<http://www.asp.net/mvc/tutorials/controllers-and-routing/creating-a-custom-route-constraint-cs> (general tutorial)

As an alternative to Web Forms, ASP.NET MVC takes a different approach when it comes to structuring web applications. This means you won’t be dealing with ASPX pages and controls, postbacks or view state, or complicated event lifecycles. Instead, you’ll be defining controllers, actions, and views. The underlying ASP.NET platform is the same, however, so things like HTTP handlers and HTTP modules still apply, and

you can mix MVC and Web Forms pages in the same application.

benefits of MVC:

■ Full control over HTML

■ Full control over URLs

■ Better separation of concerns

■ Extensibility

■ Testability: By separating application logic from the user interface, ASP.NET MVC makes it easier

to test individual components in isolation. Controller classes can be tested without

testing the aPartial page output caching

■ The Razor view engine

One of the core components of the new ASP.NET Web Pages technology is the Razor view

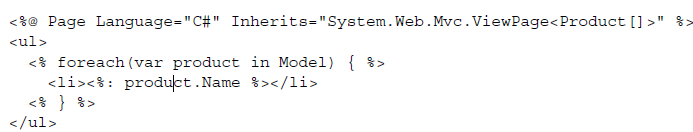
engine. This engine provides a concise way to mix code and markup within the same file.

ASP.NET MVC applications can also make use of the Razor view engine as an alternative

to the Web Forms view engine that was available in both ASP.NET MVC 1 and 2.

As an example, the following code snippet shows a simple page that constructs a

list of product names using the older Web Forms view engine:



The views we’ve seen so far only contain content that’s specific to an individual page.

All of the surrounding chrome (such as the menu and title) is defined in a layout.

Layout is like a master page. You can modify the values like header text,title etc..

■ Package management with NuGet

ASP.NET MVC also comes with the NuGet package manager. NuGet simplifies the management

of dependencies by providing a facility that can be used to install components,

libraries, and other utilities directly into your project without needing to

manually visit a website to download the library that you’re looking for.

■ Improved extensibility

■ Global action filters

■ Dynamic language features  
■ Ajax improvements  
■ Enhancements to the validation infrastructure  
■ Mobile templates  
■ Web APIctual user interface.

# Routing:

Eg: <http://www.testurl.com/product/details/99>

Normally URLs refer to physical files on the server   
eg. ‘hxxp://www.testurl.com/products/Default.aspx’ it simply meant there was a Default.aspx file in a ‘products’ folder at the root of the website

* MVC Routing helps abstract URLs away from physical files which by default maps to Controller/Action method pairs by default.
* Routing basic helps in giving cleaner and understandable URL to user.
* **Keep URLs discoverable by end-users:** Having URL parameters baked into routes makes URLs easier to understand and encourages users to play around and discover available functionality

### How does the Route Handler work

Fact is the RouteHandler is first to be executed. It follows these steps (not evident from the stack trace)

1. Check if route is static file on disk, if so the resource is served directly
2. If it’s **not a static route**, check if there is a custom route handler, if so it hands off the request **to the custom route handler**
3. If there are **no custom route handlers it hands over to the default MVC** Routing handler. Now that we have reached the route handler let us see how it treats routes

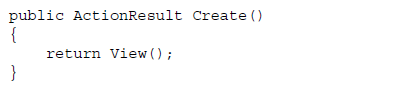
### Adding New Routes

New routes can be registered in the “RegisterRoutes” function, say for eg. The default route provided takes id as parameter as shown below,

Now if a user visited ‘hxxp://www.testurl.com/product/edit/1’ it would match the ‘Default’ route defined above and MVC would look for a controller called ProductController with an action Edit that takes an input parameter called id.

Rather than using hard-coded links, we can use the ActionLink HTML Helper in order to render a

hyperlink to a particular controller action. They are basically used to point links to Controller action methods, hence in mvc application you donot use the normal link tag of HTML.



In this case, the Views/Guestbook/Create.cshtml view file is rendered. But how does

the MVC Framework know to render this particular view rather than one of the other

views in the application (such as Index.cshtml)?

Calling the View method returns a ViewResult object that knows how to render a particular view. When this method is called with no arguments, the framework infers that the name of the view to render should be the same as the name of the action (Create). Later in the MVC pipeline, the framework’s ControllerActionInvoker class executes the ViewResult and tells it to render the view. At this point, the framework asks the ViewEngineCollection to locate the appropriate view for rendering. (As you’ve already seen back in chapter 2, by default the view engine will look for views within the Views/<Controller Name> directory and the Views/Shared directory).  
  
**You can override the convention** for using the action name as the view name if you want to. For example, if your view **was called New.cshtml rather than Create.**cshtml, you could call a second overload of the View method that accepts an explicit view name:

return View("New");

Alternatively, you can specify an application-relative path to the view if it doesn’t reside within the subdirectory with the same name as the controller:

return View("~/Views/SomeOtherDirectory/New.cshtml");

## View engines

Different view engines are responsible for rendering views with different formats. By

default, ASP.NET MVC ships with two view engines—the RazorViewEngine and the

WebFormViewEngine. The Razor view engine is responsible for rendering views in the

Razor format (either .cshtml files or .vbhtml files), while the Web Form view engine is

used to support the older-format Web Form views (.aspx and .ascx files)

Throughout the ASP.NET releases, advancements in ASPX files included

better data-binding syntax and other items more geared towards development

with controls. In various MVC frameworks, view development encourages and requires code written

directly alongside markup. Because the ASPX view engine was not designed with this

goal in mind, the ASP.NET team decided to build an entirely new view engine with a

code-focused templating approach. The result was a more intelligent parsing engine

that is able to very easily figure out where code stops and where markup begins, without

the developer needing to be very explicit.

It’s also possible to plug in additional view engines, so you can use third-party

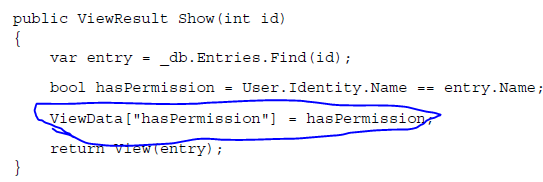
formats for rendering views. In chapter 10 we’ll look at using the popular open-source

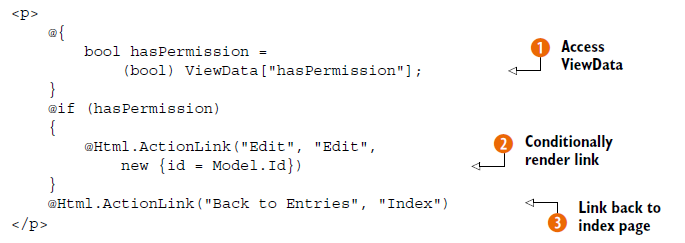
Spark view engine for rendering views.

Passing data to viewsThree different ways in which data can be passed to a view by using the

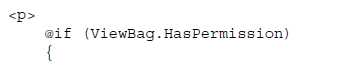
### ViewDataDictionary

ASP.NET MVC exposes a dictionary to enable the controller action to pass any number of model objects and information to the view. With a dictionary object, we can pass as many items as need be for the view to

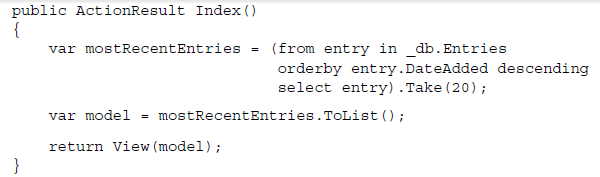
render appropriately.

information accessed in view via. Razor code, generally used to pass some extra bits of information, apart from the main data

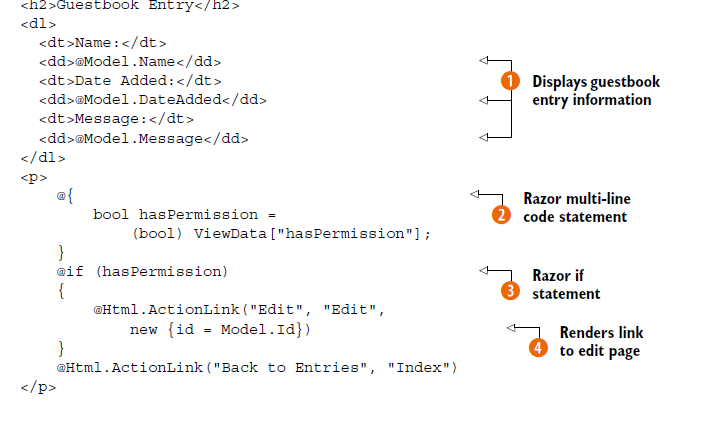
ViewBag  
Like the ViewDataDictionary, the ViewBag provides a way to pass data from the controller to the view, but the ViewBag makes use of the dynamic language features of C# 4. Instead of storing items in a dictionary using a string key, you can simply set properties on the dynamic ViewBag property within your controller:

  
In the view,  


Same like viewDictionary, difference you need not do casting while retrieving.

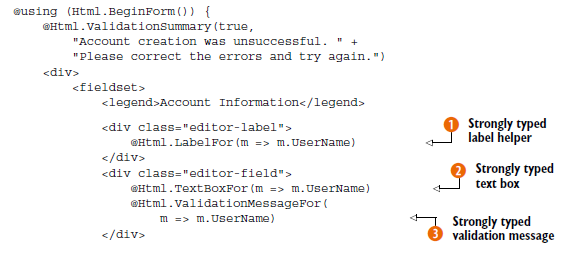
strongly typed views  
In viewbag and viewdictionary one can you use intellisense, so prone to errors due to spelling mistakes also it wont allow to use validation blocks provided by MVC framework..hence strongly typed views usage is recommended..  


then in view..  

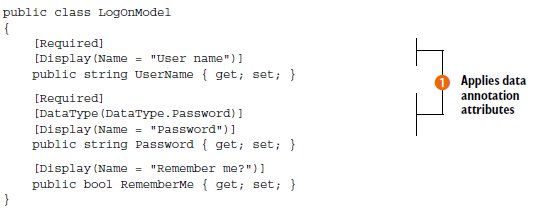



strongly typed view for the Log On screen, we can use the built-in helpers to render the HTML for each input element. Instead of using loosely bound strings to represent the action parameters, we can take advantage of the

expression-based HtmlHelper extensions to create various types of input elements, as follows.



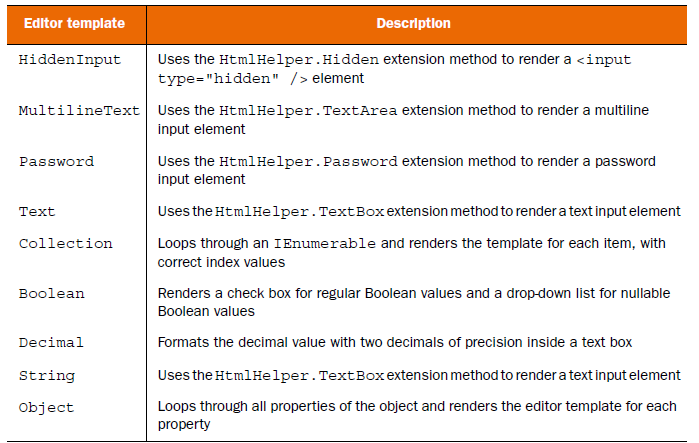
Also use the datavalidation block..



Templated helpers, and it’s designed to assist in generating HTML based on strongly typed views. Templated helpers can be used to generate HTML for the entire model or for one member at a time.for eg EditFor template generates an editable form. You can override existing builtin templates or build new templates,,templates can be shared across entire project,,,but the shared template can be overridden for a particular folder of views..

***EditorFor and DisplayFor templates***

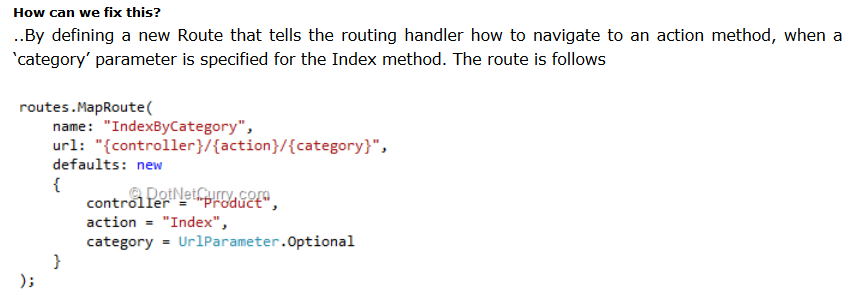
These two different sets of templates are separated into editor and display templates. The editor and display templates are generated from the following methods:





#### But now say, http://localhost:11618/Product/Index/Electronics

We wan this url to filter by category it wont work with default routing



**Other eg:**

Imagine, for example, that you are building a blog application. You might want to handle incoming requests that look like this:  
/Archive/12-25-2009

When a user enters this request, you want to return the blog entry that corresponds to the date 12/25/2009. In order to handle this type of request, you need to create a custom route as below:



The order of the routes that you add to the route table is important. Our new custom Blog route is added before the existing Default route. If you reversed the order, then the Default route always will get called instead of the custom route.

## ActionResult Subtypes

* **ViewResult** - Renders a specifed view to the response stream (this is subtype of Action result). Represents HTML and markup.
* **PartialViewResult** - Renders a specifed partial view to the response stream
* **EmptyResult** - An empty response is returned
* **RedirectResult** - Performs an HTTP redirection to a specifed URL
* **RedirectToRouteResult** - Performs an HTTP redirection to a URL that is determined by the routing engine, based on given route data
* **JsonResult** - Serializes a given ViewData object to JSON format
* **JavaScriptResult** - Returns a piece of JavaScript code that can be executed on the client
* **ContentResult** - Writes content to the response stream without requiring a view
* **FileContentResult** - Returns a file to the client
* **FileStreamResult** - Returns a file to the client, which is provided by a Stream
* **FilePathResult** - Returns a file to the client

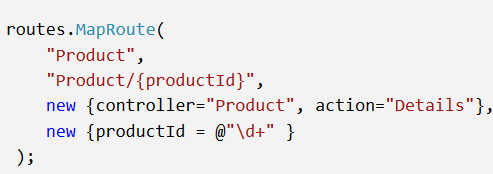
If you want to redirect the user from one controller action to another, you call the **RedirectToAction**() method.

ROUTES—MAPPING URLS TO ACTIONS  
Global.asax file’s RegisterRoutes method. This method defines routes that map a URL pattern to a controller or action. The implementation of this method is shown next.

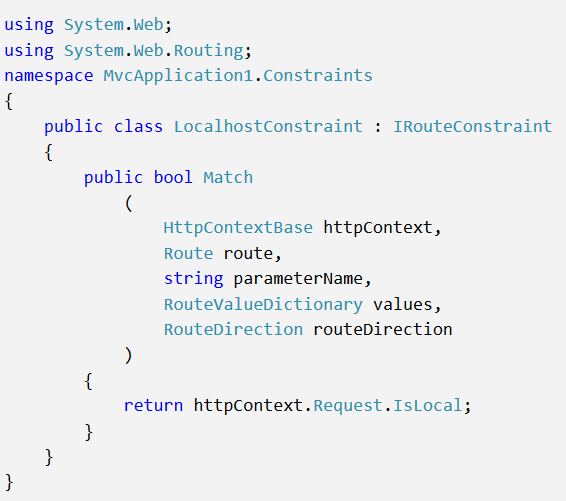
Three different ways in which data can be passed to a view by using the

* ViewDataDictionary,
* ViewBag
* strongly typed views.

Route constraints **are** to restrict the browser requests that match a particular route. You can use a regular expression to specify a route constraint.  
What you really want to do is only match URLs that contain a proper integer productId. You can use a constraint when defining a route to restrict the URLs that match the route. The modified Product route in Listing 3 contains a regular expression constraint that only matches integers.

  
The regular expression \d+ matches one or more integers. This constraint causes the Product route to match the correct URls containing integers

You use a custom constraint within a route defined in the Global.asax file. The Global.asax file in Listing 2 uses the Localhost constraint to prevent anyone from requesting an Admin page unless they make the request from the local server.

<http://www.asp.net/mvc/tutorials/controllers-and-routing/creating-a-custom-route-constraint-cs>  


Controllers and Action methods

Aaction could have a void return type and instead write out to the response stream directly  
This works because the ControllerActionInvoker ensures that the return value of an

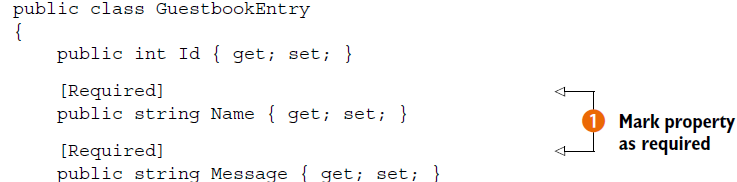
action is always wrapped in an ActionResult. If the action returns an ActionResult

already (such as a ViewResult), then it is simply invoked. However, if the action

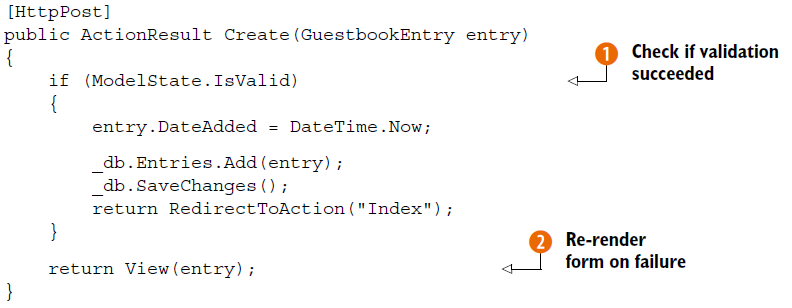
returns a different type (in this case, a string) then the return value is wrapped in a

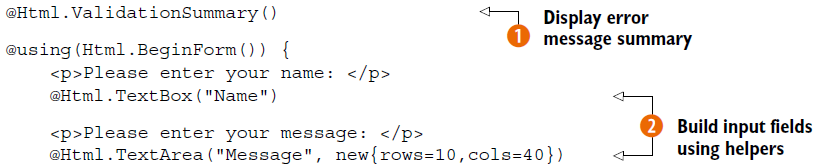
## ContentResult object that simply writes it out to the response stream.

## Validation



There are other validation attributes to check length etc.. This attributes are applied to the view model class.



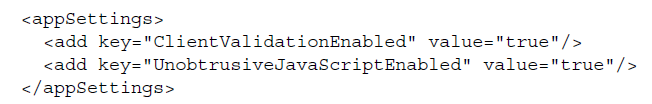
We can display the error messages generated by the validation failure in our view by calling the Html.ValidationSummary method.  


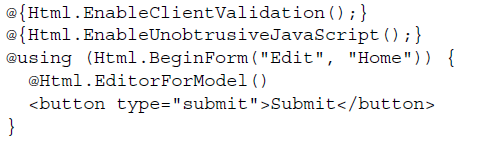


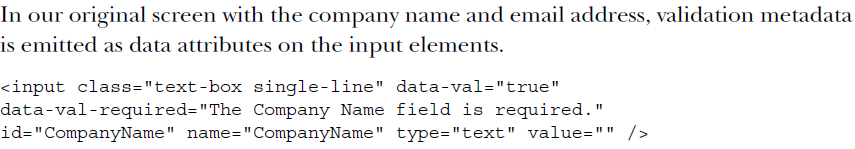
We can our own error messages..

ASP.NET MVC ships with support for using the jQuery Validate library for performing client-side validation. Another new feature of MVC is support for unobtrusive client-side validation, which will render input elements with data attributes that scripts can reference.

Another new feature of MVC is support for unobtrusive client-side validation, which will render input elements with data attributes that scripts can reference. Which means after enabling the unobstrusivejavascript as below, special tags are emited along with the controls markup, and the client side validation is done automatically.



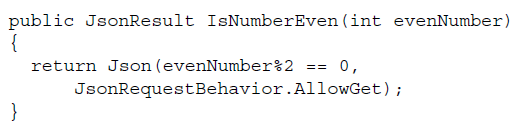




## Using RemoteAttribute

A new validation attribute in ASP.NET MVC 3 is RemoteAttribute. Decorating a model property with this attribute will instruct jQuery Validate to make an HTTP request to a given action method for server-side checking. The result is transmitted back to the client, this way we can do some quick server side validation which requires some business logic processing, before the result is acutally posted back.

## 

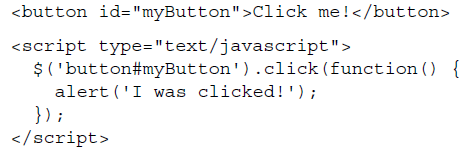


## Creating custom client-side validators

When a validation attribute implements IClientValidatable, the DataAnnotations ModelMetadataProvider (and any derivations, like our ConventionProvider) will instruct the framework to emit those data attributes on associated HTML elements. Using this mechanism, we can customize client-side validation and use our own JavaScript code to do the work. This is useful for application-specific behavior, when

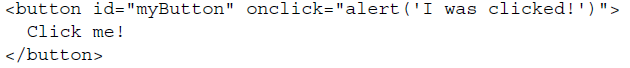
the validators provided by the jQuery Validate library aren’t enough.

Using jQuery, you can attach an event handler to the button’s click event externally.

 **This approach is known as** unobtrusive JavaScript. By keeping the site’s markup separate

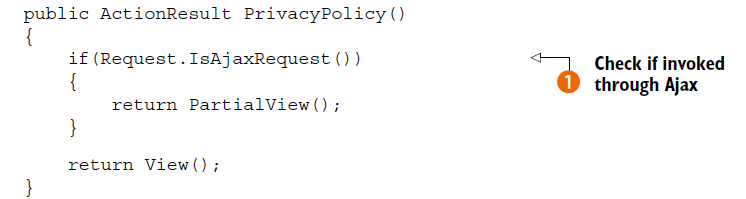
from its behavior (code), maintainability is improved and it’s easier to follow the

flow of the code. As opposed to previous days, where



## Ajax with Jquery

Using Ajax, in MVC one can do partial view updates..



## 

ASP.NET MVC Ajax helpers  
These helpers are available as extension methods on the AjaxHelper class and can be used to generate markup that will automatically use Ajax to send and retrieve data.

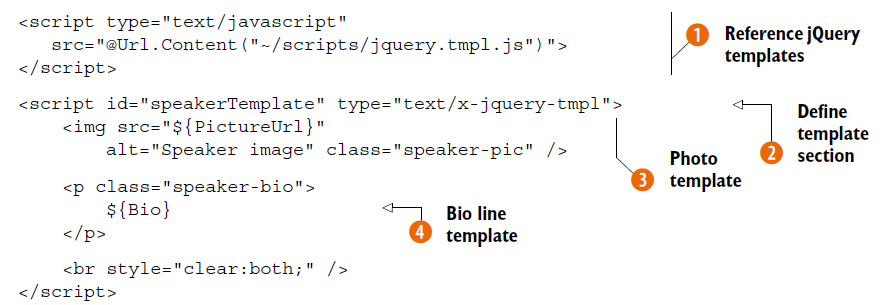
## 

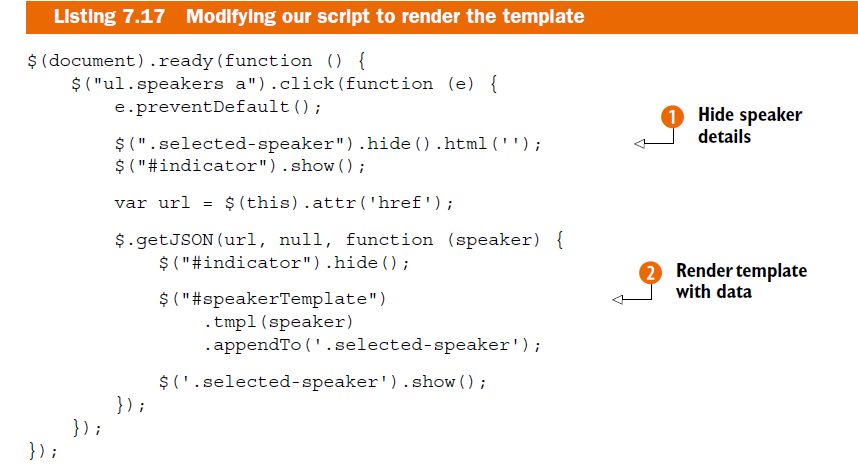
## Client-side templates

Much like we create server-side templates in the form of Razor’s .cshtml files, we can also create templates on the client. Client-side templates allow us to generate markup on the fly in the browser without having to go back to the server or having to manually construct elements using JavaScript. There are several client-side templating libraries available, but we’ll be using jQuery-tmpl, a templating library for jQuery that was written by Microsoft and

then contributed to the jQuery project as open source

This is something similar to js-render,,, where a template is created ..and then is rendered on the fly with data ..the data is obtained in json..via a ajax call..refer to screenshots below..





## View model

The purpose of a view model is quite straightforward—it is a model that is specifically designed for use within a view.

## Security

**Authentication** ensures that a user has supplied the proper credentials to access a system. Once a user logs in (typically by providing a username and password, or maybe some other token such as an SSH key or a cryptographic token) then they are authenticated.

**Authorization** takes place after authentication and involves making a decision as to whether a given user has permission to do something with the system, such as viewinga page or editing a record.