minor fix

used \_ in parameter names

------------------ API Headers ----

//Put in controller

[FromHeader(Name = "header-name-xxx")]

public string HeaderNameXxx { get; set; }

//For response headers

HttpContext.Response.Headers["header -response-id"] = " some value 111";

===>------------ End API Headers ------------

---------- For swagger headers

public void Apply(Operation operation, OperationFilterContext context)

{

operation.Parameters.Add(new NonBodyParameter

{

Name = "X-MYHEADER",

In = "header",

Type = "string($date-time)",

Required = false

});

operation.Parameters.Add(new NonBodyParameter

{

Name = "X-MYHEADER\_OtherOne",

In = "header",

Type = "integer",

Required = true

});

//if (operation.Parameters == null)

// operation.Parameters = new List<Parameter>();

//operation.Parameters.Add(new Parameter

//{

// name = "X-User-Token",

// @in = "header",

// type = "string",

// required = false

//});

}

--- End swagger headers

**Middleware** sits in between client and server pipe line, and has the ability to inspect all incoming requests and outgoing responses, and if necessary, return a custom response.

CommonHttpResponseHandler

Shared

The order that middleware components are added in the Startup.Configure

Startup.Configure method adds **middleware** components for common app scenarios:

1. Exception/error handling
2. HTTP Strict Transport Security Protocol
3. HTTPS redirection
4. Static file server
5. Cookie policy enforcement
6. Authentication
7. Session
8. MVC

------------ HttpContextAccessor vs HttpContext

HttpContext:

HttpContext has had a bit of a shifting around in ASP.net core.

Inside a controller, you can still access HttpContext by doing the following :

var myUser = HttpContext.User;

Inside services is a little tricker (by using HttpContextAccessor)

First in your startup.cs, you need to register **IHttpContextAccessor** as a service like so :

services.AddMvc();

services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();

When you create a helper/service class, you can then inject in the IHttpContextAccessor and use it. It would look like something not too dissimilar to this :

var context = \_httpContextAccessor.HttpContext;

return context.User.Identities.Any(x => x.IsAuthenticated);

public class UserService : IUserService

{

private readonly IHttpContextAccessor \_httpContextAccessor;

public UserService(IHttpContextAccessor httpContextAccessor) {

\_httpContextAccessor = httpContextAccessor;

}

public bool IsUserLoggedIn() {

var context = \_httpContextAccessor.HttpContext;

return context.User.Identities.Any(x => x.IsAuthenticated);

}

}

---- End HttpContext

-------- SQL ------------

------ **Get column names from a table in SQL Server**?

SELECT \* FROM INFORMATION\_SCHEMA.COLUMNS

WHERE TABLE\_NAME = 'Courses'

and COLUMN\_NAME like '%lo%'

--- **Script to write data**

SELECT

'(''' + [int\_XYZ\_Field] + ''','

+ '''' + [Char\_XYZ\_Field] + ''''

+ ', ' + ISNULL('''' + CONVERT(VARCHAR(50), CAST([Date\_XXXXX\_Field] AS DATETIME), 121 ) + '''', 'NULL') + ''

+ ', ' + ISNULL('''' + CONVERT(VARCHAR(210), CAST([char\_XYX\_Name] AS VARCHAR(210) )) + '''', 'NULL') + '' --=> Nullable char

+'),'

FROM [dbo].[SomeTable]

-- REPLACE([char\_XYX\_Name], '''', '''''' )

------End script to write data

----------- End SQL --------

Transforming the Collection of Student Information (TCSI)

Project Overview: The TSCI project team are tasked with transforming how student information is collected, improve administration and data collection activities for education providers and ensure student claims are simpler and more accurate. TCSI will replace the Higher Education Provider Client Assistance Tool (HEPCAT) for all education providers and the Centrelink Academic Reassessment Transformation (CART) for universities. TCSI is a joint project between DHS, DET and education providers.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\* React \*\*\*\*\*\*\*\*\*\*\***

playground for JavaScript and React: <https://jscomplete.com/repl>

**Reactjs**: is for building user interfaces. Complex React applications require the use of additional libraries for state management, routing, and interaction with an API.

Using React, writing UI test cases become extremely easy.

Components are the building blocks of a React application’s UI. These components split up the entire UI into small independent and reusable pieces. Then it renders each of these components independent of each other without affecting the rest of the UI. Components can contain other components.

**render()** in React: Each React component must have a render() mandatorily. It returns a single React element which is the representation of the native DOM component. If more than one HTML element needs to be rendered, then they must be grouped together inside one enclosing tag such as <form>, <group>,<div> etc. This function must be kept pure i.e., it must return the same result each time it is invoked.

We can embed two or more components into one class by **extends** key word.

With Reactive we write html in Java script. In other liberties like jQuery, angular enhances html.

**Virtual DOM:** uses the virtual DOM instead of the real DOM.

virtual DOM is a lightweight JavaScript object which originally is just the copy of the real DOM. It is a node tree that lists the elements, their attributes and content as Objects and their properties. React’s render function creates a node tree out of the React components.

This Virtual DOM works in three simple steps.

Whenever any underlying data changes, the entire UI is re-rendered in Virtual DOM representation.

Then the difference between the previous DOM representation and the new one is calculated.

Once the calculations are done, the real DOM will be updated with only the things that have actually changed.

React follows uni-directional data flow or one way data binding.

**Props** is the shorthand for Properties in React. They are read-only components which must be kept pure i.e. immutable. They are always passed down from the parent to the child components throughout the application. A child component can never send a prop back to the parent component. This help in maintaining the unidirectional data flow and are generally used to render the dynamically generated data.

**States** are the heart of React components. States are the source of data and must be kept as simple as possible. Basically, states are the objects which determine components rendering and behaviour. They are mutable unlike the props and create dynamic and interactive components. They are accessed via this.state().

**Class and Function components:** Function is simple one while Class component is more featured which holds its private internal state along with props.

We need other liberties for a complete solution.

**JSX** is a shorthand for JavaScript XML. JSX is XML-like (HTML like template) syntax but JSX is optional. To enable a browser to read JSX, first, we need to transform JSX file into a JavaScript object using JSX transformers like Babel and then pass it to the browser.

We can modularize code by using the export and import properties. They help in writing the components separately in different files.

We can use playground like CodePen or CodeSandbox.

**React Router** is a powerful routing library built on top of React.

**Flux** is an architectural pattern which enforces the uni-directional data flow. It controls derived data and enables communication between multiple components using a central Store which has authority for all data. Any update in data throughout the application must occur here only. Flux provides stability to the application and reduces run-time errors.

**Redux** is libraries for front-end development. It is a **predictable state container** for JavaScript applications and is **used for the entire applications state management**. Applications developed with Redux are easy to test.

Redux uses ‘**Store’** for storing the application’s entire state at one place. So all the component’s state are stored in the Store and they receive updates from the Store itself. The single state tree makes it easier to keep track of changes over time and debug or inspect the application.

Redux is composed of the following components:

Action – It’s an object that describes what happened.

Reducer – It is a place to determine how the state will change.

Store – State/ Object tree of the entire application is saved in the Store.

View – Simply displays the data provided by the Store.