**Create initial project**

TODO:

* Clone this repository in order to get local copy of it on your machine
* Create new project (and a solution) named **ToDoApi** by using [C#] ASP.NET *Core* Web API project template
* Before first commit use gitignore file template from [here VisualStudio.gitignore](https://github.com/github/gitignore/blob/master/VisualStudio.gitignore)

Out of scope:

* Database configuration
* Database models
* Controllers

**Create initial model structure**

TODO:

* Create new **ToDoCore** .NET *Standard* Class Library project inside the solution to host database models structure.
* Inside the project define class model structure based on requirements defined in README.md.

Out of scope:

* Database configuration
* Controllers

**Create API endpoints (Controllers) with mocked data**

TODO:

* Create controllers, which follow REST convention, for resources defined in README.md.
* Search endpoint should be case insensitive & enable partial match.
* Controllers should work with mocked data, for now, no database connection.

Out of scope:

* Database configuration

Resources for REST conventions:

* [RESTful Tutorial](https://www.restapitutorial.com/), especially [Using HTTP Methods for RESTful Services](https://www.restapitutorial.com/lessons/httpmethods.html)
* [REST Resource Naming Guide](https://restfulapi.net/resource-naming/)

# Enable API documentation

Introduce & configure swagger documentation for the API. Set swagger as the default loaded page.

use: <https://github.com/domaindrivendev/Swashbuckle.AspNetCore>

**Configure ToDo database**

TODO:

* Create new **ToDo.Infrastructure** project, to separate infrastructure (database context, migrations) from API and models/core.
* Set database connection string in API settings file
* In the project, create **ToDoDbContext** (DbContext implementation) and register it in .net core IOC container with connection string (you can follow this [tutorial](https://docs.microsoft.com/en-us/ef/core/get-started/aspnetcore/new-db))
* In the project, configure relation between the 2 models using Entity Framework Fluent API configurations as separate classes (implements *IEntityTypeConfiguration<>*). Tutorials: [EFCoreRelationships](https://docs.microsoft.com/en-us/ef/core/modeling/relationships) & [FluentAPIConfigurations](https://www.learnentityframeworkcore.com/configuration/fluent-api). Also, using FluentApi configurations, configure ToDoList.Title to be required filed with maximal length of 20. ToDoItem.Description should be required with maximal size limit.
* In the project, in *Migrations/* folder, add & execute **Initial** migration (before making commit/push, consult someone about code review).

# Integrate ToDoDbContext into Controller

Include **TODODbContext** in **ToDoListsController** and replace mocked list with real data. Use lambda expressions on DbContext sets to query the resources.

Task #6 must be completed before beginning with this one !!!

**Implement db migrations execution on startup**

TODO:

* Find a way to enable automatic db migration execution on application startup (so we do not need to execute Update-Database command manually anymore)
* Add new int property Position to test migration
* Validate that the migration was executed on startup

**Enable reminder functionality (optional)**

1. Update database (with new migration) to support "reminder" functionality on *ToDoList* level. "Reminder" functionality implies sending an email to specified email address if "reminder criteria" (specified *DateTime* has expired) is satisfied. Also, it should be possible to know if the user has opened a "reminded" *ToDoList* or not.
2. Implement Reminder service (the service) which will start on Api start but run in background as separate process (keyword: *IHostedService*). The service should do the flowing:
   * Every 15s (configurable) it should query the database for ToDoList entries that match the "reminder criteria"
   * Send an email, for each of the selected ToDoLists, containing an link to ToDoList (e.g. http://localhost:[port]/api/to-do-lists/[guid id]). Use [SendGrid](https://app.sendgrid.com/) mail server
   * Update each of the selected ToDoLists so that the "reminder criteria" is not valid anymore. Also, update each of the selected ToDoLists to that they have status as not opened by user
3. Store all of the configurations inside appsettings.json file (interval, SendGrid key ...)

**Enable log files (optional)**

The system should log certain actions inside log file. The following actions should be logged:

DEBUG level:

* Application start/end (e.g. "ToDoApi started!")
* Database CRUD operations (e.g. "ToDoList.GetAll() executed!", "ToDoITem.Update() executed!", etc.)
* Reminder service start/end (e.g. "ReminderService started!")
* Reminder service action cycle (e.g. "ReminderService found [x] ToDoLists")

ERROR level:

* All exceptions with stack trace

Log file should have date cycle (new log file for each new date)

You can use net core longing implementation or appropriate third party library.

# Implement position update logic in ToDoList/ToDoItem

Create 2 API endpoints to cover ToDoList & ToDoList position update.

Create List/Item position UPDATE logic as separate action (regular UPDATE must not include Position anymore).

Update List/Item CREATE logic so that each new List receive highest position of all lists, and each new Item receives lowest postilion of all items in a list (like in Google Keep).

(OPTIONALLY) Create unit tests to cover this logic (create new project **ToDoCoreTest**).

**Create initial Web application**

A suggestion, [Angular 6 ng-book-2 (ng-book2-angular-6-r68.pdf)](https://github.com/zfynote/ng2-book), very good book about angular. Go through *"Writing Your First Angular Web Application"* chapter before making your own app. Official Angular [HeroEditor tutorial](https://angular.io/tutorial) is also very good.

TODO:

1. Create new folder **to-do-web-app/** in root of the repository (not in Visual studio solution)
2. Inside the folder create Angular (v6) "to-do-web-app" application
3. Add Bootstrap (v4) node module to the app

Consult someone before making a commit (to check if .gitignore settings were applied)

**Create Dashboard landing page**

TODO:

1. Create initial project folder structure (e.g. [Folders-by-feature structure](https://angular.io/guide/styleguide#folders-by-feature-structure))
2. Create Dashboard page that will load on app start
3. Enable loading of all ToDoList's from back-end on Dashboard init. You don't need to implement any UI for this, just show the json of loaded to-do's on Dashboard.

**Preview a list of ToDoLists on Dashboard**

TODO:

1. Create *to-do-list/to-do-item* models in web app **models/** folder. It is good practice to have all of the back-end models in root **models/** folder, other helper models should go within hosted component folder. Also, each model file should end with \**.model.ts* (e.g. to-do-list.model.ts). Back-end models must also take care of deserialization from json to ts model class (check out [Working with models in Angular](https://nehalist.io/working-with-models-in-angular/))
2. In *Dashboard* module, create **ToDoPreview** component (should only preview List title and a list of Item descriptions, no need for nested components) that will receive to-do-list.model.ts class as input value and map it to html template (keyword: *@Input()*)
3. In *Dashboard* component, create a list of **ToDoPreview** components that will receive deserialized models from api service (keyword: \*ngFor)

(BONUS) Introduce Remove ToDoList functionality (like archive button on Google Keep)!

**Create ToDoList create/edit page (just read functionality)**

TODO:

1. Create new module to host *ToDoList* create/edit page (the page), and add **ToDoList** component that will be used for creating/editing.
2. Add new route that will direct the user to the page (http://localhost:port/to-do-list). Also, configure the route to support IDs in the url (e.g. http://localhost:port/to-do-list/4343).
3. On **ToDoList** component init, load *ToDoList* from back-end if ID is provided in the url, or just init empty component if ID is not provided (check out [Using Route Parameters](https://angular-2-training-book.rangle.io/handout/routing/routeparams.html))
4. In app header, introduce "Add" button that will redirect the user to empty ToDoList page via route without ID.
5. In *ToDoPreview* component, introduce click event handler that will redirect the user to ToDoList page via route containing selected *ToDoList* ID.

**Introduce Edit functionality on ToDoList page**

TODO:

* Editable inputs: ToDoList title, ToDoItem description & ToDoITem isCompleted (we will add other fields in the future)
* Text inputs should be triggered when the input lose focus. Check box input should be triggered on change.
* Changing IsCompleted should have additional *UI functionality*, if changed to selected the item should go to the bottom of the list, if changed to not selected the item should return to its original position.
* Whole list should be saved when the user leaves the page (to prevent losing data if input is focuses when the user clicks Back)
* All updates should be done "optimistically", meaning:
  + Execute API call
  + Update the layout
  + Catch the error in API call and reload the component data (call api get)

(bonus) Add remove item functionality

**Introduce Create functionality in ToDoList page**

TODO:

1. Create new *ToDoList* entry on creation of first *ToDoITem* (like in Google Keep)

**Introduce RemindMe logic (optional)**

TODO:

1. On *Dashboard* page, make Previews that have *Reminded* flag on false visually different and on top. After the user opens remind me list for the first time, the list becomes regular to-do-list.
2. On *ToDoList* page, introduce add reminder functionality. Compared to other input fields on the page, the reminder fields (email, date and time) should be saved on button click (e.g. AddReminr button). The fields should be validated so that user can enter only emails and future date/times (including current DateTime). For date & time you can use bootstrap components (*Datepicker* & *Timepicker*) or you can find some other package in npm library.

**Introduce List/Item position change**

TODO:

* Try to implement position update via drag/drop functionality. You can use third party npm packages (e.g. [ng2-dnd](https://shekhargulati.com/2017/09/21/building-an-angular-4-drag-and-drop-application-in-15-minutes/)) or create the logic manualy (up/down arrow buttons can be alternative)
* Position update should be done optimistically (api call, UI update, reload if error)
* ToDoItem position update has higher priority (do ToDoItem first, please)

**Enable ToDoList search functionality**

TODO:

* Add search box in application header. The search box should be visible on *Dashboard* page but hidden on *ToDoList* page.
* Search functionality must use service for communication between the components

**Introduce Web Client authentication/authorization**

Use [Auth0](https://auth0.com/) identity provider (with the free account) to enable "Single Page App" authentication/authorization.

TODO:

* Enable user authentication (prompt *Auth0 Login*) on application load, or redirect the users to *Dashboard* if the user is already authenticated.
* Show authenticated user *Name* in the header of the application.
* Register your API to enable back-end calls (don't do back-end authentication/authorization yet). To add *Bearer Token* to request headers use angluar's *HttpInterceptors*.
* Enable web client authorization via angular's *RouteGuards*.

Out of scope:

* Authorization/authentication on API.

**Introduce API authorization**

Use [Auth0](https://auth0.com/) identity provider (with the free account) to enable "Back-end API" authorization.

TODO:

* Configure authorization & protect the API endpoints

**Introduce the users in the API**

TODO:

1. Try to fiend a way to enrich *JWT token* with username or some other unique identifier.
2. Find a way to extract the user unique identifier from *JWT Token*.
3. Expand the tables with *Owner* column of string type which will hold unique user identifier (e.g. *username*). This field must be mandatory!
4. Update *ToDoService* so it applies *Principal.Identity.Name* in all of its CRUD db queries!

(BONUS) Enrich access token with user email address. Update "RemindMe" logic so that user email is saved inside the list on Create and never deleted. Remove email from UI and DTO models.

**Enable ToDo List sharing via Url link**

TODO:

* On *ToDoListPerivew* component add share button. On click, the link to shared component should be copied to clipboard.
* The link (e.g. [http://your\_domain/share/[generated\_guid]](http://your_domain/share/%5Bgenerated_guid%5D)) should enable unauthenticated user to view shared ToDoList (no write/delete rights). This could be new component, e.g. *ToDoListShare*. ToDoList sharing guid should be generated and stored in the API.
* The link should be valid for 2h beginning with the moment of creation.

**Deploy application and database to Azure (optional)**

Open

**Deploy application and database to Azure (optional)**

Use [Azure Portal](https://portal.azure.com/) (ask for credentials).

**Setting up database**

Create new Azure Resource: SQL Server. You will need to configure:

* database name (can be the same name used locally)
* resource group - create a new resource group and name it something like "Novalite Internship" (resource groups are used to group resources, we are going to put database and web applications in this group)
* create a new database server. db server usually holds multiple db's, we don't have to name it specific to ToDo app, so let's name it also novalite-internship. set location in western europe. configure admin account for the server (user name can be novalite\_admin)
* wait for creation of the resource and the navigate to it. find connection string and copy it to application.json of the api application (replace username and the password in the connection string with the real ones)

**Setting up web application**

Create new Azure Resource: Web App. You will need to configure:

* App name
* Use the same resource group (Novalite Internship)
* Set the App Service plan (this is the location of the server where app will be hosted. choose WestEurope)
* After resource is created, navigate to it and download the publish profile. This publish profile we'll use in Visual Studio to publish the application

You need to create 2 Web App resources: one for front end app, second for the api. Both can use same App Service Plan

In the environment.prod.ts of the front end application, configure path to the published api.

**Publishing API from VisualStudio**

* Right click on Api project and select Publish - use the downloaded profile for web api app

**Publishing Angular app (manually)**

* run production build of the angular app
* zip all files in the dist directory
* [publish the zip by following procedure](https://docs.microsoft.com/en-us/azure/app-service/app-service-deploy-zip)

# Pagination