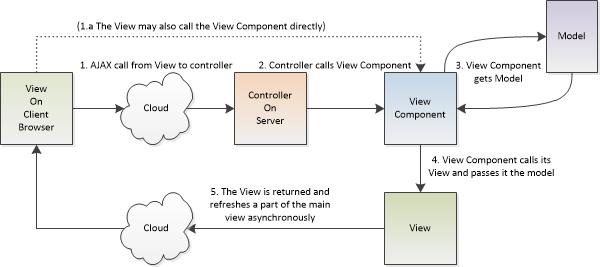
This document will provide a step by step work flow for working with View Components, those partial views that we work with in ASP.NET Core MVC. View Components allow us to refresh just a part of the page asynchronously.

They are developed in the same way a Controller and a View are developed together. A Controller calls a View and a View Component calls a View. Both the Controller and the View Component may get the Model to pass to the View before calling it.

A View Component may be called directly from View code or it may be called from a Controller method. In both cases data may be passed to the View Component such as an id value.



In this work flow, we will assume the following use case:

***“As school representative, I want to select a student from a drop-down list and view course section data so that I can determine what courses the student is enrolled in.”***

Technically, the whole page should not be refreshed and only the display of filtered course section data needs to be refreshed. A View Component can also be reused on other pages (views) of the web application.

**THE MAIN VIEW**

Using the CTTI application that was created using the Entity Framework Core Database First document as a guide, we will create a method called EnrollmentList in the *StudentsController* class that will return a View called EnrollmentList. The code for the controller class and cshtml view file follows:

Add the following method to the Students controller class code:

public async Task<IActionResult> EnrollmentList(int id = 0)

{

//Transform the Student collection from the entity framework

//context object to a SelectListItem collection--setting the

//Selected boolean property based if id matches or not

var result = \_context.Student.Select(s => new SelectListItem

{

Value = s.Id.ToString(),

Text = s.FullName,

Selected = s.Id == id

});

//pass the collection as the model to the view

return View(await result.ToListAsync());

}

Note the use of async, Task<> and await. This syntax is used for asynchronous operations, in this case, the call to result.ToListAsync().

Add the EnrollmentList view. It has two sets of nested div elements. The first nested div contains a Select element populated with *SelectListItem* objects with the values and text coming from *Student* objects in the controller. The asp-items tag helper uses the model passed to the view. The second nested div is where data will be refreshed asynchronously. The View code follows:

<h4>Student Enrollment List</h4>

<div class="row">

<div class="col-md-4">

<div class="form-group">

<label class="control-label">Select Student</label>

<**select** id="uxStudents" **asp-items**="@Model" class="form-control"></**select**>

</div>

</div>

</div>

<div class="row">

<div class="col-md-10">

<div id="uxEnrollmentData" class="form-group">

<!-- View Component data will be displayed here asynchonously -->

</div>

</div>

</div>

@section Scripts{

<script>

//When document is loaded and ready...

$(document).ready(function () {

//...call the method to load enrollment data

getEnrollmentData();

//...and set focus on the drop down

$("#uxStudents").focus();

//When selection changes, load enrollment data

$("#uxStudents").change(function () {

getEnrollmentData();

});

});

//Function loads enrollment data asynchronously based on selected student id

function getEnrollmentData() {

//get student id using jquery val() method

var studentId = $("#uxStudents").val();

//AJAX call is a GET to the controller method that calls the

//view component--the id comes from the student selection.

//The done method handles the callback asynchronously.

$.ajax({

method: 'GET',

url: '/Students/EnrollmentData',

data: { id: studentId }

}).done(function (result, statusText, xhdr) {

//the response returned from the view component will display

//in the div with id of uxEnrollmentData

$("#uxEnrollmentData").html(result);

});

}

</script>

}

Note that the url in the ajax method above is calling the *EnrollmentData* method of the *Students* controller (This is #1 in the flow diagram on page 1).

**THE CONTROLLER CODE:**

public IActionResult EnrollmentData(int id)

{

return ViewComponent("EnrollmentData", id);

}

This method requires the id of the Student. The Controller then calls the *ViewComponent* called *EnrollmentData* and passes it the id value (this is #2 in the flow diagram).

**THE VIEW COMPONENT CODE:**

Add the View Component called *EnrollmentData* to the MVC project (in Models folder or a new folder called Components). It inherits from *ViewComponent*. Use following code:

public class EnrollmentDataViewComponent : ViewComponent

{

private readonly CTTIContext \_context;

public EnrollmentDataViewComponent(CTTIContext context)

{

\_context = context;

}

public async Task<IViewComponentResult> InvokeAsync(int id)

{

var result = \_context.StudentCourse.Where(sc => sc.StudentId == id)

.Select(s => new CourseSectionViewModel

{

Course = s.CourseSection.Course.CourseCode + " " +

s.CourseSection.Course.CourseName,

Section = s.CourseSection.SectionCode,

Start = s.CourseSection.StartDate.ToShortDateString(),

Instructor = s.CourseSection.Instructor.FullName

});

return View(await result.ToListAsync());

}

}

Things to note:

* Dependency injection of the context through the constructor
* Only one method called InvokeAsync that returns the view
* StudentCourse objects transformed to CourseSectionViewModel objects for a specified student id value
* The CourseSectionViewModel collection is the model passed to the view

**THE COMPONENT’S VIEW CODE:**

Next we add the View called Default.cshtml to the project. Views for View Components have to be in a specific place in order to be found at runtime. There is only one view per View Component. One of the following folder structures must be used:

1. Views/Shared/Components/<View Component Name>/<View Name>
2. Views/<Controller>/Components/<View Component Name>/<View Name>

Our folder structure following option 1: Views/Shared/Components/EnrollmentData/Default.cshtml

@model IEnumerable<CPRG102.Ctti.App.Models.CourseSectionViewModel>

<table class="table">

<thead>

<tr>

<th>

@Html.DisplayNameFor(model => model.Course)

</th>

<th>

@Html.DisplayNameFor(model => model.Section)

</th>

<th>

@Html.DisplayNameFor(model => model.Start)

</th>

<th>

@Html.DisplayNameFor(model => model.Instructor)

</th>

</tr>

</thead>

<tbody>

@foreach (var item in Model) {

<tr>

<td>

@Html.DisplayFor(modelItem => item.Course)

</td>

<td>

@Html.DisplayFor(modelItem => item.Section)

</td>

<td>

@Html.DisplayFor(modelItem => item.Start)

</td>

<td>

@Html.DisplayFor(modelItem => item.Instructor)

</td>

</tr>

}

</tbody>

</table>

Links for more information on ViewComponents:

<https://docs.microsoft.com/en-us/aspnet/core/mvc/views/view-components?view=aspnetcore-2.2>

<https://www.telerik.com/blogs/why-you-should-use-view-components-not-partial-views-aspnet-core>