|  |
| --- |
|  |
| C# Flow Control |
| Branching Throwing Looping |
|  |
|  |
|  |

|  |
| --- |
|  |

# 

# C# Flow Control

## Objectives

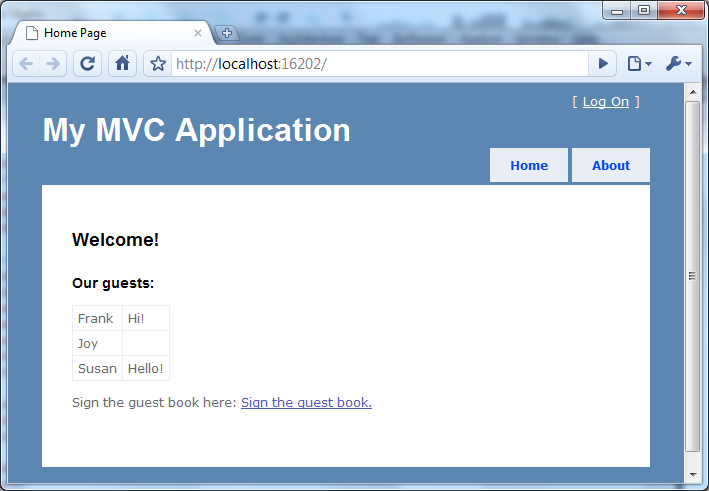
In this lab we’ll work with an ASP.NET MVC application and build a guest book. By the end of this lab you should understand how to:

* Catch Exceptions
* Compare exception flow control to branching
* Throw exceptions

## Part 1 - Try & Catch

1. Open the **Flow.sln** file in the **before** directory of this lab.
2. Press Ctrl+F5 to **run** the application.

The application should run without errors and display the home page.



1. Click on “**Sign the guest book**” and enter some information.
2. Click on **Save**.

The logic behind the save operation is not implemented yet, so any information you entered won’t appear in the guest book. We’ll implement this logic in the following steps. Note: in this lab the guest book is not persisted to a file or database. Instead the guest book is entirely in-memory to keep the lab simple. If you restart the application the guest book will lose changes. This is not a problem, but just be aware of the behavior.

1. Open **GuestController**.cs from the project’s Controllers folder.

There are two Sign methods inside. The first method responds to HTTP GET and displays the view the guest will use to enter their information. The second Sign method responds to an HTTP POST and should put the guest’s information in the guest book. It is this second method we will work with.

1. Import the Flow.Models namespace.

using Flow.Models;

1. Inside the Sign method with an HttpPost attribute, create a new instance of the **Guest** class and pass the instance to the **UpdateModel** method.

Update model will push values from the request into the new object.

1. Create a new instance of the **GuestBook** class.
2. Pass the new guest object to the **AddGuest** method of the guest book.

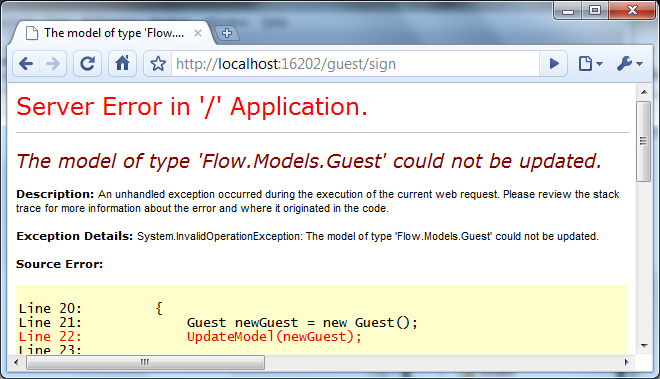
[HttpPost]  
public ActionResult Sign(FormCollection c)  
{  
    Guest newGuest = new Guest();  
    UpdateModel(newGuest);  
  
    GuestBook book = new GuestBook();  
    book.AddGuest(newGuest);  
  
    return RedirectToAction("index", "home");  
}

1. Press Ctrl+F5 to **run** the application, go to **sign** the guest book.
2. Enter the information for a guest (a name and a message) and click **Save**.

The application should save the new guest entry.

1. Return to sign the guest book, but this time **click Save without entering a name.**
2. Make a note of the **exception details** in the error page that appears.

You should see an ASP.NET error page appear. There is metadata associated with our Guest class telling the MVC runtime that the name property is required. We need to handle this error condition and allow the user to retry the save operation.



1. Notice the exception is of type System.**InvalidOperationException**.
2. Return to **GuestController**.cs
3. Surround all the logic inside the Sign method in a **try** block.
4. In the event of any exception, return View().

 [HttpPost]  
 public ActionResult Sign(FormCollection c)  
 {  
     try  
     {  
         Guest newGuest = new Guest();  
         UpdateModel(newGuest);  
  
         GuestBook book = new GuestBook();  
         book.AddGuest(newGuest);  
  
         return RedirectToAction("index", "home");  
     }  
     catch  
     {  
         return View();  
     }  
 }

1. **Run** the application again. **Sign** the guest book without a name. The ASP.NET error page **should not** appear.



What we’ve done in the last bit of code is catch ANY exception. Let’s try catching specific exceptions.

1. Return to GuestController.cs
2. Change the catch clause to catch an ApplicationException.

try  
{  
    // ...  
}  
catch(ApplicationException)  
{  
    return View();  
}

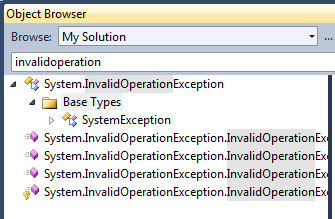
Note we are not putting the exception object into a variable because we are not inspecting the exception being thrown. If we want to put the exception into an object we could use the following code (and then use properties on the ex object).

catch(ApplicationException **ex**)  
{  
    return View();  
}

1. Run the application again and try to sign the guest book without a name.

The error page appears again. Do you know why?

1. In VisualStudio go to **View** -> **Object** **Browser**.
2. **Search** for invalidoperation
3. Expand the System.InvalidOperationException node to look the type’s base type.



Notice the base type is SystemException (and it’s base type is Exception). ApplicationException is unrelated to InvalidOperationException, thus our catch statement could not catch the invalid operation.

1. Return to the sign method and change the catch statement to catch SystemException.

catch(SystemException)  
{  
    return View();  
}

1. Run and test the application to ensure the error page does not appear.
2. Finally, change the implementation to catch an InvalidOperationException. This version should also work.

catch(InvalidOperationException)  
{  
    return View();  
}

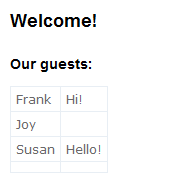
## Part II – Try a different approach

1. Return to GuestController.cs.
2. Change the Sign method to invoke **TryUpdateModel** instead of UpdateModel.

try  
{  
    Guest newGuest = new Guest();  
**TryUpdateModel(newGuest);**  
  
    GuestBook book = new GuestBook();  
    book.AddGuest(newGuest);  
  
    return RedirectToAction("index", "home");  
}

1. Run the application and sign the guest book without entering a name.

This time the application will allow the guest in the guest book.



In our previous version, the UpdateModel call throws an exception if the requirements for our guest book are not met. TryUpdateModel does not throw, but sets a flag we can check to see if the update was successful. This is a standard tradeoff in error handling scenarios.

* Do you want to force the execution path to change by throwing an exception?

or

* Do you want to set a flag to let someone check for an error using an if statement?

The answer depends on the specific scenario for your application.

Let’s work with TryUpdateModel and see how our implementation will change.

1. Write an if statement to check the ModelState.IsValid property. Only if the property is true should the controller add the guest to the guest book and redirect the browser.

[HttpPost]  
public ActionResult Sign(FormCollection c)  
{  
    try  
    {  
        Guest newGuest = new Guest();  
        TryUpdateModel(newGuest);  
  
        if (ModelState.IsValid)  
        {  
            GuestBook book = new GuestBook();  
            book.AddGuest(newGuest);  
            return RedirectToAction("index", "home");  
        }                  
    }  
    catch(InvalidOperationException)  
    {  
        return View();  
    }  
}

If we compile the above code we’ll see a compiler error: “not all code paths return a value”. We need some additional changes to ensure the logic is structured to always return a value.

1. Remove the try – catch statements and blocks.

[HttpPost]  
public ActionResult Sign(FormCollection c)  
{  
    Guest newGuest = new Guest();  
    TryUpdateModel(newGuest);  
  
    if (ModelState.IsValid)  
    {  
        GuestBook book = new GuestBook();  
        book.AddGuest(newGuest);  
        return RedirectToAction("index", "home");  
    }  
    return View();  
  
}

1. Run the application and test your guest book by saving entries both with and without a name.

# Part III – Throwing

1. Open GuestBook.cs from the Models directory.
2. Add a using for the System namespace.
3. Change the add method to ensure a guest with the name of “Joe” will generate an ArgumentException.

public void AddGuest(Guest newGuest)  
{  
    if(newGuest.Name == "Joe")  
    {  
        throw new ArgumentException("Joe is not allowed");  
    }  
    \_guests.Add(newGuest);  
}

1. Run the application to ensure Joe can’t sign the guest book.
2. See if you can handle this error in the controller and give the user an error message (one solution shown below).

if (ModelState.IsValid)  
{  
    GuestBook book = new GuestBook();  
    try  
    {  
        book.AddGuest(newGuest);  
    }  
    catch(ArgumentException e)  
    {  
        ModelState.AddModelError("", e.Message);  
        return View();  
    }  
    return RedirectToAction("index", "home");  
}

# Conclusion

In this lab we’ve looked at throwing and catching exceptions, and compared using exceptions for control flow versus branching. We saw the same method implemented using both approaches. Which approach did you prefer (and why?)