Design Patterns in ASP.Net Web Appications

Kien Quoc Mai

103532920

30/10/2022

COS20007 - Object Oriented Programming

Abstract

Website development is a dynamic and constantly evolving field. A web application can be as simple as a static HTML page or can be designed in an OOP way with complex architectures and design patterns. This research will investigate the use and advantages of design patterns in the ASP.Net MVC framework with regard to testability, flexibility and performance.

Table of Contents

1 Introduction

ASP.Net MVC framework is an extensive OOP framework for developing web applications and APIs. The framework relies heavily on design patterns including Model-View-Controller (MVC) and Dependency injection (DI).

MVC is an architectural pattern that divides an application into three main parts: Model, View and Controller (Ding, Liu, & Tang, 2012). According to Ding et al. (2012), model is a representation of data and provides interfaces to interact with that data. View is a display of model that provides external access to the application. On the other hand, controller is an intermediary layer between model and view. Its responsibility is to map user input from view to application behaviour, which then is dispatched to model. In addition, controller also controls which view to display the data. Overall, MVC creates a nice separation of interface logic and business logic which makes code extendable and reusable (Sarcar, 2020).

Dependency injection is a design pattern that enables the development of loosely coupled and easily maintainable programs (Seemann & van Deursen, 2019). When using DI, a class can declare its dependencies via interfaces. Then, other classes implementing those interfaces can be injected into the former class. Seemann and van Deursen (2019) stated that DI allows the use of late binding and increases code extensibility and reusability. In addition, code that implements DI is highly testable as dependencies can be replaced with mock implementations in unit tests.

In this research, the use and implementation of MVC and DI in the ASP.Net framework will be investigated via a small web application. Moreover, different aspects, including testability, flexibility, and performance, will be analysed to identify the key benefits of those patterns. In order to achieve that, a comparison between an MVC web application and a non-MVC counterpart will be conducted to highlight any advantages and drawbacks.

2 Method

2.1 To-do list application

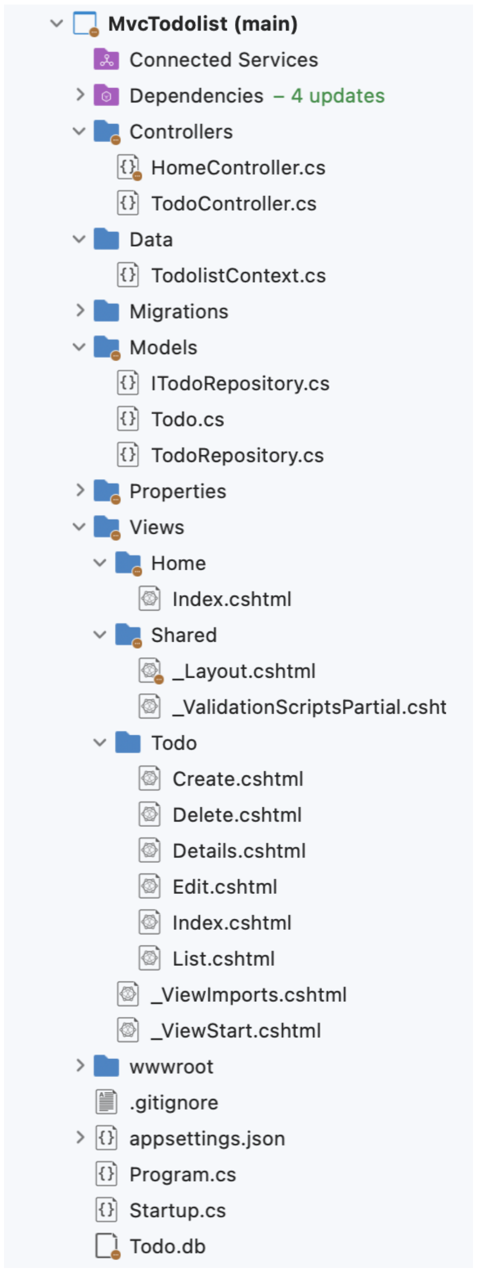
To investigate the use of MVC and DI patterns, a to-do list web application was chosen to be created using the ASP.Net Core MVC framework. To-do list is a common and simple application that is suitable for exploring a web framework since it applies all four CRUD actions (create, read, update and delete) on a single data schema. The data schema for this application only consists of 3 fields: “name”, “dueDate” and “done”. The to-do list application was designed to provide 4 basic functionalities: create a new to-do, view the to-do list, update a to-do, and delete a to-do. During the development process, the uses of MVC and DI were analysed and documented. In addition to the base features, a search feature was developed to investigate the extensibility. This is an extension of the view to-do list feature with a lot of shared logic. Lastly, the testability of the application was explored by making a few test cases for unit testing.

2.2 Non-MVC approach

In order to highlight the advantages and disadvantages of MVC patterns in ASP.Net web applications, a replica of the to-do list application was created using Razor Pages. Razor Pages is an ASP.Net Core framework that embraces a more page-focused approach for creating web applications over the MVC architecture. A comparison between the two applications was conducted with regard to performance, extensibility and testability. Firstly, the performance was measured in terms of speed, CPU and Ram usage using a benchmark program and Microsoft Visual Studio. Secondly, extensibility was compared by investigating the ease of implementing the search feature.

3 Results

3.1 MVC to-do list web application



```
MvcTodolist (main)
  Connected Services
  Dependencies – 4 updates
  Controllers
    HomeController.cs
    TodoController.cs
  Data
    TodolistContext.cs
  Migrations
  Models
    ITodoRepository.cs
    Todo.cs
    TodoRepository.cs
  Properties
  Views
    Home
      Index.cshtml
    Shared
      _Layout.cshtml
      _ValidationScriptsPartial.csht
    Todo
      Create.cshtml
      Delete.cshtml
      Details.cshtml
      Edit.cshtml
      Index.cshtml
      List.cshtml
    _ViewImports.cshtml
    _ViewStart.cshtml
  wwwroot
  .gitignore
  appsettings.json
  Program.cs
  Startup.cs
  Todo.db
```

Figure 1: MVC application's solution explorer

The application is generated with some boilerplate code using the ASP.NET Core MVC template provided by Visual Studio. As demonstrated by Figure 1, there are four primary parts separated into different modules: Data, Model, View and Controller. The relations between them as well as how they interact with each other are depicted in the following diagrams.

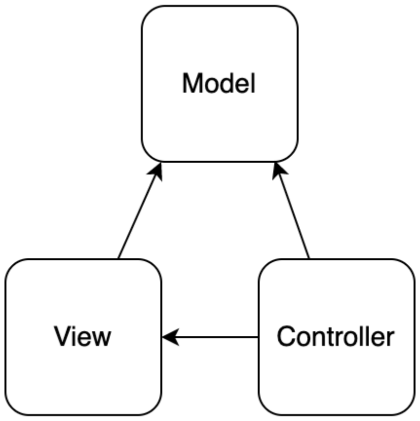


Figure 2: How components reference each other
Note. Adapted from Anderson (2022a)

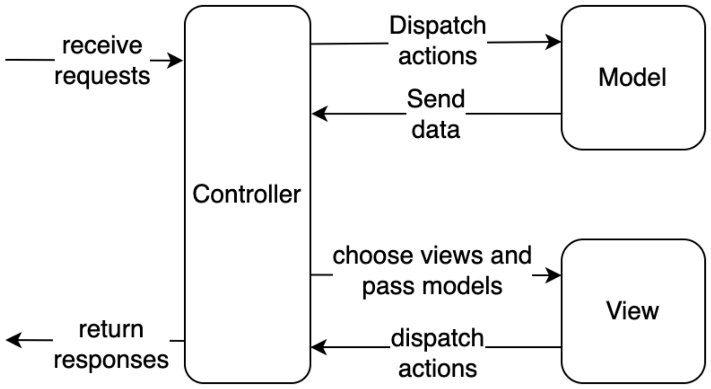


Figure 3: The application flow
Note. Adapted from Ragupathi (2019)

As can be seen from Figure 2 and Figure 3, the architecture of this application is designed based on a variation of the MVC pattern where the controller acts as an intermediary between the model and the view components. At first, the initial application only consisted of a home page. The process of creating the to-do list feature is as follows.

1. Create TodolistContext to represent the to-do data entity.
2. Create Todo model and TodoRepository, which is an interface for the Todo model. Then create TodolistContext which provides database connections.
3. Create TodoController that handles different routes of the application.
4. Create multiple views under Views/Todo.

The model is a simple class that represents the to-do entity with different properties. Below is the snippet of the Todo model.

```
public class Todo
{
    public int Id { get; set; }
    public string Description { get; set; }
    [DataType(DataType.Date)]
    public DateTime DueDate { get; set; }
    public bool Done { get; set; } = false;

    public Todo()
    {
    }
}
```

The views are special files that allow the use of C# code inside HTML files. Those views can have references to the models and can dispatch actions to different controllers. As views contain a mix of C# and HTML, the syntax is a little bit different. The "@" symbol is used to indicate C# statements. An example of one of the to-do list pages is as follows:

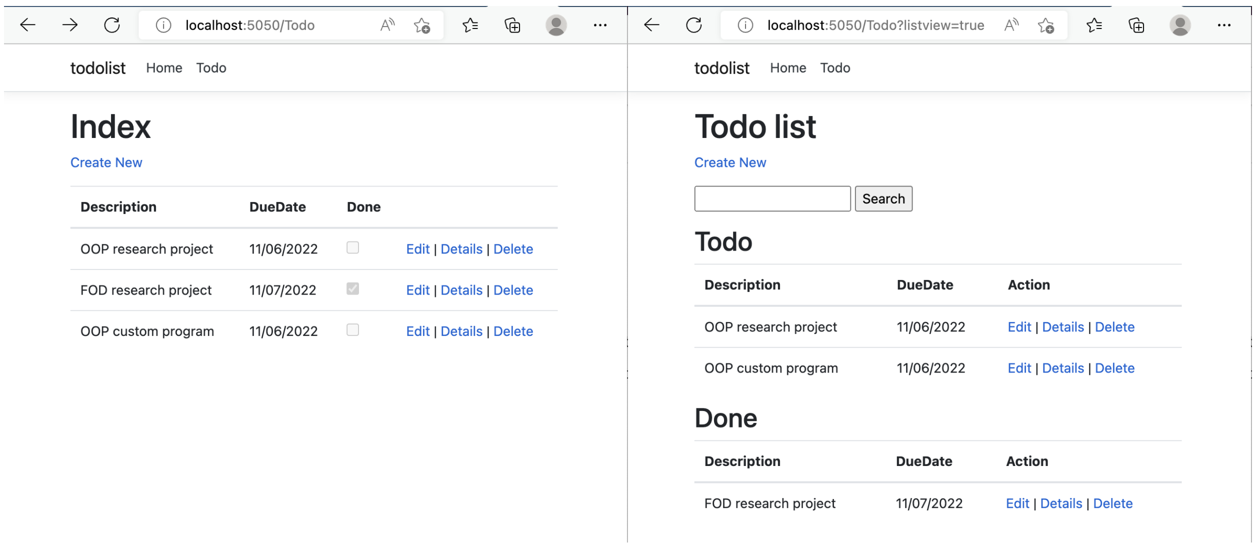
```
@model IEnumerable<MvcTodolist.Models.Todo>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Description)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.DueDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Done)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Description)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.DueDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Done)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.Id">
                    Edit
                </a> |
                <a asp-action="Details" asp-route-id="@item.Id">
                    Details
                </a> |
                <a asp-action="Delete" asp-route-id="@item.Id">
                    Delete
                </a>
            </td>
        </tr>
}
    </tbody>
</table>
```

In order to achieve the controller's intermediary responsibility, different design patterns are utilised such as strategy and dependency injection. To communicate and dispatch actions to models, a controller needs to declare all dependencies in its constructor. Then, model objects are instantiated and passed to the controller. The process of injecting objects via constructors is called constructor injection (Seemann & van Deursen, 2019). Moreover, the ability to choose which views to display is enabled through the use of strategy pattern. All the views can be treated in the same way and the controller specifies a view by its name. Below is a snippet of the todo controller.

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;

    public TodoController(ITodoRepository todoRepository)
    {
        _todoRepository = todoRepository;
    }

    // GET: Todo
    public async Task<IActionResult> Index(string searchString)
    {
        if (!String.IsNullOrEmpty(searchString))
            return View(
                "Index",
                await _todoRepository.SearchTodo(searchString)
            );

        return View("Index", await _todoRepository.GetAllTodos());
    }
}
```

As mentioned in the Method section, the List view is an extension of the default Index view. To select the List view instead of the Index view, the Todo controller only needs to change the first parameter to "List" when calling the View function. In addition, that value can be dynamically changed at runtime.

```
// GET: Todo
public async Task<IActionResult> Index(
    string searchString,
    bool listView
)
{
    string view = listView ? "List" : "Index";

    if (!String.IsNullOrEmpty(searchString))
        return View(
            view,
            await _todoRepository.SearchTodo(searchString)
        );

    return View(view, await _todoRepository.GetAllTodos());
}
```



localhost:5050/Todo

todolist Home Todo

Index

Create New

Description	DueDate	Done	
OOP research project	11/06/2022	☐	Edit | Details | Delete
FOD research project	11/07/2022	☑	Edit | Details | Delete
OOP custom program	11/06/2022	☐	Edit | Details | Delete

localhost:5050/Todo?listview=true

todolist Home Todo

Todo list

Create New

Search

Todo

Description	DueDate	Action
OOP research project	11/06/2022	Edit | Details | Delete
OOP custom program	11/06/2022	Edit | Details | Delete

Done

Description	DueDate	Action
FOD research project	11/07/2022	Edit | Details | Delete

(a) Default index view *(b) Improved list view*

Figure 4: To-do list index page with the default and the improved list views

Note. Notice in the address bar that those two are the result of the same controller

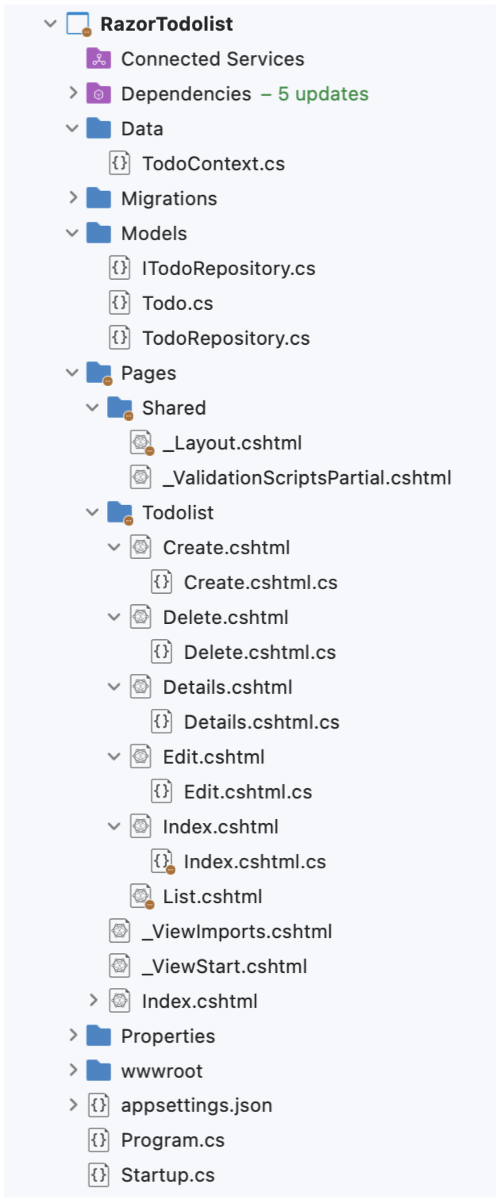
3.2 Unit test

The code snippet for the controller unit test is as follows.

```
[SetUp]
public void Setup()
{
    _mockRepository = new Mock<ITodoRepository>();
    _mockRepository
        .Setup(r => r.GetAllTodos())
        .Returns(Task.FromResult((new List<Todo>
        {
            new Todo
            {
                Id = 1,
                Description = "todo1",
                Done = false,
                DueDate = System.DateTime.Now
            },
            new Todo
            {
                Id = 2,
                Description = "todo2",
                Done = true,
                DueDate = System.DateTime.Now
            },
        }).AsEnumerable()));

    _todoController = new TodoController(
        _mockRepository.Object
    );
}

[Test]
public void TestGetAllTodos()
{
    var view = _todoController.Index("");

    _mockRepository.Verify(
        r => r.GetAllTodos(),
        Times.Once()
    );
}
```

The unit test was implemented using the NUnit test framework. In the Setup method, a TodoController object was instantiated with a mock version of the TodoRepository. Then, in the test method, in the assertion part, the Verify method was called to ensure that the GetAllTodos method was called once by the Todo controller.

3.3 Razor Pages version



```
RazorTodolist
  Connected Services
  Dependencies – 5 updates
  Data
    TodoContext.cs
  Migrations
  Models
    ITodoRepository.cs
    Todo.cs
    TodoRepository.cs
  Pages
    Shared
      _Layout.cshtml
      _ValidationScriptsPartial.cshtml
    Todolist
      Create.cshtml
        Create.cshtml.cs
      Delete.cshtml
        Delete.cshtml.cs
      Details.cshtml
        Details.cshtml.cs
      Edit.cshtml
        Edit.cshtml.cs
      Index.cshtml
        Index.cshtml.cs
      List.cshtml
    _ViewImports.cshtml
    _ViewStart.cshtml
    Index.cshtml
  Properties
  wwwroot
  appsettings.json
  Program.cs
  Startup.cs
```

Figure 5: Razor Pages application's solution explorer

A non-MVC version of the to-do list application was created using Razor Pages. The structures of the two are similar and the only difference being that the Razor Pages version does not have any controllers. Instead, page models are used to implement the application's logic. This resembles the Model-View-View Model architecture. Below is the code snippet for the index page's model.

```
public class IndexModel : PageModel
{
    private readonly ITodoRepository _todoRepository;

    public IndexModel(ITodoRepository todoRepository)
    {
        _todoRepository = todoRepository;
    }

    public List<Todo> Todo { get; set; }
    [BindProperty(SupportsGet = true)]
    public string SearchString { get; set; }

    public async Task OnGetAsync()
    {
        if (!String.IsNullOrEmpty(SearchString))
            Todo = (List<Todo>)await _todoRepository.SearchTodo(
                SearchString
            );
        else
            Todo = (List<Todo>)(
                await _todoRepository.GetAllTodos()
            );
    }
}
```

Similar to controller, page model also uses the dependency injection pattern. A page can specify which page model to use like this.

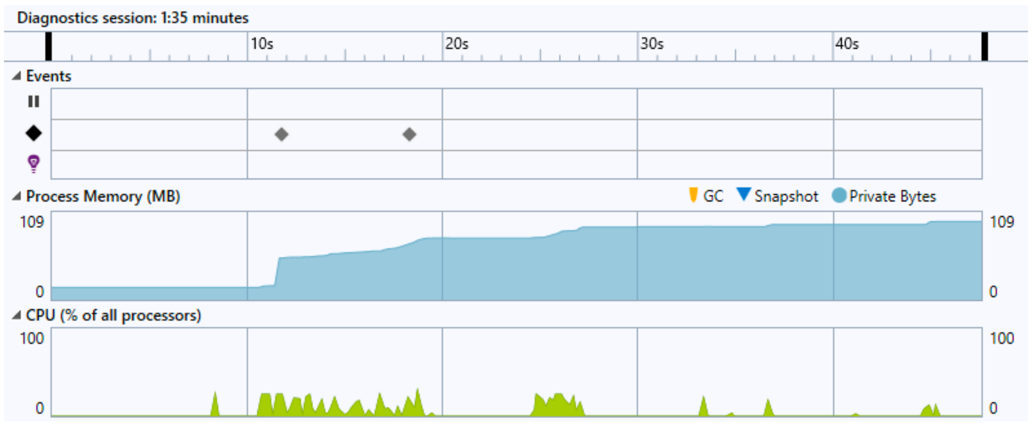
```
@model RazorTodolist.Pages.Todolist.IndexModel
```

The key difference between this architecture and the MVC counterpart is that a page is statically bound to a single page model which cannot be changed at runtime. As a result, the new List page needs to specify its page model so as to share the same model with the Index page.

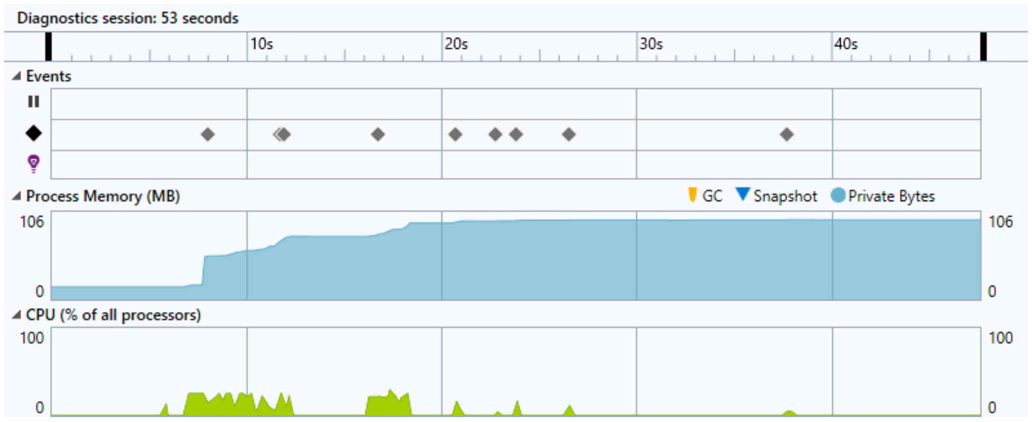
3.4 Comparison between MVC and non-MVC architectures

Application	Average Load Time (millisecond)
MVC	176
Razor Pages	182

Table 1: Performance comparison results



(a) MVC application



(b) Razor Pages application

Figure 6: CPU and RAM usage comparison results

Firstly, Table 1 shows that the loading times for both versions are relatively similar with an insignificant difference of a few milliseconds. Similarly, RAM and CPU usages are nearly identical at around 100 MB of memory and 25% of all processors at peak as demonstrated by Figure 6. Lastly, the process of building a new page, which extends the default index page, was relatively easy and flexible in the case of the MVC application since the controller can choose what view to display. Meanwhile, the Razor Pages application requires a new page to specify its page model. This allows the new List page to reuse the existing page model, but this behaviour cannot be modified at runtime.

4 Discussion

4.1 Findings interpretation

The implementation of Model-View-Controller and Dependency injection patterns brings several benefits to ASP.Net web applications. First and foremost, MVC provides great architecture as it embraces flexibility and reusability. As demonstrated in the to-do list application, data, business logic and user interface are kept separate from each other. This greatly increases the flexibility of the program since components can be developed in parallel and the connections between them are very flexible. For instance, a view can easily be replaced with a different view without making any major changes to the model. In addition, MVC pattern enhances the application's cohesion and reduces coupling. High cohesion is achieved by grouping components, such as views and controllers, together. To achieve loose coupling, abstraction is applied by making classes depend on interfaces instead of concrete implementations. Secondly, DI pattern also enables loosely coupled and extensible programs. The practice of injecting models into controllers is according to the Liskov substitution principle: an implementation of an interface can be replaced with another implementation without breaking either the client or the implementation (Seemann & van Deursen, 2019). As a result, the program is more testable since unit tests can easily be conducted on a controller by replacing its dependencies with mocked implementations. Additionally, DI pattern also supports parallel development and makes a program more maintainable thanks to the decoupling of controllers and other services. As shown in the to-do list application, TodoController and TodoRepository were developed parallelly with the common ITodoRepository interface.

Despite the aforementioned crucial advantages, MVC and DI patterns come with a few drawbacks. First of all, those patterns introduce complexity to an application. The number of classes and boilerplate code greatly increases as the program is separated into several major parts. Although complexity is not always considered a disadvantage, it creates a high learning curve that requires more skilful developers. Moreover, such complexity may not suitable for small applications, such as a prototype, since high complexity can lead to a slow development process. There are also no significant performance benefits in terms of execution speed and resource usage.

4.2 Limitations

This study has some limitations that might affect the findings. Firstly, the actual implementation of the project's boilerplate code was not examined closely. That part of the program is what enables the use of MVC components and Dependency injection. Investigation into it could potentially yield more insights into how MVC and DI patterns are set up. Secondly, the demo program used in this study was relatively small and simple. Hence, some benefits, including maintainability and performance, could not be observed easily.

4.3 Further research recommendations

As already mentioned, a study on how MVC and DI patterns are applied in complex web applications, which have a large number of pages and computational-heavy actions, can yield interesting findings. Some of the key aspects to investigate include maintainability and performance. Furthermore, other alternative architectures to MVC, such as Hierarchical Model-View-Controller (HMVC), Model View Presenter (MVP) and Presentation Abstraction Control (PAC), are great topics to explore. A comparison between different architectures could provide insightful knowledge of the advantages and disadvantages of each architecture.

5 Conclusion

This research has discussed the use of Model-View-Controller and Dependency injection patterns in ASP.Net core web applications. Two versions of a simple to-do list application were implemented using ASP.Net MVC and ASP.Net Razor Pages frameworks. Different aspects were investigated, such as the implementation, the process of extending and writing unit tests, in order to identify the role of those patterns in a web application. To summarise, MVC and DI patterns are essential to an ASP.Net web application as they make it more flexible, extensible and testable.

References

Anderson R. (2022a). *Get started with ASP.Net core MVC*. Retrieved November 1, 2022, from https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc

Anderson R. (2022b). *Tutorial: Create a razor pages web app with ASP.Net core*. Retrieved November 1, 2022, from https://learn.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/

Ding Y. H., Liu C. H., & Tang Y. X. (2012). MVC pattern based on Java. *Applied Mechanics and Materials*, *198-199*, 537–541. https://doi.org/10.4028/www.scientific.net/AMM.198-199.537

Microsoft. (2022a). *Razor pages unit tests in ASP.Net core*. Retrieved November 1, 2022, from https://learn.microsoft.com/en-us/aspnet/core/test/razor-pages-tests

Microsoft. (2022b). *Testing without your production database system*. Retrieved November 1, 2022, from https://learn.microsoft.com/en-us/ef/core/testing/testing-without-the-database

Ragupathi M. (2019). *How web works & ASP.Net MVC fits into web application development*. Retrieved November 2, 2022, from https://www.dotnetodyssey.com/asp-net-mvc-5-free-course/how-web-works-asp-net-mvc-fits-web-application-development/

Sarcar V. (2020). MVC pattern. In V. Sarcar (Ed.), *Design Patterns in C#: A Hands-on Guide with Real-world Examples* (pp. 495–519). Apress. https://doi.org/10.1007/978-1-4842-6062-3_26

Seemann M., & van Deursen S. (2019). *Dependency injection principles, practices, and patterns*. Manning Publications. Retrieved October 31, 2022, from https://learning.oreilly.com/library/view/dependency-injection-principles/9781617294730/

6 Appendix

6.1 Full implementation

All the code done for this study is inspired by great tutorials from Anderson (2022a & 2022b) and can be found in this Github repository:

- MVC implementation
- Razor Pages implementation
- NUnit tests
- Benchmarking code

6.2 Comparison data

Test No.	MVC	Razor Pages
1	134	174
2	203	211
3	165	208
4	204	209
5	160	175
6	172	153
7	199	174
8	201	172
9	155	166
10	167	177
Average	176	182

Table 2: Loading time results for MVC and Razor Pages to-do list (millisecond)