UCN dmai0916

FINAL PROJECT REPORT

14th of January 2019

Andreas Richardsen

Zahro-Madalina Khaji

University College of Northern Denmark

Technology and Business

Computer Science AP Degree Programme

Final Project Report

dmai0916

Project participants:

Andreas Richardsen

Zahro-Madalina Khaji

Supervisor:

Simon Kongshøj

Submission Date:

14th of January 2019

Contents

Introduction

In this report we will discuss the process of developing our final project for the degree of AP Computer Science. We chose to build a Personal Task Manager that can easily be used to replace the classic TO DO list. This project is a web application that we implemented using technologies that we learned through this degree and personal research. Throughout this report we will present the different stages of development that we experienced, our choices of technology and the implementation of this project.

Our goal with this project was to offer an improved organisation method that can help anyone live a simpler and more stress-free life.

Problem statement

The old TO DO list method of organising to solve tasks has several deficiencies that could be improved.

To create a TO DO list the old way a user requires access to paper and a writing instrument. In our modern age, access to the internet is very common and technology is very wide spread. Therefore, a digital TO DO list could be a better option with possibly more functionality.

Here are some as the aspects that is lacks: not easily accessible. E.g., when a user goes grocery shopping and forgets his list at home. Therefore, an improved list should be available to access anywhere, anytime with just an internet connection.

Another issue is that of security. Suppose the user wants to keep the TO DO list private – a list written on a plain piece of paper might be seen by anyone around him who might be snooping or can be lost and who knows who might find it. This shows the importance of keeping your data safe.

Writing the tasks on a list on paper means that it is hard to edit or rearrange them if the need arises and takes unnecessary space. Important tasks for the user might also not be properly evident so they might be overlooked. Therefore, a method of organising, categorising and highlighting list tasks is necessary.

- What are some ways in which we can improve the old-fashioned TO DO list?

    1) How to present the tasks in a more organised manner?
    2) How to highlight the user's most important tasks?
    3) What are some ways to make sure that our list is accessible to the user anytime and anywhere with an internet connection?
    4) Which are some ways to keep our data secure?

Analysis

Gathering requirements

For this project we have not partnered with a company to build a solution for their needs but instead we have decided to work on our own idea. This meant that during our preliminary study we could not obtain any requirements from a customer, such as a company.

Identifying requirements

From our problem statement we identified the following requirements:

- First question: "How to present the tasks in a more organised manner?"
  - Management of tasks
  - Categorising tasks as lists
- Second question: "How to highlight the user's most important tasks?"
  - Favorite system for tasks
- Third question: "What are some ways to make sure that our list is accessible to the user anytime and anywhere with an internet connection?"
  - A database for storing data
  - A service to connect and utilise the database
  - A web client that can connect and use the service over the internet
- Forth question: "Which are some ways to keep our data secure?"
  - Implementing security measures to restrict unwanted access to the system

System vision

The way our system is built is with the idea of organisation in mind. Users would have the ability to manage their tasks in several ways. One could choose to have individual tasks and they would also have the possibility of grouping their tasks into lists. Lists consists of one or more tasks. Each task can have other tasks called subtasks. A task's subtask has a maximum depth limit of four (See Appendix A). To highlight a certain task a user can favorite it.

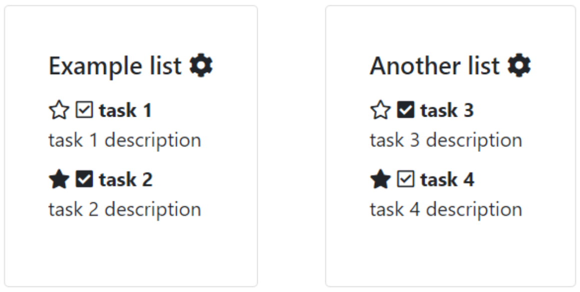


Figure 1: Task lists

Brief use cases

We decided to represent our functional requirements as brief use cases. During our analysis we discovered that none of our use cases were complex enough to necessitate making a fully dressed use case.

Manage account (Create account): User navigates to the login page, selects the create account option, the user inputs the required information for creating the account, the system creates the users account and the user is redirected to the log in page.

Manage account (View account): The user selects the account settings option, and then chooses the option View account; the system displays the account information on the page.

Manage account (Update account): User selects the account settings option, and then he changes the information he wishes to change and saves it, the system records the changes.

Manage account (Delete account): The user selects the account settings option, then selects Delete account and is prompted with a popup to confirm that he wants to delete his account; the system removes the users account information and redirects the user to the login page.

Log in: The user inputs their credentials and the system redirect them to their dashboard.

Log out: The user selects the log out option from the contextual menu and the system log out the user.

Manage task (Create task): The user presses the add task button on the view list page, then inserts a name for the task and presses save, the task is saved to the system and is displayed in the list.

Manage task (Update task): The user selects the task on the view list page, then updates the desired information and saves the task, the system records the task and the task is updated.

Manage task (Delete task): The user selects the task on the view list page, then presses the delete task where he is prompted with a popup that ask if he is sure, when the user presses yes, the task is deleted from the records and the view is refreshed.

Add subtask: The user presses the add subtask button on a selected task, then inputs the name for the subtask and presses save, the system records the task and the subtask is displayed on the task.

Remove subtask: The user presses the remove subtask button on a selected subtask, the system asks for confirmation and then deletes the subtask from the task.

Favorite task: The user presses the favorite button on a selected task, the system marks the task as a favorite and highlights it.

Search task: The user inputs the name of the desired task in the search bar; the system displays the results that match the input provided by the user.

Mark done: The user marks a selected task or subtask as done; the system displays a check mark next to it.

Prioritisation of use cases

1) Manage task: is the most important use case because tasks are the core of a TO DO list.
2) Manage account: is the second most important use case; accounts are where all the user's data is stored.
3) Add subtask: is a moderately important use case because it adds another level of organisation to the tasks.
4) Remove subtask: is on the same level as the use case before it.
5) Mark done: this use case is fairly important as it helps the user differentiate between tasks that are completed and tasks that are not.
6) Favorite task: has a mild importance; tasks that are favorites should be properly highlighted.
7) Log in: low importance; basic need for a log in.
8) Log out: also, low importance; necessary for logging out.
9) Search task: not very important unless one has many lists.

Non-functional requirements

During the analysis we decided that these are non-functional requirements that the system should fulfill:

- User friendly
- Accessible
- Fast
- Secure

Domain model

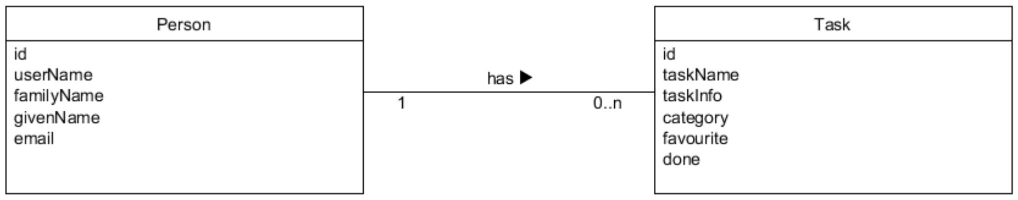


Figure 2: Domain model

Associations and patterns

The domain model contains an association between the Person class and the Task class. One person can have zero or many tasks, but a task can only belong to one person. Our domain model does not have any patterns as it is too simple.

Business considerations

To conclude the Analysis part of this report we have included some business analysis to gather information about the marketplace that our application would be launched in.

In terms of competitors, the market offers a lot of different variations of task management applications.

Our application would be free to use, and the users have the option to support the developers through a tip page for upkeep of the servers and other maintenance jobs.

Design

Architecture

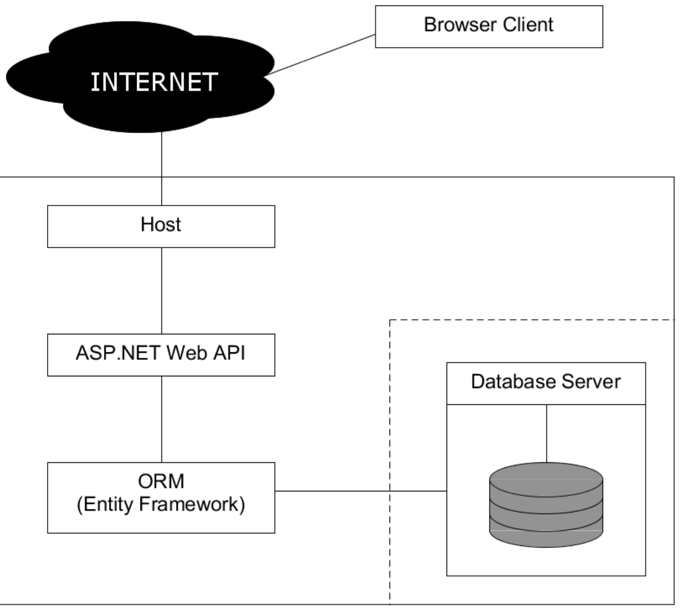


Figure 3: Architecture

Web application architecture is a complex topic that is fundamental for the development of maintainable and good quality applications. Nowadays there are many types of web architectures, and for our Personal Task Manager we tried to adopt an architecture best suited for the needs of our users and the scope and scale of our project.
A web application can be generally split into two main components: the client-side and the web server side. The client side is what the user sees and interacts with when they are using our application. For our client side we have chosen a Javascript framework called Angular which is used to build Single Page Applications or SPAs. A simple explanation of a SPA is this: “an app that works inside a browser and does not require page reloading during use.” [1]
Therefore, on the client side we have a SPA architecture.
Angular is an opinionated framework which means that are certain rules and guidelines that must be followed when developing. It used the Model View Controller (MVC) pattern to structure its code, which

is a widely spread software architecture pattern. MVC "separates out the application logic into three separate parts, promoting modularity and ease of collaboration and reuse. It also makes applications more flexible and welcoming to iterations" [2].
On the web server side, we have an ASP.NET web application architecture, which is also service oriented. The server side was build using ASP.NET Web API which is "a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices. ASP.NET Web API is an ideal platform for building RESTful applications on the .NET Framework." [3] Web API also relies heavily the MVC pattern for structuring its code.
The client side and server side of the application use Json to communicate.

User interface

To help us visualise our UI in a better way we have created a simple mockup. This mockup was quickly build using HTML and CSS and it represents our original vision for the UI.

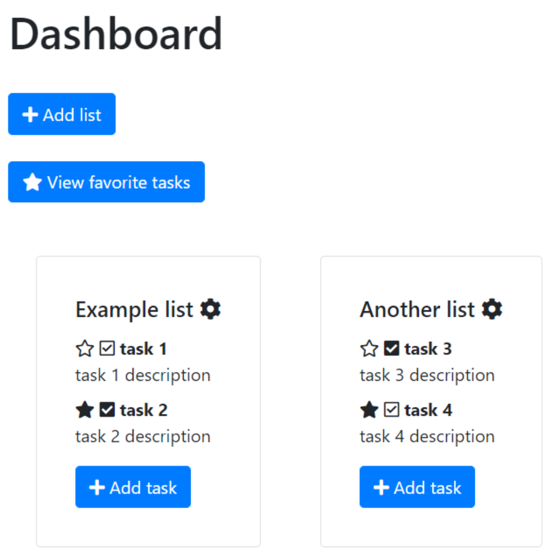
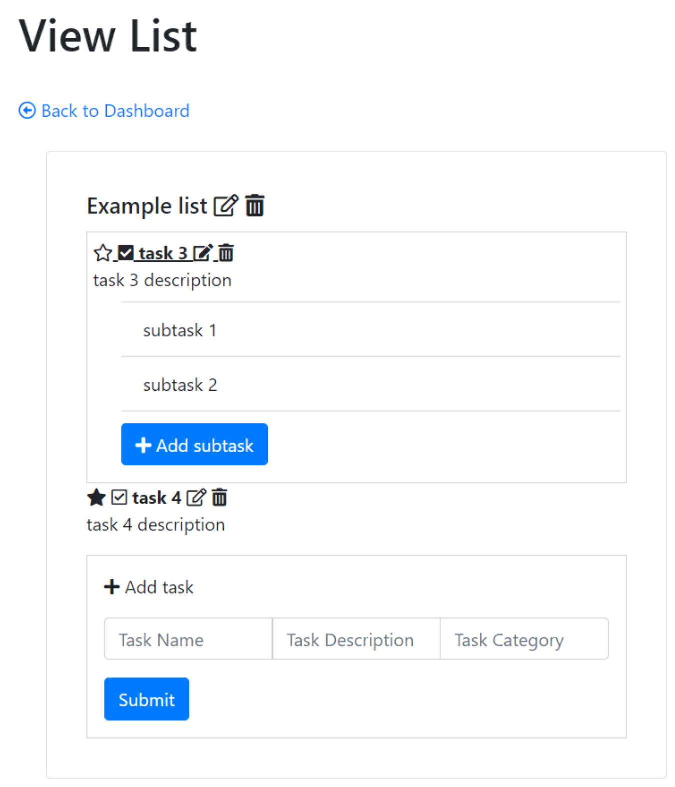


Figure 4: Dashboard Mockup

This first figure is the design for our Dashboard. The Dashboard is a central part of our application and it offers a large view of all the users' tasks and lists. We added many icons in our design to try to improve the user's experience by having a more intuitive and easier to use UI.

This design is tasked with helping us solve these following requirements. The first requirement it addresses is the requirement for implementing a favorite system the tasks. Each task has a small star icon which can be either filled or empty. By clicking on the star icon, a user can mark a task a favorite and make it stand out from other tasks that are by default assigned an empty star. This very simple yet effective favorite system is our take on solving this requirement.

Secondly, the requirement concerning categorising the tasks as lists, is derived from the need for organisation. The way we offer to solve it is by having the tasks displayed in a list format that is very clean and easy to use thanks to our design. Each list of tasks is simple yet distinctive and therefore, we consider, that it satisfies this requirement.


View List

Back to Dashboard

Example list

task 3

task 3 description

subtask 1

subtask 2

Add subtask

task 4

task 4 description

Add task

Task Name | Task Description | Task Category

Submit

Figure 5: List Mockup

This image our mockup represents a single list view. This view is accessed by clicking on a list on the Dashboard. It is supposed to be an extension to the view of a list located on the Dashboard. The list shown in this figure also has many intuitive icons. This design allows us to view more details and it offers us more options regarding the management of a list of tasks. At a glance the user can add or create a task and edit or delete tasks. This design along with other elements of our implementation, are part of our solution for the Manage tasks requirement.

Relational model

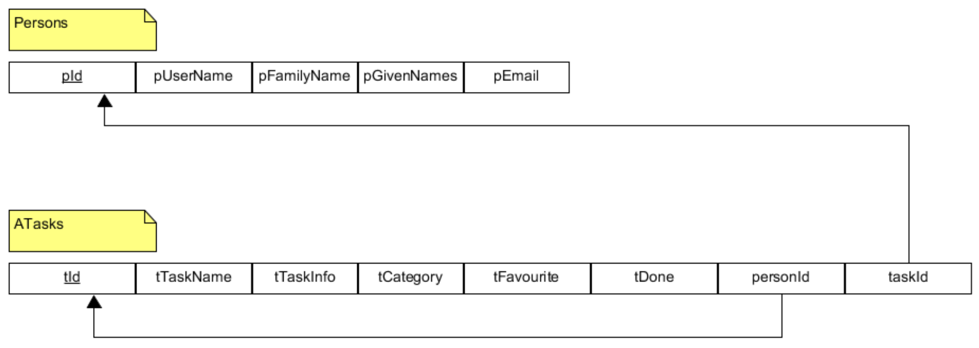


Figure 6: Relational model

Implementation

Back End

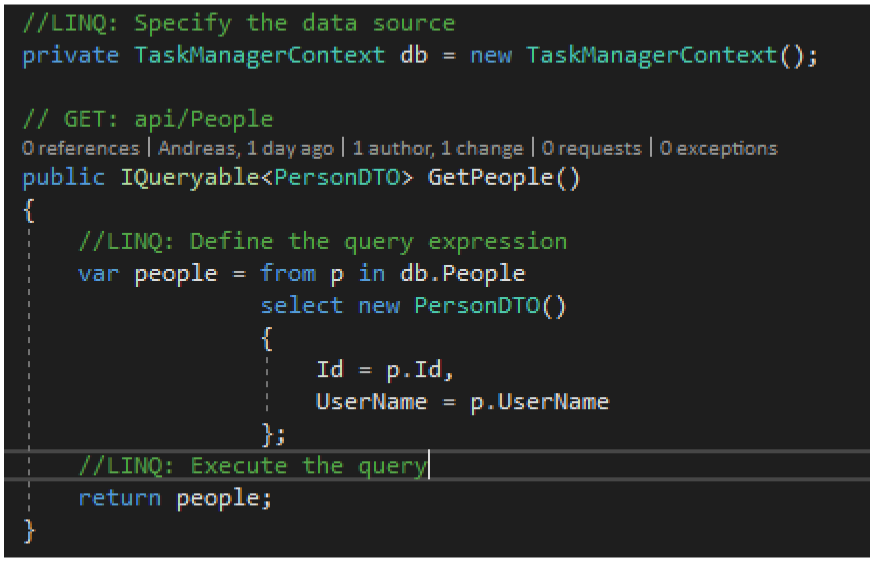
Entity Framework

We use Entity Framework 6, which is the latest version of Entity Framework. EF is an object-relational mapper (O/RM), EF eliminates the need for a large amount of data access "plumbing" code that developers would else have to write and allows developers to write applications which interact with data in relational databases using strongly-typed .NET objects which represent the domain. [6]

LINQ

Language-Integrated Query (LINQ) is a set of technologies based on the integration of query directly into C#. Using LINQ removes the need to write SQL as strings, removes the need to learn different query languages, because LINQ is a first-class language construct just like classes, methods and events.

Developers use the query expression to write the query in a declarative query syntax. By using query syntax, you can perform filtering, ordering and grouping operations on data sources with a low amount of code. LINQ makes it so the query expression is used to query and transform data in SQL databases, ADO.NET datasets, XML documents and streams, and .NET collections. See Figure X for the complete query operation. [7]



```
//LINQ: Specify the data source
private TaskManagerContext db = new TaskManagerContext();

// GET: api/People
0 references | Andreas, 1 day ago | 1 author, 1 change | 0 requests | 0 exceptions
public IQueryable<PersonDTO> GetPeople()
{
    //LINQ: Define the query expression
    var people = from p in db.People
                 select new PersonDTO()
                 {
                     Id = p.Id,
                     UserName = p.UserName
                 };
    //LINQ: Execute the query
    return people;
}
```

Figure 7: LINQ query from PeopleController

Query expression overview

- Query expression can be used to transform and query data from all LINQ-enabled data source.
- Query expression is easy to use and learn because they use C# language constructs.
- Even though variables is strongly typed, you do not have to give the type explicitly because the compiler can deduct it.
- You cannot execute until you iterate over the query variable.
- Query expressions are converted to Standard Query Operator method calls according to the rules set in the C# specification at compile time.

- Query expressions are compiled to expression trees or to delegates, decided by the type that the query is applied to. IEnumerable<T> queries are compiled to delegate. JQueryable and JQueryable<T> queries are compiled to expression trees.

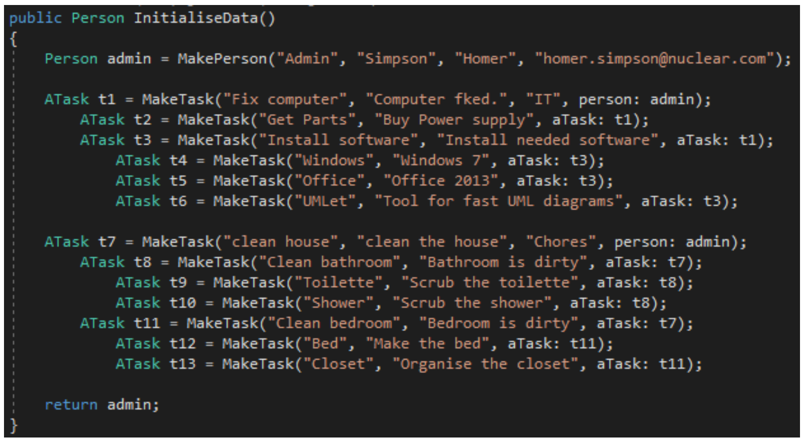
MSSQL

Microsoft SQL Server (MSSQL) is a relational database management system, its primary function is storing and retrieving data. The reason we chose MSSQL is because we have great experience with it and it works well with the other technologies we are using.

EF model: Code first

The developer writes code which specifies the model. EF generates the models and mappings at runtime based on entity classes and additional model configuration. [6]

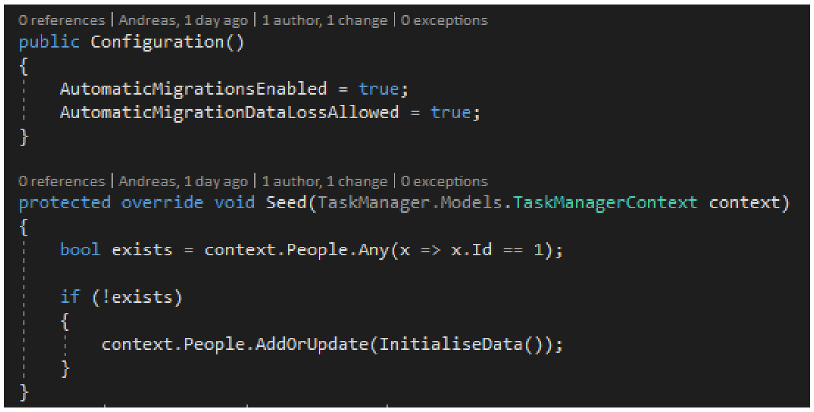
We decided on using code first to model the database, as we like to work with code more than database.



```
public Person InitialiseData()
{
    Person admin = MakePerson("Admin", "Simpson", "Homer", "homer.simpson@nuclear.com");

    ATask t1 = MakeTask("Fix computer", "Computer fked.", "IT", person: admin);
        ATask t2 = MakeTask("Get Parts", "Buy Power supply", aTask: t1);
        ATask t3 = MakeTask("Install software", "Install needed software", aTask: t1);
            ATask t4 = MakeTask("Windows", "Windows 7", aTask: t3);
            ATask t5 = MakeTask("Office", "Office 2013", aTask: t3);
            ATask t6 = MakeTask("UMLet", "Tool for fast UML diagrams", aTask: t3);

    ATask t7 = MakeTask("clean house", "clean the house", "Chores", person: admin);
        ATask t8 = MakeTask("Clean bathroom", "Bathroom is dirty", aTask: t7);
            ATask t9 = MakeTask("Toilette", "Scrub the toilette", aTask: t8);
            ATask t10 = MakeTask("Shower", "Scrub the shower", aTask: t8);
        ATask t11 = MakeTask("Clean bedroom", "Bedroom is dirty", aTask: t7);
            ATask t12 = MakeTask("Bed", "Make the bed", aTask: t11);
            ATask t13 = MakeTask("Closet", "Organise the closet", aTask: t11);

    return admin;
}
```

Figure 8: Migrations InitialiseData

We create some data to be inserted into the database at initialisation.



```
0 references | Andreas, 1 day ago | 1 author, 1 change | 0 exceptions
public Configuration()
{
    AutomaticMigrationsEnabled = true;
    AutomaticMigrationDataLossAllowed = true;
}

0 references | Andreas, 1 day ago | 1 author, 1 change | 0 exceptions
protected override void Seed(TaskManager.Models.TaskManagerContext context)
{
    bool exists = context.People.Any(x => x.Id == 1);

    if (!exists)
    {
        context.People.AddOrUpdate(InitialiseData());
    }
}
```

Figure 9: Migrations Configuration

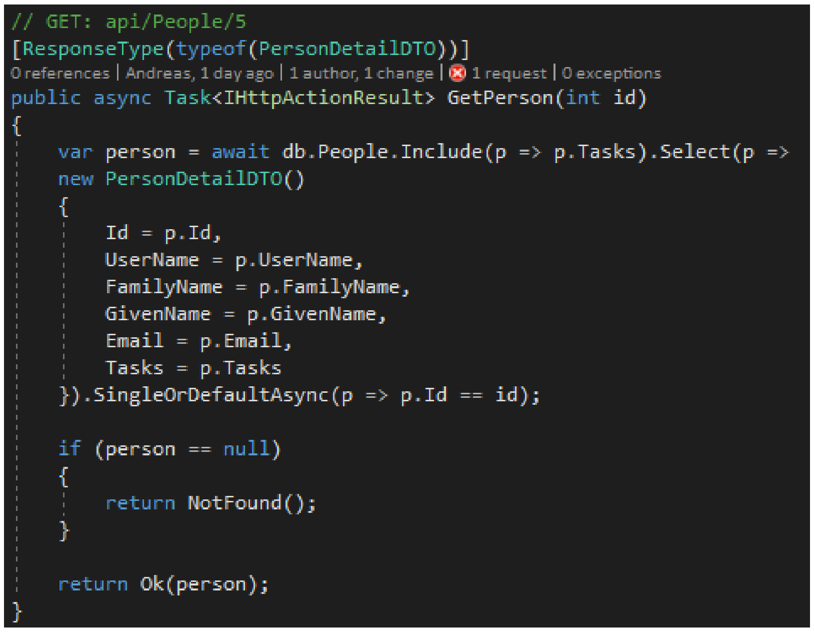
Checks if a person with id 1 exist, if it does not adds the data from InitialiseData() to the database.

ASP.NET Web API

ASP.NET Web API is a framework for creating web APIs on top of the .NET Framework that makes it easy to build HTTP services that can be connected by a various amount of clients. It is an ideal platform for building RESTful applications on. [8]

We decided on using Web API because it creates HTTP services which returns data and we wanted to make our client in the browser using Angular.js. Our other choice which we considered was MVC, but it creates web applications that return both view and data compared to Web API which only returns data, it could be JSON, XML, ATOM or other formatted data, self-hosting which is not in MCV.

Action Results

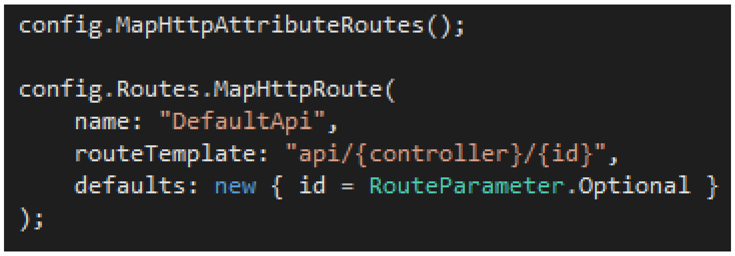


```
// GET: api/People/5
[ResponseType(typeof(PersonDetailDTO))]
0 references | Andreas, 1 day ago | 1 author, 1 change | 1 request | 0 exceptions
public async Task<IHttpActionResult> GetPerson(int id)
{
    var person = await db.People.Include(p => p.Tasks).Select(p =>
    new PersonDetailDTO()
    {
        Id = p.Id,
        UserName = p.UserName,
        FamilyName = p.FamilyName,
        GivenName = p.GivenName,
        Email = p.Email,
        Tasks = p.Tasks
    }).SingleOrDefaultAsync(p => p.Id == id);

    if (person == null)
    {
        return NotFound();
    }

    return Ok(person);
}
```

Figure 10: GetPerson

The GetPerson(int id) in PeopleController gets a Person by their Id, it includes the tasks of the person and uses the DTO class of Person PersonDetailDTO (See DTO for more information). It then converts the return value into an HTTP response message, this method returns IHttoActionResultm which means it calls ExecuteAsync to create an HttpResponseMessage, then converts it to a HTTP response message. [8]

Routing



```
config.MapHttpAttributeRoutes();

config.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

Figure 11: Default route

When the Web API framework receives a request, it routes the request to an action. To decide what action to invoke, the framework uses a routing table. In this remplate, “api” is literal path segment, {controller} and {id} are placeholders. When the Web API framework receives an HTTP request, It tries to match the URI to one of the route templates in the routing table. If no route matches, the client receives a 404 error. The {controller} is replaced with one of the controller classes and {id} is used when you want to find a specific object. To find the action, Web API looks at the HTTP verb, and then looks for an action whose name begins with that HTTP verb name.

DTO

A Data Transfer Object (DTO) is an object that defines how the data will be sent over the network. It is used to change the shape of the data you send to clients. You might want to: [8]

- Remove circular references
- Hide some properties that clients are not supposed to view
- Omit some properties in order to reduce payload size
- Flatten object graphs that contain nested objects, so they are more convenient for clients
- Avoid "over-posting" vulnerabilities
- Decouple the service layer from the database layer

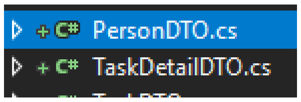


Figure 12: PersonDTO & PersonDetailDTO

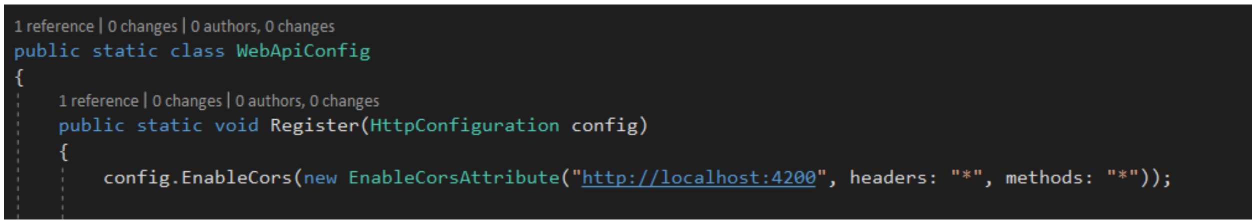
We uses two DTO classes, one contain only the Id and UserName, while the other contains all the fields. In "Figure 7: LINQ query from PeopleController", we use PersoDTO because we want a list of all the people we have and we only want to show the Id and UserName. In "Figure 10: GetPerson", we use PersonDetailDTO because we want all the information of the person we specified.User Authorisation and Authentication

Security

To satisfy the requirement concerning implementing security we decided that we need to have user authorisation and authentication. First, we created a database to store the user's necessary account information. To achieve the database we wanted, we used Asp.Net Identity which is a tool with many features including user authorisation and authentication. Along with Asp.Net identity we have also used Entity Framework with the Code First option. Once we have defined and connected to the database, we created an Account Controller with which to register users with.

Enabling CORS

A very common issue in web development is Cross-Origin Resource Sharing or CORS. To enable CORS we have used the NuGet package WebApi.Cors. In order to allow requests from applications located on localhost:4200 which is the browser client's location, we had to add the line below to the WebApiConfig file of our API.

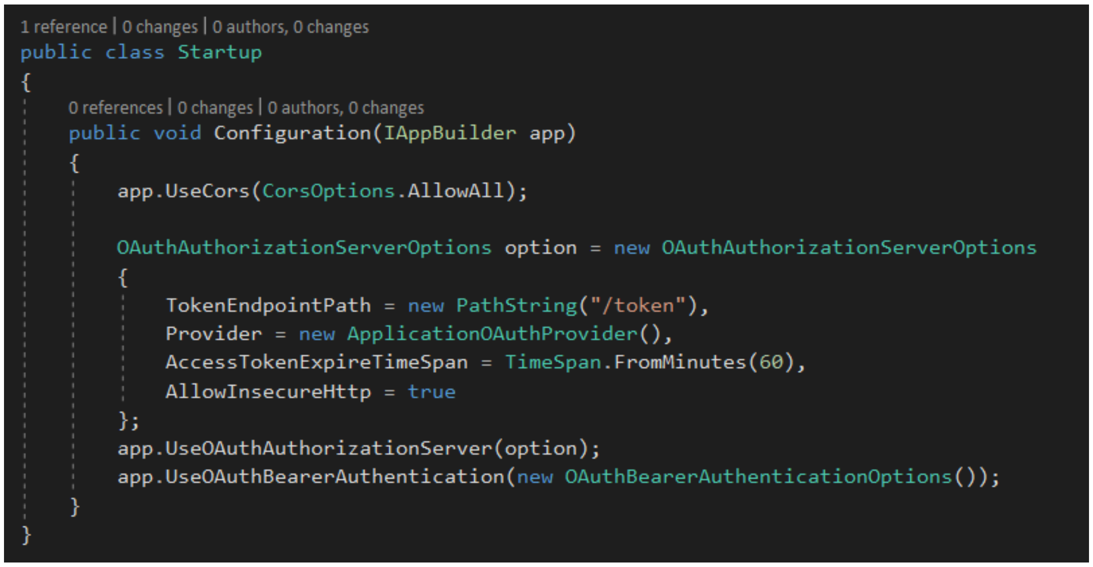


```
1 reference | 0 changes | 0 authors, 0 changes
public static class WebApiConfig
{
    1 reference | 0 changes | 0 authors, 0 changes
    public static void Register(HttpConfiguration config)
    {
        config.EnableCors(new EnableCorsAttribute("http://localhost:4200", headers: "*", methods: "*"));
```

Figure 13: Configuring CORS in the

Token Based User Authentication

For our web application we have chosen to implement token-based authentication. We used the middleware called OWIN (Open Web Interface For .Net Applications) for this implementation. OWIN is installed through NuGet package manager. A startup class is required in which to specify the authorisation options.



```
1 reference | 0 changes | 0 authors, 0 changes
public class Startup
{
    0 references | 0 changes | 0 authors, 0 changes
    public void Configuration(IAppBuilder app)
    {
        app.UseCors(CorsOptions.AllowAll);

        OAuthAuthorizationServerOptions option = new OAuthAuthorizationServerOptions
        {
            TokenEndpointPath = new PathString("/token"),
            Provider = new ApplicationOAuthProvider(),
            AccessTokenExpireTimeSpan = TimeSpan.FromMinutes(60),
            AllowInsecureHttp = true
        };
        app.UseOAuthAuthorizationServer(option);
        app.UseOAuthBearerAuthentication(new OAuthBearerAuthenticationOptions());
    }
}
```

Figure 14: OWIN Startup

Frontend

Angular

Our choice for implementing the front-end of our application was Angular. When we are talking about the subject of Angular development is it important to clear up the confusion between the different versions of Angular. The term AngularJS is used when referring to the first version of Angular. The later versions of Angular, starting with version 2 are simply called Angular. One reason for this distinction is according to the 'ng-book', the best book on learning Angular in our opinion, that ”Angular used TypeScript (instead of JavaScript) as the primary language, the 'JS' was dropped, leaving us with just Angular” [5].

The frontend of our Personal Task Manager was built using the latest version as of now, Angular 7. This new version has many impressive improvements such a better Angular CLI (Command Line Interface), faster speed and a smaller size. [4]

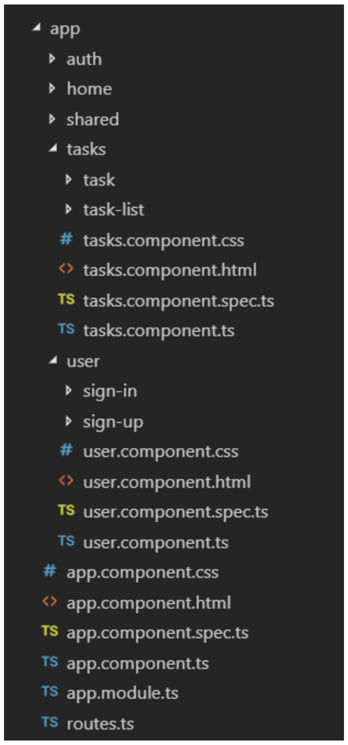
We have chosen the Bootstrap framework for the styling of our application and FontAwesome icons for improving the look of some of the UI elements. Bootstrap was very easy to add to our application and it offered the application a smooth mobile experience.

Advantages of using Angular

- SPA
- Mature Javascript Framework with many features
- Used to built complex UIs
- TypeScript
- Supported by Google

Component structure

The building blocks of Angular are called components. Components are one of the best features of Angular and this is how the components in our frontend were structured:



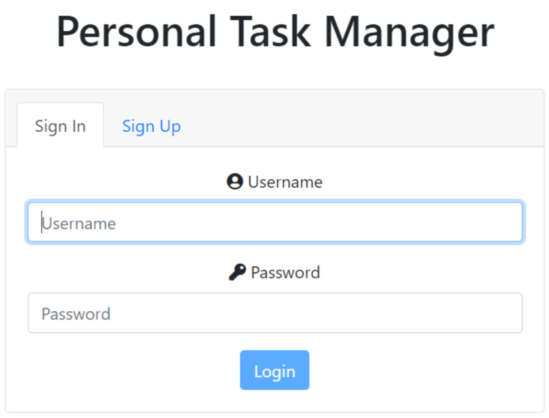
```
app
  auth
  home
  shared
  tasks
    task
    task-list
    tasks.component.css
    tasks.component.html
    tasks.component.spec.ts
    tasks.component.ts
  user
    sign-in
    sign-up
    user.component.css
    user.component.html
    user.component.spec.ts
    user.component.ts
  app.component.css
  app.component.html
  app.component.spec.ts
  app.component.ts
  app.module.ts
  routes.ts
```

Figure 15: Contents of the app folder

The User Component

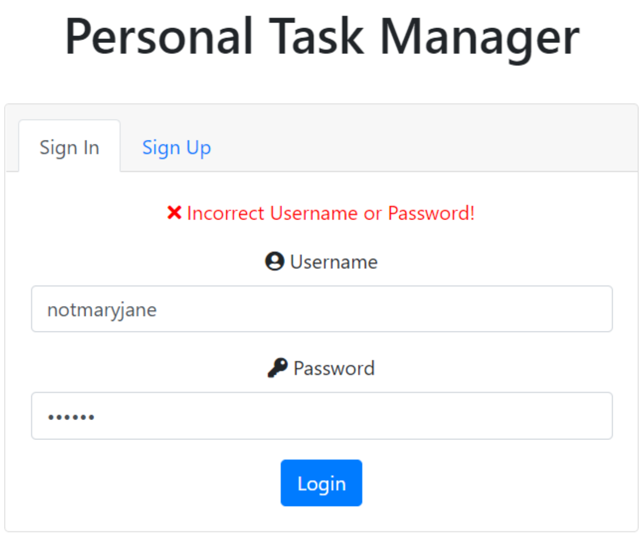
The user component also has two child components:

- Sign-up component used for registering a new user
- Sign-in component used for logging in



Personal Task Manager

Figure 16: View of the User component; this is the first screen that the user sees when opening the application



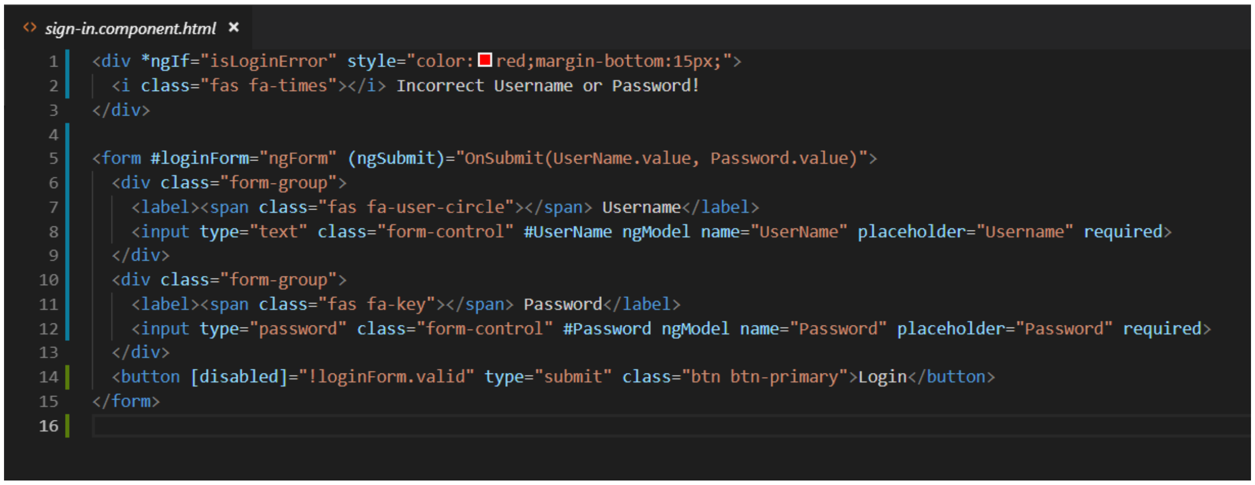
Personal Task Manager

✗ Incorrect Username or Password!

Username: notmaryjane

Password: ••••••

Figure 17: View of the system message when user inputs an incorrect username or password



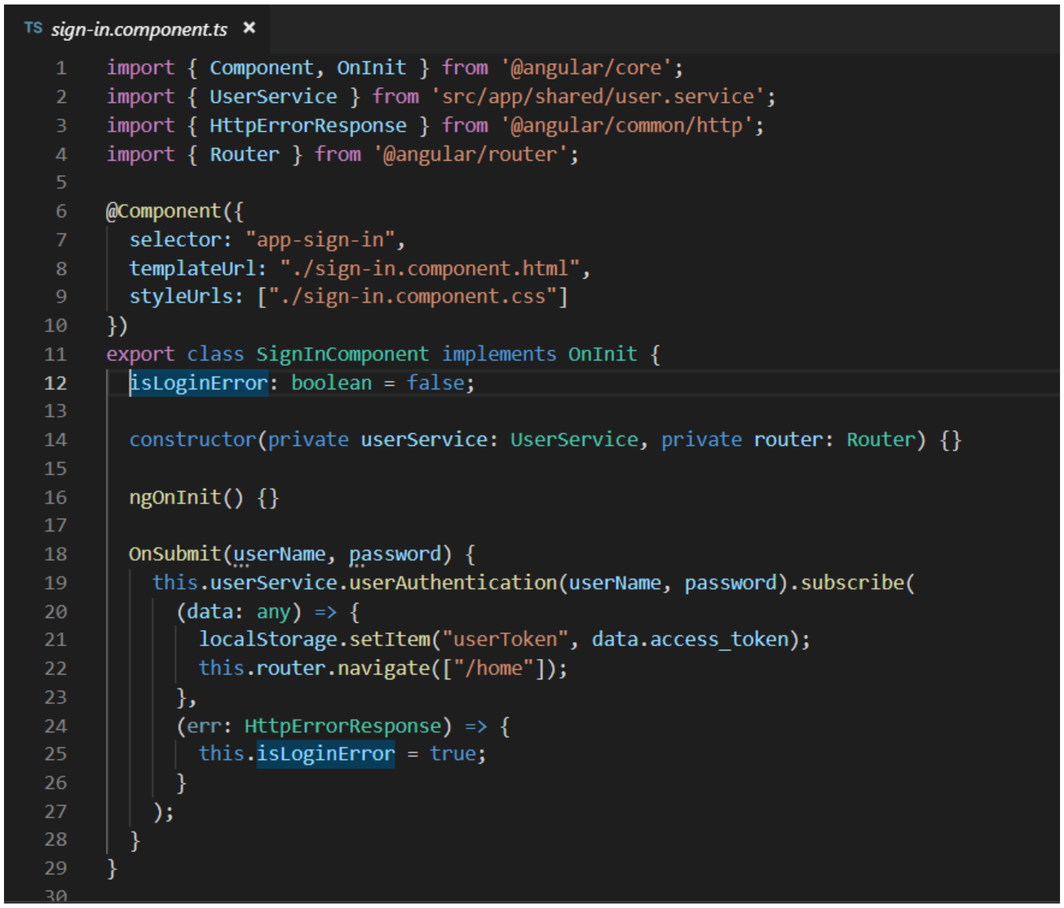
sign-in.component.html

```html
<div *ngIf="isLoginError" style="color:red;margin-bottom:15px;">
  <i class="fas fa-times"></i> Incorrect Username or Password!
</div>

<form #loginForm="ngForm" (ngSubmit)="OnSubmit(UserName.value, Password.value)">
  <div class="form-group">
    <label><span class="fas fa-user-circle"></span> Username</label>
    <input type="text" class="form-control" #UserName ngModel name="UserName" placeholder="Username" required>
  </div>
  <div class="form-group">
    <label><span class="fas fa-key"></span> Password</label>
    <input type="password" class="form-control" #Password ngModel name="Password" placeholder="Password" required>
  </div>
  <button [disabled]="!loginForm.valid" type="submit" class="btn btn-primary">Login</button>
</form>
```

Figure 18: The HTML layer

The first div is the message that is being displayed in the above image. We have used a common Angular directive called ngIf that only displays the div if the condition is true.

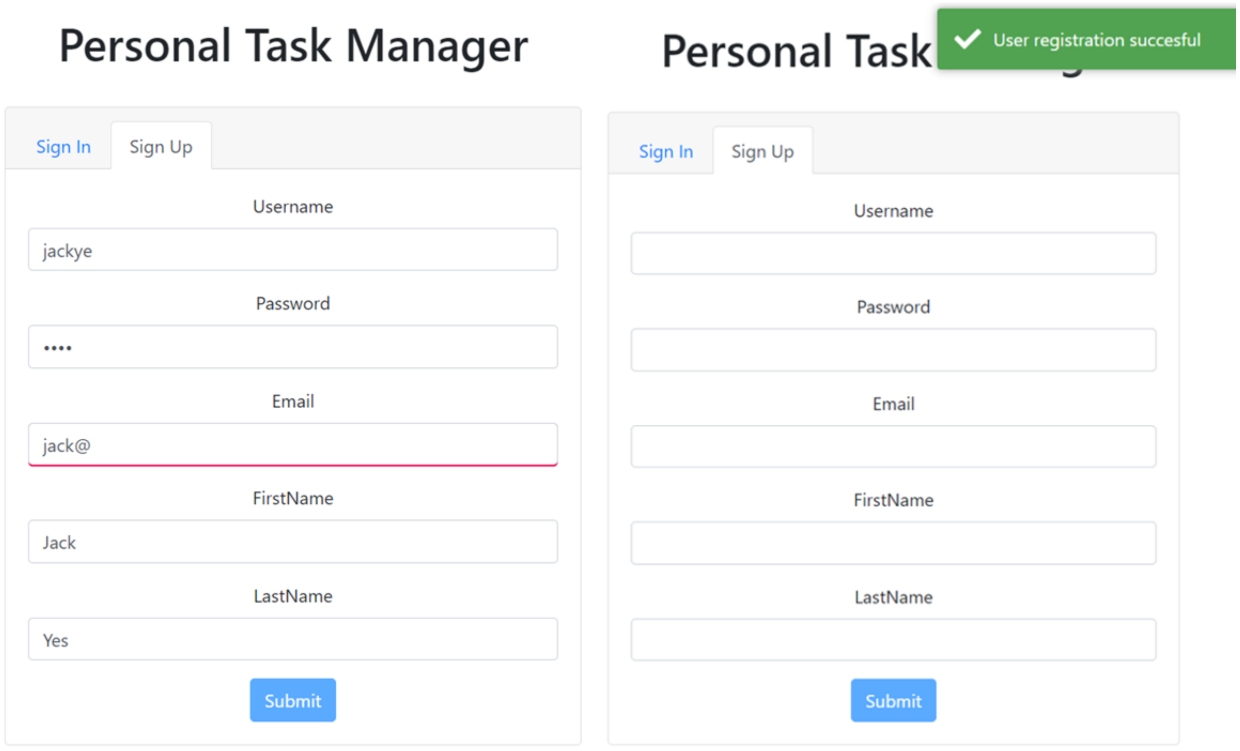
In the same code example, we can see that the submit button is disabled if the form is invalid.


sign-in.component.ts

```
import { Component, OnInit } from '@angular/core';
import { UserService } from 'src/app/shared/user.service';
import { HttpErrorResponse } from '@angular/common/http';
import { Router } from '@angular/router';

@Component({
  selector: "app-sign-in",
  templateUrl: "./sign-in.component.html",
  styleUrls: ["./sign-in.component.css"]
})
export class SignInComponent implements OnInit {
  isLoginError: boolean = false;

  constructor(private userService: UserService, private router: Router) {}

  ngOnInit() {}

  OnSubmit(userName, password) {
    this.userService.userAuthentication(userName, password).subscribe(
      (data: any) => {
        localStorage.setItem("userToken", data.access_token);
        this.router.navigate(["/home"]);
      },
      (err: HttpErrorResponse) => {
        this.isLoginError = true;
      }
    );
  }
}
```

Figure 19: TypeScript layer of the Sign Component

The figure above shows the implementation behind the sign in form. The local variable isLoginError is declared in the SignInComponent and its value is used to display the error message in case there is an error. The OnSubmit function is the one that executes the submit button is pressed. The function utilises instances of both the UserService private variable and the Router private variable. It first calls the authentication function on the service and then it uses subscribe which is part of the Observer pattern. The data returned which is the access token for the user is then stored in the browser's local storage. Lastly the user is redirected to the HomeComponent. Should an error occur then the isLoginError is set to true and the message is displayed on the HTML view of the component.


Personal Task Manager

Sign In | Sign Up

Username: jackye

Password: ••••

Email: jack@

FirstName: Jack

LastName: Yes

Submit

Personal Task

✓ User registration succesful

Sign In | Sign Up

Username:

Password:

Email:

FirstName:

LastName:

Submit

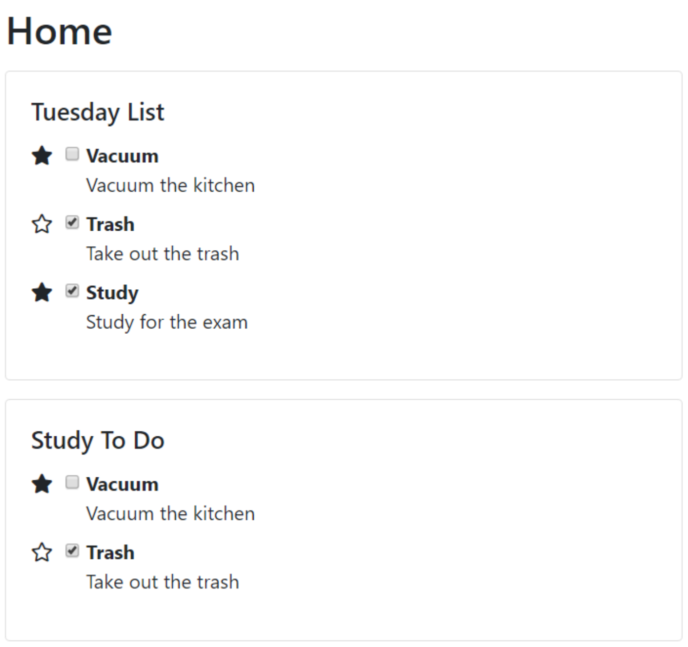
Figure 20: Image depicting the Sign Up Component in two cases

The form on the left shows an example of the form validation for the email input. Users cannot submit invalid emails and therefore the submit button is disabled and the email is highlighted in red.

The form on the right displays the system message that the user is shown after successfully registering a new account in the system. The form also automatically resets each time. To display that message, we have used a node module, that we haven't used before, called ngx-Toastr. This module is used for the purpose of displaying various types of messages to the users of a system. It is very easy to install and use and the messages it creates are very pleasing.

The Home Component

After a successful login, a user is redirected to the home component where the lists of tasks are displayed. The way that are these lists are created is by filtering and separating all the tasks by category which represents the 'list name'.


Home

Tuesday List

★ ☐ **Vacuum**
Vacuum the kitchen

☆ ☑ **Trash**
Take out the trash

★ ☑ **Study**
Study for the exam

Study To Do

★ ☐ **Vacuum**
Vacuum the kitchen

☆ ☑ **Trash**
Take out the trash

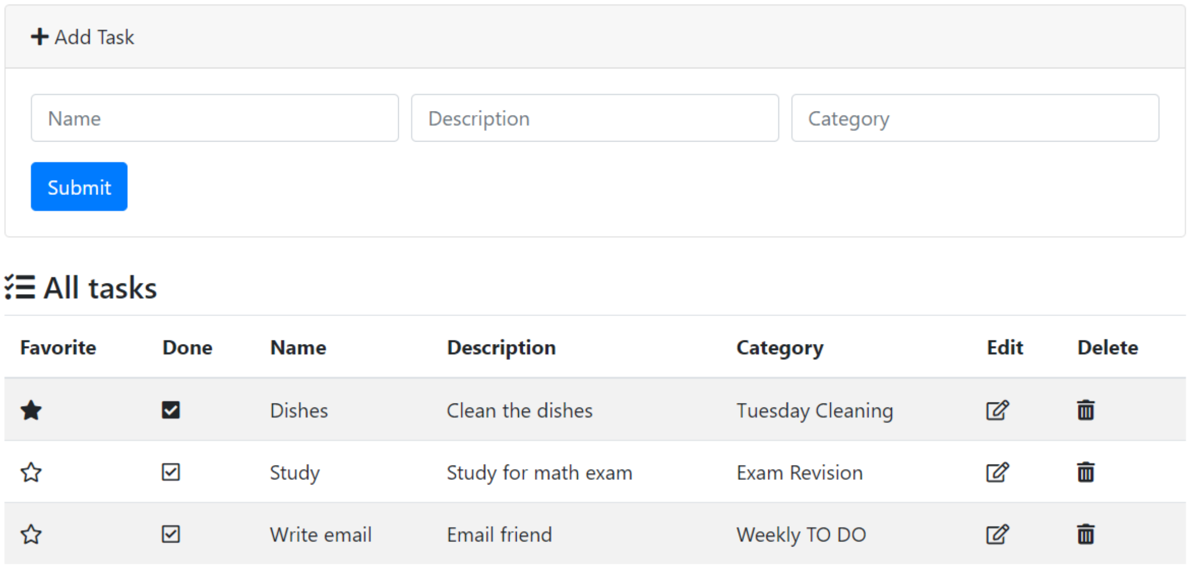
Figure 21: UI view of the Home Component

The lists of tasks displayed on the Home component are very simple. Each task has a favourite mark and a check done mark that when clicked toggle their values accordingly. In this figure, it can be noticed that the task's subtasks are not shown. The subtasks have not been implemented on the client side because of lack of time but will hopefully be added later.

The Tasks Component

This component has two child components:

- Task component responsible for the form used for adding or updating tasks;
- Task-list component responsible for displaying the table of tasks.



Favorite	Done	Name	Description	Category	Edit	Delete
★	☑	Dishes	Clean the dishes	Tuesday Cleaning	✎	🗑
☆	☑	Study	Study for math exam	Exam Revision	✎	🗑
☆	☑	Write email	Email friend	Weekly TO DO	✎	🗑

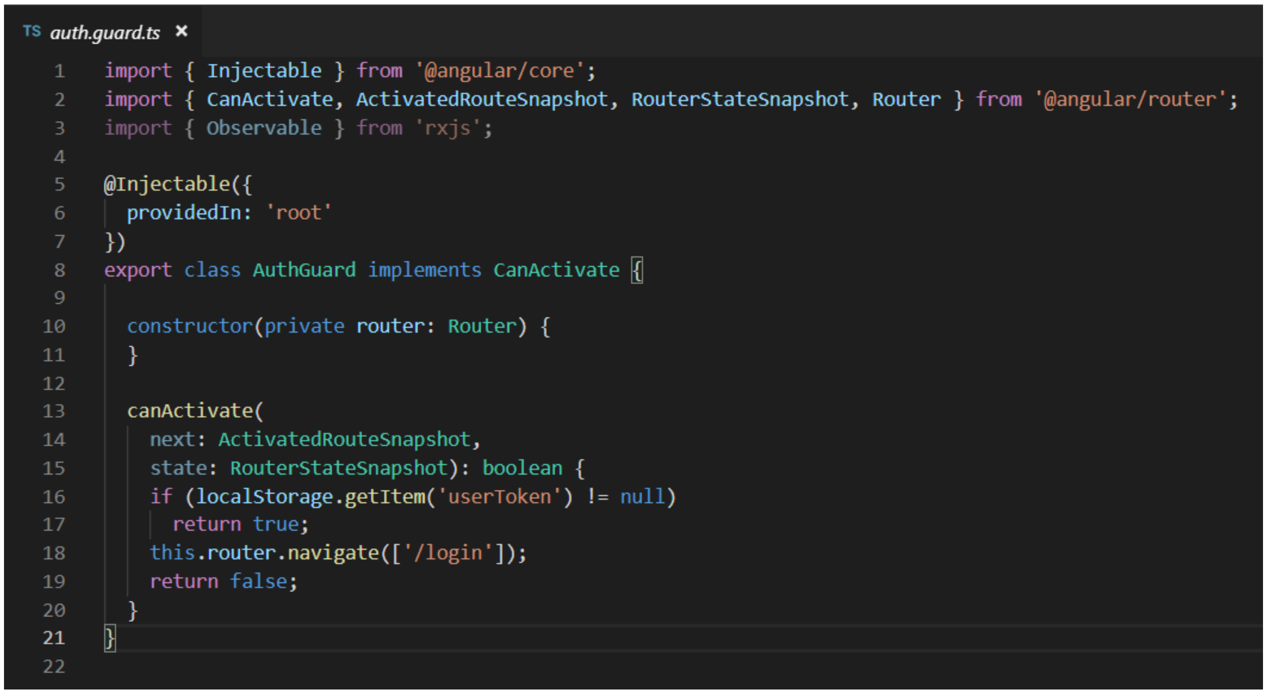
Figure 22: View of the Tasks component

The figure shows the form for adding/updating tasks that is contained in the task component and the list of tasks that is in the task-list component. In same manner as the tasks on the Home component view, the favourite and check done marks can be clicked to toggle their values. Adding a task is done through the form at the top of the page. After adding a task, the list is refreshed, and it now contains the new task. All the tasks can be viewed on this page and not their subtasks as they have not added to the client side yet. Updating a task is done by clicking the edit icon of a task, then the task detail will be loaded in the form above. Pressing submit will update the task's details and the changes will be shown immediately as the list will refresh itself. To delete a task, the user must press the delete icon of a task. Clicking it will trigger a prompt that will ask the user to confirm the deletion of the task. If the user presses ok, then the task is deleted and the list of task refreshes.

Our implementation of the CRUD for the tasks is not complete and it can be improved for example by adding the subtasks, however we consider that it can satisfy the Manage tasks requirement to a decent extent.

Authentication guard

The authentication guard is an Angular tool that can be used to protect routes from unauthorised access. It is by implementing the canActivate function that we check if the user has the necessary token that is created whenever a user signs into our system. If the user has the right token stored in his browser then the user can access and view the route that they requested. However, if they cannot prove their identity then they are redirected to the login page.


auth.guard.ts

```typescript
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router } from '@angular/router';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {

  constructor(private router: Router) {
  }

  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): boolean {
    if (localStorage.getItem('userToken') != null)
      return true;
    this.router.navigate(['/login']);
    return false;
  }
}
```

Figure 23: The implementation of the Auth Guard

Routing

Routing in Angular is done following the Angular guidelines. We have chosen to store our array of routes in a separate TypeScript file, instead of storing them in the app module file. The routing of our web application is basic except for the Auth Guard which protects the home and tasks routes from unwanted access. We have two paths that have children, this implementation is used in the User component and it allows us to switch between the Sign up component and the Sign in component when viewing their parent component User component.

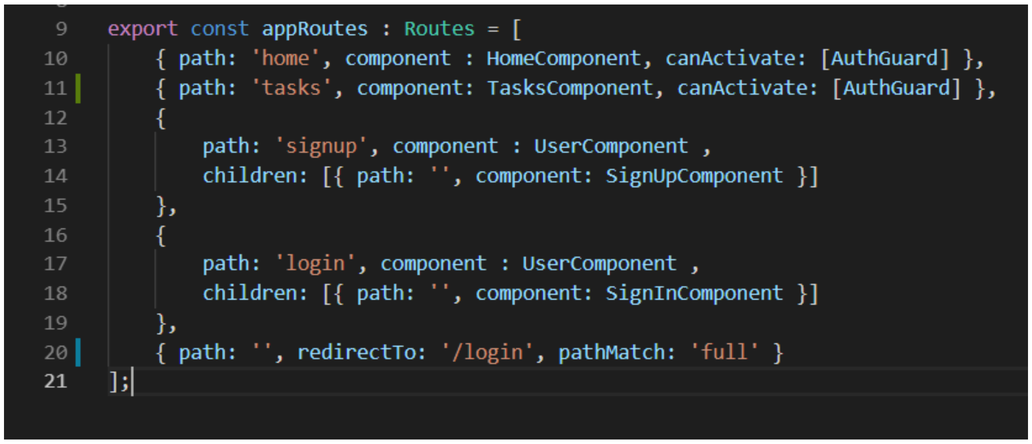
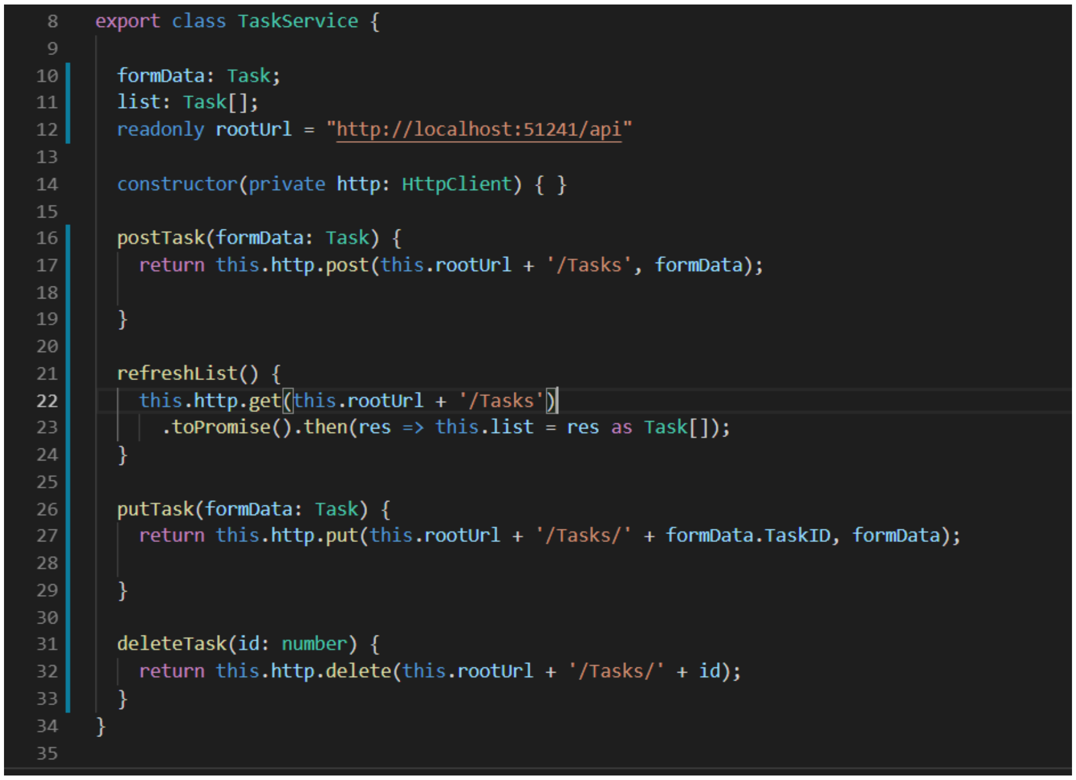

```typescript
export const appRoutes : Routes = [
    { path: 'home', component : HomeComponent, canActivate: [AuthGuard] },
    { path: 'tasks', component: TasksComponent, canActivate: [AuthGuard] },
    {
        path: 'signup', component : UserComponent ,
        children: [{ path: '', component: SignUpComponent }]
    },
    {
        path: 'login', component : UserComponent ,
        children: [{ path: '', component: SignInComponent }]
    },
    { path: '', redirectTo: '/login', pathMatch: 'full' }
];
```

Figure 24: The application routes used in the web

Task service

Angular uses services to inject data from outside systems such as APIs into its components. In the Task service file, we have implemented the functions that request data with the help of the HttpClient module. The Task service is responsible for all the data concerning the tasks and it was a crucial part of implementing the Manage tasks use case and fulfilling the requirement with the same title.

To communicate effectively the service must be provided with the root URL corresponding to the desired web API.



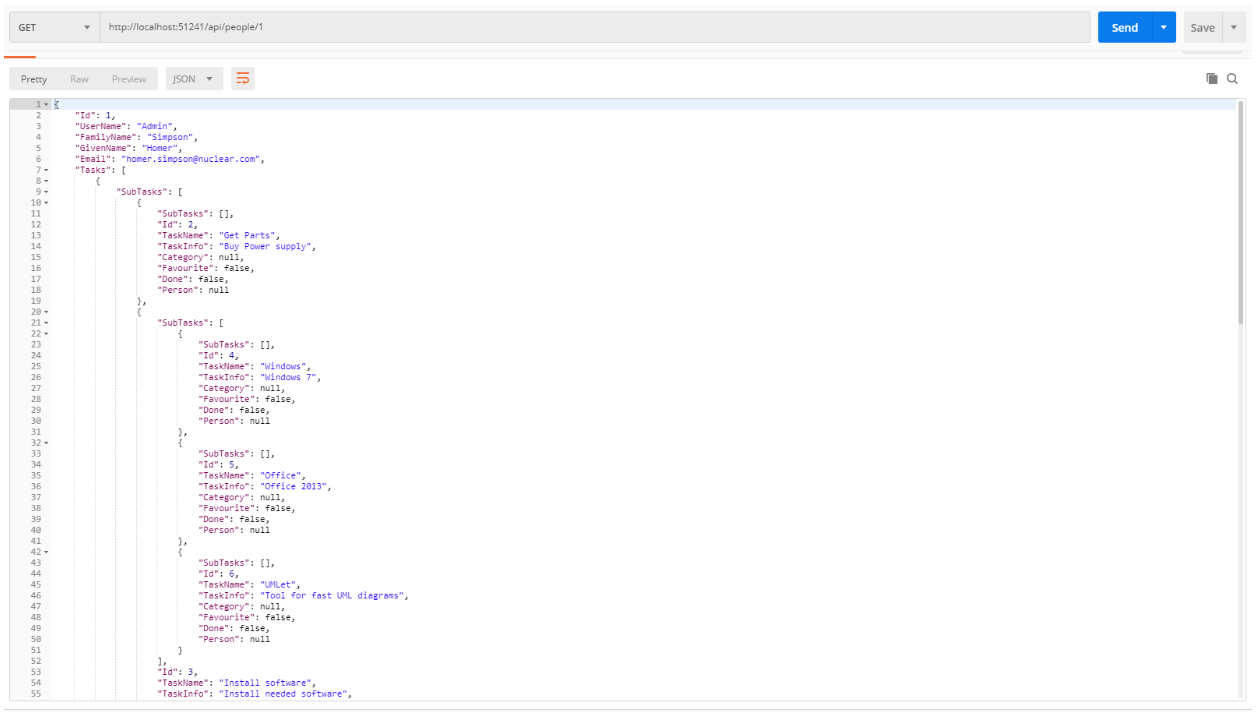
```
export class TaskService {

  formData: Task;
  list: Task[];
  readonly rootUrl = "http://localhost:51241/api"

  constructor(private http: HttpClient) { }

  postTask(formData: Task) {
    return this.http.post(this.rootUrl + '/Tasks', formData);

  }

  refreshList() {
    this.http.get(this.rootUrl + '/Tasks')
      .toPromise().then(res => this.list = res as Task[]);
  }

  putTask(formData: Task) {
    return this.http.put(this.rootUrl + '/Tasks/' + formData.TaskID, formData);

  }

  deleteTask(id: number) {
    return this.http.delete(this.rootUrl + '/Tasks/' + id);
  }
}

```

Figure 25: The Task service implementation

Testing

Testing with Postman

We tested our web API using Postman, a tool for interacting with HTTP APIs.



```
http://localhost:51241/api/people/1
```

```
{
    "Id": 1,
    "UserName": "Admin",
    "FamilyName": "Simpson",
    "GivenName": "Homer",
    "Email": "homer.simpson@nuclear.com",
    "Tasks": [
        {
            "SubTasks": [
                {
                    "SubTasks": [],
                    "Id": 2,
                    "TaskName": "Get Parts",
                    "TaskInfo": "Buy Power supply",
                    "Category": null,
                    "Favourite": false,
                    "Done": false,
                    "Person": null
                },
                {
                    "SubTasks": [
                        {
                            "SubTasks": [],
                            "Id": 4,
                            "TaskName": "Windows",
                            "TaskInfo": "Windows 7",
                            "Category": null,
                            "Favourite": false,
                            "Done": false,
                            "Person": null
                        },
                        {
                            "SubTasks": [],
                            "Id": 5,
                            "TaskName": "Office",
                            "TaskInfo": "Office 2013",
                            "Category": null,
                            "Favourite": false,
                            "Done": false,
                            "Person": null
                        },
                        {
                            "SubTasks": [],
                            "Id": 6,
                            "TaskName": "UMLet",
                            "TaskInfo": "Tool for fast UML diagrams",
                            "Category": null,
                            "Favourite": false,
                            "Done": false,
                            "Person": null
                        }
                    ],
                    "Id": 3,
                    "TaskName": "Install software",
                    "TaskInfo": "Install needed software",
```

Figure 26: GetPerson: success

Development process

Kanban

Because our group was small (contain only two members) our choice was between Kanban and Scrum. We both appreciated the agility, the Kanban board and freedom of Kanban and a major thing in Scrum was Sprint which we saw as unnecessary for our project.

Group contract

- Work from 9:00 to 16:00 with a one-hour lunch break from 12:00 to 13:00.
- Workdays are Monday to Friday with weekends off.
- Members will be assigned individual tasks to work on.
- Punishment for failing to comply with the rules established in this contract will be punished by having to eat the undesired food chosen beforehand for each member. (Madalina = 1 piece of liquorice, Andreas = 1 tsp. of balsamic vinegar).

Conclusion

In the end we learned many new technologies during this project. We consolidated our web development knowledge and are overall pleased with the resulting system that we created.

Regarding the questions that were raised in our problem statement:

- How to present the tasks in a more organised manner?
    - By implementing the CRUD functions in both the frontend and backend using Http
    - By creating a good UI design, with which we displayed the tasks in lists according to category
- Second question: “How to highlight the user’s most important tasks?”
    - By implementing a favorite system in the frontend that can be easily used to favorite tasks
- Third question: “What are some ways to make sure that our list is accessible to the user anytime and anywhere with an internet connection?”
    - By having a MSSQL database
    - By creating a ASP.NET Web API
    - By creating an Angular browser client
- Forth question: “Which are some ways to keep our data secure?”
    - By implementing user authorisation and authentication in our web application.

References

[1] “Single-page application vs multiple.page application,” [Online]. Available: https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58. [Accessed January 2019].

[2] “MVC architecture,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Apps/Fundamentals/Modern_web_app_architecture/MVC_architecture. [Accessed January 2019].

[3] “Learn about ASP.NET Web API,” [Online]. Available: https://www.asp.net/web-api. [Accessed January 2019].

[4] “Angular 7,” [Online]. Available: https://alligator.io/angular/angular-7/. [Accessed January 2019].

[5] Nate Murray, ng-book: The Complete Book on Angular 4st Edition, Fullstack io, 2017

[6] “Entity Framework (EF) Documentation,” [Online]. Available: https://msdn.microsoft.com/enus/library/ee712907(v=vs.113).aspx. [Accessed January 2019].

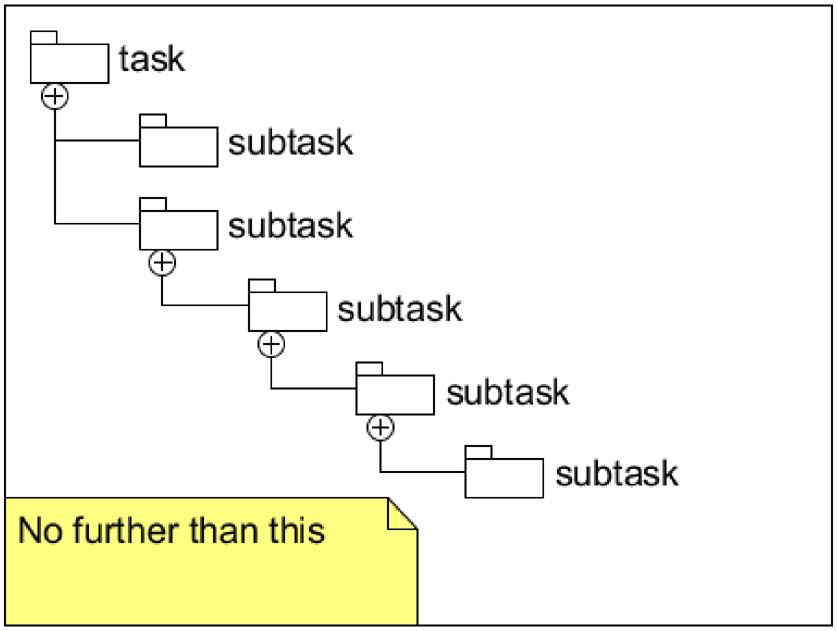
[7] “Language Integrated Query (LINQ) Documentation,” [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/. [Accessed January 2019].

[8] “ASP.NET Web API Documentation,” [Online]. Available: https://docs.microsoft.com/en-us/aspnet/web-api/. [Accessed January 2019].

[9] “Postman Documentation,” [Online]. Available: https://www.getpostman.com/. [Accessed January 2019].

Appendix

Appendix A: Subtask Depth


task
subtask
subtask
subtask
subtask
subtask
No further than this