Staff Hours WTS Task

This solution contains my approach to solving the problem provided by Working Time Solutions.
The purpose of this document is to explain why I chose to do certain things a certain way and what I would do differently given more time.

I appreciate that this is a long read, so I have tried to make the code speak for itself as much as possible, as well as provide relevant comments. This document aims to go a bit more in depth.

Before running

The connection string for the database is stored in Web.Config for the web api project, and App.Config for the Tests project, these will need to be configured appropriately.

Dependencies

Microsoft.Net.Compilers (nuget)

This project was developed in Visual Studio 2017 and uses features of C# 7, however it should compile and build on older versions as the nuget package Microsoft.Net.Compilers is included to provide backwards compatibility - You may see intellisense errors in older versions of Visual Studio, however the build should succeed.

Newtonsoft.JSON (nuget)

Microsoft's shipped json serializer is very basic in its capabilities, so to improve this and support potential future development, Newtonsoft.JSON was chosen as a substitute.

.Net 4.6

This application was built and tested against .Net 4.6

Architecture

This project is made up of three components, the front end (html, js, css), the back end (.Net, MVC, Web API), and the Database (SQL Server)

The front end

location: `\StaffHours\StaffHours\DIST`
The front end is independent of .Net and is built with Bulma CSS and Vue JS front end frameworks. There is a temporary controller configured in the back end project to automatically serve the front end for ease of use during development, however it

could be hosted anywhere provided it has access to the api, the endpoint can be configured in `scripts\sh_Custom.js`

The design of the front end was heavily influenced by the color schemes and layout of the WTS website and product screenshots on the website.

Bulma CSS was chosen as it's an easy to use front end library for rapid application development, with a simple flexbox based column system.

Vue JS was chosen as it's an extremely lightweight front end framework, great for this scale of project that allowed very simple data binding of UI elements to the data.

The back end

The back end was developed, as required, in C#, MVC & Web API.
There are 3 sections to the back end, the controllers, the models & the utilities.

Controllers

The controllers are extremely light on business logic and simply provide an entry point into the application.
There are 4 main API's developed using these controllers, however only one is consumed by the front end. I chose to do this, as it made sense to expose the data we had in a meaningful way, even if it wasn't used in this iteration. the api's are as follows:

- **{host}/api/employee**: returns all employee rows from the database as JSON
  - **{host}/api/employee/{id}**: can also be used to get a specific employee by ID
- **{host}/api/shifts**: returns all shifts rows from the DB as JSON
- **{host}/api/employee_works_shift**: returns all employee_works_shift rows from the DB as json
- **{host}/api/employeeoverview**: this api is the one consumed by the front end and returns a list of all employees with the following properties: ID, Fullname & Total hours worked per month. The logic to calculate this information is done serverside in Utils.EmployeeOverviewFactory
  - **{host}/api/employeeoverview*/{id}**: can also be used to return a specific employees total hours

Utilities

The utilities namespace contains three main components, the DataAccess static class, the EmployeeOverviewFactory & the SqlCommandExtensions.

The data access and SqlCommandExtensions work together to provide an extremely basic custom built ORM inspired by the way Dapper ORM works. I chose to implement a custom ORM to map tables to classes using reflection in .Net as I thought for such a small application, keeping it simple would be sensible. There are some assumptions made when deserializing a table into a class which could cause errors if the assumptions are not true. These are outlined in the comments in the project.

There are multiple methods in the DataAccess class that I think could be combined and generalized to be more DRY.

The EmployeeOverviewFactory provides methods for creating EmployeeOverviews as described previously. In here is the logic used to calculate the total hours an employee will be working in a given month. The factory is instance based, meaning that the data is collected from the server when the class is instantiated. If the data needs to be refreshed at any point in time, there is a method to do so.

Models

Besides the EmployeeOverview model, the other models are designed to be 1:1 copies of the provided tables, this means ClasName = TableName and the properties = columns. This was done so that we can use reflection in .Net to map the tables to the classes as previously described.

Looking back, it would make sense to separate the models designed to replicate the DB from the rest.

The models contain no advanced logic and are used simply to store data.

Database

The database remains unchanged from the one in the brief. Given more time and an established database, I would have moved various parts of the logic away from the back end, and utilized stored procedures with SqlServer. Some of the business logic that needed to be on the back end project could still be enhanced with the use of stored procedures.

The database has no authentication currently, besides a simple read only user. In production I would recommend tighter individual user based security that doesn't require a plain text password in the connection string.

Evaluation

This project was completed over three evenings as a weekend project. Given more time there are many improvements I would make to the application.

The front end is is very basic but provides what was asked in the brief. This could be expanded further to provide more information, such as a drill down option to see exactly what shifts an employee is working.

On the back end, there are many places where the logic could be improved and tidied up to be made more testable.
My main concern with some of the logic is in the data access class, where

assumptions are made about property names matching the columns of tables. If a class contained a property not included in the table, currently an exception is thrown and caught, and the property in question is simply ignored.
This doesn't happen in this particular project as I've ensured the property names match, however if it were to happen, it would be an inefficient & wasted call to the database so it would be worth implementing custom attributes that we can decorate a property with to tell the "ORM" to look up that property in the DB. Or we could use a pre-existing ORM that does all of this for us :)

In short, I feel the requirements in the brief were met, however there is definitely room for improvement across the application.

Testing

Each controller has a test, that simply matches the count of the results to the count of the related items in the column. This could be flawed when the tests are using the same database as the main project, because an update to the table could cause the counts to mismatch, causing the tests to fail. The fix for this would be to implement a specific test database with its own connection string.

The logic for calculating the amount of hours worked per employee is tested in depth against some pre-arranged values as well as a comparison to the data in the database. The database test shares the same flaw as the controller tests in that an update to the DB could cause the tests to fail.