Webben vNext

Lab 3 – Frontend with TypeScript

# Intro

In this lab you will learn how to enhance the website client with TypeScript. We will enable a preview of the images before they are being uploaded.

Techniques and concepts you will use:

* TypeScript
* NPM
* Gulp
* Run NPM and Gulp from a build server (VSTS)

Once you are finished with the lab your code should look similar to this branch in the GitHub repo. Feel free to have a look if you need inspiration.

<https://github.com/ActiveSolution/lab-webben-vnext/tree/lab-3-frontend>

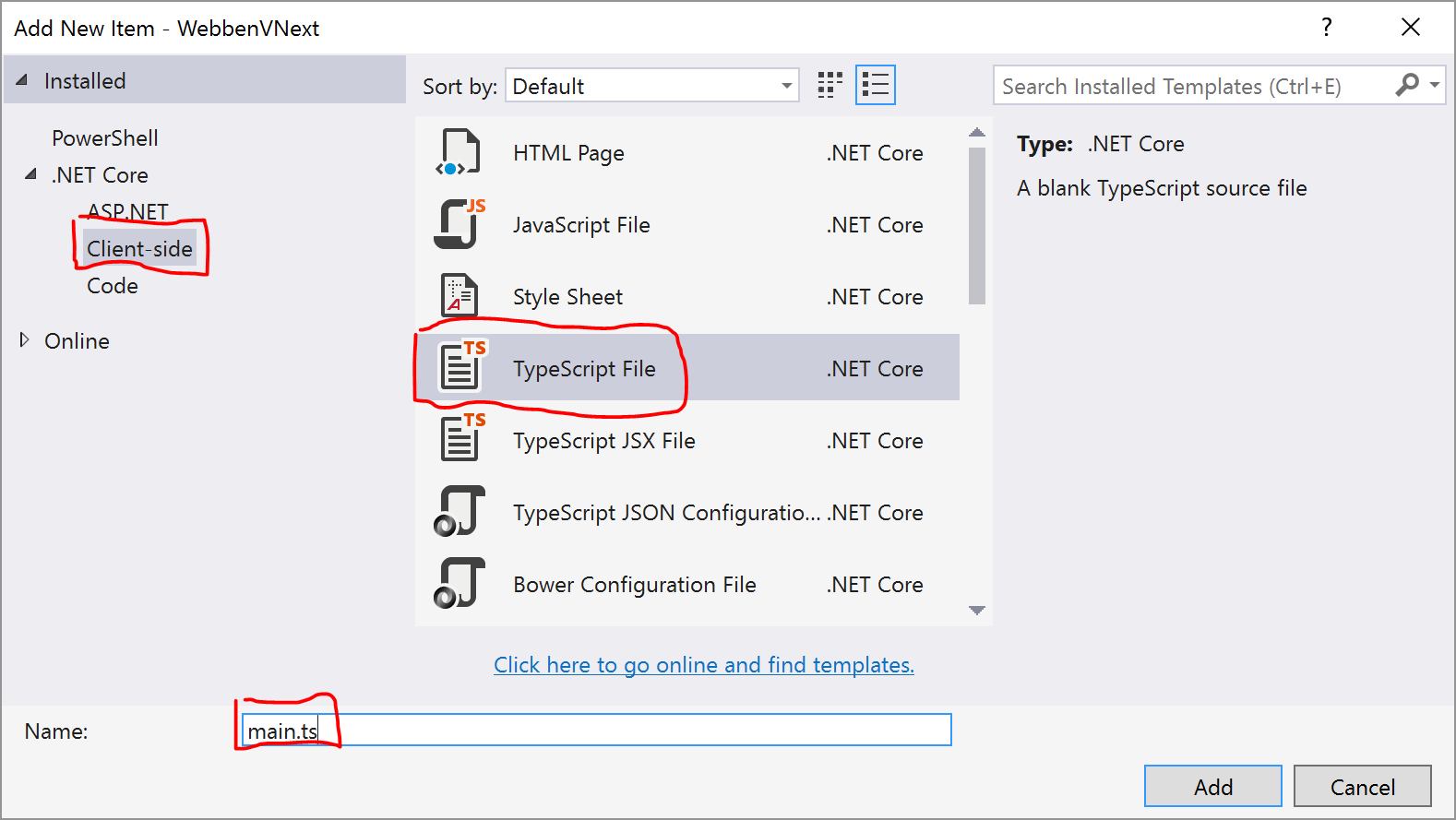
# Lab Instructions

## Configure the front en build pipeline

Our goal is to write some TypeScript will preview the images before being uploaded to the server. TypeScript is a superset of JavaScript and a language that the browsers does not understand and therefore we need to convert it to JavaScript. This process is called transpiling, a combination of “transforming” and “compiling”.

### Create a simple typescript file

At first we need to have a TypeScript file to transpile, before we get into the more advanced stuff, we will start with a simple Hello World.

1. In your project, create a folder called *src*. I’s a common pattern to store the source files of a client side application in such folder.
2. In the *src* folder, create a TypeScript file called *main.ts*.  
   
3. Add some Hello World-code to *main.ts* that uses TypeScript features, for example:

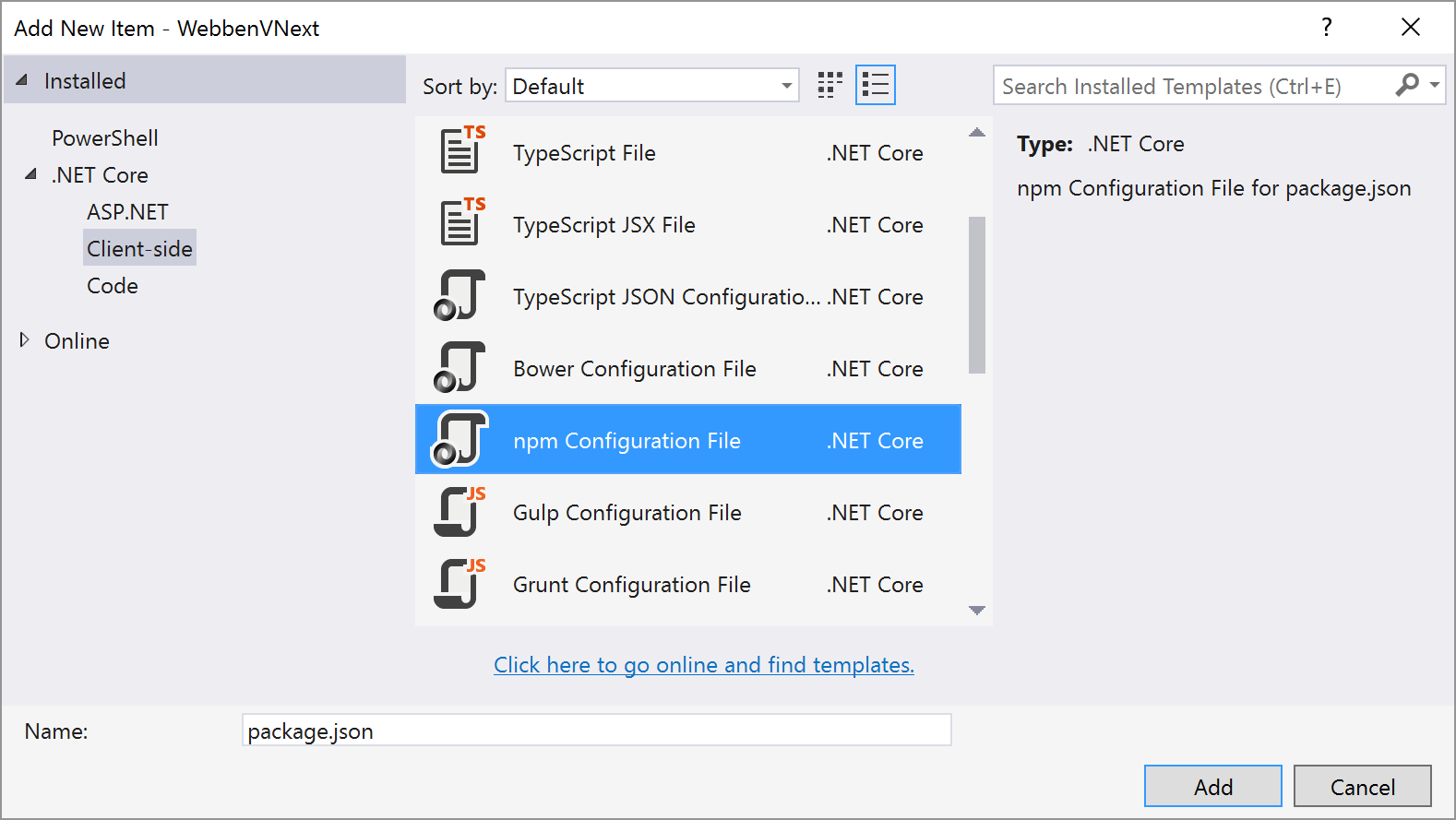
|  |
| --- |
| let userName: string;  userName = prompt('What your name?');  alert(`Hello ${userName}!`); |

Now you have created your first TypeScript file, later on we will add some more code to it. Right now you can’t run it, but we will make sure you can using Gulp later on.

### Configure NPM

NPM (Node Package Manager) is a tool that handles client side packages and their dependencies. We will use it to download the tooling and dependencies needed to transpile our TypeScript file.

1. In the root of your project, add a file called *package.json*

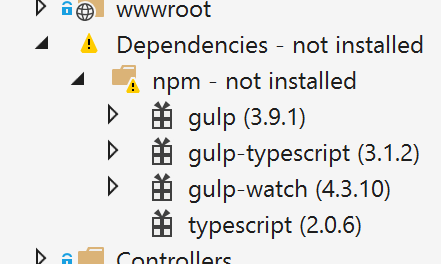


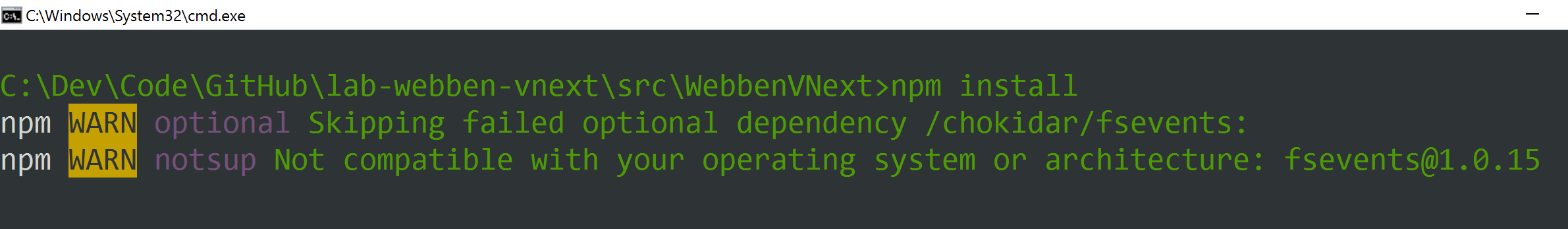
1. Visual Studio will give you some default content such as version, name and so on. What you need to configure not is the section for devDependencies. This is the place we can specify what packages/tools our application needs to work. It’s also specified what version of the package we want, feel free to try newer versions if available, but we can’t guarantee that the combination will work.

Add the following rows in the devDependencies section:

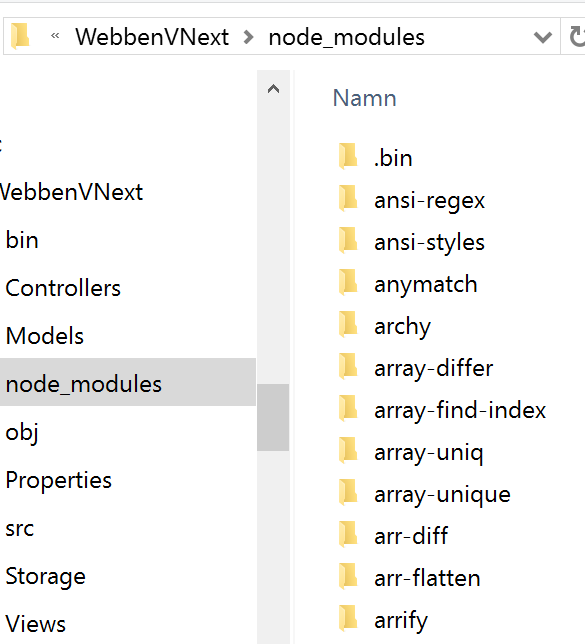
|  |
| --- |
| "typescript": "2.0.6",  "gulp": "3.9.1",  "gulp-typescript": "3.1.2",  "gulp-watch": "4.3.10" |

* The package “typescript” contains the core compiler for TypeScript.
* The packages starting with “gulp” contains our task runner and extensions to it, more on that later.

1. Visual Studio should now start to restore these packages. Don’t get scared if a warning symbol appears in the UI, it’s a known bug but it should still work. 
2. What VS does behind the scenes is to run *npm install* in the folder where the *package.json* is located. Feel free to do that yourself from the command line to see what happens.



1. Verify that the packages have been restored by locating the folder node\_modules using windows (or the tool of your choice). It should contain a list of subfolders of packages.

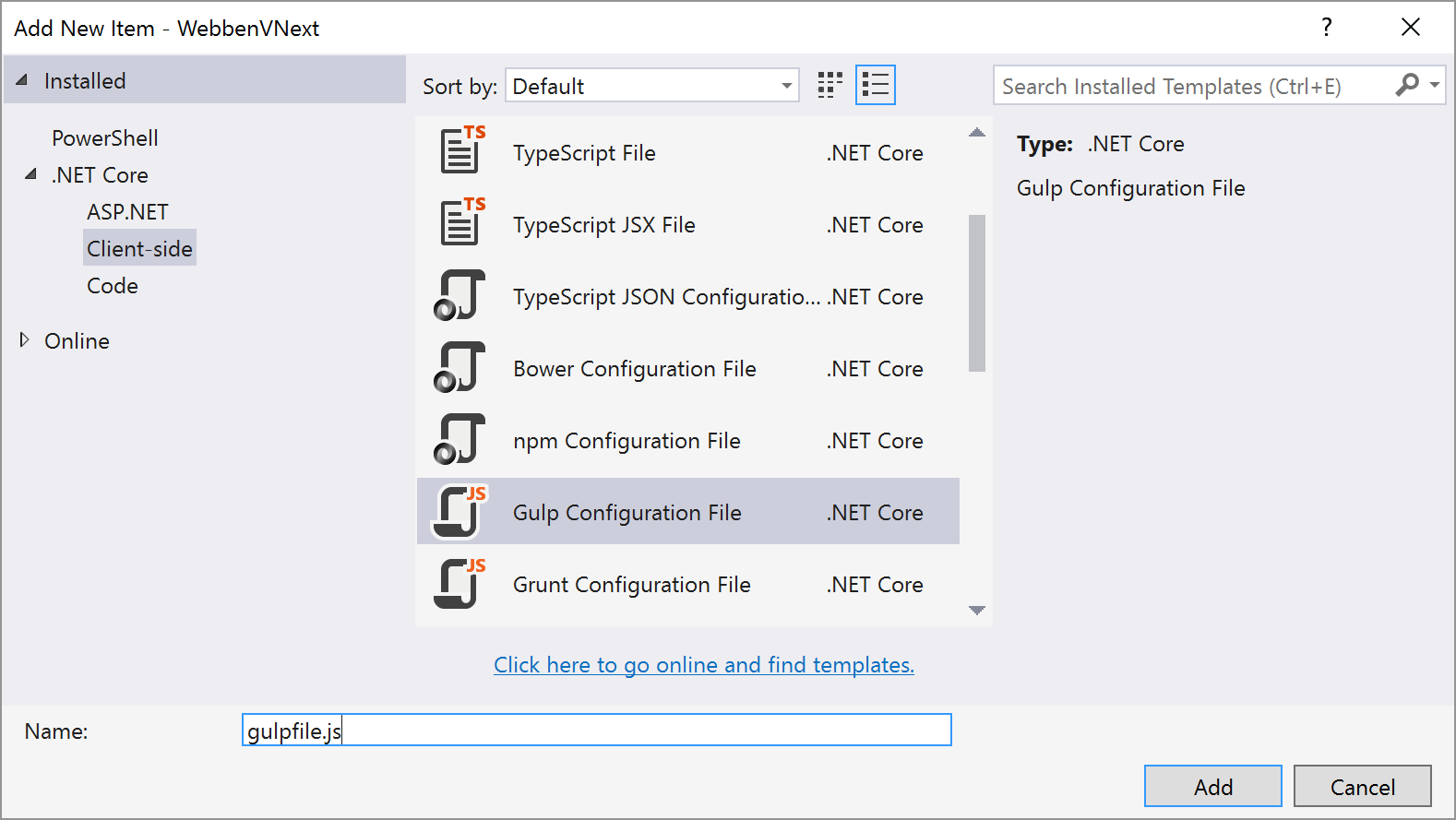


Your project now has the tools that we need to transpile our TypeScript-file.

### Configure the task runner Gulp

When working with client side assets such as JavaScript, CSS, images and so on It’s a good practice to preprocess them in some way. Maybe you’d like to minify, bundle or validate your code before releasing it. To do so you need some kind of task runner and in this lab we will use Gulp as the tool. Gulp will be the task runner that calls the TypeScript transpiler and converts it from TypeScript to JavaScript that the browser understands.

1. In the root of your project, add a file called *gulpfile.js*



1. Visual Studio will add some boilerplate code, but we are not in need of that now, so begin with removing all the code in that file.
2. Gulp runs on top of Node.js and that’s the conventions we need to follow. So to begin with, we need to import the packages that we want to use. This is similar to the “import” statements you find in C# .NET. We will import the core of gulp as *gulp* and the typescript extension as *ts*. Add these lines to do so.

|  |
| --- |
| var gulp = require('gulp');  var ts = require('gulp-typescript'); |

1. Next up we must define a gulp task. Each task has its unique name and can be called from the command line and therefore from our build server, more on that later. Create a task called default by adding these lines of code:

|  |
| --- |
| gulp.task(**'default'**, function () {  return gulp.src('src/\*\*/\*.ts')  .pipe(ts({  out: 'app.js'  }))  .pipe(gulp.dest('wwwroot/dist'));  }); |

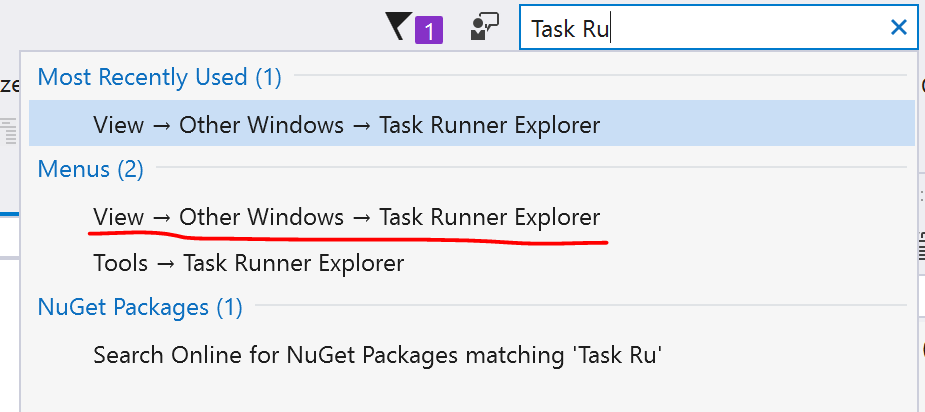
* gulp.src('src/\*\*/\*.ts') what files to look for, in our case all files ending with .ts in the folder src.
* .pipe() is a convention in gulp where you pipe the stream of files through a series of extensions
* In our case, that extension is ts, which we provide with the configuration of what to name the final javascript file
* Finally we pipe the result to a new destination. Because ASP.NET Core (by default) only will serve files located in *wwwroot* we need to place our files here. The preferred way is to place the result of a “build” in a folder called dist.

1. Last but not least, we will ease the development by adding a task that will make sure to automatically transpile the TypeScript whenever a change is detected. This is a must for rapid development. Add this at the enf of your file:

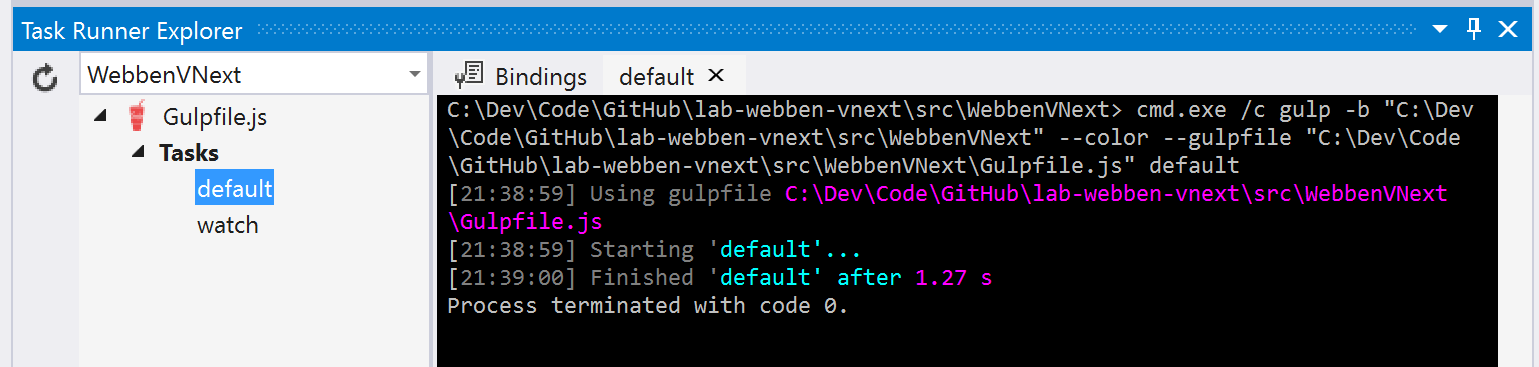
|  |
| --- |
| gulp.task(**'watch'**, function () {  gulp.watch('src/\*\*/\*.ts', ['default']);  }); |

* gulp.watch is basically a feature that listens to changes in files and whenever a change occurs it triggers another task. In our case we listen for changes in \*.ts files and triggers the default task.

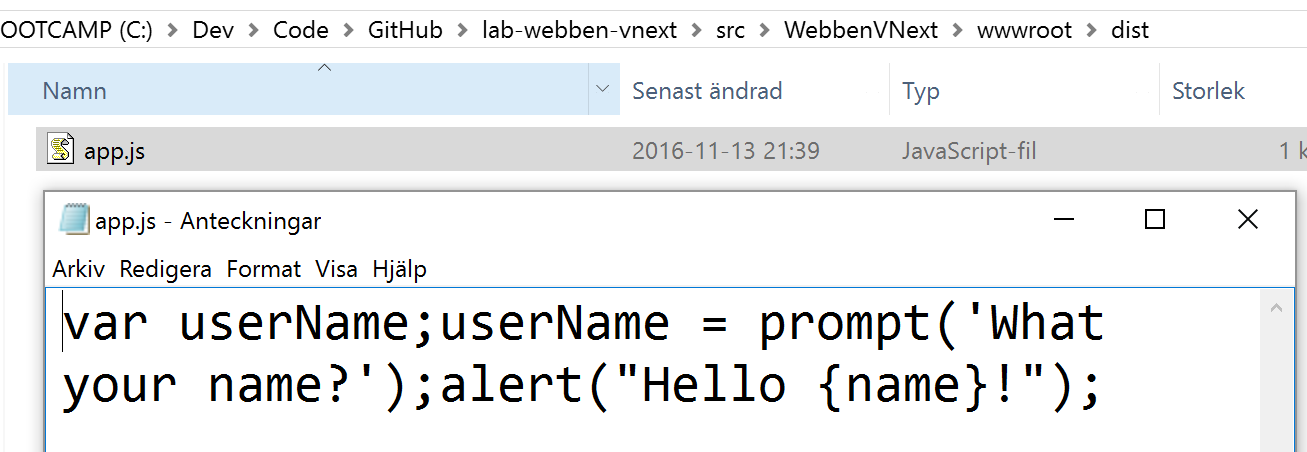
1. To run your tasks and verify that they work we will use the built in tool in Visual Studio called *Task Runner Explorer*. You can find it using the Quick launch bar in the upper right corner.



1. Once the window is opened you can launch your tasks by double clicking them in the list. If your tasks do not appear, press the refresh button. First launch *default* and then *watch*.



1. If everything is working properly the script should have generated app.js in the *wwwroot/dist* folder and you should see that the TypeScript specific features have turned into pure JavaScript.



1. Now it’s time to run our generated JavaScript and to do so we need to add it to our ASP.NET view. Open *Views/Shared/\_Layout.cshtml* and add the following line just before the ending *</body>*-tag.

|  |
| --- |
| <script src="dist/app.js"></script> |

1. Launch the application in the browser. If it prompts for your name everything is working fine ☺

You now have a basic task runner pipeline configured and we are ready to write some serious TypeScript.

### Implement the file upload preview

TypeScript is a superset of JavaScript and adds some advantages where the addition of types is one of the best. It gives you a better tooling experience and compile time errors when the types do not add up. We will use TypeScript to write a “vanilla” implementation of an image previewer. In a bigger project it’s common to use a framework such as Angular or React (which both play well with TypeScript) to save development time.

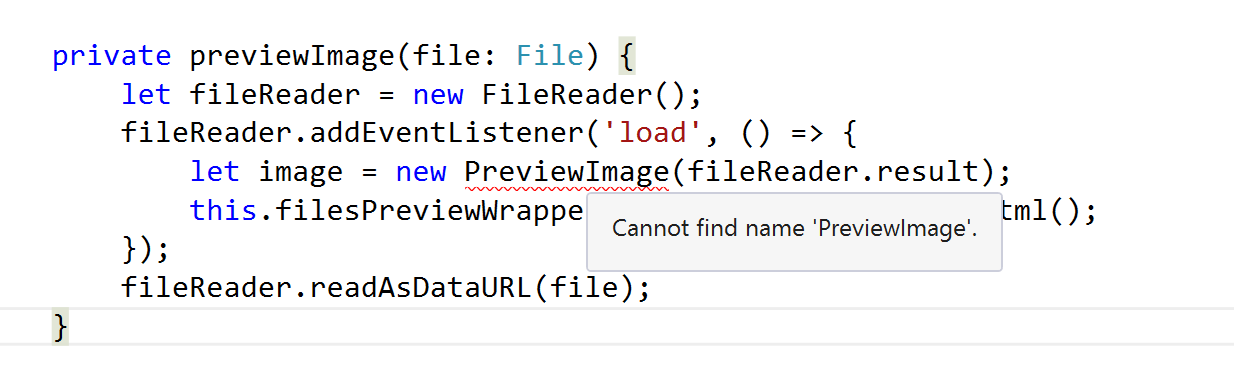
1. First we need to add some markup to our html so our script has somewhere to place the preview images. In *Views/Home/Index.cshtml,* Add these lines of code after the card that contains the file upload form.

|  |
| --- |
| <h3>Förhandsgranskning</h3>  <div id="files-preview"></div> |

* The id *files-preview* will be the key to access this DOM element from code.

|  |
| --- |
| class FileApplication {  private filesInput: HTMLInputElement;  private filesPreviewWrapper: HTMLDivElement;  constructor() {  this.filesInput = document.querySelector('#files') as HTMLInputElement;  this.filesPreviewWrapper = document.querySelector('#files-preview') as HTMLDivElement;  }  public run(): void {  if (this.filesInput !== undefined && this.filesPreviewWrapper !== undefined) {  this.filesInput.addEventListener('change', event => {  this.previewImages();  });  }  }  private previewImages(): void {  this.filesPreviewWrapper.innerHTML = '';  let files = this.filesInput.files;  for (let i = 0; i < files.length; i++) {  this.previewImage(files[0]);  }  }  private previewImage(file: File) {  let fileReader = new FileReader();  fileReader.addEventListener('load', () => {  let image = new PreviewImage(fileReader.result);  this.filesPreviewWrapper.innerHTML += image.toHtml();  });  fileReader.readAsDataURL(file);  }  }  var app = new FileApplication();  app.run(); |

1. Go back to *src/main.ts* and clean out the code we wrote there. Instead replace it with the following code:
2. Did you see that TypeScript actually already helped you? It actually tells you that there seems to be something missing, the PreviewImage-class. That’s the class we will implement and explain in more detail.



1. At the top of the file, add an empty class called *PreviewImage*. Classes is not an exclusive feature of TypeScript anymore, but actually part of the ECMAScript 6 specification. But because only modern browsers understand them natively, TypeScript is a great tool to transpile it into code that older browsers understand.

|  |
| --- |
| class PreviewImage {  } |

1. The purpose of the class is to take a URL as a string and expose a function to generate the HTML for an image that can be used as preview. Therefore we need to have a constructor that can store the URL. Add the constructor in the class using this line:

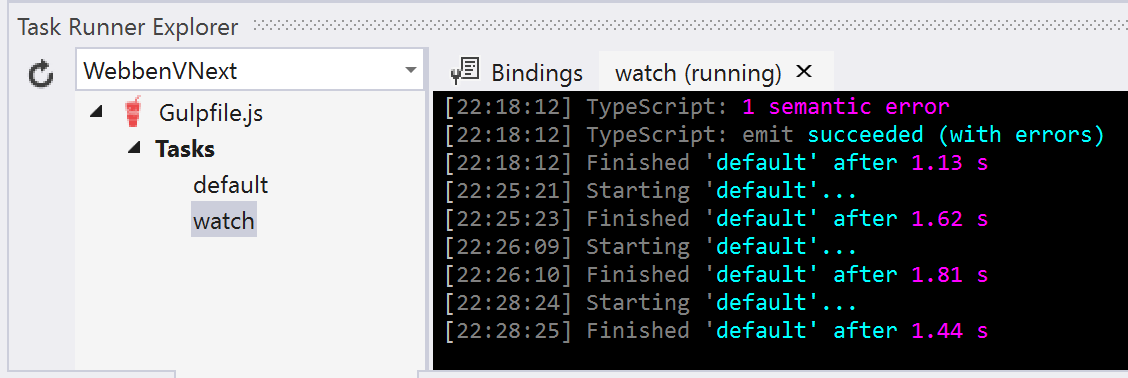
|  |
| --- |
| constructor(private url: string) { } |

* To begin with we specify that the constructor expects a string as input, therefore we won’t be able to pass numbers, objects or other values by mistake. Try changing the value when the PreviewImage is being called and see how Visual Studio shows that it’s wrong.

1. Last but not least we need to implement the function that produces the HTML, do so by adding this code in the class:

|  |
| --- |
| public toHtml(): string {  return `<div class="card">  <img src="${this.url}" class="img-fluid" />  </div>`;  } |

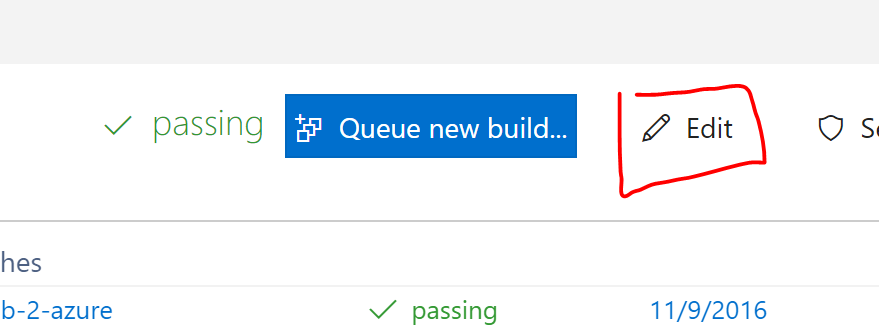
* public tells TypeScript that this function should be available from outside of the class
* : string tells TypeScript that this function is expected to return a string.
* Finally we return a string using ECMAScript string literals. It’s similar to the feature available in C# 6.

1. If you did let watch run in the background Gulp has already generated the final app.js file for you, otherwise run the *default* task again manually.  
   
2. It’s now time to verify the final application. Launch the application in the browser and try it out! Browse for some images and they should appear as previews under the preview section.

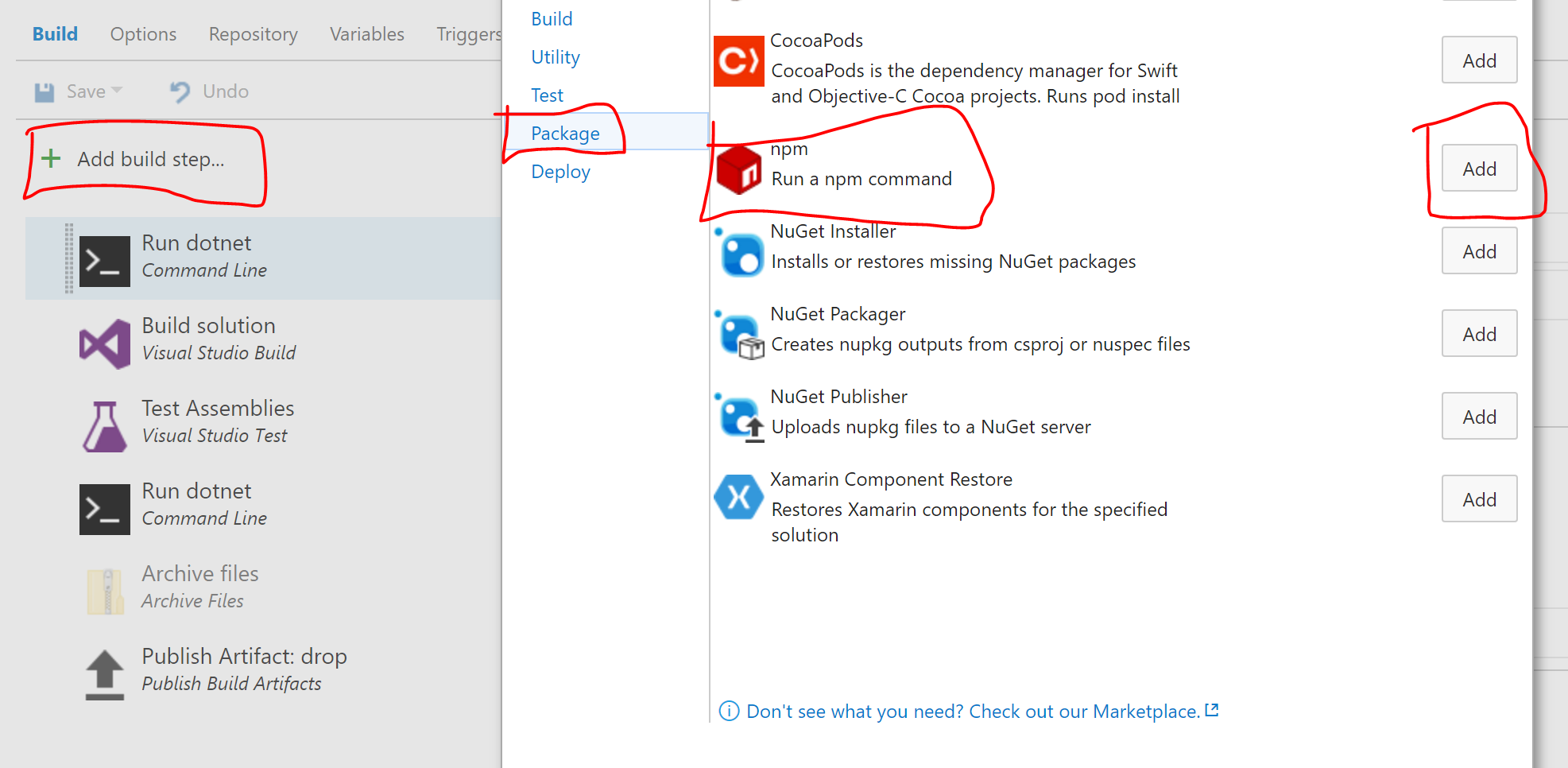
## Run Gulp in Visual Studio Team Services

So far we have let the *Task Runner Explorer* run our Gulp tasks, but that’s a tool that is part of Visual Studio and something that won’t be available on a build server, in our case Visual Studio Team Services (VSTS). The final part of this lab will focus on running npm and gulp from VSTS.

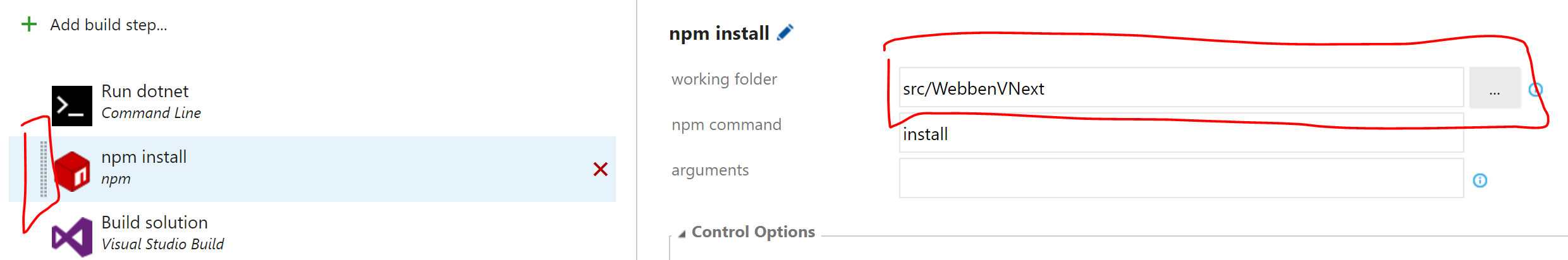
1. Open Visual Studio Team Services in your browser and navigate to the Build definition that you created in the previous lab. Once you’ve located it, click Edit.



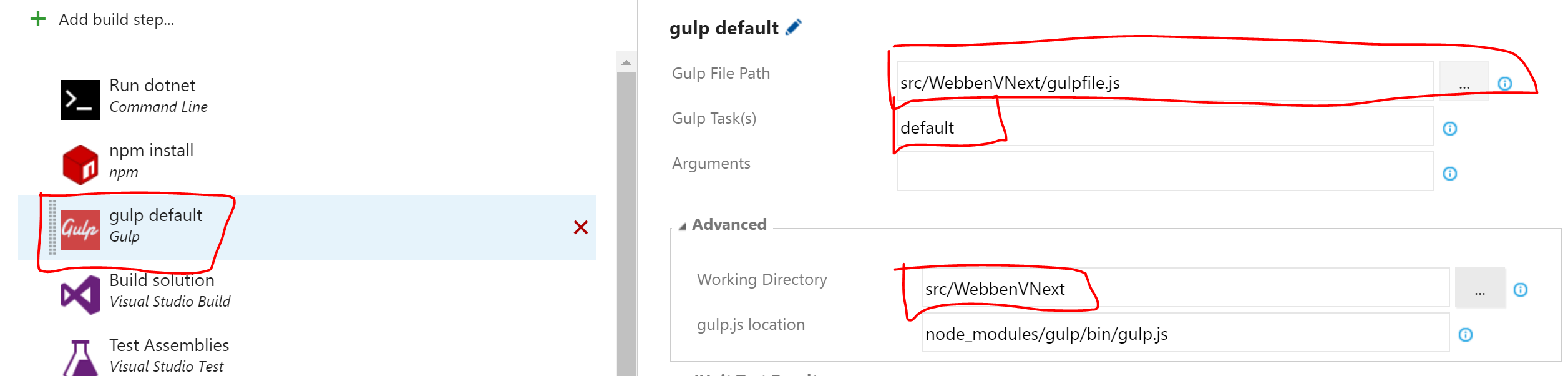
1. To begin with we need to ensure that the build server restores all the tools it needs, and the tools is managed by npm. In VSTS there is a built in build task that does this for us. Click *Add build step…* and locate the *npm* task under the *Package* category. Then click *Add*.



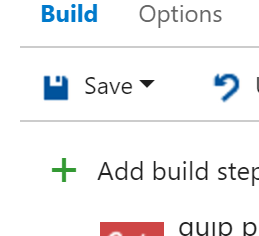
1. The npm task needs to be rearranged and configured. Move it right after the first dotnet task and set the *Working folder* to *src/WebbenVNext.*



1. The last task we need to add is the one for Gulp, place it after the npm task. Follow the same concept as you did for npm. The gulp task is located under the *Build* category. Make sure to configure Gulp File Path, Gulp Task(s) and Working directory according to the image below. The Gulp task *default* is referring to the task name we gave it in our gulpfile.js.



1. Finish up by saving your work and commit the changes you have done throughout this lab. If it works correct VSTS should transpile your TypeScript to JavaScript using Gulp and the final app.js will be published to Azure.



# Next step

## Challenges

In case you have time left, here are some challenges that you can try to implement using the skills you’ve learned in this lab.

* Validate the picture file type and/or file format on the client using TypeScript before it’s being sent to the server.
* Style the preview of the images to look nicer.
* Display metadata such as file size next to the images before they are being uploaded.
* Show a progress bar of the upload while the image is being uploaded.

## Sessions during TechDays

If you wish to dig deeper into the subjects that you’ve learned throughout this lab, here are some suggestions of related sessions during TechDays.

* Write TypeScript With Visual Studio Code, Rachel Appel (<http://tdswe.se/session/write-typescript-with-visual-studio-code-level-200/>)
* I say a "front-end build pipeline" – You say wat!?, Chris Klug (<http://tdswe.se/session/i-say-a-front-end-build-pipeline-you-say-wat/>)