# ASP.NET 5 Tutorial

I denne tutorialen skal vi se nærmere på ASP.NET 5 og MVC 6:

- Ny prosjektstruktur
- MVC 6 WebAPI og routing
- Dependency injection
- Nytt konfigureringsrammeverk
- Bruk av Node.js, NPM og Bower
- Task runner: Gulp

Vi vil bruke Visual Studio 2015 som har innebygd støtte for disse nye verktøyene. Løsningsforslag til oppgavene er vedlagt.

## Systemkrav

De følgende verktøyene kreves:

- Visual Studio 2015
- ASP.NET and Web Tools 2015 (Beta8). Dette er den nyeste versjonen og resten av tutorialen vil anta at denne versjonen er installert. Dersom du ikke har Beta8 må du passe på at du har samme versjon i referansene dine som den runtime-versjonen du kjører.

Valgfrie verktøy:

- Web Essentials 2015 (gjør opplevelsen av å arbeide med ASP.NET 5 hakket bedre)
- Postman / Fiddler (for testing av WebAPI)

## Oppgave 1 – ASP.NET 5 Prosjekt

Det nye prosjektet kan legges i den samme løsningen som dere allerede har fått utdelt. Dette web-prosjektet kan dere bruke videre under GeekRetreat.

1. New Project > Visual C# > Web > **ASP.NET Web Application**
2. ASP.NET 5 Preview Templates > **Empty**
3. «Host in the cloud» er huket av som standard. Dette er vi ikke interessert i akkurat nå, så fjern krysset i denne boksen.

Du vil nå få et helt tomt ASP.NET 5 prosjekt som bare har det minste som trengs for å spinne opp en webapplikasjon. Det følgende er verdt å bemerke seg:

- wwwroot — Denne mappen er til statisk HTML, JavaScript, CSS og andre ressurser.
- project.json — Dette er prosjektfilen. Avhengigheter legges inn her.
- Startup.cs — Denne filen inneholder metoder som bootstrapper webapplikasjonen. Dependency injection settes opp her.

Dersom dere starter opp webapplikasjonen nå, vil dere kun se en hvit side med teksten «Hello World!». Dette kommer av innholdet i metoden *Configure(…)* i Startup.cs. Webapplikasjonen vil, slik som denne metoden ser ut nå, alltid returnere «Hello, World!». Dette må vi gjøre noe med.

4. Fjern innholdet i *Configure(…)*-metoden

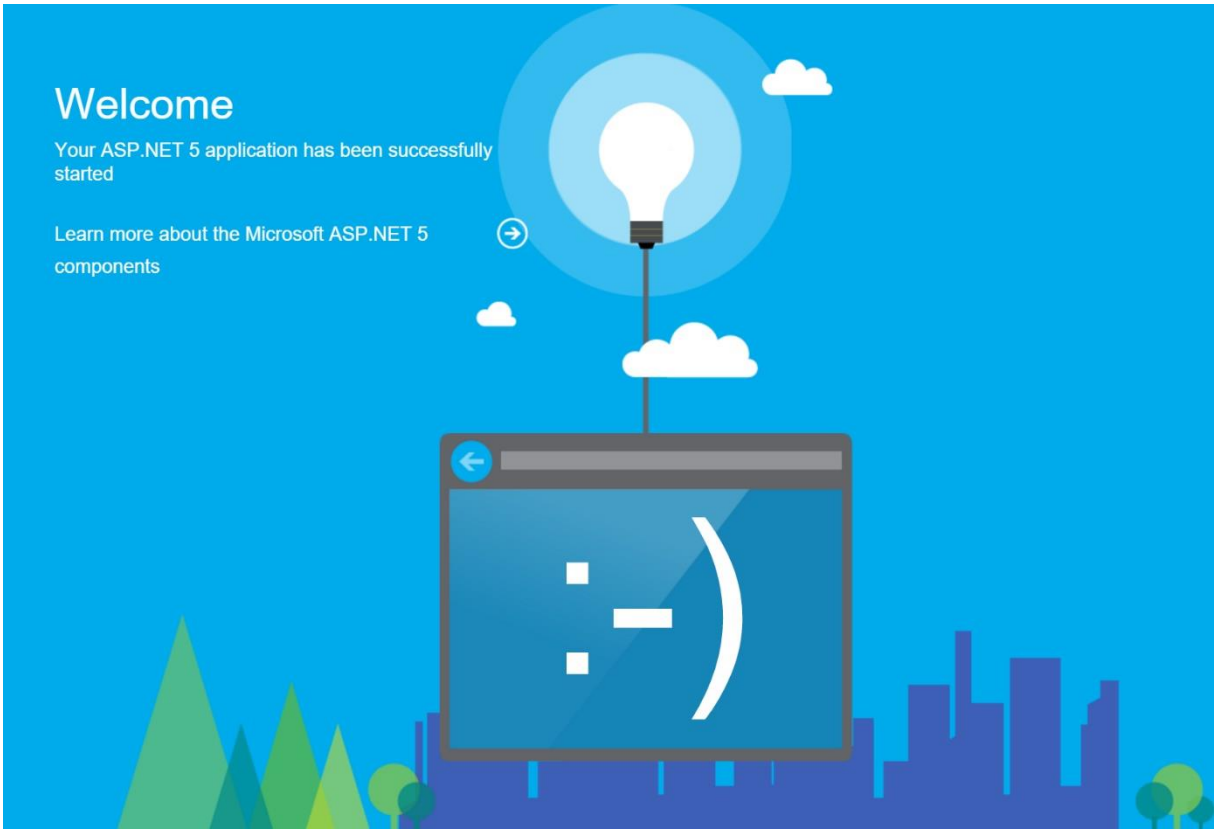
Dette fører til at ingenting vises når man starter applikasjonen. Det kan være greit å ha noe der, slik at man lettere vet at applikasjonen er oppe og kjører. Vi vil bruke en innebygd modul for å vise en fremside.

5. Legg til en referanse til «`Microsoft.AspNet.Diagnostics`» (1.0.0-beta8) i project.json.

Legg merke til at man har IntelliSense i json-filen og at Visual Studio automatisk laster ned denne pakken når man lagrer filen. Vi har nå det som trengs for å vise en simpel fremside.

6. Legg til «app.UseWelcomePage()» i *Configure(...)*

Dette registrerer en modul i webapplikasjonen vår som returnerer den følgende siden i alle requester til websiden:




## Oppgave 2 – WebAPI og Routing

I denne oppgaven vil vi jobbe med MVC 6 WebAPI og routing-systemet. Det første vi trenger er MVC.

1. Legg til en referanse til «`Microsoft.AspNet.Mvc`» (6.0.0-beta8)
2. Registrer MVC-modulen (Hint: *vi gjorde nettopp det samme for WelcomePage*)

Vi er ikke helt ferdig med å sette opp MVC enda. Siden ASP.NET 5-rammeverket er bygd opp vha. dependency injection, må vi registrere MVC i IoC-kontaineren.

3. Legg til «services.AddMvc()» i *ConfigureServices(…)*

Nå har vi gjort alt som skal til for å kunne ta i bruk MVC og er klar til å lage vår første API-kontroller. Den første kontrolleren vi skal jobbe med vil bare bli brukt til å adressere routing og vil ikke ha noe bruksformål senere i GeekRetreat.

4. Legg til klassen «`TodoController`» som arver fra Controller (i Microsoft.AspNet.Mvc)

Siden vi ikke har konfigurert noe annet, er det den standard navnekonvensjonen som er i bruk. Det vil si at alle kontrollere må ha et navn som slutter på *Controller*. Vi har enda ikke fortalt MVC hvilken route som tar oss til kontrolleren vi nettopp opprettet.

5. Legg til attributtet «`[Route("/api/todo")]`» på *TodoController*-klassen

Dette forteller MVC at dersom man navigerer til «http://localhost:[port]/api/todo» så kommer man til denne kontrolleren. Vi har ingen metoder som håndterer requester enda, så det skjer ikke mye dersom man navigerer dit nå.

For å ha noe data å sende til og fra API-kontrolleren fortsetter vi med å lage en modell.

6. Legg til klassen «`TodoItem`» med feltene Id (int) og Text (string)
7. Initialiser en statisk liste av *TodoItem* i *TodoController* med et par verdier
8. Legg til en metode for å hente alle elementene i TODO-listen, som returnerer de i JSON-format

```
[HttpGet]
public IActionResult Get()
{
    return Json(_todoItems);
}
```

Attributtet «`[HttpGet]`» gjør at GET-requester mot «/api/todo» blir rutet til denne metoden. Ved å ha returtype «`IActionResult`» og bruke metoden *Json(…)* får vi returner alle elementene i listen som JSON. Dette kan vi verifisere ved å bruke Postman eller Fiddler til å sende en GET-request mot «http://localhost:[port]/api/todo».

9. Legg til en metode som returnerer elementet med en gitt ID (som JSON). (Hint: «`Get(int id)`»)

I et REST API gjøres det ved å gjøre en GET-request til «/api/todo/[id]». Vi må derfor bruke samme attributt som vi på metoden vi nettopp lagde for å hente alle elementer, men på følgende måte: «`[HttpGet("{id:int}")]`». Dette attributtet forteller at det er denne metoden som skal brukes dersom routen slutter med et tall, f.eks. «/api/todo/1». Vi kan returnere resultatet fra *HttpNotFound()*-metoden dersom elementet vi forespør ikke finnes i listen. Verifiser metoden ved å bruke Postman eller Fiddler.

10. Legg til en metode for å legge til et element i listen. (Hint: HttpPost) (Hint 2: Bruk attributtet «`[FromBody]`» på parameteret til denne metode for å fortelle at verdien skal leses fra body-delen av POST-requesten)

For å teste denne metoden i Postman/Fiddler må man sette «ContentType: application/json» for at objektet skal bli deserialisert riktig. Innholdet i request body blir da, f.eks.:

```
{
  "Text": "Kjøp seigmenn"
}
```

11. Legg til en metode for å slette elementet med gitt ID. (Hint: Denne metoden blir ganske lik metoden som henter elementet med gitt ID)

12. Bonus oppgave: Legg til en metode som oppdaterer elementet med gitt ID. (Hint: Denne metoden er ganske lik metoden som legger til et nytt element)

## Oppgave 3 – Dependency Injection og Konfigurering

I denne oppgaven vil vi fokusere på dependency injection i ASP.NET 5 og det nye konfigureringsrammeverket. Vi kommer til å lage en tjeneste som integrerer webapplikasjonen vår mot Azure Search. Til denne tjenesten trenger vi navnet til søketjenesten, API-nøkkel og navnet på indeksen som inneholder alle twittermeldingene som vi har indeksert opp. Dette vil vi nå legge i en konfigureringsfil som leses inn og registreres i IoC-kontaineren.

1. Legg til filen «config.json» i rotmappen til web-prosjektet

I denne filen vil dere se at Visual Studio foreslår «Data» gjennom IntelliSense. Det er fordi Microsoft forsøker å lage en standard for plassering av connection strings. Vi kommer ikke til å ha noen connection strings, så denne seksjonen vil ikke bli brukt i denne tutorialen.

2. Legg til det følgende i *config.json*

```
{
  "AppSettings": {
    "SearchServiceName": "SERVICE_NAME",
    "SearchServiceApiKey": "API_KEY",
    "TweetIndexName": "tweets"
  }
}
```

«SERVICE_NAME» og «API_KEY» byttes ut med navnet og API-nøkkelen til søketjenesten dere har satt opp i Azure, henholdsvis.

Konfigurasjonsfilen er nå klar, så det gjenstår bare å ta denne i bruk. Vi vil nå flytte oss over i *Startup.cs* der vi skal sette opp konfigurasjonsrammeverket til webapplikasjonen vår.

Hele ASP.NET 5-rammeverket bruker dependency injection, og vi kommer til å dra nytte av noen tjenester som allerede er registrert. I tillegg til tjenestene «`IApplicationBuilder`» og «`IServiceCollection`» som kommer inn som argument i *Configure(…)* og *ConfigureServices(…)*, har vi bl.a. tilgang til følgende tjenester:

- «`IApplicationEnvironment`» - Inneholder informasjon om miljøet til den kjørende applikasjonen
- «`IHostingEnvironment`» - Inneholder informasjon om hvor applikasjonen er hostet

I denne tutorialen vil vi kun bruke «`IApplicationEnvironment`», som ligger i pakken «`Microsoft.Dnx.Runtime`».

3. Legg til en referanse til «`Microsoft.Dnx.Runtime`» (1.0.0-beta8) i *project.json*
4. Legg til en konstruktør som tar inn «`IApplicationEnvironment`» i *Startup.cs*

I konstruktøren vil vi opprette konfigureringsrammeverket.

5. Legg til en referanse til «`Microsoft.Framework.Configuration`» (1.0.0-beta8)
6. Legg til en referanse til «`Microsoft.Framework.Configuration.Json`» (1.0.0-beta8)
7. Legg til et statisk felt av typen «`IConfigurationRoot`» i *Startup.cs*

8. Bruk den følgende koden til å gi det statiske feltet en verdi

```
_configurationRoot = new ConfigurationBuilder()
    .SetBasePath(appEnv.ApplicationBasePath)
    .AddJsonFile("config.json")
    .Build();
```

Denne kodesnutten bruker argumentet som kommer inn i konstruktøren til å sette filstien der letingen etter eventuelle konfigurasjonsfiler starter, og registrerer *config.json*-filen vi la til tidligere.

JSON-filer er bare et av alternativene man kan bruke. Det er støtte for XML-filer, INI-filer (!) og miljøvariabler, og man kan uten problemer bruke alle alternativene på samme tid. Dersom flere konfigurasjonsfiler inneholder den samme verdien, er det verdien i den siste filen som blir gjeldene.

Det neste vi skal gjøre er å registrere konfigurasjonsinstansen i IoC-kontaineren slik at vi kan ta i bruk denne hvor det skulle ønskes. Siden instansen er statisk og vil leve så lenge webapplikasjonen kjører, kan vi like godt registrere instansen som en singleton.

9. Legg til den følgende kodesnutten i *ConfigureServices(…)*

```
services.AddSingleton(serviceProvider => _configurationRoot);
```

Det denne kodesnutten gjør er å alltid returnere den statiske instansen vi har definert i *Startup.cs* dersom en klasse tar inn «`IConfigurationRoot`» som parameter.

Nå er vi klar til å lage tjenesten som bruker Azure Search.

10. Legg til klassen «`AzureSearchService`»
11. Legg til en konstruktør som tar inn «`IConfigurationRoot`»

Det neste steget er å hente ned Azure Search SDK. Denne pakken bygger ikke under DNX Core, så denne tar vi like greit å fjerner fra prosjektet.

12. Fjern «dnxcore50» fra «frameworks»-seksjonen i *project.json*
13. Legg til referanse til «`Microsoft.Azure.Search`» (0.13.0-preview)

Tilbake i *AzureSearchService.cs* er vi klar til å opprette søkeklienten. For å gjøre dette, trenger vi navnet på søketjenesten i Azure, API-nøkkel og navnet på Twitter-indeksen fra konfigurasjonsfilen. For å nøste seg nedover i konfigurasjonsfilen, brukes kolon (:) til å skille lagene. Siden vi la verdiene vi trenger under «AppSettings» blir da nøkkelen for API-nøkkelen «AppSettings:SearchServiceApiKey».

14. Bruk den følgende kodesnutten til å opprette søkeklienten i konstruktøren:

```
var searchServiceName = configurationRoot["AppSettings:SearchServiceName"];
var searchApiKey = configurationRoot["AppSettings:SearchServiceApiKey"];
var tweetIndexName = configurationRoot["AppSettings:TweetIndexName"];

var serviceClient = new SearchServiceClient(
    searchServiceName, new SearchCredentials(searchApiKey));

_indexClient = serviceClient.Indexes.GetClient(tweetIndexName);
```

Feltet «_indexClient» er av typen «`SearchIndexClient`» og lagres på et privat felt på klassenivå. Det er denne klassen vi skal bruke til å søke i Twitter-indeksen vi har i Azure. Så la oss legge til metoden

for å gjøre et enkelt søk. For å gjøre resultatet fra søket sterkt typet, trenger vi en klasse som ligger i prosjektet «Entities.Package».

15. Legg til en referanse til «`Entities.Package`» (1.0.0-*) i *project.json*
16. Legg til den følgende metoden i *AzureSearchService.cs*:

```
public virtual DocumentSearchResponse<FlattendTweet> Search(
    string query, string username = null)
{
    var result = _indexClient.Documents.Search<FlattendTweet>(
        query,
        new SearchParameters
        {
            OrderBy = new List<string> { "CreatedAt" },
            Filter = string.IsNullOrEmpty(username)
                ? null
                : $"Username eq '{username}'"
        });

    return result;
}
```

Denne metoden utfører et simpelt søk etter teksten som sendt inn via parameteren «query». Den har også støtte for å legge til et filter på brukernavn (dersom det er spesifisert) vha. et OData-query. Resultatet er sortert etter «CreatedAt» feltet.

For at denne klassen skal være tilgjengelig for andre klasser via dependency injection må vi registrere den i IoC-konteineren. Denne klassen vil vi at skal nyes opp når den trengs og forkastes med en gang vi er ferdig med den. Derfor registrerer vi denne som «transient».

17. Registrer «`AzureSearchService`» som «transient» i *ConfigureServices(…)*

Den siste delen av oppgaven er å lage API-kontrolleren som front-end kan kalle for å utføre et søk.

18. Opprett API-kontrolleren «`SearchController`» med route «/api/search»
19. Legg til en konstruktør som tar inn «`AzureSearchService`»
20. Legg til en action som kaller *Search(…)* på «`AzureSearchService`»

Back-end skal nå være klar for bruk og vi kommer nå til å bevege oss over til front-end. For at webapplikasjonen vår skal servere oss statiske filer må vi legge til én modul til.

21. Legg til en referanse til «`Microsoft.AspNet.StaticFiles`» (1.0.0-beta8)
22. Legg til de to følgende modulene i *Configure(…)*-metoden i *Startup.cs*

```
app.UseDefaultFiles();
app.UseStaticFiles();
```

«UseDefaultFiles» sørger for at «index.html» blir servert når man navigerer til roten av webapplikasjonen (http://localhost:[port]/).

## Oppgave 4 – Bower

Legg til ny bower konfigurasjonsfil til Web-prosjektet

1. Add > New Item > Bower Configuration File (under Client Side)

Default konvensjon er at alle pakker som bower laster ned, havner i *wwwroot* mappen. Vi ønsker å heller legge det til i en annen mappe *bower_components*

2. Ekspander *bower.json* og åpne *.bowerrc*
3. Endre directory-verdien fra *wwwroot/lib* til *bower_components*

Vi ønsker å bruke bootstrap i web-applikasjonen vår.

4. I *bower.json*, legg til en ny property med navn *boostrap*
5. Intellisens bør foreslå et sett med package-versjoner, vi velger det nyeste. (Dersom ingenting vises, sett versjonen til «3.0.0»

Når filen nå lagres, vil *Dependencies* automatisk oppdateres. Mer info om hva som ble kjørt kan leses utifra Output vinduet for *Bower/npm*.

## Oppgave 5 - Gulp

Legg til ny gulp konfigurasjonsfil til Web-prosjektet

1. Add > New Item > Gulp Configuration File (under Client Side)

Legg til ny package konfigurasjonsfil til Web-prosjektet

2. Add > New Item > NPM Configuration File (under Client Side)

Legg til gulp I *devdependencies* i *package.json*

3. «gulp»: «3.9.0»

Opprett task for kopiering av ønskede filer fra *bower_components* til *wwwroot/lib* i *gulpfile.js*

4.

```
gulp.task('copy', function () {

});
```

For alle mappene direkte under *bower_components*, beskriv (i 'copy'-tasken) de filtypene vi ønsker å kopiere, og i hvilke mapper vi finner de.

5.

```
var bower = {
        "bootstrap": "bootstrap/dist/**/*.{js,map,css,ttf,svg,woff,eot}"
    }
```

For å kopiere alle filene over til *wwwroot/lib,* bruker vi *rimraf* rammeverket. Vi bruker også *fs* rammeverket for å lese filer. Disse må legges til i package.json som vi gjorde med gulp.

6.

```
"rimraf": "2.4.3",
"fs": "0.0.2",
```

Require rimraf og fs-javascriptene i gulpfile.js, slik at de kan brukes.

7.

```
var gulp = require('gulp');
var rimraf = require('rimraf');
var fs = require('fs');
```

Vi kan nå hente project.json objektet ved å bruke fs sin readFileSync() metode. Vi oppretter en variabel *project* utenfor gulp-tasken.

8.

```
...
var fs = require('fs');

eval("var project = " + fs.readFileSync("./project.json"));
```

I *copy*-tasken, kopier så alle de ønskede filene fra *bower_components* til *wwwroot/lib*.

9.

```
for (var destinationDir in bower) {
        gulp.src("./bower_components/" + bower[destinationDir])
            .pipe(gulp.dest("./" + project.webroot + "/lib/" + destinationDir));
}
```

Vi kan nå åpne Task Runner Explorer og kjøre gulp-tasken.

10. Høyreklikk *copy* > Run

Bootstrap skal nå ligge i wwwroot/lib.

Vi ønsker å legge til en ny gulp-task som fjerner alt i *wwwroot/lib.* I *gulfile.js*, legg til en ny gulptask med navn *clean.*

11.

```
gulp.task(clean, function () {

});
```

Bruk så rimraf for å slette alt innenfor wwwroot/lib mappen.

12.

```
rimraf("./" + project.webroot + "/lib", callback);
```

Kjør så gulptasken vha Task Runner Explorer. *Wwwroot/lib* Skal nå være tom.

Vi ønsker å sette *copy*-tasken slik at den kjøres hver gang vi bygger. Dette gjøres via Task Runner Explorer:

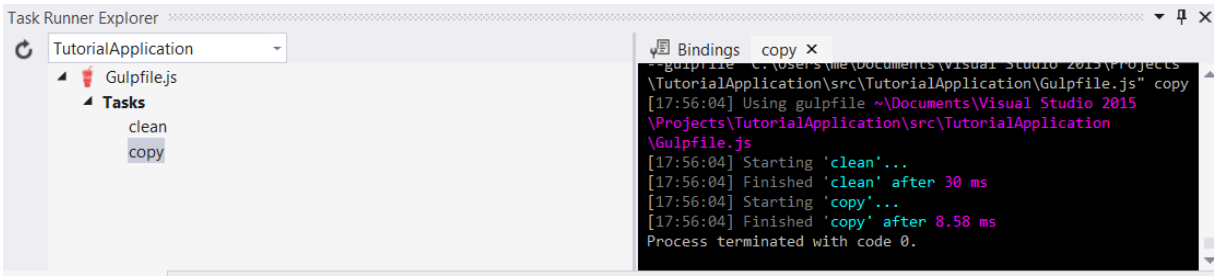
1. Task Runner Explorer > copy > Bindings > After Build

Vi ønsker at clean skal kjøres før copy, men etter hver build. Vi legger til et ekstra parameter til *copy*-tasken, som er et array av gulp-tasknavn som skal kjøres før *copy*.

*14.*

```
gulp.task('copy', ['clean'], function () {
```

Når vi nå bygger, så skal først clean bli kjørt, deretter copy.



## Løsningsforslag

Dette avsnittet har løsningsforslag til en del av oppgavene.

### Oppgave 1

project.json

```
"dependencies": {
  "Microsoft.AspNet.IISPlatformHandler": "1.0.0-beta8",
  "Microsoft.AspNet.Server.Kestrel": "1.0.0-beta8",
  "Microsoft.AspNet.Diagnostics": "1.0.0-beta8"
},
```

Startup.cs

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseWelcomePage();
    }
}
```

### Oppgave 2

project.json

```
"dependencies": {
  "Microsoft.AspNet.IISPlatformHandler": "1.0.0-beta8",
  "Microsoft.AspNet.Server.Kestrel": "1.0.0-beta8",
  "Microsoft.AspNet.Diagnostics": "1.0.0-beta8",
  "Microsoft.AspNet.Mvc": "6.0.0-beta8"
},
```

Startup.cs

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseMvc();
        app.UseWelcomePage();
    }
}
```

ViewModels/TodoItem.cs

```
public class TodoItem
{
    public int Id { get; set; }
    public string Text { get; set; }
}
```

API/TodoController.cs

```
[Route("/api/todo")]
public class TodoController : Controller
{
    private static List<TodoItem> _todoItems = new List<TodoItem>
    {
        new TodoItem { Id = 1, Text = "Kjøp Q-Tips" },
        new TodoItem { Id = 2, Text = "Ta en dusj" }
    };

    [HttpGet]
    public IActionResult Get()
    {
        return Json(_todoItems);
    }

    [HttpGet("{id:int}")]
    public IActionResult Get(int id)
    {
        var todoItem = _todoItems.FirstOrDefault(item => item.Id == id);
        if (todoItem == null)
            return HttpNotFound();

        return Json(todoItem);
    }

    [HttpPost]
    public IActionResult Add([FromBody]TodoItem todoItem)
    {
        var nextId = _todoItems.Max(item => item.Id) + 1;
        todoItem.Id = nextId;

        _todoItems.Add(todoItem);

        return Json(todoItem);
    }

    [HttpDelete("{id:int}")]
    public IActionResult Delete(int id)
    {
        var todoItem = _todoItems.FirstOrDefault(item => item.Id == id);
        if (todoItem == null)
            return HttpNotFound();

        _todoItems.Remove(todoItem);

        return Ok();
    }
}
```

API/TodoController.cs – Bonus

```
[HttpPut("{id:int}")]
public IActionResult Edit([FromRoute]int id, [FromBody]TodoItem newTodoItem)
{
    var todoItem = _todoItems.FirstOrDefault(item => item.Id == id);
    if (todoItem == null)
        return HttpNotFound();

    todoItem.Text = newTodoItem.Text;

    return Json(todoItem);
}
```

## Oppgave 3

config.json

```
{
  "AppSettings": {
    "SearchServiceName": "SERVICE_NAME",
    "SearchServiceApiKey": "API_KEY",
    "TweetIndexName": "tweets"
  }
}
```

Startup.cs

```
public class Startup
{
    private static IConfigurationRoot _configurationRoot;

    public Startup(IApplicationEnvironment appEnv)
    {
        _configurationRoot = new ConfigurationBuilder()
            .SetBasePath(appEnv.ApplicationBasePath)
            .AddJsonFile("config.json")
            .Build();
    }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();

        services.AddSingleton(serviceProvider => _configurationRoot);

        services.AddTransient<AzureSearchService>();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseMvc();
        app.UseDefaultFiles();
        app.UseStaticFiles();
    }
}
```

project.json

```
{
  "webroot": "wwwroot",
  "version": "1.0.0-*",

  "dependencies": {
    "Microsoft.AspNet.IISPlatformHandler": "1.0.0-beta8",
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-beta8",
    "Microsoft.AspNet.Diagnostics": "1.0.0-beta8",
    "Microsoft.AspNet.Mvc": "6.0.0-beta8",
    "Microsoft.Framework.Configuration": "1.0.0-beta8",
    "Microsoft.Framework.Configuration.Json": "1.0.0-beta8",
    "Microsoft.Azure.Search": "0.13.0-preview",
    "Microsoft.Dnx.Runtime": "1.0.0-beta8",
    "Microsoft.AspNet.StaticFiles": "1.0.0-beta8",
    "Entities.Package ": "1.0.0-*"
  },

  "commands": {
    "web": "Microsoft.AspNet.Server.Kestrel"
  },

  "frameworks": {
    "dnx451": { }
  },

  "exclude": [
    "wwwroot",
    "node_modules"
  ],
  "publishExclude": [
    "**.user",
    "**.vspscc"
  ]
```

API/SearchController.cs

```
[Route("/api/search")]
public class SearchController : Controller
{
    private AzureSearchService _searchService;

    public SearchController(AzureSearchService searchService)
    {
        _searchService = searchService;
    }

    [HttpPost]
    public IActionResult Search([FromBody]SearchCriteria searchCriteria)
    {
        var searchResult = _searchService.Search(
            searchCriteria.Query, searchCriteria.Username);

        return Json(searchResult);
    }
}
```

ViewModels/SearchCriteria.cs

```
public class SearchCriteria
{
    public string Query { get; set; }
    public string Username { get; set; }
}
```

Services/AzureSearchService.cs

```
public class AzureSearchService
{
    private SearchIndexClient _indexClient;

    public AzureSearchService(IConfigurationRoot configurationRoot)
    {
        var searchServiceName = configurationRoot["AppSettings:SearchServiceName"];
        var searchApiKey = configurationRoot["AppSettings:SearchServiceApiKey"];
        var tweetIndexName = configurationRoot["AppSettings:TweetIndexName"];

        var serviceClient = new SearchServiceClient(
            searchServiceName, new SearchCredentials(searchApiKey));

        _indexClient = serviceClient.Indexes.GetClient(tweetIndexName);
    }

    public virtual DocumentSearchResponse<FlattendTweet> Search(
        string query, string username = null)
    {
        var result = _indexClient.Documents.Search<FlattendTweet>(
            query,
            new SearchParameters
            {
                OrderBy = new List<string> { "CreatedAt" },
                Filter = string.IsNullOrEmpty(username)
                    ? null
                    : $"Username eq '{username}'"
            });

        return result;
    }
}
```