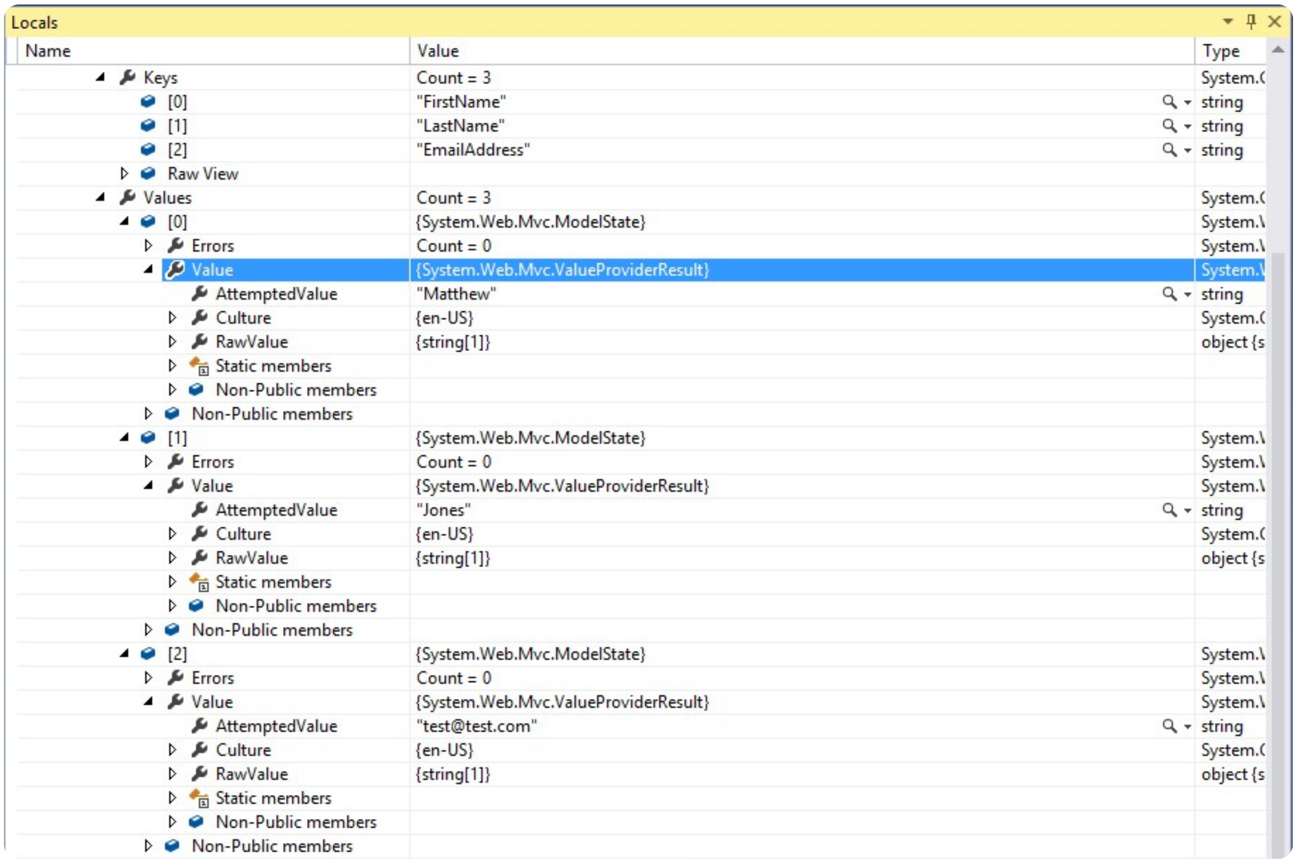

Locals

Name	Value	Type
this	{ModelStateDemo.Controllers.HomeController}	ModelSt
base	{ModelStateDemo.Controllers.HomeController}	System.\
base	{ModelStateDemo.Controllers.HomeController}	System.\
ActionInvoker	{System.Web.Mvc.Async.AsyncControllerActionInvoker}	System.\
AsyncManager	{System.Web.Mvc.Async.AsyncManager}	System.\
Binders	{System.Web.Mvc.ModelBinderDictionary}	System.\
DisableAsyncSupport	false	bool
HttpContext	{System.Web.HttpContextWrapper}	System.\
ModelState	{System.Web.Mvc.ModelStateDictionary}	System.\
Count	3	int
IsReadOnly	false	bool
IsValid	true	bool
Keys	Count = 3	System.(
[0]	"FirstName"	string
[1]	"LastName"	string
[2]	"EmailAddress"	string
Raw View		
Values	Count = 3	System.(
[0]	{System.Web.Mvc.ModelState}	System.\
[1]	{System.Web.Mvc.ModelState}	System.\
[2]	{System.Web.Mvc.ModelState}	System.\
Raw View		
Non-Public members		
Results View	Expanding the Results View will enumerate the IEnumerable	
Profile	{System.Web.Profile.DefaultProfile}	System.\

The Values property of the ModelStateDictionary contains instances that are of type System.Web.Mvc.ModelState. What does a ModelState actually contain?

What's in a ModelState?

Here's what those values look like, from the same debugger session:



```
Locals
Name                            Value                                   Type
  ◢ Keys                        Count = 3                               System.(
      [0]                       "FirstName"                             string
      [1]                       "LastName"                              string
      [2]                       "EmailAddress"                          string
    ▷ Raw View
  ◢ Values                      Count = 3                               System.(
    ◢ [0]                       {System.Web.Mvc.ModelState}             System.\
      ▷ Errors                  Count = 0                               System.\
      ◢ Value                   {System.Web.Mvc.ValueProviderResult}    System.\
          AttemptedValue        "Matthew"                               string
        ▷ Culture               {en-US}                                 System.(
        ▷ RawValue              {string[1]}                             object {s
        ▷ Static members
        ▷ Non-Public members
      ▷ Non-Public members
    ◢ [1]                       {System.Web.Mvc.ModelState}             System.\
      ▷ Errors                  Count = 0                               System.\
      ◢ Value                   {System.Web.Mvc.ValueProviderResult}    System.\
          AttemptedValue        "Jones"                                 string
        ▷ Culture               {en-US}                                 System.(
        ▷ RawValue              {string[1]}                             object {s
        ▷ Static members
        ▷ Non-Public members
      ▷ Non-Public members
    ◢ [2]                       {System.Web.Mvc.ModelState}             System.\
      ▷ Errors                  Count = 0                               System.\
      ◢ Value                   {System.Web.Mvc.ValueProviderResult}    System.\
          AttemptedValue        "test@test.com"                         string
        ▷ Culture               {en-US}                                 System.(
        ▷ RawValue              {string[1]}                             object {s
        ▷ Static members
        ▷ Non-Public members
      ▷ Non-Public members
```

Each of the properties has an instance of ValueProviderResult that contains the actual values submitted to the server. MVC creates all of these instances automatically for us when we submit a POST with data, and the POST action has inputs that map to the submitted values. Essentially, **MVC is wrapping the user inputs into more server-friendly classes** (ModelState and ValueProviderResult) for easier use.

There's still two important properties that we haven't discussed, though: the ModelState.Errors property and the ModelState.IsValid property. They're used for the second function of ModelState: to store the errors found in the submitted values.

Validation Errors in ModelState

Let's change our AddUserVM class:

ViewModels/Home/AddUserVM.cs

```
public class AddUserVM
{
    [Required(ErrorMessage = "Please enter the user's first name.")]
    [StringLength(50, ErrorMessage = "The First Name must be less than {1} characters.")]
    [Display(Name = "First Name:")]
    public string FirstName { get; set; }

    [Required(ErrorMessage = "Please enter the user's last name.")]
    [StringLength(50, ErrorMessage = "The Last Name must be less than {1} characters.")]
    [Display(Name = "Last Name:")]
    public string LastName { get; set; }

    [EmailAddress(ErrorMessage = "The Email Address is not valid")]
    [Required(ErrorMessage = "Please enter an email address.")]
    [Display(Name = "Email Address:")]
    public string EmailAddress { get; set; }
}
```

We've added validation attributes, specifically Required, StringLength, and EmailAddress. We've also set the error messages that are to be displayed if the corresponding validation errors occur.

With the above changes in place, let's modify the Add view to display the error messages if they occur:

Views/Home/Add.cshtml

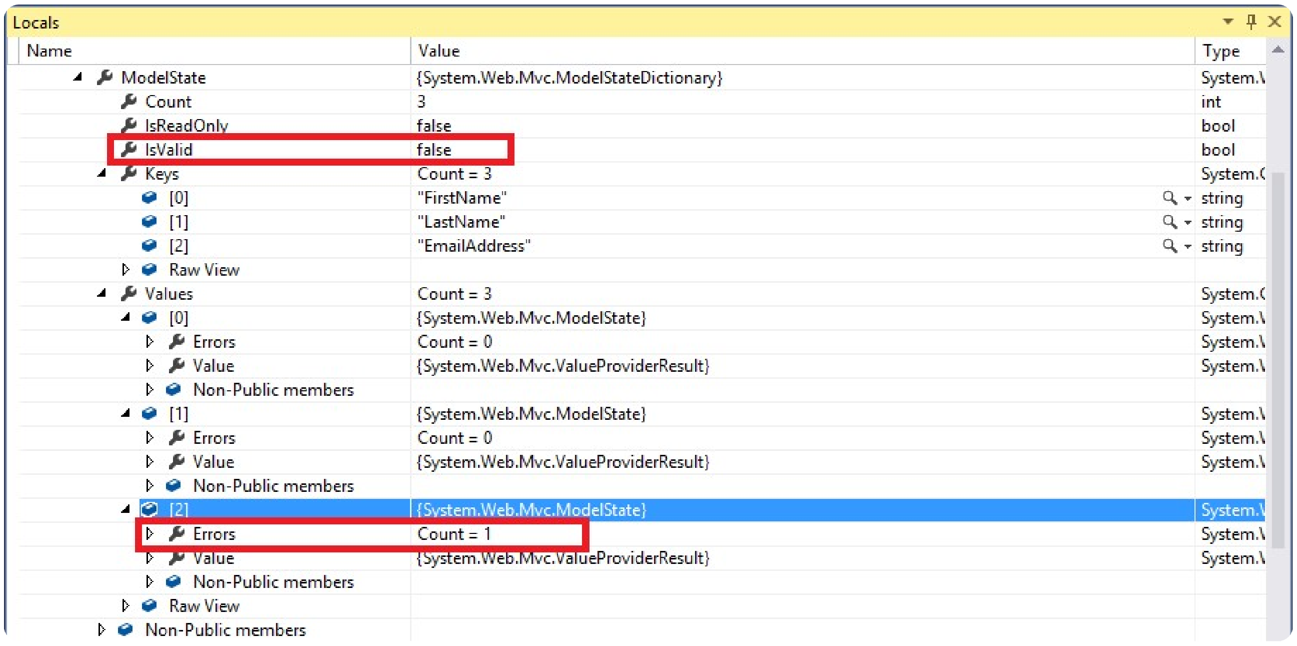
```
@model ModelStateDemo.ViewModels.Home.AddUserVM

<h2>Add</h2>

@using(Html.BeginForm())
{
    @Html.ValidationSummary()
    <div>
        <div>
            @Html.LabelFor(x => x.FirstName)
            @Html.TextBoxFor(x => x.FirstName)
            @Html.ValidationMessageFor(x => x.FirstName)
        </div>
        <div>
            @Html.LabelFor(x => x.LastName)
            @Html.TextBoxFor(x => x.LastName)
            @Html.ValidationMessageFor(x => x.LastName)
        </div>
        <div>
```

```
                @Html.LabelFor(x => x.EmailAddress)
                @Html.TextBoxFor(x => x.EmailAddress)
                @Html.ValidationMessageFor(x => x.EmailAddress)
            </div>
            <div>
                <input type="submit" value="Save" />
            </div>
        </div>
    }
```

Notice the two helpers we are using now, ValidationSummary and ValidationMessageFor.

- ValidationSummary reads all errors from the model state and displays them in a bulleted list.
- ValidationMessageFor displays only errors for to the property specified.

Let's see what happens when we attempt to submit an invalid POST that is missing the email address. When we get to the POST action while debugging, we have the following values in our ModelStateDictionary:



Locals

Name	Value	Type
ModelState	{System.Web.Mvc.ModelStateDictionary}	System.\
Count	3	int
IsReadOnly	false	bool
IsValid	false	bool
Keys	Count = 3	System.(
[0]	"FirstName"	string
[1]	"LastName"	string
[2]	"EmailAddress"	string
Raw View		
Values	Count = 3	System.(
[0]	{System.Web.Mvc.ModelState}	System.\
Errors	Count = 0	System.\
Value	{System.Web.Mvc.ValueProviderResult}	System.\
Non-Public members		
[1]	{System.Web.Mvc.ModelState}	System.\
Errors	Count = 0	System.\
Value	{System.Web.Mvc.ValueProviderResult}	System.\
Non-Public members		
[2]	{System.Web.Mvc.ModelState}	System.\
Errors	Count = 1	System.\
Value	{System.Web.Mvc.ValueProviderResult}	System.\
Non-Public members		
Raw View		
Non-Public members		

Note that the ModelState instance for the email address now has an error in the Errors collection. When MVC creates the model state for the submitted properties, it also goes through each property in the ViewModel and validates the property using attributes associated to it. If any errors are found, they are added to the Errors collection in the property's ModelState.

Also note that ModelState.IsValid is false now. That's because an error exists;

ModelState.IsValid is false if any of the properties submitted have any error messages attached to them.

What all of this means is that by setting up the validation in this manner, we allow MVC to just work the way it was designed. ModelState stores the submitted values, allows them to be mapped to class properties (or just as parameters to the action) and keeps a collection of error messages for each property. In simple scenarios, this is all we need, and all of it is happening behind the scenes!

Custom Validation

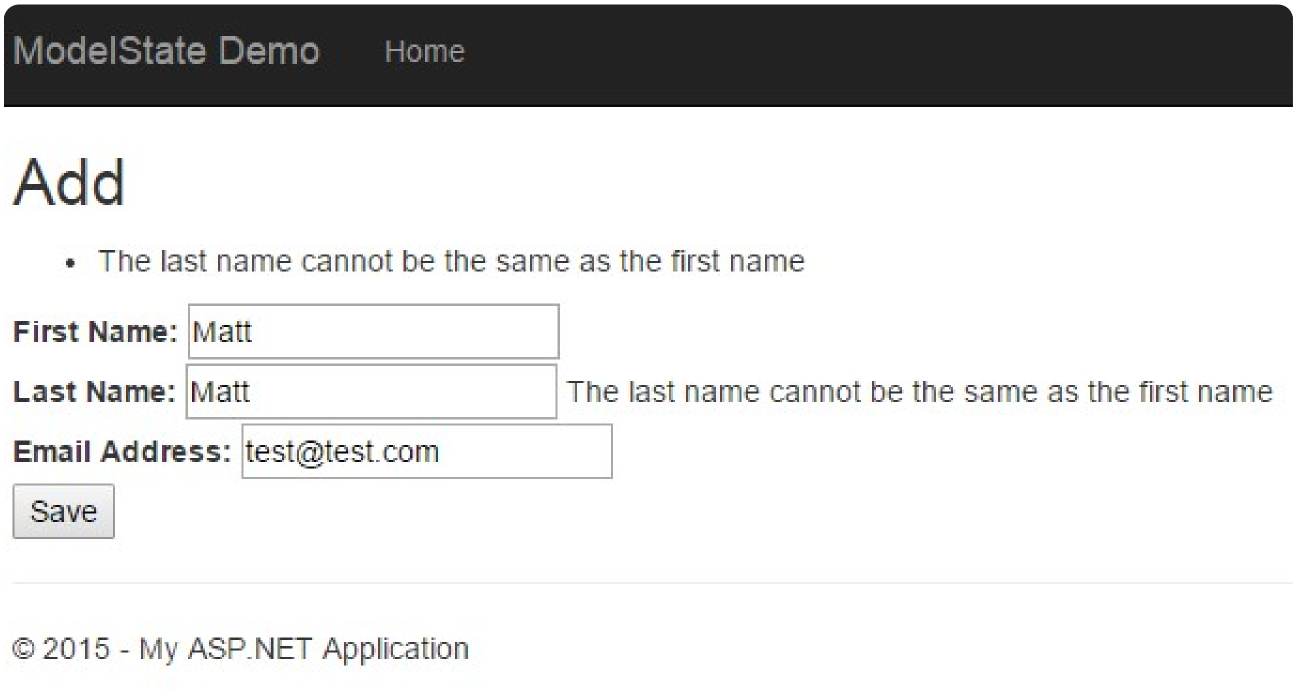
But what if we needed to perform more complex validation than what is provided by attributes? Say we needed to validate that the first and last names are not identical, and display a particular error message when this happens.

We can actually add errors to the model state via the AddModelError method on ModelStateDictionary:

```
[HttpPost]
public ActionResult Add(AddUserVM model)
{
    if(model.FirstName == model.LastName)
    {
        ModelState.AddModelError("LastName", "The last name cannot be the same as the first
name.");
    }
    if(!ModelState.IsValid)
    {
        return View(model);
    }
    return RedirectToAction("Index");
}
```

The first parameter to the AddModelError method is the name of the property that the error applies to. In this case, we set it to LastName. You could also set it to nothing (or a fake name) if you just want it to appear in the ValidationSummary and not in a ValidationMessage.

Now the error will be displayed on the page:



Add

- The last name cannot be the same as the first name

First Name:	Matt
Last Name:	Matt — The last name cannot be the same as the first name
Email Address:	test@test.com

Summary

The ModelState represents the submitted values and errors in said values during a POST. The validation process respects the attributes like Required and EmailAddress, and we can add custom errors to the validation if we so desire. ValidationSummary and ValidationMessageFor read directly from ModelState to display errors to the user.

I've got a very simple sample project on Github that demonstrates how the ModelState works and provides all the code and markup in this post. Take a look!

Also, if this post helped you, please consider buying me a coffee. Your support funds all of my projects and helps me keep traditional ads off this site. Thanks!

Happy Coding!

SHARE THIS ARTICLE

Did you enjoy this post? Become a member today!

Get all the latest posts right in your inbox! |

Your email address

SUBSCRIBE

YOU MIGHT ALSO LIKE

ASP.NET

Areas - ASP.NET MVC Demystified

ASP.NET

Partial Views - ASP.NET MVC Demystified

ASP.NET MVC DEMYSTIFIED

Attribute Routing vs Convention Routing - ASP.NET MVC Demystified

← NEWER ARTICLE

Are programmers afraid of losing control over their code?

OLDER ARTICLE

A Simple CheckBoxList in ASP.NET MVC

This domain is not registered with Commento.

Membership Newsletter Authors Categories

Guest Writer Program Speaking

EXCEPTION NOT FOUND © 2020 — ALL RIGHT RESERVED. PUBLISHED WITH GHOST.