

DNC Magazine

www.dotnetcurry.com





@suprotimagarwal

Editor in Chief

the team

Editor In Chief :

Suprotim Agarwal
(suprotimagarwal@dotnetcurry.com)

Art Director :

Minal Agarwal

Contributing Authors :

Yacoub Massad
Subodh Sohoni
Ravi Kiran
Gerald Versluis
Darren Gillis
Daniel Jimenez Garcia
Damir Arh

Technical Reviewers :

Damir Arh
Daniel Jimenez Garcia
Gouri Sohoni
Suprotim Agarwal
Vikram Pendse
Yacoub Massad

Windows, Visual Studio, ASP.NET, Azure, TFS & other Microsoft products & technologies are trademarks of the Microsoft group of companies. 'DNC Magazine' is an independent publication and is not affiliated with, nor has it been authorized, sponsored, or otherwise approved by Microsoft Corporation. Microsoft is a registered trademark of Microsoft corporation in the United States and/or other countries.

EDITOR'S NOTE

Reaching our 50th edition has been no small feat on the part of the DotNetCurry team, and for all the many people who have been involved over the past nine years of this magazine.

This milestone has led us to reflect on the achievements of our previous editions, our authors, our reviewers, contributors and our sponsors, whose support and expertise has enabled us to continue producing this high-quality magazine for developers and architects, free of cost.

On this occasion, I am happy to inform our readers that the DotNetCurry (DNC) Magazine is read in over 100 countries, and reaches out to more than 140,000 readers. I am glad that our efforts have helped our readers skill, reskill and upskill themselves, and this very feeling brings a lot of unparalleled pride and joy for everyone associated with this magazine. As always, we celebrate this new milestone with a set of tutorials that are relevant for the present and particularly future. I hope you will enjoy reading them!

I strongly believe that passion drives progress, which ultimately leads to success. But what drives passion? For us, it's a desire to constantly help the developer community. If you have the expertise and share the same passion, shoot an email to me and join our team.

Once again, THANK YOU and a hearty congratulations to the entire DotNetCurry team on the 50th edition of the DNC Magazine, and here is to the next 50 and beyond.

Cheers and Happy Holidays in advance!

Copyright @A2Z Knowledge Visuals Pvt. Ltd.

Reproductions in whole or part prohibited except by written permission. Email requests to "suprotimagarwal@dotnetcurry.com". The information in this magazine has been reviewed for accuracy at the time of its publication, however the information is distributed without any warranty expressed or implied.

CONTENTS

Server-Side JavaScript for .NET Developers Part I - NODE.JS Fundamentals 06

Language Understanding with LUIS 22

Deploying BLAZOR WEBASSEMBLY Applications TO Azure STATIC WEBAPPS 44

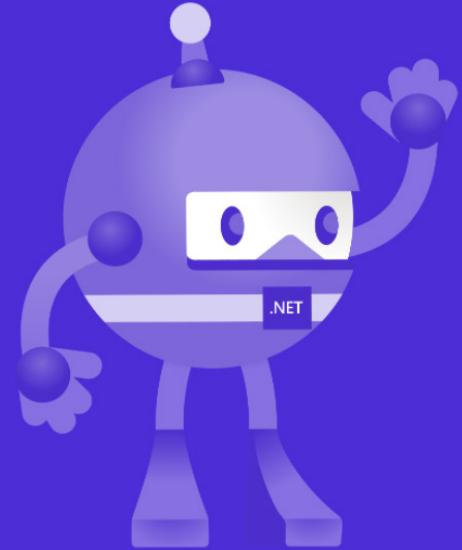
New features in ECMAScript 2021 56

Get the most out of Async/Await 64

Azure Pipelines CI / CD for developers 86

Creating and Consuming a C# Source Generator (.NET 5 and C# 9) 98

Server-Side JavaScript for .NET Developers Part II - Web Frameworks, Express and Fastify 108



.NET MAUI :
What to expect?

78



Enhance Your Application's Data Capture Performance With Accusoft's OCR Technology

Today's applications are expected to quickly and precisely extract text from a broad range of document types, image files, or forms.



Image clean-up for improved results



Full-page and zonal extraction



75+ Western and Eastern languages



Recognition of handwritten or machine text



Accusoft OCR Technology

Accusoft's OCR technologies are finely tuned to deliver fast, accurate text recognition results. For forms, Accusoft's OCR technology offers field recognition to quickly and accurately extract predefined form content like names, phone numbers, and SSNs for processing. For images, the OCR technology captures text from document or image files to create a searchable text-based PDF, plain text file, or XML.

[Learn More About How to Integrate Accusoft's OCR Technology Into Your Applications](#)

[Get Started](#)



@suprotimagarwal



@dani_djg



@subodhsohoni



@yacoubmassad



@damirrah



@darrengillis



@jfversluis



@VikramPendse



@saffronstroke



@gouri_sohoni



@sravi_kiran



THANK YOU SO MUCH FOR YOUR HARD WORK THROUGHOUT
ALL EDITIONS OF THE DNC MAGAZINE.

YOU SHOWED TREMENDOUS CHARACTER BY KEEPING COOL
AND HELPING US TO MAKE SURE EVERYTHING GOT DONE, EVEN
DURING BUSY AND DIFFICULT TIMES.

SUPROTIM AGARWAL, EDITOR IN CHIEF

Daniel Jimenez Garcia



SERVER-SIDE JAVASCRIPT FOR .NET DEVELOPERS **PART 1-NODE.JS FUNDAMENTALS**

JS

The web has been dominated by JavaScript since the last 10-15 years. In a race to help developers create increasingly complex applications, we have seen a number of frameworks rise, shine and decay.

Throughout this journey, JavaScript escaped the traditional browser boundaries. Using JavaScript, you can create web, native and desktop client applications, server-side applications and services, command line tools or even machine learning applications.

There are two technologies which played a fundamental role in its success.

One is [Node.js](#), a JavaScript runtime based on Chrome's V8 engine, which made it possible to leverage JavaScript outside of the browser. The other is [npm](#), the package manager and public registry that allowed developers to author and publish countless libraries and frameworks.

But technology is just one part of the story. An equally important role was played by the open-source community.

The community didn't limit itself to building useful frameworks and libraries. It advocated for JavaScript as a platform, showed others how it can be leveraged, and helped build, grow and nurture a user base.

And let's not forget about tech giants like Facebook, Google, Amazon or Microsoft, for the role they played in some of its more successful Open source projects, and thus contributing to the overall popularity of the platform.

The popularity of JavaScript probably isn't news to most .NET developers, particularly those building web applications. Many will have heard of or used tools such as npm or webpack, and frameworks such as React, Angular or Vue. However, there is a very rich JavaScript ecosystem outside of the browser. In fact, the design of ASP.NET Core and even .NET Core borrows many ideas from the strengths and developer experience that server-side JavaScript frameworks provided.

In this new series of articles, we will take a look at JavaScript as a compelling platform for server-side web applications and services. For the 50th Edition, we will start by covering some of the Node.js fundamentals for creating web servers. Then we will take a thorough look at two of the most popular web frameworks, Express and Fastify. In a forthcoming issue, we will look at two rising application frameworks, Next.js and NestJS, which go beyond strict web server features.

I hope these articles will give you a taste of what Node.js can do. Who knows, maybe you will begin to consider Node.js as another tool in your toolbelt!

You can find the examples discussed through the article on [GitHub](#).

For the most part I will leave TypeScript outside of the articles. If you are a TypeScript user, the information discussed in the article will be equally useful, and there is no shortage of online tutorials covering TypeScript and Node.js.

The traditional Hello world application

If you don't yet have Node.js installed, you can do so from the [official downloads page](#), either as a standalone binary, an installer or through a package manager. Alongside Node, you will also get the npm CLI installed, which is a fundamental part of the developer experience with Node. Verify both are

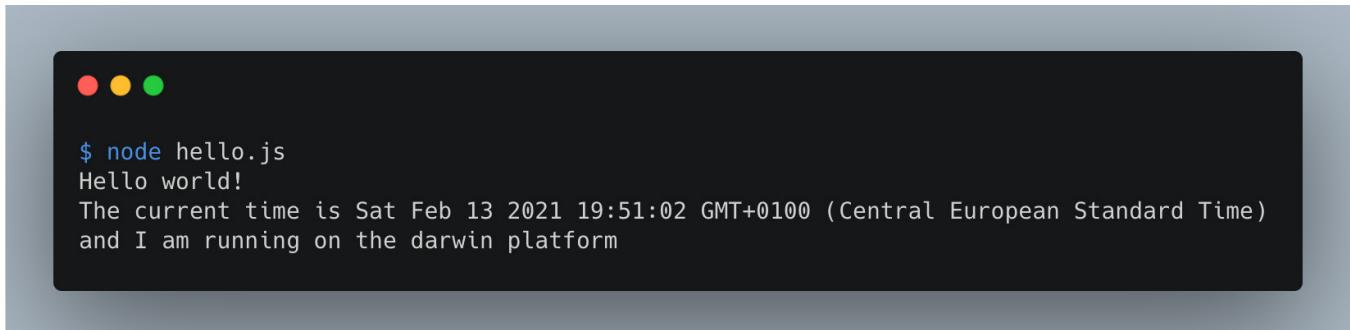
successfully installed by running:

```
node --version  
npm -version
```

To create your first project, you don't need any special files or structure. All you need is a JavaScript file! Create a new folder and create a new file named **hello.js** inside it. Add the following contents to the file:

```
const process = require('process');  
console.log('Hello world!');  
console.log(`The current time is ${ new Date() }`);  
console.log(`and I am running on the ${ process.platform } platform`);
```

You can run it using the `node hello.js` command as follows:

A screenshot of a terminal window on a Mac OS X system. The window has the characteristic red, yellow, and green close buttons at the top left. The terminal itself is black with white text. It shows the command '\$ node hello.js' being typed, followed by the output: 'Hello world!', 'The current time is Sat Feb 13 2021 19:51:02 GMT+0100 (Central European Standard Time)', and 'and I am running on the darwin platform'.

```
$ node hello.js  
Hello world!  
The current time is Sat Feb 13 2021 19:51:02 GMT+0100 (Central European Standard Time)  
and I am running on the darwin platform
```

Figure 1, running a hello world Node.js application

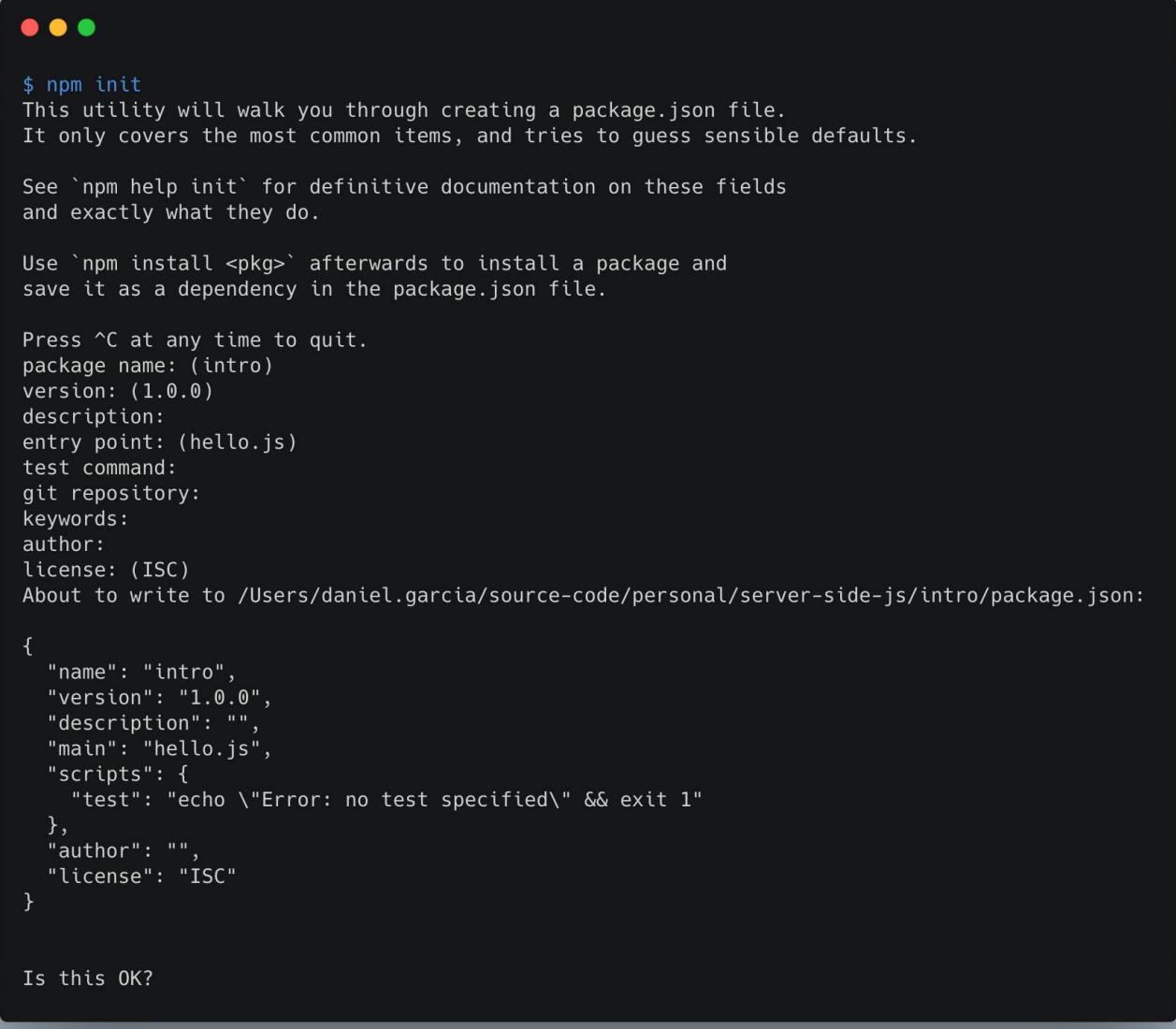
There doesn't seem to be anything remarkable about this. A standard hello world exercise, using some string interpolation and Node's standard API `process` in order to print the current date and platform. However, note the experience was straightforward and developer friendly. Write some code in a file, save and run!

Your first *project*

Normally you don't structure your projects as standalone files that are manually run. Some structure is desirable, so you can organize your source code, keep track of dependencies or define commands to run/debug your project.

To do so, Node projects typically leverage `npm`. The easiest way to create a *project* is to run the `npm init` command, which will run you through several questions about the project and create a `package.json` file.

Let's do so in the same folder where we created the `hello.js` file, accepting all the default values:



```
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (intro)
version: (1.0.0)
description:
entry point: (hello.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /Users/daniel.garcia/source-code/personal/server-side-js/intro/package.json:

{
  "name": "intro",
  "version": "1.0.0",
  "description": "",
  "main": "hello.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}

Is this OK?
```

Figure 2, initializing an empty project using npm init

By adding a `package.json` file that npm understands, we have turned our single file into a project. For example, we can now use `npm start` to run it, since the `main` file (or entrypoint) is defined as the `hello.js` file.

Note the convention is to name the main file index.js. You can rename it if you so desire, just remember to also update it in package.json.

The combination of npm with the `package.json` file will also let us manage and keep track of the dependencies in our project. For example, let's add `nodemon`, a tool able to watch our source code files and automatically restart our project, something very useful during development:

```
npm install --save-dev nodemon
```

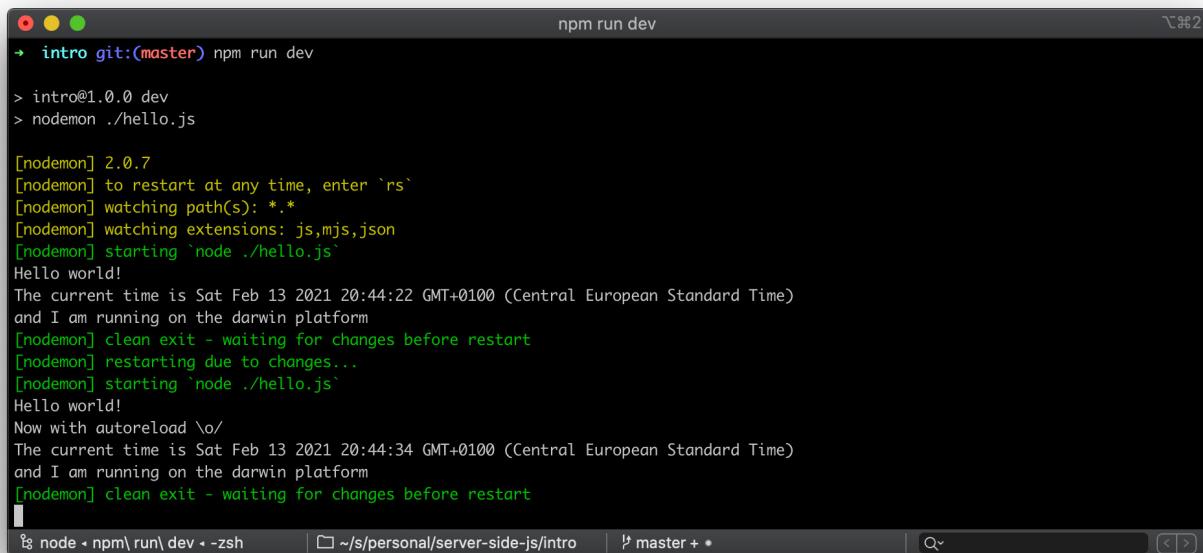
With npm, we can distinguish between *dependencies* and *devDependencies*. The former are the ones your application needs to run, while the latter are the ones only needed during development. We have added `nodemon` as a `devDependency`.

Now let's add a new script to our `package.json`. This will use `nodemon` in order to run the project in

development mode with hot reload:

```
"scripts": {  
  "dev": "nodemon ./hello.js",  
  "test": "echo \"Error: no test specified\" && exit 1"  
},
```

In your terminal run the new command `npm run dev`. You will notice nodemon runs the script and waits for file changes. Make a change, like adding a new `console.log` line and note how nodemon automatically re-runs it.



```
npm run dev  
+ intro git:(master) npm run dev  
> intro@1.0.0 dev  
> nodemon ./hello.js  
  
[nodemon] 2.0.7  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.  
[nodemon] watching extensions: js,mjs,json  
[nodemon] starting `node ./hello.js`  
Hello world!  
The current time is Sat Feb 13 2021 20:44:22 GMT+0100 (Central European Standard Time)  
and I am running on the darwin platform  
[nodemon] clean exit - waiting for changes before restart  
[nodemon] restarting due to changes...  
[nodemon] starting `node ./hello.js`  
Hello world!  
Now with autoreload \o/  
The current time is Sat Feb 13 2021 20:44:34 GMT+0100 (Central European Standard Time)  
and I am running on the darwin platform  
[nodemon] clean exit - waiting for changes before restart
```

Figure 3, running our project with automatic reload on file changes

Overall, this workflow might feel familiar to those with experience using .NET Core and its CLI tool. A similar article *.NET for Node.js developers* could be written using commands such as `dotnet new`, `dotnet run`, `dotnet watch` or `dotnet add package`.

The .NET Core tooling borrowed many successful ideas from platforms like Node.js, and for good reason!

A web server using the http module

One of the many reasons why Node.js succeeded was its ability to create a self-hosted web server, thanks to its built-in [http API](#). So much so, that it became one of the most [typical examples](#) of a Node app.

Let's turn our current console application into an HTTP server. Following the example in the official Node docs, update the contents of the `hello.js` file with the following:

```
const http = require('http');  
const process = require('process');  
  
const hostname = '127.0.0.1';  
const port = 3000;  
  
const server = http.createServer((req, res) => {  
  res.statusCode = 200;  
  res.setHeader('Content-Type', 'text/plain');  
  res.end(`Hello world!`)
```

```

The current time is ${ new Date() }
and I am running on the ${ process.platform } platform
`);
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});

```

If you had the `npm run dev` command still running, you will notice the message `Server running at http://127.0.0.1:3000/` as soon as you save. Otherwise run `npm run dev` again. Then open `localhost:3000` in your browser:

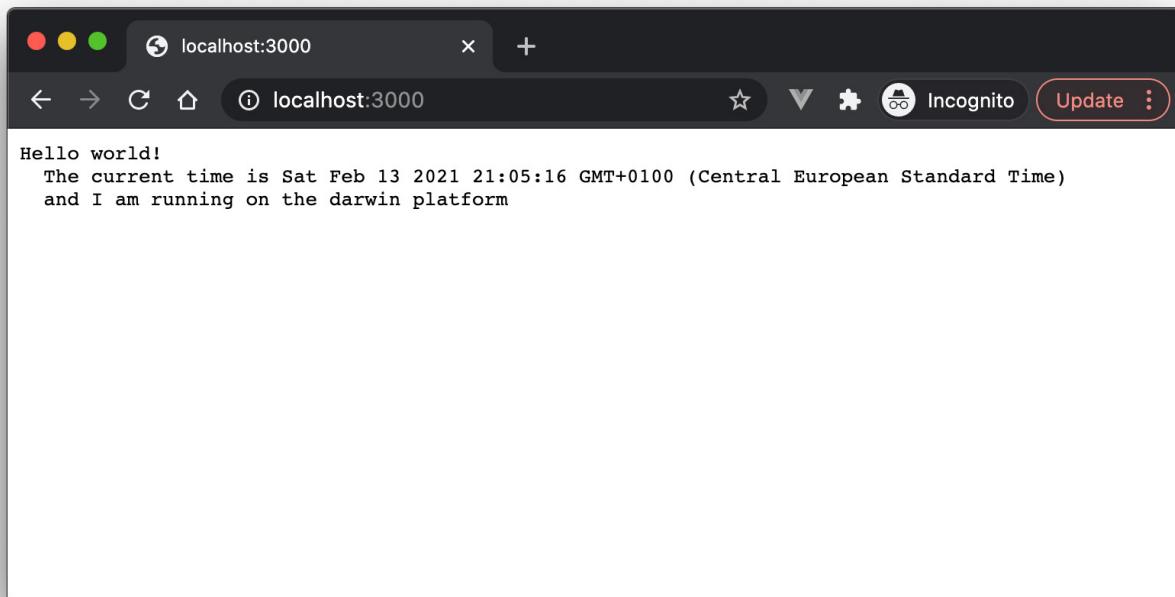


Figure 4, your first web server

Make a change to the string sent in the response and save the file, then reload the browser tab. Note how nodemon automatically restarted the server!

You will normally want nodemon to ignore test files so it does not reload because tests have changed. Update the dev command as in:

```
"dev": "nodemon --ignore 'test/**/*' ./hello.js",
```

Of course, this is a barebones HTTP server without the features you might be used to, such as routing or authentication.

But it shows how easy it is to create one!

Worry not, in the follow-up articles of the series we will take a look at several frameworks like Express or Fastify which expand on the basics.

Testing a web application

If you have worked with frameworks like React, Angular or Vue, you might have used test frameworks such as [mocha](#) or [Jest](#) to create tests. These same frameworks can be used to test web applications created with Node.

In this section, we will see some examples with [Jest](#), one of the most popular test frameworks in the JavaScript ecosystem. Note that [mocha](#) is an equally valid option, everything we will see can be also achieved with mocha (and many avoid Jest due to its interference with Node's internals).

Although our current application is simple, the testing basics we will see here apply to more complex frameworks and applications.

If you run into trouble or just want to follow along by browsing code, check the “intro” project in [GitHub](#).

Make the application testable

Before we can write the tests, let's refactor the code a bit, so it's easier to test. Let's move the request handling code to its own file so we can easily create a unit test. Create a new file **server-api.js** like:

```
const process = require('process');
module.exports = (req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end(`Hello world!
The current time is ${ new Date() }
and I am running on the ${ process.platform } platform
`);
};
```

Then we will move the code that creates the HTTP server to its own file, without listening on any port. This will let an integration test to automatically create/teardown a server on each test run. Create a new file **server.js** like:

```
const http = require('http');
const handler = require('./server-api');
module.exports = http.createServer(handler);
```

Finally, rename the original **hello.js** file as **index.js** (updating the references in **package.json**, namely the **main** file and the **dev** script) and update its contents as:

```
const server = require('./server');
const hostname = '127.0.0.1';
const port = 3000;
server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Unit testing example with Jest

As in the earlier sections, begin by installing the necessary dependencies. In this case, we will install [Jest](#):

```
npm install --save-dev jest
```

Then create a new **/test/unit** folder. Inside, create a new file **server-api.spec.js** with the following contents. We will import the server-api, mock the process object and the request/response objects, so we can test the function works as intended:

```
const { beforeEach } = require("@jest/globals");
const serverApi = require('../server-api');

jest.mock('process', () => ({
  platform: 'mockPlatform'
}));

describe('the server-api', () => {
  let mockReq;
  let mockRes;
  beforeEach(() => {
    mockReq = {};
    mockRes = {
      setHeader: jest.fn(),
      end: jest.fn()
    };
  });
  test('returns 200 status code', () => {
    serverApi(mockReq, mockRes);
    expect(mockRes.statusCode).toBe(200);
  });
  test('adds text/plain as content header', () => {
    serverApi(mockReq, mockRes);
    expect(mockRes.setHeader).toBeCalledWith('Content-Type', 'text/plain');
  });
  test('sends a hello world message', () => {
    serverApi(mockReq, mockRes);
    expect(mockRes.end.mock.calls[0][0]).toMatch(/Hello world!\s+The current time is .*\s+and I am running on the mockPlatform platform/);
  });
});
```

Next update the **test** script inside package.json. We will replace it with a command that runs Jest and executes all the test files found inside the test/unit folder:

```
"test": "jest \"test\\unit\\.*\\\\.spec\\\\.js\""
```

Finally run the tests with the command npm test as in:

```

→ intro git:(master) ✘ npm test

> intro@1.0.0 test
> jest "test/unit/.*\spec\.js"

PASS test/unit/server-api.spec.js
the server-api
  ✓ returns 200 status code (2 ms)
  ✓ adds text/plain as content header (1 ms)
  ✓ sends a hello world message

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        0.858 s, estimated 1 s
Ran all test suites matching /test\unit\.*\spec\.js/i.
→ intro git:(master) ✘

```

Figure 5, running unit tests with Jest

This example serves to showcase how you can test a module using Jest, and how to leverage its assertion and mocking features. Overall, this isn't much different from writing unit tests in .NET Core.

Note this is just meant as an example! Unit tests and the extra refactoring required are overkill for such a simple application. The integration test we will see next would be enough for such an application.

Integration testing with Jest and supertest

Integration tests are one of the most useful types of tests when creating web applications and services. Being able to easily setup these tests and for them to run quickly, is a huge advantage for developers.

Let's see how we can setup an integration test with Jest. We will leverage the library `supertest` in order to send real HTTP requests to our application. It will also let us automatically start/teardown our application in conjunction with Jest global's hooks. Install the library with

```
npm install --save-dev supertest
```

Then create a new folder `/test/integration` and create a new file `server.spec.js`. Let's build it step by step. Begin by adding the necessary imports and a describe block that will wrap the tests of that file:

```
const { beforeAll, afterAll } = require("@jest/globals");
const supertest = require('supertest');
const server = require('../..../server');
describe('the server', () => {
});
```

Next add Jest's `beforeAll/afterAll` hooks inside the describe block which will start and teardown our server. You can now see the benefit of separating the `server.js` from the `index.js`. Each test file can start its own HTTP server instance used by those file tests.

```

describe('the server', () => {
  let request;
  beforeAll(() => {
    server.listen(0); // start server on any available port
    request = supertest(server);
  });
  afterAll(done => {
    server.close(done); // stop the server
  });
});

```

Now we can use the `request` instance to send any HTTP requests to our server using the supertest API. Let's add a test that verifies that our one and only endpoint returns the expected hello world plain text message:

```

test('GET / returns a helloworld plaintext', async () => {
  const res = await request
    .get('/')
    .expect('Content-Type', 'text/plain')
    .expect(200);

  expect(res.text).toMatch(/Hello world!\s+The current time is .*\s+and I am
running on the .* platform/);
});

```

You could use other libraries to make HTTP requests in your test instead of supertest! The trick is to make sure that you adjust the setup/teardown to start and close your server.

You can also decide to start/close a server globally for the entire test run, as opposed to for each test file. In that case, use Jest's `globalSetup` and `globalTeardown` config options.

To run the tests, we need to add a new script to our `package.json` file, like the one added before in order to run unit tests. Since it might come handy to run either unit or integration tests, let's define separate scripts for unit/integration tests, as well as a single `test` script that combines them both:

```

"scripts": {
  "dev": "nodemon --ignore 'test/**/*' ./index.js",
  "test": "npm run test:unit && npm run test:integration",
  "test:unit": "jest \"test/unit/.*\\".spec\\\".js\\\"",
  "test:integration": "jest \"test/integration/.*\\".spec\\\".js\\\""
},

```

After adding the commands, you can run all the project tests with `npm run test`. Alternatively run either `npm run test:unit`, or `npm run unit:integration` to run a specific set of tests.

In case you missed it before, note how the `dev` command includes the `--ignore` option so nodemon ignores changes to test files.

```
..side-js/intro (-zsh)
→ intro git:(master) ✘ npm test

> intro@1.0.0 test
> npm run test:unit && npm run test:integration

> intro@1.0.0 test:unit
> jest "test/unit/*\spec\js"

PASS  test/unit/server-api.spec.js
the server-api
  ✓ returns 200 status code (3 ms)
  ✓ adds text/plain as content header (1 ms)
  ✓ sends a hello world message

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        0.932 s, estimated 1 s
Ran all test suites matching /test\unit\*.spec\js/i.

> intro@1.0.0 test:integration
> jest "test/integration/*\spec\js"

PASS  test/integration/server.spec.js
the server
  ✓ GET / returns a helloworld plaintext (17 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.004 s
Ran all test suites matching /test\integration\*.spec\js/i.
→ intro git:(master) ✘
```

Figure 6, running unit and integration tests with Jest

As you can imagine, this barely scratches the surface of testing in Node. However, the process of testing applications will follow these basic techniques!

Importing modules, CommonJS or ES modules

Traditionally Node.js has used **CommonJS** in order to import/export modules. The code we have seen so far uses CommonJS, exporting using `module.export` and importing using the `require` function. For example:

```
// in server.js
const http = require('http');
const handler = require('./server-api');
module.exports = http.createServer(handler);
```

```
// in index.js
const server = require('./server');
```

However, since Node.js started back in 2009, the ES6 standard was developed and with it came standard module definition and a new syntax for importing/exporting modules, known as **ES modules**. You export using the `exports` keyword and import using the `import` from syntax.

```
// in server.js
import http from 'http';
import handler from './server-api.js';
export default http.createServer(handler);
// in index.js
import server from './server.js'
```

This has quickly become the standard for client-side JavaScript code, and most examples, tutorials, articles you will see related with React, Vue and many others will use this syntax. However, Node.js still defaults to CommonJS and most of the articles, tutorials and examples you will see out there still use the CommonJS syntax.

Good news is that you can use ES modules in Node.js:

- Since v12, you can use the `.mjs` extension for modules that use the ES module syntax and start the node process adding the `--experimental-modules` flag.
- Since v13, you no longer need to add the `--experimental-modules` flag, any file with `.mjs` extension will be considered to use ES modules.

You can also add the property `"type": "module"` to your `package.json`, and then it will consider every `.js` file to be using ES6 modules, except for those with the `.cjs` extension.

Therefore if you use Node.js 13+, you can just add `"type": "module"` to your `package.json` and use the ES modules syntax across your entire project.

The real question is whether this makes a difference for developers or not?

The truth is that these are not just different syntax for the exact same behavior. While in many cases the behavior is equivalent, these modules systems have completely different implementations, which for the user, manifest in subtle differences and caveats. These are particularly important for library authors publishing modules to NPM:

- CommonJS allow for special import use cases like JSON files, implicit `index.js` file, implicit file extension, which are not supported with ES modules (or require a special flag). See [official docs](#).
- The process of parsing and evaluating an imported module is different between both module systems. CommonJS `require()` statements are dynamic (i.e. it's a function that you invoke from anywhere in your code), and evaluated synchronously. On the other hand, ES `import` statements are defined statically at the top of the file but loaded asynchronously! For a deeper comparison, see [this article](#) for example.
- A consequence is that traditional Node.js mocking libraries such as `proxyquire` do not support ES modules, even Jest's support is [limited](#). At least new libraries such as `rewiremock` are appearing to fill the gap.

- Another consequence of the different evaluation behavior is that `interoperating` isn't seamless both ways. An ES module can import a CommonJS module but not the other way around, unless you use the asynchronous `import()` function rather than the synchronous `require()` function. i.e., the CommonJS consumer would need to know that it is importing an ES6 module and use a promise or `async/await`!

This is an important point to consider for authors of libraries consumed by other developers (such as Jest or supertest). For them, there are several techniques to publish hybrid/dual modules, see [this article](#) for example. The major caveat is that a single dependency that switches to ES modules only can force you to migrate your project to ES modules.

Don't worry if all of this sounds complicated. The important part is to be aware that there are two different module implementations in JavaScript, where ES modules are dominant in client-side JavaScript, and CommonJS is still dominant in Node.js.

As an *end user*, you can continue that way, or switch your Node.js application to use ES modules too.

It is likely the ecosystem of JavaScript libraries continues to support both module systems for quite some time. Perhaps major libraries dropping support for CommonJS will be the force that decisively moves the ecosystem towards ES modules.

Asynchronous code in JavaScript

JavaScript is a single threaded language. This means for any web server written in JavaScript to be performant, it needs to avoid blocking code.

For example, you need to avoid the single thread of your application to be blocked waiting for a database operation. Instead, you want to switch context and start/resume other requests while the database operation completes.

One of the reasons Node.js was successful is that the JavaScript concurrency model based on the event loop is naturally suited to IO workloads such as web servers, databases or services. Since Node.js started, several patterns emerged over the years to let developers make the most out of this model with the minimum effort!

In the beginning there were callbacks

The initial pattern that dominated the early Node.js years (and client-side JavaScript) was the usage of callbacks. Many of the core APIs in Node.js still follow this model!

For example, let's simulate a function that reads from a database. Our function will accept a callback, which will be invoked once the operation is finished with either a successful result or an error object:

```
const loadDataFromDb = function(done) {
  const error = null;
  setTimeout(
    () => done(error, {foo: 42, bar: 'baz'}),
    1000);
};
```

Any consumers of a function with a callback-style API will need to provide a callback function, a style that resulted in nested functions:

```

const requestHandler = function (req, res) {
  loadDataFromDb((err, data) => {
    if (err) {
      res.statusCode = 500;
      return res.end();
    }
    res.end(JSON.stringify(data));
  });
};

```

This style could degrade quickly into a callback hell once multiple asynchronous functions had to be combined:

```

const requestHandler = function (req, res) {
  loadDataFromDb((err, data) => {
    if (err) {
      res.statusCode = 500;
      return res.end();
    }
    saveDataToDb((err, data) => {
      if (err) {
        res.statusCode = 500;
      }
      res.statusCode = 201;
      res.end();
    });
  });
};

```

Then came the promises

The next step in the asynchronous code evolution were Promises, which first took off in client-side JavaScript (remember jQuery?) and later made its way to Node.js.

Now asynchronous functions can return a Promise object, which consumers can wait upon:

```

const loadDataFromDb = () => {
  return new Promise((resolve, reject) =>
    setTimeout(() =>
      resolve({foo: 42, bar: 'baz'}),
      1000));
};

```

Consuming a Promise means attaching a continuation callback with either `.then` or `.catch`, depending on whether you were interested in a successful or failed execution:

```

const requestHandler = function (req, res) {
  return loadDataFromDb()
    .then(data => res.end(JSON.stringify(data)))
    .catch(err => {
      res.statusCode = 500;
      res.end();
    });
};

```

Since Promises are *chainable*, it is easy to combine multiple functions with a Promise API:

```
const requestHandler = function (req, res) {
  return loadDataFromDb()
    .then(data => makeChangeToData(data))
    .then(changedData => saveDataToDb(changedData))
    .then(() => res.end())
    .catch(err => {
      res.statusCode = 500;
      res.end();
    });
};
```

However, with Promises, it is still possible to shoot yourself in the foot. One way is to forget to add a `.catch` in the chain of handlers, which could end in an `UnhandledPromiseRejectionWarning` error bubbling through the entire application. Another typical pain point was sharing data between handlers in large promise chains.

Today we have async/await

The most recent pattern will be familiar to .NET developers since it gained popularity when first introduced in C# 5, then was adopted by TypeScript, and finally made its way into JavaScript.

Editorial Note: This edition covers two tutorials on Async and Await in C#, and how to make the most of it. Make sure to give it a read.

This solves many of the problems the previous async styles had, while remaining compatible with the existing ecosystem of libraries and functions. In a grossly simplified way, async/await is a nicer way to deal with Promises which solves most of its shortcomings. For a better overview check the [MDN](#).

We can await any function that returns a Promise, so we can write the last request handler from the previous example as:

```
const requestHandler = function (req, res) {
  try{
    const data = await loadDataFromDb();
    data = makeChangeToData(data);
    await saveDataToDb(changedData);
    res.end();
  } catch() {
    res.statusCode = 500;
    res.end();
  }
};
```

This results in code which many find more readable and natural than the one with Promises, even if under the hood we are using Promises.

However, that's not to say that async/await doesn't have its own new set of gotchas!

If you google a bit, you will find articles such as [this one](#) and will learn about scenarios and use cases you should be aware of. In the end, technology keeps moving forward and tries to find newer and hopefully

better solutions to the same problems.

Conclusion

Node.js can be a very efficient tool for building web servers. With its built-in but powerful http module, it can be easy to create a self-hosted server in a single JavaScript file. And modern JavaScript, the one supported by current versions of Node.js, can be quite a productive development environment.

You can leverage an expressive language with ES modules, async/await, Promises and more, together with its dynamic nature that reduces boilerplate and makes testing a pleasant experience.

I am fully aware the language has a bad reputation among many developers, and not everyone likes its dynamic and sometimes chaotic nature. However, I feel like many of its shortcomings come from the development experience when creating client-side JavaScript and all its tooling baggage. The good news is that you can avoid them with Node.js and just concentrate on your application.

If you are interested in Node.js, take your time to understand these fundamentals, as they would help you no matter what frameworks you later decide to use. Because let's be honest, even though the http module is great, most teams and developers will not want to build from first principles features such as routing, error handling, request body parsing, etc.

That's why in another article in this edition, we will move on from the fundamentals and explore two different web frameworks, Express and Fastify. Keep reading!

• • • • •

Daniel Jimenez Garcia

Author

Programmer, writer, mentor, architect, problem solver and knowledge sharer. In his 15+ years of experience, Daniel has lead teams building scalable systems, driven transformational changes to increase teams and developers' productivity, delivered working quality software, worked across varied industries and faced many challenges.



His DNC articles have garnered over 2 million reads. He has also published libraries, contributed to open-source projects, created many sample projects, reviewed books, answered many stack overflow questions and spoken at events.

He remains interested in many areas and technologies such as: Cloud architecture, DevSecOps, containers, Kubernetes, Node.js, Python, Vue, .NET Core, Go, Rust, Terraform, API and framework design, code quality, developer productivity and automated testing.

Twitter: https://twitter.com/Dani_djg

Github: <https://github.com/DaniJG>

Stack Overflow: <https://stackoverflow.com/users/1836935/daniel-j-g>

Personal site: <https://danijg.github.io/>



Technical Review

Damir Arh



Editorial Review

Suprotim Agarwal



Darren Gillis

LANGUAGE UNDERSTANDING WITH LUIS

As artificial intelligence (AI) and machine learning (ML) proliferate within modern software applications, the need for these advanced skillsets become more difficult to retain. However, there is a growing shift to democratize AI/ML with cloud providers offering an ever-increasing menu of managed services to satisfy access to AI capabilities such as speech recognition, image classification, and computer vision including face and object detection.

Included with Microsoft Azure's AI/ML offerings is a service called **Language Understanding or LUIS**.

In this article, I will give an overview of LUIS and demonstrate some of the features using a real-world example of natural language understanding in action.

What is LUIS?

Under the umbrella of Azure Cognitive Services, the Language Understanding Intelligent Service (LUIS) offers a simplified method of including natural language understanding into your applications.

LUIS can be used directly by your applications or in combination with other Azure Cognitive Services that are directly related to Language and Speech processing and analysis including Azure Bot Service, QnA Maker, Text Analytics, Bing Spell Check, and Azure Search – to name just a few.

In addition, there are many cases for using a combination of LUIS with the Microsoft Bot Framework for strengthening the language understanding and intent for custom Bot applications.

Role within Azure

Azure offers a grouping of services within the cognitive services space that includes Decision, Language, Speech, and Vision. LUIS is included in the cognitive services

offering under the Language category along with QnA Maker, Text Analytics, and Translator.

Benefits

The largest benefit you will receive by successfully implementing LUIS is that the alternative would require hiring a team of highly educated and experienced artificial intelligence and machine learning experts within the domain of natural language processing and understanding. These skills are in high-demand and smaller companies simply cannot afford to retain the necessary talent.

Before we dive into LUIS and the associated concepts, let's quickly consider how one might approach a language processing application without the services of experienced AI and ML resources for processing text-based content and determining a next course of action.

For example, imagine you had to create a software application that could act on the simple concept of someone saying "Goodbye". An easy solution would be to process the incoming text-based content and parse for the word "Goodbye" – if found, then do the next step.

Now consider someone saying "Goodbye", "See you later", or "I'm out of here". As a human, you would immediately be able to gather that the person is leaving (their intended action). However, processing these sentence structures to allow for a computer to immediately understand the intent and therefore, the necessary action, presents more of a challenge.

Would you really want to program the control flow of your application to look for a static string such as "See you later" to determine a leaving action and then decide based on that direct match?

You could, but then you would have to consider all the various iterations of what a leaving intent could include. Doing so would require many potential control-flow statements in your application based on static strings and would quickly become unmanageable. For this example, let LUIS intervene, as it can help by processing many similar forms of a word or grouping of words and distil them down to a singular intent or actionable decision.

Luis - Getting Started

In this section, we will briefly describe what natural language is, give a brief description of the application we will be building with LUIS, and highlight the core concepts that are important to grasp before using LUIS effectively.

Natural Language

The "L" in the LUIS acronym is for "Language". This might be clear enough immediately, but what "language" is it referring to?

Language can be viewed as naturally occurring or constructed. Constructed languages could be a software language (i.e., C#, JavaScript) – built for purpose and derived for instructing computation. Naturally occurring languages are those that have evolved over time and have changed organically based on generally accepted terms and conventions over the course of the language's lifespan. These languages that we are familiar with include English, Spanish, German, etc.

The language “type” that we (and LUIS) are focused on in this article is the natural language. LUIS supports several, but we must teach LUIS to understand how to interpret the language’s conventions.

The Application

Before we dive into the core concepts of LUIS, let’s talk about the application we are looking to support.

Let’s imagine that we currently work for a cable TV company, and we want to offer a way for our customers to send in questions or commands regarding our programming using either a direct SMS message from their mobile phone or by interacting with a Bot we will be setting up on our website. We won’t cover the client applications portion here such as the creation of the Bot or ingesting the SMS message, that will be left for another team to look after. However, after those applications are built and the text is extracted from the SMS message or the Bot application, it will be coming to our LUIS application for processing.

We are going to work with LUIS to build out the processing mechanism for the incoming text phrase. Keep in mind that we need to teach LUIS what it can expect from a customer’s language and how it should determine what the customer is requesting. With our help, LUIS will receive the proper training for success.

LUIS Core Concepts

Intents

What does the user want to do? Consider when a customer asks if he or she can find out when a favorite TV show is scheduled. They might task “when is Law & Order on tonight?” or “is Law & Order on tonight”. Intuitively, a human would know what the customer’s intent is based on the phrase used. In this case, look up the times for the TV show for tonight’s broadcast and return the answer to the customer. The human knows that the customer wants specific information based on the phrase and implied intent – the customer is wanting to know scheduled times for when a TV show airs.

Utterances

An utterance refers to the natural language phrase. As with our example above, “when is Law & Order on tonight?”. It is these utterances that LUIS will ingest, and it is up to you, as the LUIS application developer, to ensure that LUIS can understand the intent and deliver the result.

Entities

Continuing with our example, the entity would be the TV show. Again, this is easy for a human to recognize and understand, but it must be taught to LUIS. As we will see, there are four main types of entities, Machine learned, List, Regex, and Pattern.Any. The Machine learned entities can also be made up of composites, where a larger entity can be made up of child entities. This would be analogous to an object with several properties.

Models

“Models” or “Language Models”, used interchangeably, are made up of intents and entities. LUIS is continually trying to determine the intent based on a given utterance and then filter out the entities that the intent includes. Going back to our example, based on the utterance from the customer, you quickly

determine he or she wants to know a list of times when a TV show is broadcasting. Asking for the time is the intent and the TV show is the entity.

Patterns

Patterns are meant to balance the number of trained utterances across all intents. The patterns can be used to help strengthen your intents. To use a concrete example, if you have 5 utterances against one intent, and 105 against another, there is an imbalance that a pattern may be able to alleviate.

The “None” Intent

Every application created in LUIS contains a default intent called the “None” intent. This intent is important and there is no option to delete or rename it. The “None” intent is basically just that – there is no intent derived from the text phrase that is processed. It is used as a fallback intent like the “default” option in a case statement – there must be an action that can be taken if no intent is derived. From an application perspective, the user may need to provide additional or clearer details when the intent is unknown.

Active Learning

As you will see, the concept of active learning takes place as LUIS singles out utterances (text-based phrases that require validation). If the confidence or prediction score is low and the intent can't be determined, these phrases get flagged and sent to the “review” collection where you, as the administrator of the application, can provide additional direction for the next time this phrase (or fragment of the phrase) is ingested for processing.

Set it and Forget it

Remove the notion of “set it and forget it” before diving in. LUIS is not a “set it and forget it” solution. Working with LUIS requires an on-going commitment. We will see this in practice below, but keep in mind that LUIS needs to be taught continually to be effective. The great part is that the effectiveness will increase over time providing it is continually learning based on your commitment to teach it.

How Does LUIS Handle and Process Data?

Typical use cases for utilizing LUIS include conversational and human assistant bots in addition to command-and-control applications (application performs an action based on intent). It is important to note that your application data is sent to LUIS for processing only but that utterances can be logged in LUIS, if enabled. It is recommended that personally identifying or other sensitive data is limited to a flow-through for LUIS (and not logged explicitly).

In any case, this is the responsibility of the application developer but keep in mind that utterances containing sensitive data could be persisted outside of your application's data store and this scenario should be considered when applying your use case to LUIS.

We won't go into details in this article, but optional security measures can be taken where your logged data is stored in containers for you to manage directly.

Taking LUIS for a Ride

It's now time for an example to demonstrate how the LUIS services are created within Azure and how features of LUIS are used to apply the core concepts against a new language model that can be used by your applications.

Creating Resources

Creating a LUIS resource is like creating any other Azure service. A simple search for "luis" in the Azure marketplace will find the Language Understanding service.

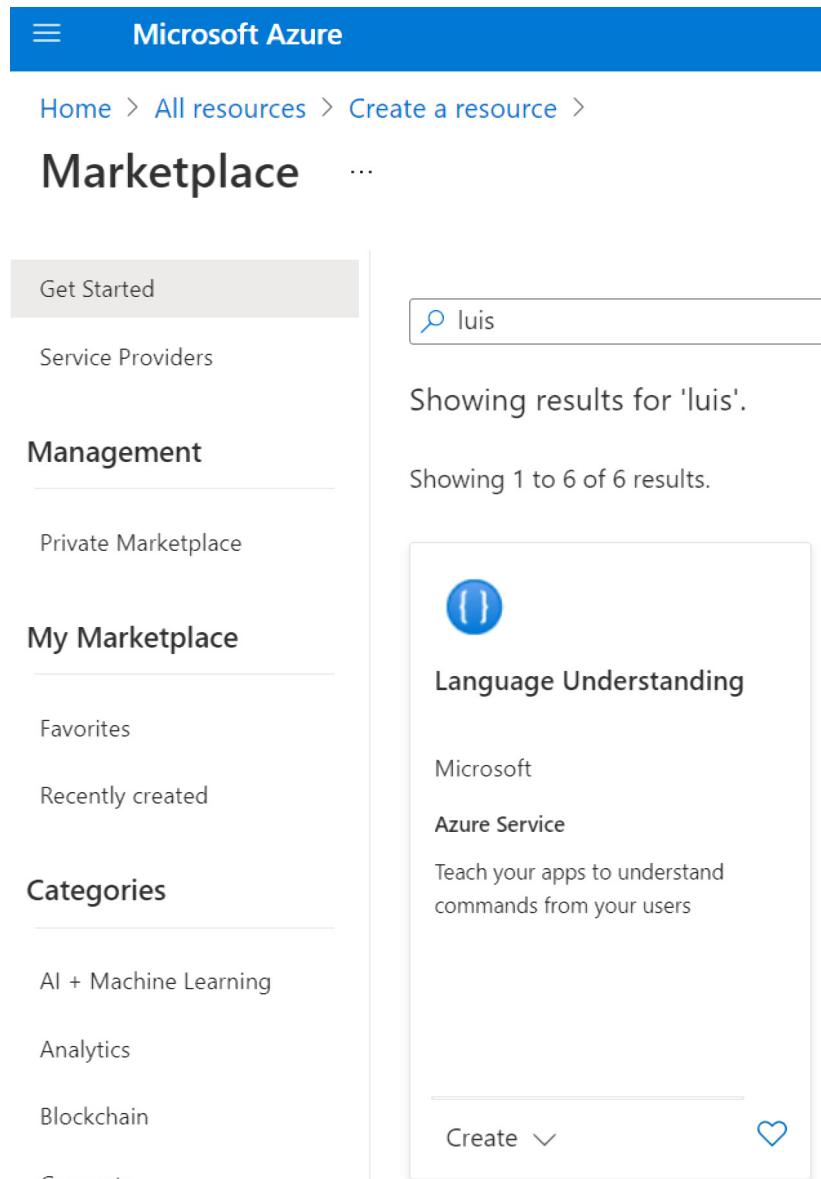


Figure 1: Searching for the LUIS Service

Clicking "Create" will bring you to the next screen where you can pick the appropriate options based on your existing Azure setup.

Create ...

Luis all in one

Basics Tags Review + create

Language understanding (LUIS) is a natural language processing service that enables you to build your own custom model to understand human language programmatically or through the UI in the LUIS portal. After you are satisfied with your LUIS model, you publish it and query its prediction endpoint through your client application for an end to end conversational flow. To build, manage, train, test and publish your LUIS Model, you will need to create the below Authoring Resource. This also gives you 1,000 requests/month endpoint requests. If you want your client app to request beyond the 1,000 requests provided by the authoring, create the below Prediction Resource. If you know from the start you will be needing more than 1000 prediction requests as well as the authoring experience, create using the "Both" option. This will create two resources, one for each type. [Learn more ↗](#)

Create options ⓘ

Both Authoring Prediction

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ

Playground ▾

Resource group * ⓘ

(New) default ▾

[Create new](#)

Name * ⓘ

dnc-tv

Authoring Resource

Select pricing and location for Authoring Resource

Authoring location * ⓘ

(US) West US ▾

Authoring pricing tier ([Learn More](#)) * ⓘ

Free F0 (5 Calls per second, 1M Calls per month) ▾

Prediction Resource

Select pricing and location for Prediction Resource

Prediction location * ⓘ

(US) East US ▾

Prediction pricing tier ([Learn More](#)) * ⓘ

Free F0 (5 Calls per second, 10K Calls per month) ▾

[Review + create](#)

[Next : Tags >](#)

Figure 2: Creating the LUIS Services

This is familiar territory when creating an Azure service, however there are a few minor differences. In the create options, there is a choice for creating an Authoring service, a Prediction service, or both. We will create both, ensuring that the appropriate resource group is either created or selected, the resource is adequately named, and the region and pricing are selected for each. Selecting the Free pricing tier is acceptable for testing the services and can also provide enough runway for an application with limited scale. We will review pricing options later in the article.

To reason about how the two services differ, consider the authoring service as the place you go to build and manage your LUIS models and the prediction service is where you go to send requests for the model to process.

Clicking “Review + Create” will create the following two resources:

Name ↑↓	Type ↑↓	Resource group ↑↓	Location ↑↓
<input type="checkbox"/> dnc-tv	Language understanding	default	East US
<input type="checkbox"/> dnc-tv-Authoring	Language understanding	default	West US

Figure 3: Newly Created LUIS Services

Azure has taken the name given in the creation options screen, “dnc-tv” and used it for both resources while appending the “-Authoring” convention for the authoring service. The prediction service used the original name as the convention.

LUIS Portal

At this point, we will leave the Azure portal and explore the LUIS.ai portal where we can connect back to the created resources and work with them directly using the features of LUIS through the portal interface. The portal address is <https://luis.ai/>.

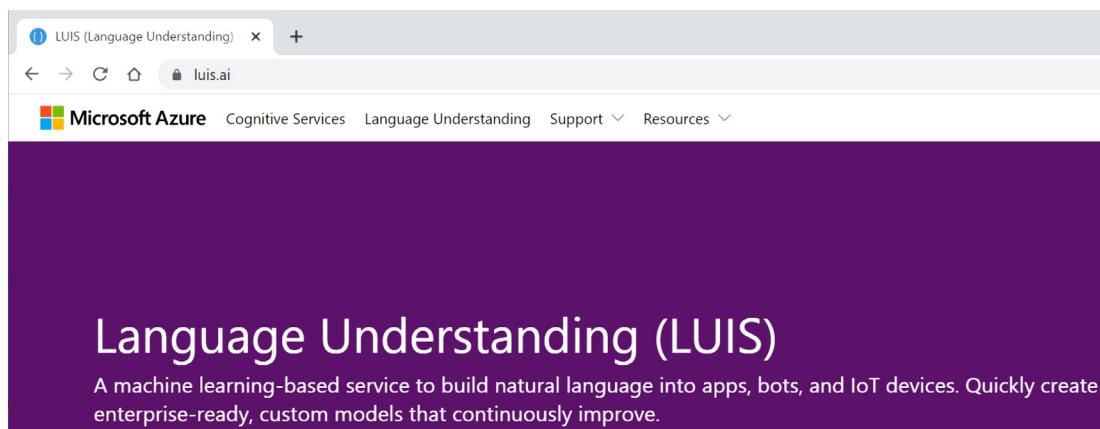


Figure 4: LUIS Portal

When signing in, you will use the same domain account to sign into the LUIS portal that you would for the Azure portal. Once signed in, you will be presented with the authoring tools that we will explore next after ensuring that you are setup with the correct subscription and Azure connection.

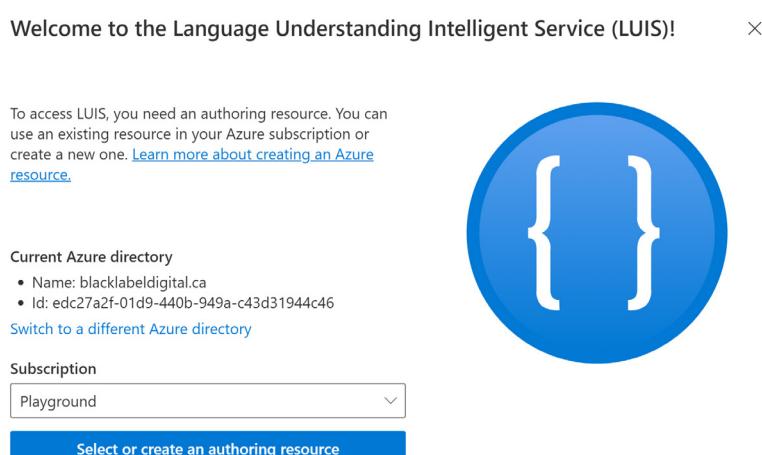


Figure 5: Welcome Modal

You will notice that LUIS has reached into the Azure account and retrieved the list of subscriptions. We will select the appropriate subscription and continue with the selection of our authoring resource we created previously in the Azure portal.

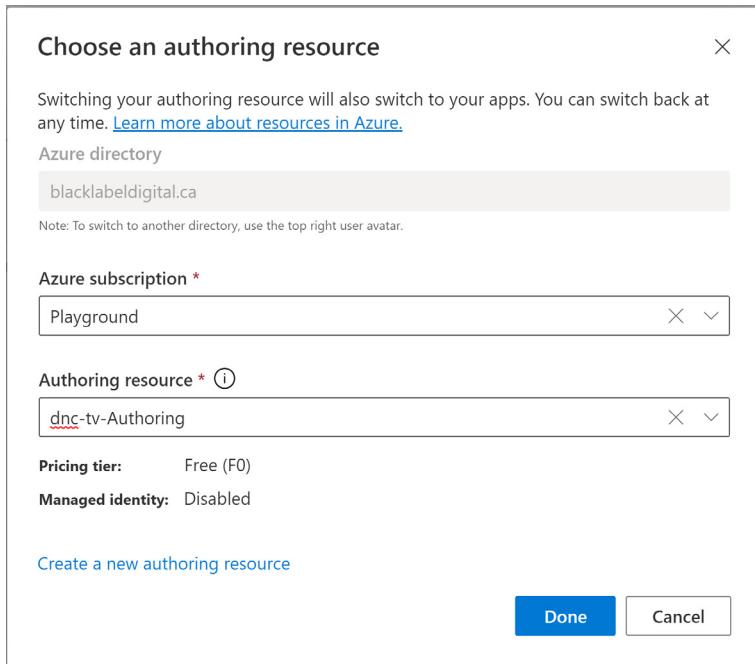


Figure 6: Authoring Service Selection

You can see here that the authoring resources are listed based on the subscription selected. There is also the option to create a new authoring resource. For now, we will select the “dnc-tv-authoring” service and click “Done”.

Conversation apps

Azure subscription: Playground / Authoring resource: dnc-tv-Authoring [Choose a different authoring resource.](#)

[+ New app](#) [Rename](#) [Export](#) [Import logs](#) [Export logs](#) [Delete](#)

Figure 7: Empty Application List

We are now in the LUIS portal and connected to our desired authoring service. The first step for us will be to create a new “Conversation app” and begin with authoring the application.

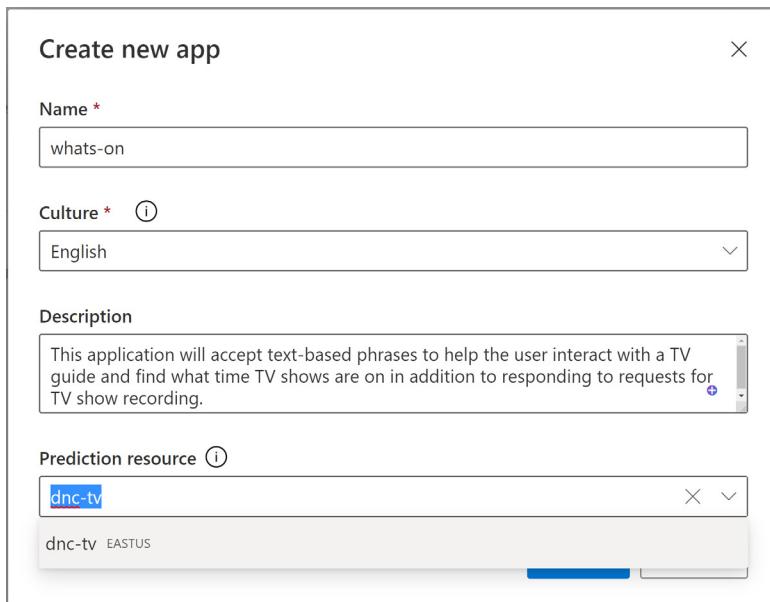


Figure 8: Creating a New Application

In the “Create new app” modal, we want to make sure that we give the application and appropriate name.

In our case, we will name the application “whats-on”.

Culture is the language that you will use for the text utterances (phrases) sent to LUIS. This would typically correlate to your client applications. Currently there are 13 supported languages and 6 in preview:

Supported	Preview
Chinese	Arabic
Dutch	Gujarati (Indian)
English (United States)	Hindi (Indian)
French (Canada)	Marathi (Indian)
French (France)	Tamil (Indian)
German	Telugu (Indian)
Italian	
Japanese	
Korean	
Portuguese (Brazil)	
Spanish (Mexico)	
Spanish (Spain)	
Turkish	

*The simplified Chinese character set is expected instead of the traditional character set. Naming for intents, entities, features, and regular expressions accept either Chinese or Roman characters.

**LUIS will not understand the difference between Keigo and informal Japanese due to the lack of syntactic analysis in this regard. Therefore, the incorporation of varying levels of formality as training examples are required.

The prediction resource list is the pulling from our Azure subscription and correlates back to the service we created earlier – along with the authoring service.

Conversation apps

Azure subscription: Playground / Authoring resource: dnc-tv-Authoring [Choose a different authoring resource](#).

+ New app | ↻ Rename | ⌂ Export | ⌄ Import logs | ⌄ Export logs | 🗑 Delete

Name	Last modified ↓	Culture
whats-on	25/9/2021	en-us

Figure 9: Application List

After our application is created, it will be listed in the applications table.

Lay of the Land

Now that we have successfully signed in, connected to our Azure instance, and created our first application, let's quickly breakdown the various menu options before diving into creating some intents and entities.



Figure 10: Top Menu

You can see that the horizontal toolbar shows that we are currently working with the “whats-on” application. You will also notice that the application has a version designation. Applications in LUIS can be cloned and versioned. This is helpful for when you have an application published for production use, but you want to explore additional possibilities with a separate version.

The three main menu options are “DASHBOARD”, “BUILD”, and “MANAGE”.

DASHBOARD provides analytics that include the number of interactions with your LUIS application. BUILD is where we will spend most of our time in this example and, you can probably guess that this is where we will build out the intents, entities, and utterances that are critical components of the application. MANAGE, which we will touch on briefly later, is useful for adjusting settings and also accessing authorization keys for integration. The “TRAIN”, “TEST”, and “PUBLISH” options, we will get to shortly.

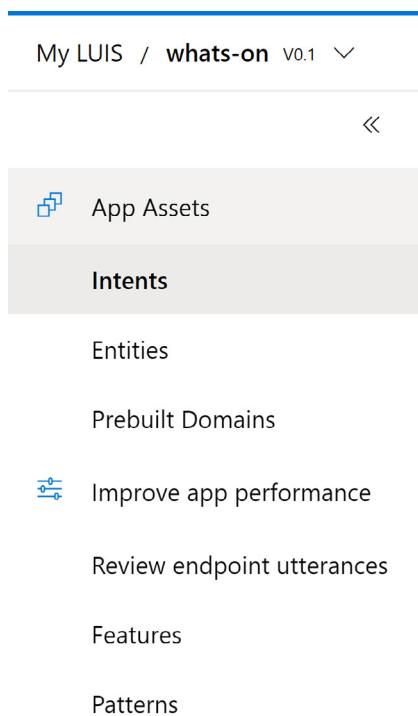


Figure 11: Side Menu

The side menu allows you to access the editing and management features for the main components of the LUIS application. Let's click on “Intents” to get started.

Intents

Name ↑ Examples Features

Name	Examples	Features
None	0	+ Add feature

Figure 12: Empty Intents List

Here you will see that the default “None” intent has already been created for us. As mentioned earlier, this intent is required for every application and cannot be renamed or deleted.

Let's create our first intent. Click the “Create” button to open the create modal.

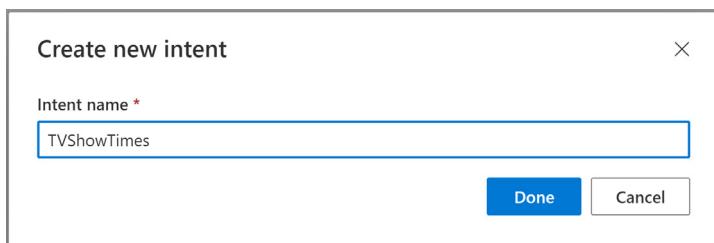


Figure 13: Create New Intent

We want our first intent to reflect the intent that a user might imply when sending in an utterance (text phrase) to indicate that he or she is looking for scheduling information regarding a TV Show. As such, we will call this intent “TVShowTimes”.

Click “Done” to create the intent.

TVShowTimes

Machine learning features

+ Add feature

Examples

✓ Confirm all entities ...

Example user input

Score

Nearest intent

Difference

Type an example of what a user might say and hit Enter.

There are no utterances found.

Figure 14: TVShowTimes Edit Screen

Here we can see our new TVShowTimes intent. We will need to enter some utterances, but before we do, we will first go and create some entities.

This can be accessed by clicking “Entities” in the side menu.

The screenshot shows the LUIS Entities list interface. On the left is a sidebar with options: App Assets, Intents, Entities (which is selected and highlighted in grey), Prebuilt Domains, and Improve app performance. The main area has a header "Entities" with a help icon. Below the header are buttons for Create, Add prebuilt entity, Add prebuilt domain entity, Rename, and Delete. A table below the buttons has columns for Name (sorted by ascending name) and Type. A message at the bottom of the table says "You haven't created any entities yet."

Figure 15: Empty Entity List

Click “Create” to create your first entity.

The screenshot shows the "Create an entity" dialog. At the top is a title "Create an entity" and a close button "X". Below the title is a "Name *" field containing "TVShow". Underneath the field are four radio buttons for "Type": "Machine learned" (unchecked), "List" (checked), "Regex" (unchecked), and "Pattern.any" (unchecked). A descriptive text follows: "List entities represent a fixed, closed set of related words along with their synonyms. A list entity is not machine-learned. It is an exact text match. LUIS marks any match to an item in any list as an entity in the response." Below this is an "Example" section showing a sentence with tokens: "Book 2 adult business tickets to Arrabury Airport". The word "Arrabury" is underlined with a blue arrow pointing to the word "airport" below it. A note below the example says "'airport' includes: AAA = {"Ana Airport", "AAA"}, AAB = {"Arrabury Airport", "AAB"} ...". At the bottom right are "Create" and "Cancel" buttons.

Figure 16: Create an Entity

Here we can see the four entity types that are available to us. We will create a Machine Learned entity later.

For now, select the “List” type and click “Create” to create the list entity.

The screenshot shows the Entities list page again. The sidebar is identical to Figure 15. The main area shows a table with one row. The first column is "Name" with the value "TVShow" and the second column is "Type" with the value "List". Above the table are buttons for Create, Add prebuilt entity, Add prebuilt domain entity, Rename, and Delete.

Figure 17: New Entity Created

With our list entity, “TVShow” now created, we can go ahead and click on it to access the edit screen.

TVShow

List

[List items](#) [Examples](#) [Roles](#)

List entities represent a fixed, closed set of related words along with their synonyms. List entities are extracted by an exact text match and can be resolved to a normalized value. [Learn more about List entities.](#)

 Import values  Delete

Normalized values ↑

Synonyms

Type in a list item ...

America's Got Talent

AGT



Type in value ...

CSI

CSI: Las Vegas



CSI: Miami



CSI: NY



CSI: New York



CSI: Cyber



CSI: Vegas



CSI Vegas



Type in value ...

Criminal Intent

CI



Type in value ...

Law & Order

Law and Order



L&O



Type in value ...

Seinfeld

Jerry Seinfeld



Type in value ...

Figure 18: Entity List Values

Here we are in the “TVShow” edit screen. To enter new items, simply enter the name in the “Type in the list item...” text box and hit enter. I have gone ahead and entered several so we can breakdown what these are and how the “Synonyms” relate. These items all identify a TV show that we, as the cable company, know we include in our schedules. When the user sends in a text request, we want LUIS to be able to easily parse these values from the text phrases. We use the synonym values to associate potential additional ways that a user may request a TV show. “America’s Got Talent” is often referred to as “AGT”, so we use synonyms to create this association.

Let’s now go back and enter a different type of entity, the “Prebuilt” entity.

Add prebuilt entities

When you add a built-in entity, its predictions will be available to you while labeling utterances.

 Search built-in entities ...

Name

age

Age of a person or thing

10-month-old, 19 years old, 58 year-old

datetimeV2

Dates and times, resolved to a canonical form

June 23, 1976, Jul 11 2012, 7 AM, 6:49 PM, tomorrow at 7 AM

dimension

Spacial dimensions, including length, distance, area, and volume

2 miles, 650 square kilometres, 9,350 feet

Figure 19: Prebuilt Entities List

LUIS comes with prebuilt entities that we can take advantage of without having to reinvent the wheel and start from scratch to train LUIS on heavily used entities such as a datetime value. Let's go ahead and select the "datetimeV2" option and click "Done".

To complete our entity list, create a "Machine learned" entity called "Recording" and we can make some modifications to it to round out our entities before going back to update our intents.

Entities ⓘ	
Create Add prebuilt entity Add prebuilt domain entity Rename Delete	
Name ↑	Type
datetimeV2	Prebuilt
Recording	Machine learned
TVShow	List

Figure 20: Entity List

From the entities list, click the "Recording" entity and we will make some slight modifications to allow for a composite entity.

Recording ⓘ

Machine learned

[Schema and features](#) [Examples](#) [Roles](#)

An ML entity can be composed of smaller subentities, each of which can have its own ML features. [Learn more about ML features.](#)

Name	Machine learning features ⓘ
Recording	+ Type name ... ×

Figure 21: Edit Recording Entity

In the edit screen for the "Recording" entity, click the "+" button to add a few child properties.

Name	Machine learning features ⓘ
Recording	+ Type name ... ×
TVShow	+ Add feature
RecordTime	+ Add feature

Figure 22: List of Child Properties

Our goal here is to create a composite entity that represents a TV show recording request. We want the

“Recording” entity to have a “TVShow” and “RecordTime” property since both values will be required for scheduling the recording. We will use the “Add feature” option to create the association to the relevant entities.

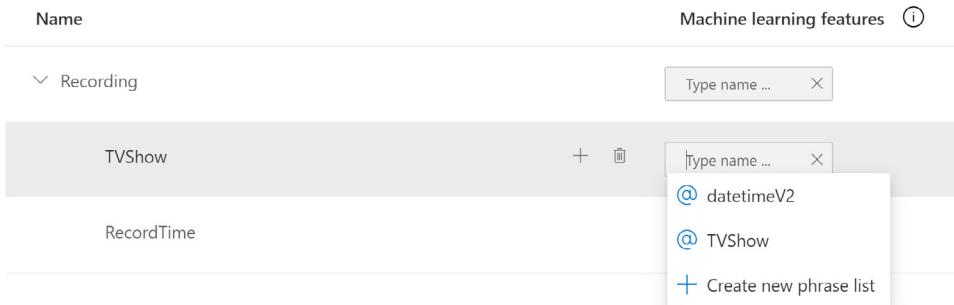


Figure 23: Select TVShow Feature

You will see that clicking on the “Add feature” section revealed a dropdown with our two existing entities we have created previously. Let’s create the associations to each child property.



Figure 24: Recording Features

At this point we will go back and add one more intent before beginning to train our model. We will name this intent “RecordTime”, and it will be used to determine from any utterance sent by the user that a TV show should be recorded and at which time.

Intents ①			
+ Create + Add prebuilt domain intent Rename Delete			
Name ↑	Examples	Features	
None	0	+ Add feature	
RecordTime	0	+ Add feature	
TVShowTimes	0	+ Add feature	

Figure 25: Intents List

Looking at our intent list now, we can see that we have three intents.

It is now time to start training LUIS. Let’s click on the “TVShowTimes” intent and start to enter utterances

which LUIS will use to create the necessary learning model when responding to real-world requests.

TVShowTimes 

Machine learning features 

+ Add feature

Examples 

✓ Confirm all entities  Move to  Delete ...

  View options  

Score Nearest intent Difference

Example user input

can you tell me when criminal intent is on today?

There are no utterances found.

Figure 26: Empty Utterance List

Enter the phrase “can you tell me when criminal intent is on today?” into the text box and hit “Enter”

Example user input

Type an example of what a user might say and hit Enter.

can you tell me when criminal intent is on today ?
TVShow dateti...

Figure 27: Example TV Show Utterance

You can see that LUIS has picked out from the utterance that the list type of TVShow has been highlighted. In addition, LUIS has also signaled out “today” as referring to a prebuilt entity type that we added to represent a datetime value.

Let’s go ahead and enter more utterances to help LUIS understand what intent is inferred from these types of text-based phrases.

is l & o on tonight ?
TVShow datetimeV2

when is agt on this week ?
TVS... datetimeV2

what time is csi las vegas on this coming wednesday ?
TVShow datetimeV2

can you tell me when criminal intent is on today ?
TVShow dateti...

Figure 28: TV Show Utterance List

We will now go and enter some utterances to help train LUIS further on the RecordTVShow intent

Figure 29: Record Utterance

You can see that LUIS has parsed the “TVShow” and “datetimeV2” entities, but we also want to ensure the “Record” object can be parsed as well. Let’s help LUIS out here. We are going to highlight a portion of the text and make some adjustments.

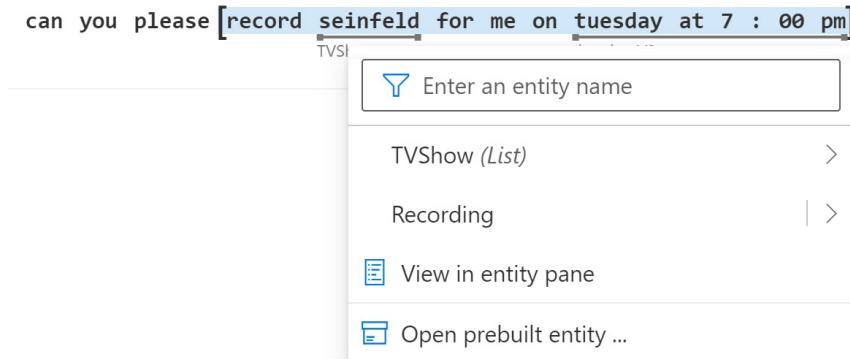


Figure 30: Select the Recording Entity

Click “Recording” and you will see that LUIS now understands that this block of text is a higher-level object with child entities that it has already detected.

can you please record seinfeld for me on tuesday at 7 : 00 pm

Recording  TVShow datetimeV2

Figure 31: Recording Utterance

Now we will continue to add some more utterances in the “Record” intent to ensure LUIS is receiving additional training.

can you record agt for me on friday at 9 : 00 pm est

Recording

TVS... datetimeV2

hello , i would like you to record csi : miami for me tonight at 8 : 00 pm

Recording

TVShow

datetimeV2

can you please record seinfeld for me on tuesday at 7 : 00 pm

Recording TVShow datetimeV2

Figure 32: Recording Utterances

Let's now explicitly tell LUIS to lock in the training based on our guidance. Click the "Train" button to train LUIS on the information provided so far.

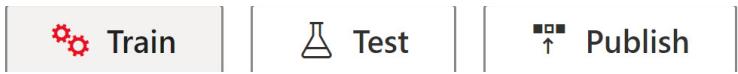


Figure 33: Train Button

From here, we can click the “Test” button to see how LUIS is doing.

The image shows the LUIS Test panel. At the top left is the word "Test". At the top right is a blue "X" icon. Below "Test" are two buttons: "Start over" and "Batch testing panel". A yellow callout box contains the text: "You have untrained changes that will not be reflected in testing." Below this is a text input field containing the query "what time is seinfeld on thursday?".

Figure 34: Test Query

The image shows a detailed view of the LUIS Test Panel. At the top left is the word "Test". At the top right is a blue "X" icon. Below "Test" are two buttons: "Start over" and "Compare with published". A yellow callout box contains the text: "You have untrained changes that will not be reflected in testing." Below this is a text input field containing the query "what time is seinfeld on thursday?". To the right of the query is a button labeled "Inspect".

The main area is divided into sections:

- Top-scoring intent:** TVShowTimes (0.672) (highlighted with a red border)
- ML entities:** No predictions
- Composite entities:** No predictions
- Other entities:** TVShow: seinfeld, datetimeV2: thursday (highlighted with a red border)

Figure 35: Test Panel

You can see that LUIS has determined that the intent from our phrase is “TVShowTimes” and that the entities have been singled out for both the “TVShow” and the “datetimeV2” entities!

Something to point out here is that the intent score is low at 0.672. This is a good indication that LUIS needs further training. In this case, additional utterances can be added to ensure that LUIS is getting smarter.

The last thing to do now is to publish so integration with the external applications can also begin.

Go ahead and click the “Publish” button.

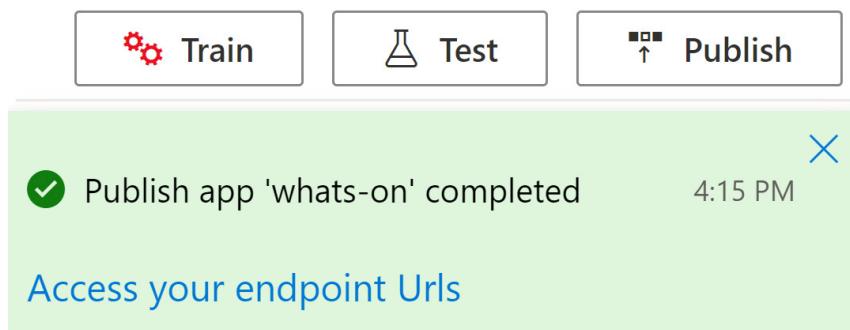


Figure 36: Publishing Completed

After publishing is completed, click the “Access your endpointUrls” to jump over to the “Prediction Resources” screen where we will find the values to use when integrating with the LUIS from your client applications. For enhanced security, the endpoint URLs enforce TLS 1.2

A screenshot of the Microsoft LUIS "Prediction Resources" screen. The top navigation bar includes "My LUIS / whats-on v0.1", "DASHBOARD", "BUILD", "MANAGE" (which is highlighted in blue), and "Train". On the left, a sidebar lists "Settings", "Publish Settings", "Azure Resources" (which is expanded to show "dnc-tv"), and "Versions". The main content area shows a "Prediction Resources" section with a sub-section for "dnc-tv". It displays various configuration details: Resource group: default; Location: eastus; Primary Key: 9a1bf57b556a4e789cd0f8a938e0ecfa; Secondary Key: 39f1c6a7577e426fba65117492a77fd9; Endpoint URL: https://dnc-tv.cognitiveservices.azure.com/; Pricing Tier: F0 (Free); and an Example Query input field containing the URL https://dnc-tv.cognitiveservices.azure.com/luis/prediction/v3.0/apps/1c30df54-764a-4dd5-a037-d9db349503c5/slots/staging/predict?subscription-key=9a1bf57b556a4e789cd0f8a938e0ecfa&verbose=true&show-all-intents=true&log=true&query=what%20time%20is%20seinfeld%20on%20thursday?. A "Change query parameters" link is located below the example query field.

Figure 37: Prediction URL

We want to copy the Example Query and include a custom natural language phrase into the “query” parameter. The query below contains the relevant domain specification, security keys, and query parameter to successfully access your LUIS instance and application.

```
https://dnc-tv.cognitiveservices.azure.com/luis/prediction/v3.0/apps/1c30df54-764a-4dd5-a037-d9db349503c5/slots/staging/predict?subscription-key=9a1bf57b556a4e789cd0f8a938e0ecfa&verbose=true&show-all-intents=true&log=true&query=what%20time%20is%20seinfeld%20on%20thursday?
```

Copy the created query and paste it into your browser's location bar and hit "Enter". The following output will be produced:

```
{  
    query: "what time is seinfeld on thursday?",  
    prediction: {  
        topIntent: "TVShowTimes",  
        intents: {  
            - TVShowTimes: {  
                score: 0.6715228  
            },  
            - RecordTime: {  
                score: 0.045393355  
            },  
            - None: {  
                score: 0.028689612  
            }  
        },  
        entities: {  
            - TVShow: [  
                - [  
                    "Seinfeld"  
                ]  
            ],  
            - datetimeV2: [  
                - {  
                    type: "date",  
                    values: [  
                        - {  
                            timex: "XXXX-WXX-4",  
                            resolution: [  
                                - {  
                                    value: "2021-09-23"  
                                },  
                                - {  
                                    value: "2021-09-30"  
                                }  
                            ]  
                        }  
                    ]  
                ],  
            - $instance: {  
                - TVShow: [  
                    - {  
                        type: "TVShow",  
                        text: "seinfeld",  
                        startIndex: 13,  
                        length: 8,  
                        modelTypeId: 5,  
                        modelType: "List Entity Extractor",  
                        recognitionSources: [  
                            "model"  
                        ]  
                    }  
                ],  
                - datetimeV2: [  
                    - {  
                        type: "builtin.datetimeV2.date",  
                        text: "thursday",  
                        startIndex: 25,  
                        length: 8,  
                        modelTypeId: 2,  
                        modelType: "Prebuilt Entity Extractor",  
                        recognitionSources: [  
                            "model"  
                        ]  
                    }  
                ]  
            }  
        }  
    }  
}
```

Figure 38: Query Result

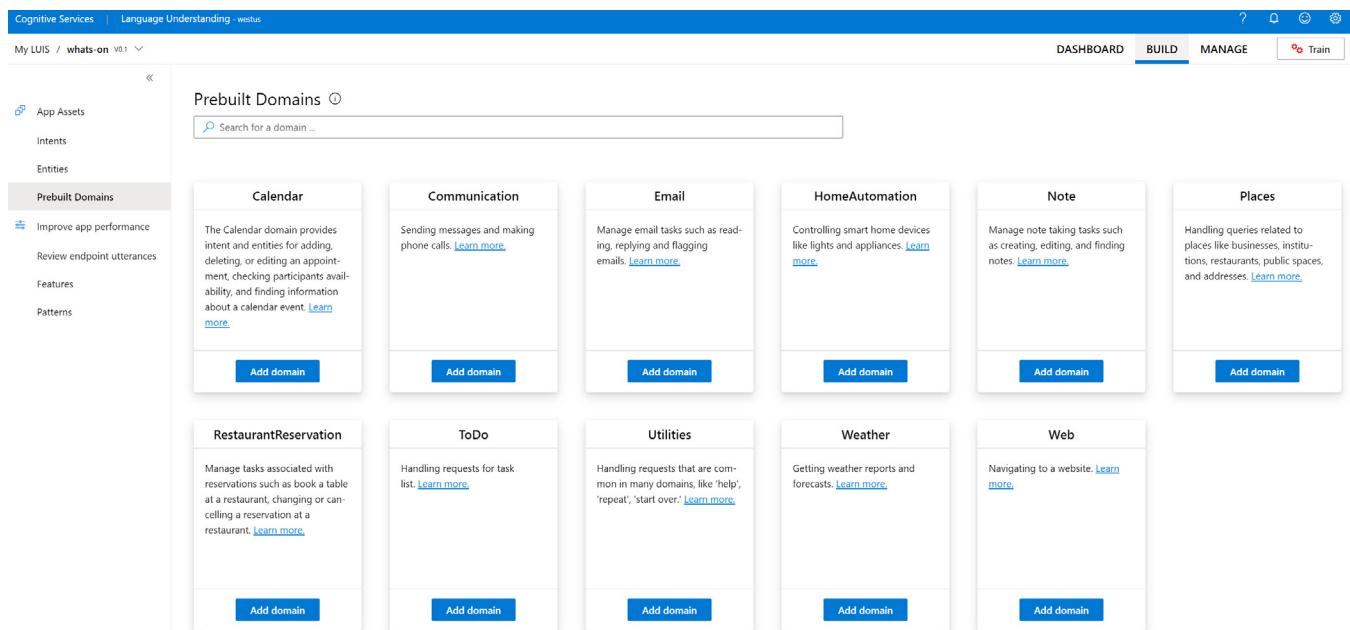
The JSON document in the response includes a nicely formatted set of data values that your client application can easily hydrate into an object and use for additional application logic. Your client application can use these values to look up the TV show for the relevant date and return all the associated times. This decisioning mechanism is possible since LUIS will do the heavy lifting of parsing the intent and relevant entities from a natural language structure that your application does not have to parse directly.

Other Ways to Interact With LUIS

In addition to using the portal for authoring, testing, and training the created language models, there are other interaction methods that can be utilized to both author and query the trained and published models. These methods include direct access using either a REST API or SDK client libraries for C#, JavaScript, and Python.

Prebuilt Domains

While you can create application models that are custom to your domain, LUIS also offers a selection of prebuilt domains that you can readily access and include in your applications.



The screenshot shows the Microsoft LUIS portal interface. The top navigation bar includes 'Cognitive Services' and 'Language Understanding - westus'. The main menu on the left has options like 'App Assets', 'Intents', 'Entries', and 'Prebuilt Domains', with 'Prebuilt Domains' currently selected. The central content area is titled 'Prebuilt Domains' and contains a search bar. Below the search bar is a grid of ten cards, each representing a different prebuilt domain:

- Calendar**: Provides intent and entities for adding, deleting, or editing an appointment, checking participants availability, and finding information about a calendar event. [Learn more](#). [Add domain](#).
- Communication**: Handles messages and phone calls. [Learn more](#). [Add domain](#).
- Email**: Manages email tasks such as reading, replying, and flagging emails. [Learn more](#). [Add domain](#).
- HomeAutomation**: Controls smart home devices like lights and appliances. [Learn more](#). [Add domain](#).
- Note**: Handles note-taking tasks such as creating, editing, and finding notes. [Learn more](#). [Add domain](#).
- Places**: Handles queries related to places like businesses, institutions, restaurants, public spaces, and addresses. [Learn more](#). [Add domain](#).
- RestaurantReservation**: Manages tasks associated with reservations such as booking a table at a restaurant, changing or cancelling a reservation. [Learn more](#). [Add domain](#).
- ToDo**: Handles requests for task lists. [Learn more](#). [Add domain](#).
- Utilities**: Handles requests common in many domains like 'help', 'repeat', 'start over'. [Learn more](#). [Add domain](#).
- Weather**: Gets weather reports and forecasts. [Learn more](#). [Add domain](#).
- Web**: Navigates to a website. [Learn more](#). [Add domain](#).

Figure 39: Prebuilt Domains

There is a nice mix of prebuilt domains available to help get you started, or you may choose to take full advantage of one of these domains if your application currently requires the language understanding capabilities provided out of the box with LUIS.

Pricing

Pricing for the LUIS service ranges from free which has a limit on transactions per second (TPS) and total transactions per month, to a paid tier that includes a higher number of TPS and allowable monthly transactions. The authoring service and prediction service are not offered in every region so you may have to pick separate regions for each of the authoring and prediction service.

Region: West US

Instance	Transactions per second (TPS)	Features	Price
Authoring – Web	5 TPS	Text requests	1 million authoring transactions and 1,000 testing prediction transaction* for free per month
Prediction – Web/Container	5 TPS	Text requests	10,000 prediction transactions free per month
Standard - Web/Container	50 TPS	Text requests	\$1.50 per 1,000 prediction transactions
		Speech requests	\$5.50 per 1,000 prediction transactions

Conclusion:

In this article, I gave an overview of the capabilities of LUIS.

LUIS is a mature service included with the growing array of managed AI/ML services that Microsoft is committed to developing under the expanding Azure Cognitive Services umbrella. Democratizing access to natural language understanding using managed services will allow your applications to include complex natural language understanding (NLU) capabilities at a fraction of the cost when compared to hiring and retaining specialized skillsets.

In a subsequent article, we will look at integrating directly with LUIS to enhance a chatbot application.

• • • • • •



Darren Gillis

Author

Darren Gillis is a Toronto-based software developer and technologist with 20+ years of experience primarily with Microsoft technologies. He is a big fan of data and Microsoft Azure, having architected numerous cloud-based software projects leveraging many of the features that Azure has to offer. He is currently developing a SaaS based compliance platform using C#, React, and PostgreSQL.

LinkedIn: www.linkedin.com/in/darrengillis,

Twitter: @darrengillis



Technical Review

Vikram Pendse



Editorial Review

Suprotim Agarwal

Damir Arh



DEPLOYING BLAZOR WEBASSEMBLY APPLICATIONS TO AZURE STATIC WEBAPPS



In this article, I compare Azure Static Web Apps with Azure App Service and explain how to publish a Blazor WebAssembly application as a Static Web App.

Since the early days of Microsoft Azure, [App Service](#) has been the platform as a service (PaaS) of choice for hosting any type of web application or website. In May 2020, an alternative was made available when Azure Static Web Apps was first previewed. A year later, in May 2021, [Azure Static Web Apps](#) became generally available (GA) and thus fully supported.

If you decide to host a web application or website in Microsoft Azure, you should consider Azure Static Web Apps now as a viable alternative to the more established Azure App Service.

What are Azure Static Webs?

Azure Static Web Apps vs Azure App Service

Both are platform-as-a-service (PaaS) offerings. Yet, the two services are significantly different:

- **Azure App Service** is a full-fledged application server on the cloud. It can run on both Windows and Linux and supports a variety of different payloads: .NET, Node.js, Python, and many others.
- **Azure Static Web Apps**, on the other hand, can only serve static files (as the name suggests).

However, the restriction to static files isn't so limiting for web applications. A large portion of today's web applications and websites don't need anything else:

- Websites built with **static site generators** like [Hugo](#), [Jekyll](#), [Gatsby](#), and others generate all the markup for their pages at build time, so you only need to serve static HTML, JavaScript, CSS, and asset files (images, fonts, etc.).
- **Single-page applications (SPAs)** written in JavaScript (vanilla or one of the popular frameworks like [Angular](#), [React](#), and [Vue](#)) or other languages thanks to WebAssembly support in modern browsers (e.g., [Blazor WebAssembly](#)) run all their code in the browser, so again you only need to deploy static files to the hosting web server.
- Even **higher-level web frameworks** like [Next.js](#) for React and [Nuxt.js](#) for Vue only need to serve their static files for two of their three modes of operation: single-page application and static site generation. Only server-side rendering requires Node.js code to run on the server and therefore cannot be hosted in Azure Static Web Apps.

You can read more about the difference between client-side and server-side rendered web apps in my [DotNetCurry article: Architecture of web applications \(with design patterns\)](#).

All of these apps (and many more) can of course be hosted in an App Service, so Azure Static Web Apps seems to be only a limited (albeit cheaper) alternative. This is contradicted by the fact that Azure Static Web Apps offer many additional services preconfigured:

- Hosting a serverless backend API, implemented as Azure Functions in .NET Core, Node.js or Python.

You can learn more about Azure Functions in an article on DotNetCurry by Gerald Versluis: [Serverless Apps with Azure Functions](#).

- A CI/CD pipeline for repositories in GitHub and (with some limitations) Azure DevOps.
- Automatic deployment to a pre-production environment for each pull request created.

- Globally distributed static files, as typically provided by CDN (Content Delivery Network) solutions.

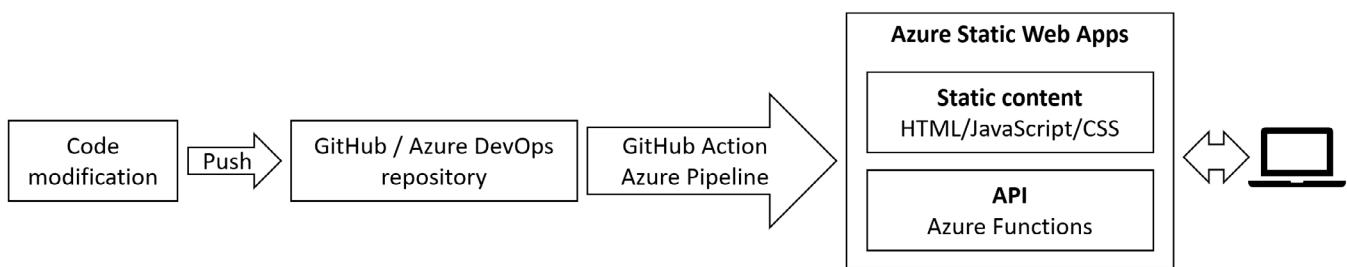


Figure 1: Azure Static Web Apps components

Although all of these features can of course be implemented with an App Service, this would require manual configuration and/or the use of additional services at an additional cost. This can put Azure Static Web Apps at a distinct advantage if their features are a good fit for your needs.

Publishing a Blazor WebAssembly application

In the rest of this article, I will walk you through the process of publishing a Blazor WebAssembly application to Azure Static Web Apps and introduce you to its various features.

Blazor is Microsoft's framework for building highly interactive web applications using .NET and C# instead of JavaScript (or TypeScript). It supports two different hosting models:

- In the **Blazor Server** hosting model, the application runs in a .NET runtime environment on the web server. The output page is rendered there and sent to the client with full HTML markup. All user triggered events on the page are sent to the server with SignalR. There they are processed, and the resulting changes to the page are sent back to the client, again via SignalR. This model requires the server to manage all page state and have an open SignalR connection for each active client.
- In the **Blazor WebAssembly** hosting model, the application runs in the browser like a normal JavaScript-based single-page application. The .NET runtime environment runs on top of the WebAssembly engine that is included in modern web browsers.

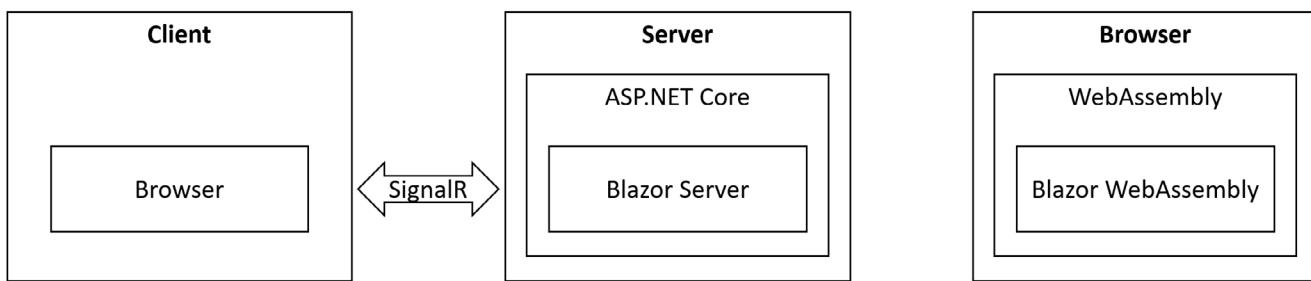


Figure 2: Blazor Server and Blazor WebAssembly hosting models

Blazor Server applications cannot be hosted in Azure Static Web Apps because they require a .NET runtime on the server. Blazor WebAssembly applications, on the other hand, can be hosted in Azure Static Web Apps because the browser only needs to download static files from the server: assets, HTML, CSS, and .NET assemblies (instead of JavaScript files). This is what we will use in the next two sections.

You can learn more about Blazor WebAssembly in the DotNetCurry article from Daniel Jimenez Garcia:

[Using Blazor WebAssembly, SignalR and C# 9 to create full-stack real time applications.](#)

However, it's worth noting that the application from this article cannot be hosted in Azure Static Web Apps, as it relies on an ASP.NET Web API backend.

Using the Blazor starter template from GitHub

There is not yet a project template in Visual Studio, which is pre-configured for publishing to Azure Static Web Apps.

If you are looking for a sample project to try with minimal effort, [the Blazor Starter Application template on GitHub](#) is your best choice. Provided you are logged in to GitHub with your account, you can use the template to initialize a new GitHub repository with its contents.

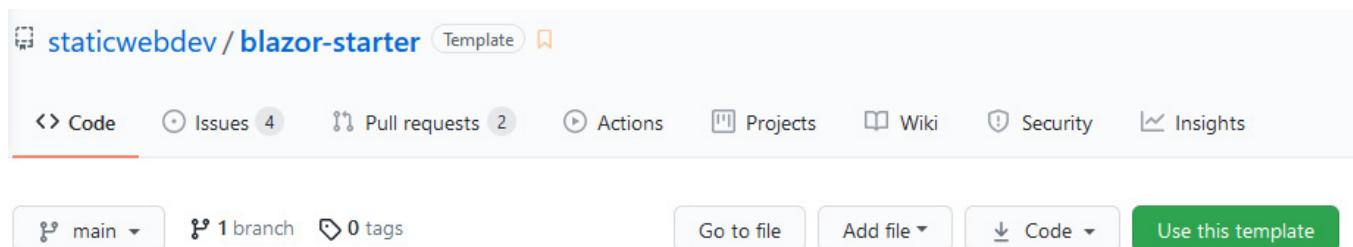


Figure 3: Blazor Starter Application template in GitHub

The solution consists of three projects that provide you with a general guide on how to best structure your code:

- **Client** is a Blazor WebAssembly application, which is very similar to the application created from the Blazor WebAssembly project template in Visual Studio. Only the data source for the weather forecast page is different: an Azure Function instead of a static JSON file.
- **Api** is an Azure Functions project with a single function that returns the weather forecast data.
- **Shared** is a .NET Standard class library with the **WeatherForecast** data type used by the other two projects.

A screenshot of a web application titled 'Blazor Starter'. On the left is a sidebar with 'Home', 'Counter', and a highlighted 'Fetch data' button. The main content area is titled 'Weather forecast' and contains a table with five rows of weather data. The table has columns for Date, Temp. (C), Temp. (F), and Summary.

Date	Temp. (C)	Temp. (F)	Summary
27.07.2021	41	105	Hot
28.07.2021	14	57	Cold
29.07.2021	1	33	Cold
30.07.2021	30	85	Mild
31.07.2021	39	102	Hot

Figure 4: Weather forecast page from the Blazor Starter Application

To run the application locally, clone the repository to your local machine and open it in your favorite code editor:

- In Visual Studio 2019, select multiple startup projects: Api and Client, and then start the solution with or without debugging.

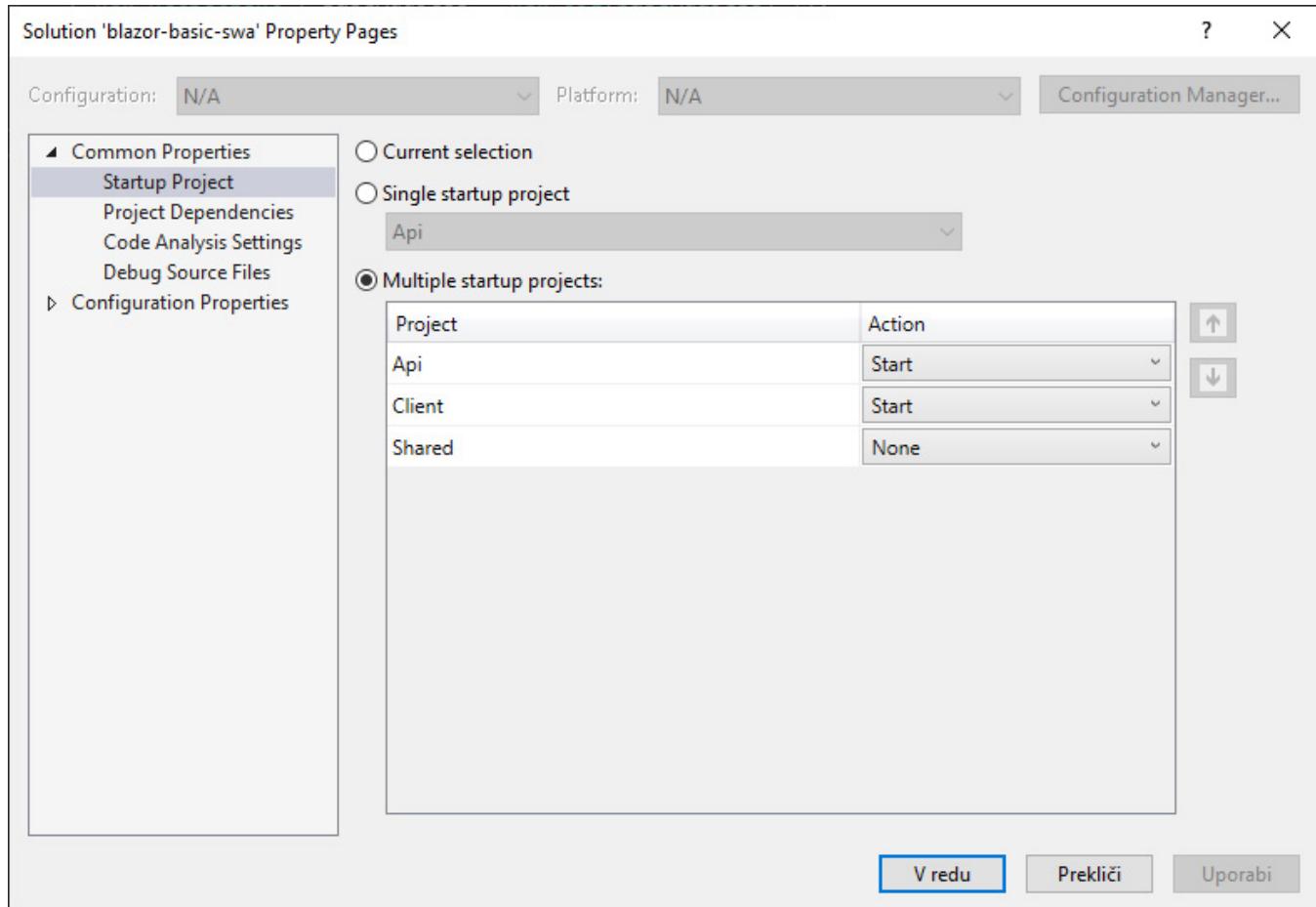


Figure 5: Multiple startup projects configuration in Visual Studio 2019

- In Visual Studio Code, select the **Client/Server** launch configuration from the dropdown and start debugging.

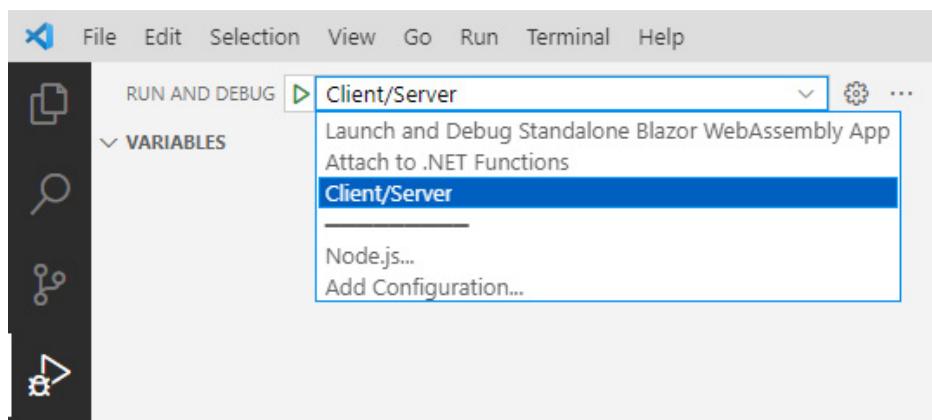


Figure 6: Selecting Client/Server launch configuration in Visual Studio Code

This ensures that the Azure Functions and Blazor WebAssembly run concurrently, just like when deploying the solution in Azure Static Web Apps.

Once you've verified that the application is running correctly, it's time to publish it to Azure Static Web Apps. There aren't too many options when creating a Static Web App resource in Azure Portal. The most important part is the *Deployment details* needed to enable the automatic CI/CD feature.

Deployment details

Source GitHub Other

GitHub account damirah

Organization * damirah

Repository * dnc-swa-blazor-starter

Branch * main

Build Details

Enter values to create a GitHub Actions workflow file for build and release. You can modify the workflow file later in your GitHub repository.

Build Presets Blazor

App location * Client

Api location Api

Output location wwwroot

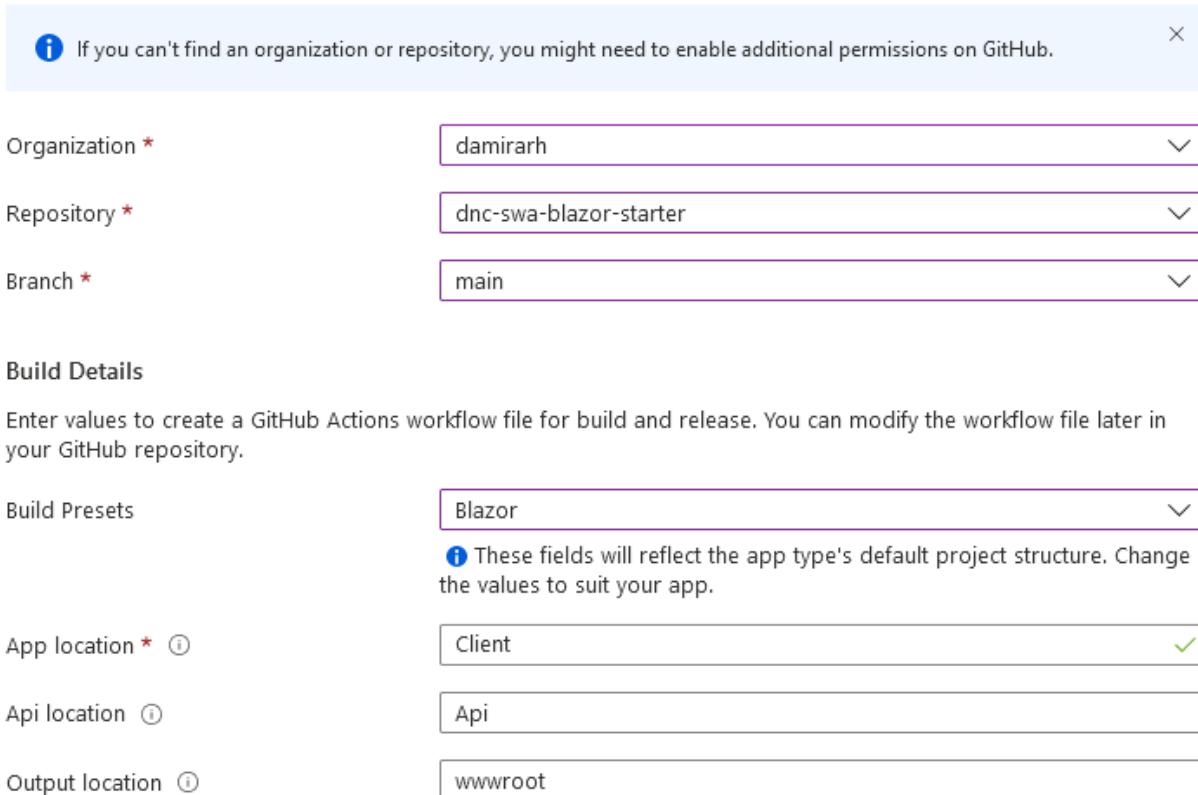


Figure 7: Configuring the deployment details for an Azure Static Web App

If you select **GitHub** as the source for your application, you must first authorize access to your GitHub account and then select the repository and branch from which you want to deploy. There are a number of build presets for different frameworks that you can choose from. Blazor is one of them.

In general, you will need to further configure it with paths to different parts of your full solution, although the default settings match those in the Blazor Starter Application template:

- *App location* is the path to the folder containing the Blazor WebAssembly project.
- *Api location* is the path to the folder containing the Azure Functions project.
- *Output location* is the path within the Blazor WebAssembly project that contains the final build output to be deployed to the web server.

Based on this information, a GitHub workflow file for GitHub Actions is created and committed to your repository. If you want to change any of the values later, you will need to modify this file. Fortunately, the configuration is well documented.

After the resource is created, a GitHub Action is automatically triggered to build and deploy the application. Until completed, a link will appear at the top of the resource page in Azure Portal that will take you to the GitHub page with details about the build in progress. The URL of your Azure Static Web App is also on this resource page, which you can click to test the application once it is deployed.

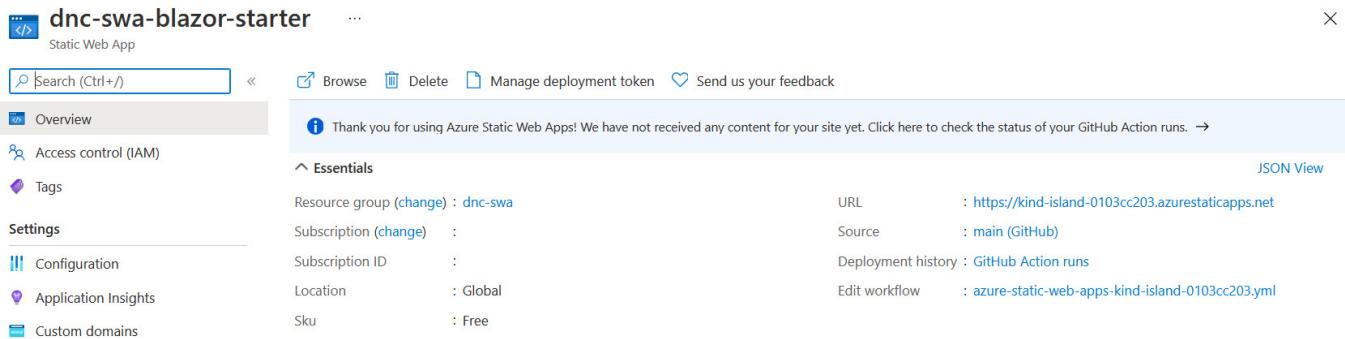


Figure 8: Azure Static Web App resource page in Azure Portal

If you select Other as the source for your app when creating the resource, no further configuration is possible on the Azure Portal. You must then manually configure the build and deployment process in your CI/CD tool.

Azure DevOps is the only other CI/CD tool that is [officially supported](#) with a readily available custom build task. However, there are no Azure Pipelines presets for Static Web Apps yet. You need to create a Starter pipeline and replace the default YAML with the following:

```
trigger:
  - main

pool:
  vmImage: ubuntu-latest

steps:
  - checkout: self
    submodules: true
  - task: AzureStaticWebApp@0
    inputs:
      app_location: '/Client'
      api_location: '/Api'
      output_location: '/wwwroot'
      azure_static_web_apps_api_token: $(deployment_token)
```

You may need to change the following values:

- trigger contains the name of the branch in your repository from which you want to deploy.
- app_location, api_location, and output_location are the same paths you can configure for GitHub repositories in Azure Portal.

Before you save and run the pipeline, you need to configure the `deployment_token` variable referenced in the last line of the YAML file. To get the value, click the *Manage deployment token* button on the toolbar of the Static Web App resource page in Azure Portal.

When creating the variable in Azure Pipelines, you should enable the option to keep the value secret.

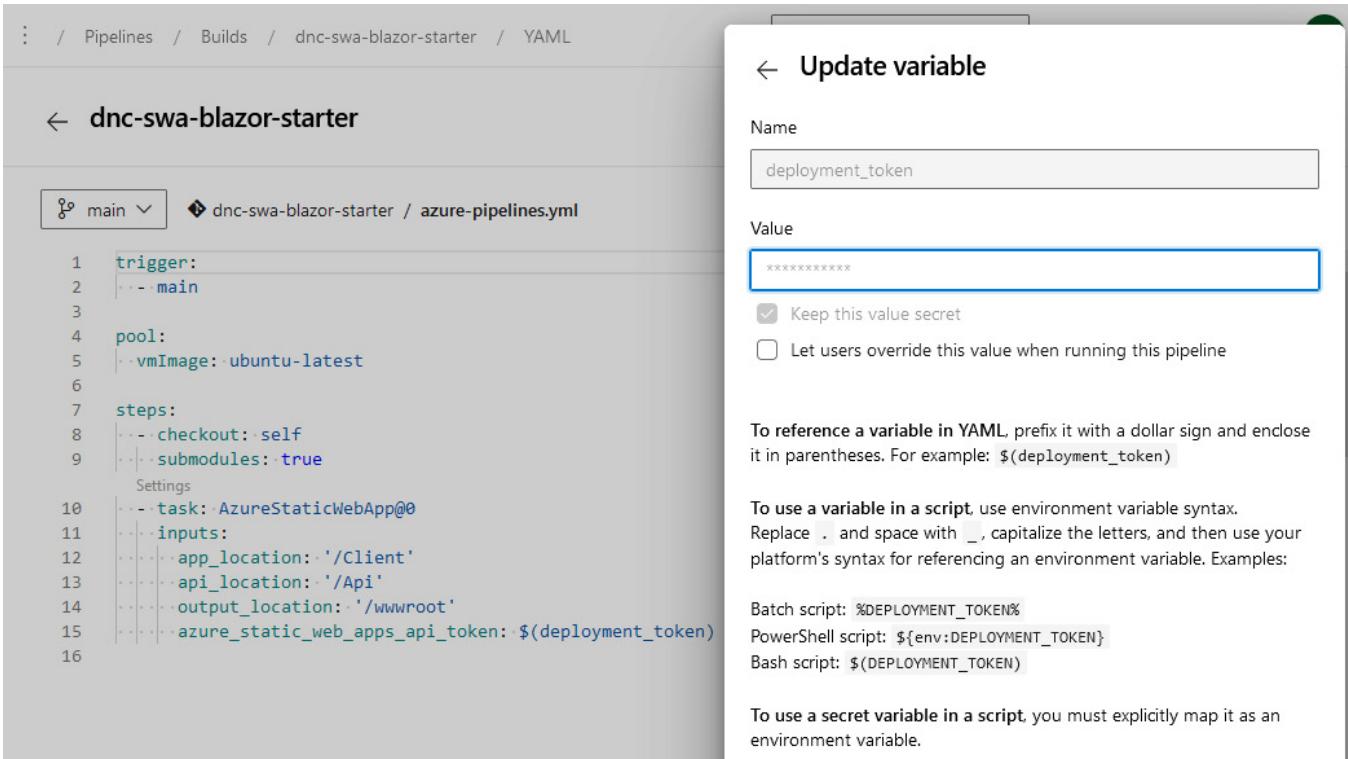


Figure 9: Configuring the deployment token variable in Azure Pipelines

When this pipeline is configured, the application from the Azure DevOps repository is automatically published to Azure Static Web Apps, just like the application from the GitHub repository.

Creating a custom solution for a Blazor WebAssembly application

Although creating a solution from the Blazor Starter Application template in GitHub makes deploying Azure Static Web Apps very easy, it's not perfect and its limitations might bother you:

- The template repository seems to be updated even less frequently than the project templates in Visual Studio 2019. For example, the Blazor WebAssembly app still uses version 3.2, so if you want to use version 5, you'll have to update it yourself.
- If you already have a Blazor WebAssembly app that you want to deploy to Azure Static Web Apps, it's not easy to start with a completely new repository and migrate your app there, especially if you want to keep your Git history.

Fortunately, it's not that difficult to deploy any Blazor WebAssembly application to Azure Static Web Apps using either GitHub or Azure DevOps. In this section, we'll take a look at the steps required.

Deploying Blazor WebAssembly application to Azure Static Web Apps

Any existing application should be easy enough to deploy, as the Blazor WebAssembly application folder within the repository is fully configurable in the Azure Portal wizard or in the generated YAML file:

```
app_location: "dnc-swa-blazor" # App source code path
api_location: "Api"           # Api source code path - optional
output_location: "wwwroot"    # Built app content directory - optional
```

Don't worry if you don't have an Azure Functions API project. If the build script doesn't find it in the configured folder, it'll simply skip this step.

However, after deployment, you'll notice that the server responds with a 404 (missing page) error if you navigate directly to a non-root page of your Blazor WebAssembly application.

For example:

- Opening the root page of your application and navigating to a subpage works.
- Refreshing that subpage in your browser returns 404.

This is because there's no static page on the web server that matches the URL of the subpage. When you navigate there from the root page in the browser, the Blazor WebAssembly application in the browser does the routing and makes it work. When you request the same page from the server, it doesn't know which page to return.

You can fix this by adding a `staticwebapp.config.json` file with the following content to the root of your Blazor WebAssembly application folder:

```
{  
  "navigationFallback": {  
    "rewrite": "/index.html"  
  }  
}
```

This tells Azure Static Web Apps to serve the `index.html` page if there is no static page that matches the requested URL. The Blazor WebAssembly application in the browser will then ensure that the correct page is rendered once it is loaded.

If you experimented with Azure Static Web Apps during the preview, this file was originally named `routes.json`. The old name is still supported, but it is recommended that you use the new name instead.

If you want to add an **Azure Functions API** to your Blazor WebAssembly application, you need to make sure that .NET Core 3.1 (or alternatively Node.js 12 or Python 3.8) is used. Other runtime versions are not supported for Azure Functions, which are provided as part of Azure Static Web App. Don't forget to properly configure the `api_location` in your project's GitHub or Azure DevOps YAML file.

If you are using the Standard (non-free) hosting plan for Azure Static Web Apps, you can [use another Azure Functions resource](#) to host the API. However, you will have to deploy and pay for it separately.

Azure Static Web Apps use a **reverse proxy** to map the Azure Functions API to the `/api` subpath of your domain. One advantage is that you do not need to configure [CORS \(Cross-Origin Resource Sharing\)](#) to make Azure Functions work in the browser.

However, if the Azure Functions are run locally during development, they will usually be accessible at `http://localhost:7071/api` without remapping through a reverse proxy. This requires additional configuration to make it work.

When running locally, the Blazor WebAssembly application must use a different base path for API calls. In the Blazor Starter Application mentioned above, this is accomplished by a `BaseAddress` application

configuration value in `appsettings.Development.json` file which is only in use during local development:

```
{  
    "BaseAddress": "http://localhost:7071/  
}
```

When it is not present, the code falls back to the base URL of the application:

```
var baseAddress = builder.Configuration["BaseAddress"] ??  
    builder.HostEnvironment.BaseAddress;  
builder.Services.AddScoped(_ => new HttpClient  
{  
    BaseAddress = new Uri(baseAddress)  
});
```

In addition, the Azure Functions application must be launched with the following arguments to disable CORS: `start --cors *`. You can configure this with a `launchsettings.json` file in the `Properties` folder of the Azure Functions project:

```
{  
    "profiles": {  
        "Api": {  
            "commandName": "Project",  
            "commandLineArgs": "start --cors *"  
        }  
    }  
}
```

You can avoid all these special configurations for local development if you want to use [Azure Static Web Apps CLI](#) instead. It's still in preview, but it already does a good job of emulating the same setup locally as when deploying the app.

To use it, you need to have Node.js installed. You can then install CLI with npm:

```
npm install -g @azure/static-web-apps-cli
```

From the root of your solution, you can launch CLI with the following command:

```
swa start http://localhost:5000 --api http://localhost:7071
```

For this to work, make sure you are running the Blazor WebAssembly application with Kestrel on port 5000. Otherwise, you will need to change the port in the command accordingly.

This command maps the Blazor WebAssembly application to `http://localhost:4280` and the Azure Functions API to `http://localhost:4280/api` so that the same code can run both locally and in Azure after deployment. CLI can also read the configuration in `staticwebapp.config.json` to further emulate the behavior of Azure Static Web Apps after deployment.

Other Azure Static Web Apps features

There are other features in Azure Static Web Apps that I haven't covered yet. Many of them can be configured via [the staticwebapp.config.json file](#): URL redirects and rewriting, global and route-specific

response headers, overrides for error pages, and more. Built-in support for user authentication with Azure Active Directory, GitHub and Twitter as identity providers is also available.

Pre-production environments

When deployed from GitHub, code from open pull requests is automatically deployed to a separate pre-production environment with its own URL that doesn't affect the production version of the application published from the main branch in any way.

Change intro text #1

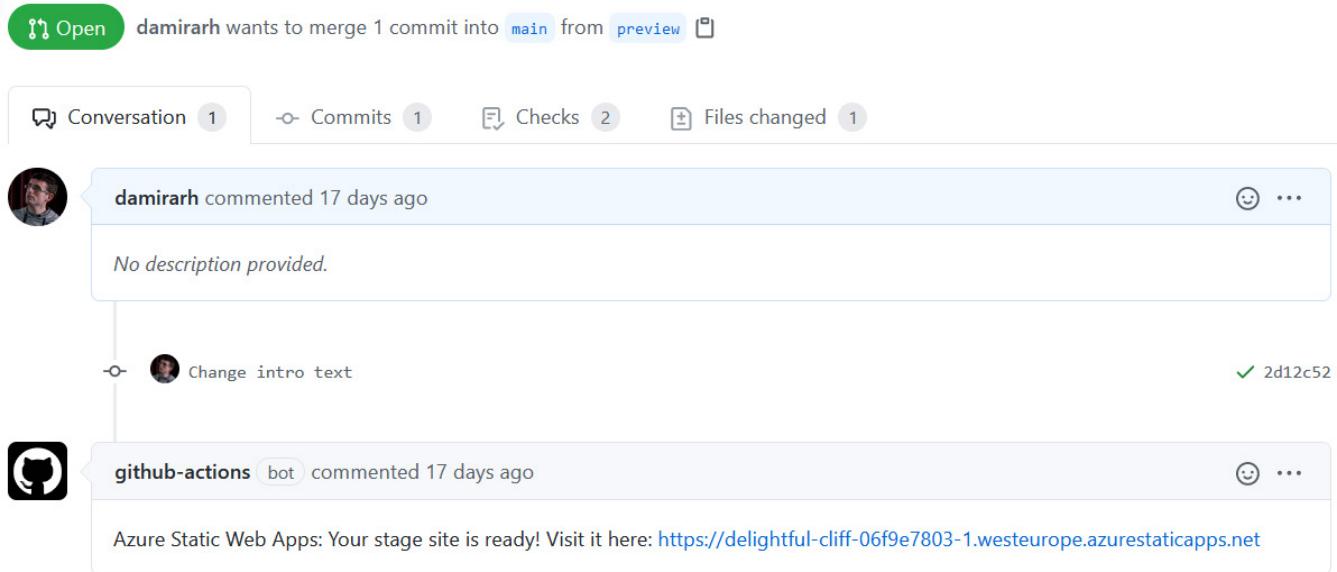


Figure 10: Pre-production URL in GitHub pull request as published by the GitHub action

This simplifies testing of the code in the pull request. Whenever new code is pushed to the underlying branch, it's automatically redeployed. When the pull request is merged (or closed), the pre-production environment is also automatically removed.

Unfortunately, this feature isn't (yet) available when deploying from Azure DevOps.

Custom domains

Unlike App Service, which generates an `azurewebsites.net` subdomain that matches the resource name, Azure Static Web Apps don't generate their URLs based on the name you used for the resource. Instead, they automatically generate a random subdomain at `azurestaticapps.net` (for example, `delightful-cliff-06f9e7803.azurestaticapps.net`).

For this reason, it's even more important that you can configure a custom domain for the site without incurring any additional costs (other than the domain, of course). All you need to do is add a CNAME record with your DNS provider. Once this is validated, the configuration will be applied in Azure Portal.

A free SSL certificate will also be automatically created for this domain.

Advantages and disadvantages

You can create up to ten Azure Static Web Apps per Azure subscription for free (with some additional quotas) if you do not need SLA (Service Level Agreement). For most hobby projects, this should be acceptable.

If you need more reliability for your application, you can upgrade to the Standard plan where you are charged per application. In both cases, you get a lot of features: pre-configured CI/CD, CDN, custom domain, SSL certificates, etc. If you are used to App Services, this is a great value add.

On the other hand, Azure Static Web Apps have their own limitations:

- Your code must preferably be in GitHub or at least Azure DevOps.
- The route configuration is only flexible to a limited extent. Pattern-based redirects, for example, are not supported.
- Most importantly, only a subset of applications is supported. Web pages cannot be generated on the server.

Any of these limitations could be a showstopper and force you to continue using App Service if you want to host your site in Azure PaaS.

Conclusion

This article introduced Azure Static Web Apps. It compared its features with Azure App Service, i.e. the other PaaS option for hosting websites in Azure and described the benefits and limitations of the new service.

The process of publishing and configuring a Static Web App is presented using a sample Blazor WebAssembly single-page application. To help you get started, common pitfalls and solutions to them are highlighted.

• • • • • •



Damir Arh
Author



Damir Arh has many years of experience with software development; from complex enterprise software projects to modern consumer-oriented mobile applications. Although he has working knowledge of many different languages, his favorite one remains C#. He is a proponent of test-driven development, continuous integration, and continuous deployment. He shares his knowledge by speaking at local user groups and conferences, blogging, and writing articles. He is an awarded Microsoft MVP for Developer Technologies since 2012. You can find his blog at <https://www.damirscorner.com/> Twitter: @DamirArh <https://twitter.com/DamirArh>.



Technical Review
Daniel Jimenez Garcia



Editorial Review
Suprotim Agarwal



JavaScript gets a new version every year and as developers we need to be aware of these features and their usage. This article is a quick walkthrough of the new features added to ECMAScript 2021 and demonstrates these features, with examples.

NEW FEATURES in **ECMAScript** — 2021

JavaScript as a language is changing every year now. Starting with the major changes brought into the language in 2015 with ES6 or ES2015, the language design team has been putting efforts to add new features to the language every year. The new features are added with the goal of making the language better for its usage.

Some of these features introduce newer syntax to the language while the others add new APIs or extend the existing APIs. Overall, these new features help JavaScript developers write meaningful code and improves their productivity.

Any new features to JavaScript have to go through a [process](#) with the TC39 team. The proposals made to the TC39 team go through four stages of evaluations and when a proposal reaches stage 4, it makes it to the language specifications. The proposals at different stages can be found in the [proposals repository](#). The proposals that reach stage 4 and are considered to be included in the coming version of the language can be found in the [finished proposals](#).

The set of features to be included in ECMAScript 2021 were finalized on 22nd June. These features bring in some conveniences to the JavaScript programmers and also help in dealing with the objects based on the garbage collectors.

Let's dive in and explore these features now.

New features in ES 2021

`String.prototype.replaceAll`

The strings in JavaScript have a `replace` method, which replaces the first occurrence of the substring in a given string. Let's consider the following example:

```
let str = "Doctors say apples are very good for health and you should eat an apple everyday.";
console.log(str.replace("apple", "orange"));

// Output: Doctors say oranges are very good for health and you should eat an apple everyday.
```

Snippet 1: JavaScript `replace()`

The `replace` method replaces the first occurrence of apple with the orange and the second one is left unchanged. Thus, the sentence doesn't sound logical anymore. The `replaceAll` method introduced in ES 2021 extends this and replaces all the occurrences of the substring with its substitute. Snippet 2 here modifies Snippet 1 to use `replaceAll` and give it a proper meaning:

```
let str = "Doctors say apples are very good for health and you should eat an apple everyday.";
console.log(str.replaceAll("apple", "orange"));

// Output: Doctors say oranges are very good for health and you should eat an orange everyday.
```

Snippet 2: JavaScript `replaceAll()`

`Promise.any`

The `Promise.any` method is a promise aggregator. It takes an array of promises and returns a single promise when any of the promises are resolved. If all the promises fail, then it results in an `AggregateError` - a new error created to wrap a set of errors into a single error object.

Snippet 3: `Promise.any`

```
let promises = [
  Promise.resolve(10),
  Promise.resolve({ value: 100 }),
  Promise.reject('Something went wrong')
];

Promise.any(promises).then(
  result => {
```

```

        console.log(result);
    }
);

//Output: 10

```

Snippet 3: Promise.Any()

As we see here, the `promises` array contains three promises, of which two are resolved and the third one is rejected. As the first promise is resolved first, the promise created by the `Promise.any` is resolved with its value and the output is 10.

Let's modify the example shown in Snippet 3 and reject all the promises. Now we need to handle the rejection of the resulting promise to see what error is thrown when all the promises are rejected. Snippet 4 shows the modified example:

```

let promises = [
  Promise.reject(10),
  Promise.reject({ value: 100 }),
  Promise.reject('Something went wrong')
];

Promise.any(promises).then(
  result => {
    console.log(result);
  },
  error => {
    console.log(error.errors, error.message, error.name);
  }
);

// Output:
// [10, Object, "Something went wrong"]
// All promises were rejected
// AggregateError

```

Snippet 4: Modified promise example

As we see, the `AggregateError` object received contains the rejection objects of all the three promises.

This will be really helpful when a set of APIs are invoked using `Promise.any` and when all of them fail, the error object would contain details of failures of all the APIs. This will help in debugging the issues with each of them.

Numeric Separator

Numbers are one of the most fundamental types of variables we handle in any project. Some projects may need to work with large numbers and store their values in numeric constants. If the source code contains initialization of constants with large numbers, those statements are not easily readable. Let's consider the following snippet:

```
let num = 1000000000;
```

Snippet 5: num declaration

The variable `num` is assigned with the value one billion. But it is very hard to know this, as one has to count the number of zeros to know the value of the number. The numeric separator feature can be used here to make the number more readable.

To write this number on a paper we use commas to make it readable. Using commas this number can be written as: 1,000,000,000.

To write it this way in JavaScript, the commas have to be replaced with underscores(_). Following is the modified assignment using underscores:

```
let num = 1_000_000_000;
```

Snippet 6: num declaration with separators

Now at the first look, one can understand the value is a billion. Positions of the separators can be changed based on how you interpret the numbers. If your team consists of people who are more familiar with lakhs and crores (Indian numbering system) than millions and billions, the separators can be repositioned accordingly. The following snippet shows the modified separators showing the number as 100 crores:

```
let num = 100_00_00_000;
```

Snippet 7: numeric separators with localization

The separator can be used after the decimal points and also with binary, octal or hexadecimal numbers. Snippet 8 shows some more examples:

```
let num = 126_182_872.19_283;
let binaryNum = 0b1_001_101;
let octalNum = 0o10_271_152;
let hexaDecimalNum = 0x1_81A_F0E;
```

Snippet 8: separators in various numeric formats

Logical Assignment Operators

Quite often, we see conditions checking if a variable is set with a value, or if an object has a property initialized before using the value. Logical assignment operators are introduced to reduce the number of statements in the code to perform such operations. ES 2021 adds three logical assignment operators.

While writing a certain business logic, it is often necessary to check if a variable is set to a value before assigning it. The following snippet shows how it is done:

```
if(!a) {
  a = 20;
}
```

This can be simplified to a single statement using the “Or or equals” operator, it is shown below:

```
a ||= 20;
console.log(a);
```

The above operator has a counterpart using the ampersand, it is called “And and equals operator”. This operator assigns value to the variable if it has a value. The following snippet shows this:

```
a &&= 30;  
console.log(a);
```

The above snippet assigns the value 30 to a only if a already has a value. If it is not set, the variable remains unassigned.

The third logical assignment operator is called “QQ Equals”. It can be used to assign value to a property in an object by checking if the property has a value. Here’s an example:

```
let person = {  
    name: "Ravi",  
    city: undefined  
};  
person.city ??= 'Hyderabad';  
console.log(person);  
  
// Output: Hyderabad
```

The interesting part about this operator is, it doesn’t evaluate the expression to the right if the target has a value. Meaning, it gets expanded to a statement similar to the following snippet:

```
person.city ?? (person.city = "Hyderabad");
```

It clearly shows that the assignment would happen only when `person.city` doesn’t have a value. So, if the expression to get the value is a function call, the function wouldn’t be invoked when the target has value.

WeakRefs

Every object created in JavaScript is created with a strong reference. Which means, the object cannot be garbage collected unless the reference is destroyed.

And when the objects are created at one place and then the reference is stored in an array or sent as an argument to a function, the object will have multiple references. These objects cannot be garbage collected when any of these references are still available. Memory management in JavaScript is explained pretty well on [MDN](#).

`WeakRef` is a type added to ES 2021 to create weak references. As the name itself suggests, a weak reference doesn’t hold its object strongly. It allows the object to be garbage collected even when the weak reference still points to it.

The following snippet shows how to create a `WeakRef` and use it:

```
let wr = new WeakRef({  
    name: 'Ravi',  
    profession: 'Software Engineer'  
});  
  
console.log(wr.deref());
```

Snippet 9: Using WeakRef

The `deref` method on the `WeakRef` object returns the object to which it holds the reference. It returns `undefined` once the object is garbage collected.

`WeakRef` will hold the reference to the object unless it is garbage collected. It is not guaranteed when the object will be garbage collected, as the implementation of garbage collection depends on the JavaScript engine where the code is being executed. It is recommended to not use `WeakRef` when the object has to be available for a given period of time or for certain functionality in the program.

A snippet similar to the one below can be used to check for how long a weak reference stays in the memory for a trial, though the result may not be completely reliable as the garbage collector would run at different instances:

```
let remove = setInterval(() => {
  let ref = wr.deref();
  if(!ref) {
    console.log('Reference destroyed');
    clearInterval(remove);
  }
  else {
    console.log("Reference exists");
  }
}, 3000);
```

Snippet 10: Testing Garbage Collection on WeakRef

FinalizationRegistry

The `FinalizationRegistry` is an API introduced to support `WeakRef`. The `FinalizationRegistry` provides a way to attach a callback that can perform cleanup operations when the object is garbage collected. So the callback gets executed only when the object is removed from the memory.

Registering a `FinalizationRegistry` is a two-step process. First is to create a `FinalizationRegistry` object and the second is to attach it to an object. The following snippet shows the first step:

```
let fr = new FinalizationRegistry((value) => {
  console.log("Cleaning up the value: ", value);
  // Clean up logic
});
```

As the above snippet shows, the `FinalizationRegistry` accepts a callback during instantiation, the callback can receive a reference of the resource object held by the underlying object. The logic to release the resource has to be implemented in the callback.

Check the following snippet:

```
let wr = new WeakRef({
  name: 'Ravi',
  profession: 'Software Engineer'
});

fr.register(wr, "Some object");
```

Though this snippet uses a `WeakRef`, any object can be registered with the `FinalizationRegistry`.

If the role of `FinalizationRegistry` does not hold good anymore for an object because the resource has been cleared by some piece of business logic, it is possible to unregister the `FinalizationRegistry`. The following snippet unregisters the registry added above:

```
fr.unregister(wr);
```

As the execution of `FinalizationRegistry` depends on the garbage collector, it is not guaranteed when it is going to invoke the callbacks attached. It is [recommended to avoid the usage](#) unless you are absolutely sure about it.

Conclusion

The features added in ECMAScript 2021 make some of the most frequently used snippets more compact and readable.

Also, with the introduction of `WeakRef` and `FinalizationRegistry`, the language is providing ways to make the objects ready for garbage collection and also release the resources to avoid memory leaks. I hope this article provides a good start to these features.

• • • • •



Ravi Kiran
Author



Ravi Kiran is a developer working on Microsoft Technologies at Hyderabad. These days, he is spending his time on JavaScript frameworks like AngularJS, latest updates to JavaScript in ES6 and ES7, Web Components, Node.js and also on several Microsoft technologies including ASP.NET 5, SignalR and C#. He is an active blogger, an author at SitePoint and at DotNetCurry. He is rewarded with Microsoft MVP (Visual Studio and Dev Tools) and DZone MVB awards for his contribution to the community.



Technical Review
Daniel Jimenez Garcia



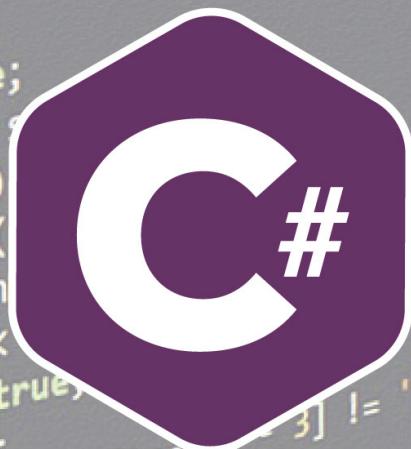
Editorial Review
Suprotim Agarwal

COVERS C# v6, v7 AND .NET Core

THE ABSOLUTELY AWESOME

NOW INCLUDES
CHAPTERS ON
.NET Core 3.0
& C# 8.0

BOOK ON



AND

.NET

DAMIR ARH



Yacoub Massad

GET THE MOST OUT OF ASYNC /AWAIT

Introduction

The **async** and **await** keywords were added to the C# language in version 5, about nine years ago. They enable developers to write asynchronous methods.

Note: This article is not an introduction to `async/await`. I assume the reader already knows what these keywords mean and how to use them.

Editorial Note: For a quick introduction to `Async/Await` in C#, take a look at [Async & Await in C# 5.0](#) and [Asynchronous Programming in C# using Async/Await – Best Practices](#)

It seems to me that many developers do understand `async/await` and know how to use them, when they must. For example, if they are assigned to work on an existing piece of code that uses `async/await`, they know how to work with this existing code.

At times, they could also be forced by the libraries they are using to use `async/await`. For example, the [HttpClient class](#) (before .NET 5) did not have synchronous methods; only asynchronous

In this article I try to show how `async/await` can be used to solve problems in situations where developers may not think about `async/await` as an immediate solution.



ones. Even today, it has only a synchronous **Send** method. It does not have synchronous versions of the more convenient **GetAsync**, **PostAsync**, **PutAsync**, or **DeleteAsync** methods. This means that when you use the **HttpClient** class, you are kind of forced to use **async/await**. You can use **.Result** or **.Wait()** on the returned tasks from the **async** methods of **HttpClient** to mimic a synchronous experience, but that is not recommended.

However, I have also seen developers miss chances to use this feature when they did not expect **async/await** to solve the problem. This resulted in complex code that could have been simplified with **async/await**.

This article is **not** about how to use **async/await** when you are supposed to.

Instead, in this article I try to show **how you can use async/await to solve problems when you might not have expected** it to provide a solution.

A simple example: A timer

I will start with a simple example: a timer.

```
private void StartButton_OnClick(object sender, RoutedEventArgs e)
{
    async Task Start()
    {
        int count = 1;

        while (true)
        {
            this.Title = count + " iterations";

            count++;

            await Task.Delay(1000);
        }
    }

    Start();
}
```

This **StartButton_OnClick** method is an event handler in a WPF application. It runs when a “Start” button is clicked.

This method defines a local function called **Start**, which it calls in a fire and forget manner. This local function is asynchronous. When this function is run, it basically starts counting from 1 to infinity. Each second, it increments the counter by one and displays the result on the title of the window.

Almost every second, the title of the window will change.

Using **Task.Delay**, I was able to basically create a timer that runs every second.

If on each iteration we made some calculation that takes half a second, then the counter will be updated every 1.5 seconds.

```

private void StartButton_OnClick(object sender, RoutedEventArgs e)
{
    async Task Start()
    {
        int count = 1;

        while (true)
        {
            var calculationResult = RunCalculationThatTakesHalfASecond();

            this.Title = count + " iterations and result is :" + calculationResult;

            count++;

            await Task.Delay(1000);
        }
    }

    Start();
}

```

We can do the following to try to increment the counter every second:

```

private void StartButton_OnClick(object sender, RoutedEventArgs e)
{
    async Task Start()
    {
        int count = 1;

        while (true)
        {
            var sw = Stopwatch.StartNew();

            var calculationResult = RunCalculationThatTakesHalfASecond();

            this.Title = count + " iterations and result is :" + calculationResult;

            count++;

            var timeToWait = TimeSpan.FromSeconds(1) - sw.Elapsed;

            if (timeToWait > TimeSpan.Zero)
            {
                await Task.Delay(timeToWait);
            }
        }

        Start();
    }
}

```

In the code above, I measure the time it takes to process each iteration using the `Stopwatch` class. After processing, I do not delay the next iteration for a full second. Instead, I delay the next iteration one second minus the time spent on processing the current iteration.

Instead of `async/await`, I can use a timer class like this:

```

private void StartButton_OnClick(object sender, RoutedEventArgs e)
{
    int count = 1;

    var timer = new System.Windows.Threading.DispatcherTimer();

    timer.Tick += (_, __) =>
    {
        var sw = Stopwatch.StartNew();

        var calculationResult = RunCalculationThatTakesHalfASecond();

        this.Title = count + " iterations and result is :" + calculationResult;

        count++;

        var timeToWait = TimeSpan.FromSeconds(1) - sw.Elapsed;

        timer.Interval = timeToWait > TimeSpan.Zero ? timeToWait : TimeSpan.Zero;
    };

    timer.Interval = TimeSpan.FromSeconds(1);
    timer.Start();
}

```

The DispatcherTimer class is configured to raise the Tick event every second. But it starts measuring time after the Tick event handler completes. Therefore, like before, I am using a Stopwatch to calculate the time each iteration takes and adjusting the timer wait interval for the next iteration.

The difference between the two examples is that in the async/await case, the code is modeled as a *single procedure* while in the timer class case, we have basically *two procedures*: the procedure that starts the timer (StartButton_OnClick), and the event handler lambda that runs every time a new iteration begins. Modeling logic as a single procedure gives us more control and is cleaner.

For example, in the async/await version, I am using a while loop to model the iterations. The logic in the async/await version is clearly modeled using procedural code.

For example, compare how we specify the waiting time for each iteration in the two examples. In the async/await version, we “wait” for the remaining time and continue the loop while in the timer class version, we set the Interval property.

In this later version, it’s like we are communicating state between two different things. The code in the lambda is telling the timer object how much to wait before executing the next iteration. In the async/await version, there is no such communication. There is a single procedure.

I call the approach that uses the timer class an **event-based approach** because we write code to react to events.

In the async/await version, I can exit the loop using a simple break statement. In the event-based version, I would need to call timer.Stop() which is a form of communication between two things.

In summary, the async/await version allowed us to model the timer logic, while the event-based version required us to coordinate things.

This difference will become more apparent in the next example.

A bigger example: A transaction system

In this section, I will share with you a bigger example. You can find the source code here:
<https://github.com/ymassad/AsyncAwaitExample/tree/main/AsyncAwaitExample>

Event Based Approach

The idea here is basically this: we want to provide an ASP.NET Web API service with three APIs to allow consumers of the service to insert multiple pieces of data into the database within a single database transaction.

It is expected that the whole transaction might span several minutes because a few seconds (or minutes) might pass in between posting one piece of data to the server, and the other. Here are the APIs:

```
[HttpPost]
[Route("Event/StartTransaction")]
public Guid StartTransaction()

[HttpPost]
[Route("Event/Add")]
public void Add(Guid transactionId, [FromBody] DataPointDTO item)

[HttpPost]
[Route("Event/EndTransaction")]
public void EndTransaction(Guid transactionId)
```

The consumer of the service would start by calling the StartTransaction API to start a new transaction. A unique transaction id is returned to the consumer. The consumer would then call the **Add** API multiple times, each time passing the transaction id and some piece of data. Once the consumer wants to commit the transaction, the logic would call EndTransaction passing the transaction id.

The three method signatures above are the signatures of actions (methods) in the **EventController** class in the TransactionWebService1 project. I named this class EventController because it uses the event-based approach I discussed in the previous section.

Although the three methods are not C# event handlers, they are basically events handlers. They get called when an HTTP request arrives. The logic of the transaction is distributed among the tree methods.

I have included the implementation of these methods here for easier understanding of the article:

```
[HttpPost]
[Route("Event/StartTransaction")]
public Guid StartTransaction()
{
    var transactionId = Guid.NewGuid();

    var connectionString = configuration.GetConnectionString("ConnectionString");

    var context = new DatabaseContext(connectionString);

    context.Database.EnsureCreated();
```

```

        statePerTransaction.InitializeState(transactionId, new StateObject(context));

        return transactionId;
    }

```

The `StartTransaction` method creates a new random transaction id, creates a new Entity Framework Core database context object, makes sure the database is created, and then stores the database context object in an in-memory state object that maps the state to a specific transaction id. This allows us later to retrieve the database context object in the other methods.

Note: In many cases, it is not a good idea to store state in memory in web services.

One reason is that web services might be restarted and therefore in-memory state would be lost. Another reason is that you might want to have multiple servers respond to requests, and requests belonging to the same transaction might be processed differently by different servers that don't share memory. This article is not on web service design. The examples I provide may be valid for some scenarios and not others.

```

[HttpPost]
[Route("Event/Add")]
public void Add(Guid transactionId, [FromBody] DataPointDTO item)
{
    var stateObject = statePerTransaction.GetStateObjectOrThrow(transactionId);

    try
    {
        var context = stateObject.DatabaseContext;

        var dataPointEntity = new DataPoint()
        {
            Value = item.Value
        };

        context.DataPoints.Add(dataPointEntity);
    }
    catch
    {
        statePerTransaction.RemoveStateObject(transactionId);
        throw;
    }
}

```

The `Add` method starts by retrieving the state object based on the transaction id. This state object will contain the database context object. A new `DataPoint` entity is created based on the data provided to the `Add` method. Such an entity is then attached to the database context. If there is an error processing the data, the state object will be removed from memory.

```

[HttpPost]
[Route("Event/EndTransaction")]
public void EndTransaction(Guid transactionId)
{
    var stateObject = statePerTransaction.GetStateObjectOrThrow(transactionId);

    try
    {
        var context = stateObject.DatabaseContext;

        context.SaveChanges();
    }
}

```

```

        }
        finally
        {
            statePerTransaction.RemoveStateObject(transactionId);
        }
    }
}

```

The EndTransaction method retrieves the state object, invokes SaveChanges on the database context (thus committing the transaction), and finally removes the state object from memory.

Note: The TransactionWebService1 project (and the TransactionWebService2 project) is a console application. You can run it to test running a complete transaction. If you do run it, the ASP.NET controller (AsyncAwaitController or EventController) will be hosted and ready to accept requests, and an HttpClient based consumer will call the APIs to run a complete transaction that includes five data points. See the code in [Program.cs](#) to see how to select which controller (AsyncAwaitController or EventController) to use and how the [FakeClient](#) class is used to run a complete transaction.

The implementation of the EventController controller seems reasonably clean.

Async/Await Alternative

Now, let's look at the async/await alternative of the same controller. The code is inside the [AsyncAwaitController](#) class.

```

[HttpPost]
[Route("AsyncAwait/StartTransaction")]
public Guid StartTransaction()
{
    var transactionId = Guid.NewGuid();

    HandleTransaction(transactionId);

    return transactionId;
}

[HttpPost]
[Route("AsyncAwait/Add")]
public void Add(Guid transactionId, [FromBody] DataPointDTO item)
{ ... }

[HttpPost]
[Route("AsyncAwait/EndTransaction")]
public void EndTransaction(Guid transactionId)
{ ... }

```

These three methods have the exact same signature as the ones in the EventController class. The implementation is different though.

The StartTransaction method generates a new random transaction id, and then calls a method named HandleTransaction. The HandleTransaction method is an asynchronous method. The StartTransaction method does not wait for the HandleTransaction method to complete (does not await the returned Task or call the Wait function on the returned task).

Let's take a look at this HandleTransaction method before we discuss the other two APIs.

```
private async Task HandleTransaction(Guid transactionId)
{
    statePerTransaction.InitializeState(transactionId, new StateObject(new
AsyncCollection<DataPointDTO>()));

    try
    {
        var connectionString = configuration.GetConnectionString("ConnectionString");

        await using var context = new DatabaseContext(connectionString);

        await context.Database.EnsureCreatedAsync();

        while (true)
        {
            var item = await WaitForNextItem(transactionId).ConfigureAwait(false);

            if (item.ended)
            {
                //Transaction complete
                await context.SaveChangesAsync();

                return;
            }
            else
            {
                var dto = item.obj ?? throw new Exception("Unexpected: obj is null");

                var dataPointEntity = new DataPoint()
                {
                    Value = dto.Value
                };

                context.DataPoints.Add(dataPointEntity);
            }
        }
    }
    finally
    {
        statePerTransaction.RemoveStateObject(transactionId);
    }
}
```

The HandleTransaction method is an asynchronous method that models a single transaction from the beginning to the end. It contains all the logic that we saw in the three API methods in the EventController class. It first initializes a state object and maps it to the transaction id. This state object includes an `AsyncCollection<DataPointDTO>` object. More on this soon.

The HandleTransaction method then creates a new database context object. Notice how we are using the using declaration feature of C# 8 to make sure that the context object is disposed of before the HandleTransaction method exits.

In the EventController class, creating the context object was in the StartTransaction method, and disposing of it was in the EndTransaction method. Just now (as I am writing this), I wanted to check if I am also

disposing of in the EventController.Add method in case there is an exception, but I am not.

I forgot.

In the AsyncAwaitController class, since all of the logic is in a single method (HandleTransaction), it's harder to make this mistake.

Also, the call to RemoveStateObject in HandleTransaction is done once at the end in a finally block. In the EventController class, we had to do that twice: once in the Add method in case there was an exception, and once in the EndTransaction method.

After making sure the database is created, I am doing a loop. In the loop, I am waiting for the next data item to arrive. I am basically waiting for the Add or EndTransaction methods to be called. Let's go back to the implementations of these methods to see how this works.

```
[HttpPost]
[Route("AsyncAwait/Add")]
public void Add(Guid transactionId, [FromBody] DataPointDTO item)
{
    statePerTransaction
        .GetStateObjectOrThrow(transactionId)
        .Collection
        .Add(item);
}

[HttpPost]
[Route("AsyncAwait/EndTransaction")]
public void EndTransaction(Guid transactionId)
{
    statePerTransaction
        .GetStateObjectOrThrow(transactionId)
        .Collection
        .CompleteAdding();
}
```

The `Add` method simply adds the posted data object (of type `DataPointDTO`) to the `AsyncCollection` object associated with the transaction id. The `EndTransaction` method simply marks the `AsyncCollection` object as completed (that no more items can be added to it).

The `AsyncCollection` class is part of a NuGet package called `AsyncEx`. I don't want to talk about this class in detail. But you can think of it as a producer consumer collection. Someone can add items to it, and another one can take items from it. The nice thing about this collection is that we can asynchronously wait for items to be available in this collection. The following `WaitForNextItem` method does exactly that. It is called from the `HandleTransaction` method.

```
private async Task<(bool ended, DataPointDTO? obj)> WaitForNextItem(Guid
transactionId)
{
    var collection = statePerTransaction.GetStateObjectOrThrow(transactionId).
Collection;

    if (!await collection.OutputAvailableAsync())
    {
        return (true, null);
    }
}
```

```

    }

    return (false, collection.Take());
}

```

The AsyncCollection.OutputAvailableAsync method is an asynchronous method. It returns a Task<bool> that will complete when the collection becomes non-empty or when the collection is marked as completed and there are no more items to take. This method asynchronously returns true if there are available items, and false, if there are no more items and the collection is marked as complete.

In the WaitForNextItem method, we asynchronously return (`true, null`) if the collection is empty and marked as complete (`EndTransaction` was called), or (`false, item`) when there is an available item (`Add` was called).

Now back to `HandleTransaction` - in the loop, we handle two cases:

1. There is an item: we add it to the database context object
2. There will be no more items because `EndTransaction` was called: we call `SaveChangesAsync` and exit the loop.

I hope you see the value in using `async/await` in modeling such a relatively long-running operation.

Note: At the end of the `Main` method in `Program.cs`, I wait two seconds before exiting the application. This is required because when ASP.NET exits, it doesn't know about active invocations of `HandleTransaction` because such invocations are not necessary part of any active requests, and I want to make sure such invocations are complete before I exit the application. There are better ways of handling this. One way is to use an [ASP.NET hosted service](#), but this is out of scope of this article.

Now, let's say that we want to handle the case where a transaction is abandoned. That is, what if a consumer calls `StartTransaction`, calls `Add` a couple of times, and then some time elapses without any more calls to `Add` or `EndTransaction`? We don't want to keep the state in memory, so we want to implement some timeout functionality. That is, if we don't receive a related request within X seconds, we want to remove the transaction from memory.

With the `async/await` based implementation, the fix is really easy. The only line that changed in the `HandleTransaction` method (see the updated `AsyncAwaitController` class in the `TransactionWebService2` project) is this:

```

var item = await WaitForNextItem(transactionId)
    .TimeoutAfter(Program.TimeoutSpan)
    .ConfigureAwait(false);

```

I simply called an extension method named `TimeoutAfter` on the task returned from `WaitForNextItem`.

This method returns a new task object that is guaranteed to complete within the specified timespan; either successfully if the original task completed, or unsuccessfully if the specified timespan elapsed before the original task completes.

Here is the code of this method for your reference:

```

public static async Task<TResult> TimeoutAfter<TResult>(this Task<TResult> task,
TimeSpan timeout)
{

```

```

using var timeoutCancellationTokenSource = new CancellationTokenSource();
var completedTask = await Task.WhenAny(task, Task.Delay(timeout,
timeoutCancellationTokenSource.Token));
if (completedTask != task)
    throw new TimeoutException();

timeoutCancellationTokenSource.Cancel();
return await task;
}

```

I based this method implementation on the following Stack Overflow answer:

<https://stackoverflow.com/a/22078975/2290059>

Now, if we don't receive a call to the Add or EndTransaction methods within some period of time (currently 5 seconds as specified by the Program.TimeoutSpan field), an exception will be thrown on the line that calls TimeoutAfter. As per the rules of the C# language, the database context will be disposed of properly and the state will be removed since a call to do so is in the finally block.

Now, when I tried to implement this new feature in the event-based controller, it was really hard. You can find the code in the [EventController](#) class in the TransactionWebService2 project.

I spent good time trying to implement the new code in EventController. And I am not really sure if the code I have covers all edge cases. Here is the code for convenience:

```

[HttpPost]
[Route("Event/StartTransaction")]
public Guid StartTransaction()
{
    var transactionId = Guid.NewGuid();

    var connectionString = configuration.GetConnectionString("ConnectionString");

    var context = new DatabaseContext(connectionString);

    context.Database.EnsureCreated();

    var reset = TimeoutManager.RunActionAfter(Program.TimeoutSpan, () =>
    {
        try
        {
            context.Dispose();
        }
        finally
        {
            statePerTransaction.RemoveStateObject(transactionId);
        }
    });
}

statePerTransaction.InitializeState(transactionId, new StateObject(context,
reset));

return transactionId;
}

[HttpPost]
[Route("Event/Add")]

```

```

public void Add(Guid transactionId, [FromBody] DataPointDTO item)
{
    var stateObject = statePerTransaction.GetStateObjectOrThrow(transactionId);
    var context = stateObject.DatabaseContext;

    try
    {
        var dataPointEntity = new DataPoint()
        {
            Value = item.Value
        };

        context.DataPoints.Add(dataPointEntity);
    }
    catch
    {
        try
        {
            context.Dispose();
        }
        finally
        {
            statePerTransaction.RemoveStateObject(transactionId);
        }

        throw;
    }

    stateObject.ResetTimeout(cancel: false);
}

[HttpPost]
[Route("Event/EndTransaction")]
public void EndTransaction(Guid transactionId)
{
    var stateObject = statePerTransaction.GetStateObjectOrThrow(transactionId);

    try
    {
        var context = stateObject.DatabaseContext;

        try
        {
            context.SaveChanges();
        }
        finally
        {
            context.Dispose();
        }
    }
    finally
    {
        statePerTransaction.RemoveStateObject(transactionId);
    }

    stateObject.ResetTimeout(cancel: true);
}

```

I implemented a new class called TimeoutManager to help me run an action after some time in a resettable

way. That is, I want to register an action to run after say 5 seconds, but then I want to be able to reset the timer if I receive an **Add** request before the time out elapses.

For example, say I receive an **Add** request, so I start counting from 1. After 4 seconds, I receive a new **Add** request, which means that I need to reset the timer. I should start counting from 1 again, not continue the counting from 4. See the **TimeoutManager** class for more details. I am not 100% sure about the correctness of this class.

Observe how I am setting up the cleaning code to run *after* the timeout period in the **StartTransaction** method. Also observe how I am resetting the timeout in the **Add** method. In the **EndTransaction** method I am cancelling the timeout.

I do not like the code in the updated **EventController** class. It is not clean, and I am not sure about its correctness. Adding the timeout feature in the **AsyncAwaitController** was much cleaner and verifying correctness is much easier there.

I hope I was able to show you the value of modeling an asynchronous operation using **async/await**.

Conclusion:

In this article, I have shown examples of how **async/await** can be used to model asynchronous operations in a clean way.

There may be asynchronous operations in your code that you are not modeling explicitly as asynchronous. You might be responding to multiple events and coordinating between multiple event handlers using state.

If you can detect such implicit asynchronous operations and model them using **async/await**, you are likely to end up with cleaner code.

• • • • •

Yacoub Massad

Author



Yacoub Massad is a software developer and works mainly on Microsoft technologies. Currently he works at Zeva International where he uses C#, .NET, and other technologies to create eDiscovery solutions. He is interested in learning and writing about software design principles that aim at creating maintainable software. You can view his blog posts at criticalsoftwareblog.com. He is also the creator of DIVEX (<https://divex.dev>), a dependency injection tool that allows you to compose objects and functions in C# in a way that makes your code more maintainable.

Youtube: <https://www.youtube.com/channel/UCKUnsjT09KRINzJoLv3KmDQ/>

Twitter: @yacoubmassad (<https://twitter.com/yacoubmassad>).



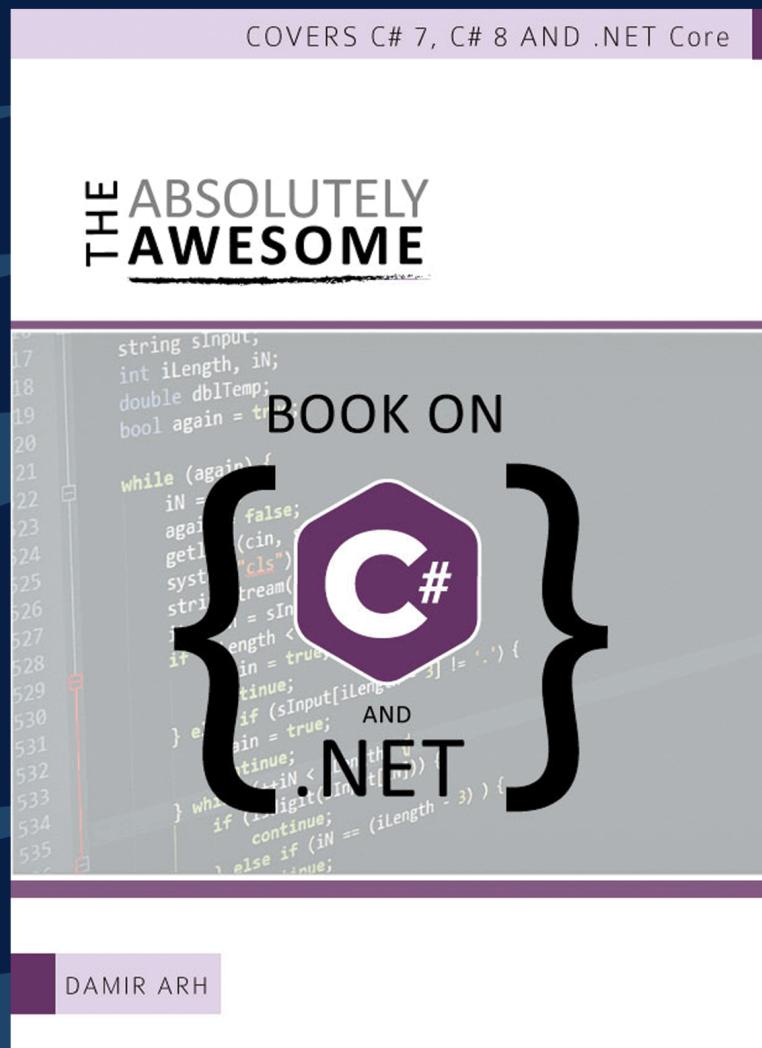
Technical Review

Damir Arh



Editorial Review

Suprotim Agarwal



Features

- .NET Framework and CLR
- New features in .NET
- Type System
- Generics and Collections
- C# 6,7 and 8
- Parallel Programming
- Async Programming
- LINQ

It's got it all!

Crack your next .NET Interview

Build a Solid Foundation

Strengthen Concepts

THE ABSOLUTELY AWESOME BOOK ON
C# and .NET

ORDER NOW !

PDF, EPUB and MOBI

Gerald Versluis



.NET MAUI : WHAT TO EXPECT

A lot of development has been going on lately in .NET MAUI. I believe it's about time that I update you with the latest and greatest that you can expect from this cross-platform app development framework!

What is .NET MAUI?

Let me quickly refresh your memory on what we're talking about over here..NET MAUI is the spiritual successor of Xamarin.Forms.[Xamarin.Forms](#) is the well-known framework by Microsoft that can be used to write cross-platform applications for iOS, Android, UWP and even others beyond that.

The name Xamarin came from the company with the same name that was acquired by Microsoft several years ago. For the product, they kept the Xamarin name up until now, but with the journey to [one .NET](#), Xamarin will be unified with the .NET ecosystem.

Xamarin as a name will fade away, but the product will live on as a first-class citizen right inside the .NET framework (note: the “new” .NET framework, .NET 6, these terms are going to be confusing for some time). Xamarin Traditional (basically Xamarin.iOS and Xamarin.Android) will not change much as they were already using the platform native namespaces., Xamarin.Forms will have a bigger transition into .NET MAUI.

That is what I will be focusing on for this article.

You might also wonder what is happening with Xamarin.Essentials, the library with all those handy APIs that you could use to access things like contacts, the GPS sensor and much more. Good news: Essentials is also becoming part of .NET and will live under the Microsoft.Maui.Essentials namespace.

Editorial Note: For those who are very new to .NET MAUI, read [Goodbye Xamarin.Forms, Hello .NET MAUI!](#)

Why .NET MAUI?

Over the years, Xamarin.Forms has become known and loved in the industry and is used by individuals and companies, big and small. You can imagine that backwards compatibility is a big theme with products on this scale. This means the team had little to no possibilities to make a lot of changes in areas where improvements could be made. Let alone any architectural changes.

Now with the move to .NET 6, there is the unique opportunity to take all the learnings from past years and put that into a complete rewrite which will be known as .NET MAUI.

New Handler Architecture

If you have worked with Xamarin.Forms before, you will know about the renderers.

The renderers are at the very core of the product. The renderer is what turns the abstract control into a native control. For instance, you define a button in XAML, this is the abstract layer Xamarin.Forms offers, that goes through the ButtonRenderer for the platform that you are running on, and that renderer will map all the properties of that Button on the native button properties.

While functionally this approach worked well, over time, some problems cropped up. First: renderers are registered and discovered through reflection. And as you might know, reflection is rather slow. Also, the renderers are tightly coupled to the platform counterpart of that control. Additionally, if you, as the developer, wanted to influence the renderer behavior, you could, but the experience wasn't always straight-forward.

To overcome this, the architecture is now rewritten to use so-called handlers. You can see a schematic overview of the old and new scenario below.

Evolving the Architecture

Renderers are tightly coupled to Xamarin.Forms and Xamarin.Forms components

Handlers are unwrapped, decoupled, portable, and reusable

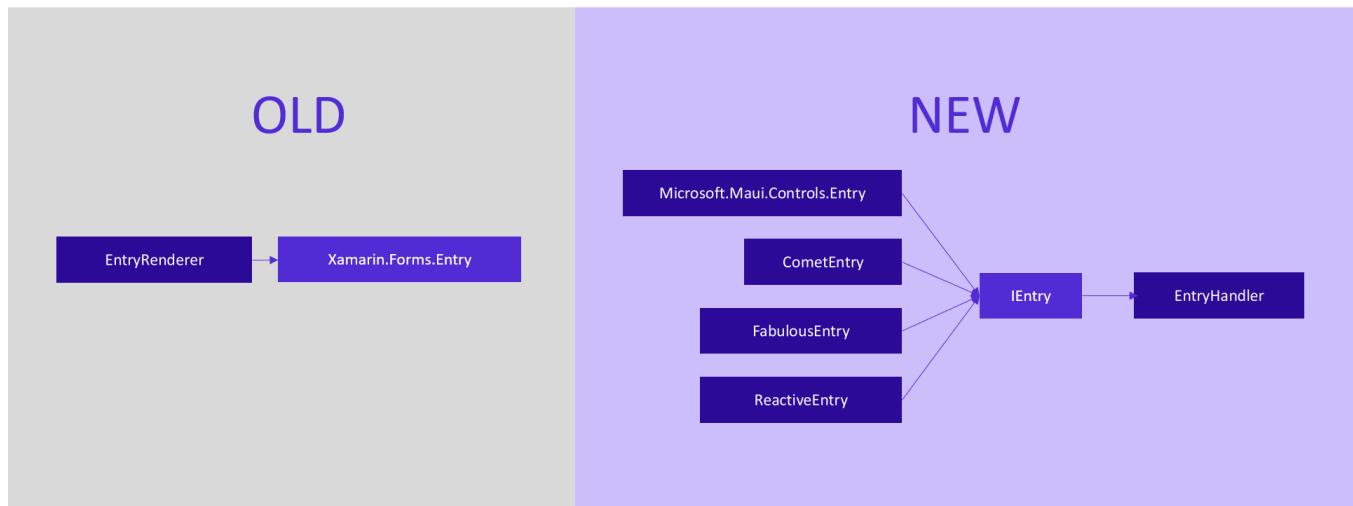


Figure 1: On the left the old/current Xamarin.Forms approach, on the right the new .NET MAUI extensible approach

You will notice that the control now is abstracted away by using an interface. This means the handlers will interact with that interface as opposed to the actual concrete control. By doing it this way, the architecture is much easily pluggable with different backends such as a completely drawn interface with SkiaSharp or Microsoft.Maui.Graphics. And simultaneously it makes it easier to implement other code design patterns like MVU with Comet or,...something else that comes up in the future.

The possibilities are endless!

I realize I am dropping some names here (SkiaSharp, Microsoft.Maui.Graphics, MVU and Comet) that might be new to you, these are beyond the scope of this article, but be sure to use a search engine and look them up. Or find links at the end in the Useful Links section.

Introducing .NET Generic Host

The name .NET Generic Host might not immediately ring any bells, but I'm quite sure you know about it. Let's have a look at the code snippet below.

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
```

```

    {
        webBuilder.UseStartup<Startup>();
    });
}

```

Looks familiar, doesn't it?

If you are an ASP.NET Core developer, you will recognize this code snippet. This is where you bootstrap your whole application. This will now be integrated with .NET MAUI as well.

This way you will have a unified and familiar way of configuring your applications whether it's a web app or a mobile app. The .NET MAUI variation of the above code snippet would look something like the one below.

```

public class Startup : IStartup
{
    public void Configure(IApplicationBuilder appBuilder)
    {
        appBuilder
            .UseMauiApp<App>()
            .ConfigureFonts(fonts =>
        {
            fonts.AddFont("Lobster-Regular.ttf", "Lobster");
        });
    }
}

```

Since this is more of an overview article than an actual deep-dive, I won't go into too much of the details here but you can see the similarities. In this case the .NET MAUI app registers a class ([App](#)) that acts as the applications' host and we are configuring one custom font. We'll talk about the font and other resources in a short while.

All Apps from a Single-Project

Another new feature is the single-project approach. Earlier, if you would build a Xamarin.Forms app, you would have a solution with at least three projects: the shared project, the iOS project, and the Android project. Ideally you would want to have all your code in the shared project because that means that you can share it across all platforms.

The iOS and Android (and UWP, Tizen, macOS, etc. if you would add those as well) projects are basically nothing more than bootstrap projects that hold the platform specific bits. For those platform specific bits, think of your app metadata or the resources like images or fonts. Or, if you still have a need to write platform-specific code, that would go here as well.

With .NET MAUI, all those projects will be combined into one. By just choosing the target that you want to run, the IDE will know which files to compile, and it will start the right emulator or send it to the right device. If you've been working with Xamarin.Forms libraries, you might know this approach as **multi-targeting**.

An extra advantage of this approach is that we can now also share resources. Before, for each platform, you had to provide its own set of images in a multitude of different dimensions. Now, with the single-project, you just import your image once and the build tools will resize the image to all the right formats for iOS, Android and WinUI.

You might notice that I suddenly mention WinUI. [WinUI](#) is a new UI framework by Microsoft that more or less is the successor of UWP, but also WPF. Where Xamarin.Forms supported UWP, this support will now change to be WinUI. Note that WinUI was previously known as Project Reunion. You might see that name pop up somewhere as well.

.NET MAUI + Blazor == <3

Something else that is completely new is the possibility to combine **.NET MAUI and Blazor**. It's unlikely that you've never heard of Blazor at this point but let me explain for if you didn't pick up on it yet.

In short, Blazor is the .NET solution for creating single-page web applications without having to write JavaScript. By leveraging WebAssembly, browsers can now interpret C# code directly with great performance and developers can stick with a language that feels more familiar to them. To make this even more powerful, there is the possibility to actually interop with JavaScript and thus all the libraries and frameworks out there can be used just as well. Since this article is not about Blazor, please refer to this page for more information on how it all works exactly: <https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor>.

There are two ways to use Blazor in .NET MAUI, with slight nuances. Let me explain.

Blazor Mobile Bindings

The Blazor Mobile Bindings is something really magical!

With this feature, you get to use the Blazor programming model inside of your mobile app. You can mix code with XAML (not HTML!) in a razor page and use all the syntax that you know and love from Blazor.

But it doesn't stop there!

You can also add the [BlazorWebView](#), a new component, (more on that below) which allows you to use *actual* Blazor inside of your app. And with actual, I mean the Blazor that you might currently know where you use HTML.

This means you get to use the mix of XAML and HTML and code altogether and leverage the strength of each platform while still being able to access the devices' native capabilities. When you use XAML, it will still render to native controls depending on the platform that you are running on. When in a [BlazorWebView](#) you will get a hybrid HTML approach.

You can see a small code sample of mixing XAML and code in a razor file below.

```
<StackLayout>
    <Label FontSize="30"
        Text="@($"You pressed " + count + " times")" />
    <Button Text="+1"
        OnClick="@HandleClick" />
</StackLayout>
@code {
    int count;
    void HandleClick()
    {
```

```
        count++;
    }
}
```

You can find more information on the Blazor Mobile Bindings in the Microsoft Docs here: <https://docs.microsoft.com/mobile-blazor-bindings/>. While experimental, you can already use it with Xamarin.Forms, today!

BlazorWebView

I already talked a little bit about the **BlazorWebView**. This is a new component in .NET MAUI, you can load your (existing) Blazor web application inside of a native .NET MAUI app. Effectively with this, you can build a hybrid app.

With the **BlazorWebView**, you can take your existing Blazor app, including all the HTML and CSS and put that in a .NET MAUI project. With a minimum amount of configuration, you must wire up your Blazor app and suddenly it's inside of a native mobile app. The difference is that this is a XAML control that is specifically used to render Blazor inside of your mobile app. Without the Blazor Mobile Bindings described above, you will still write all your XAML (or code if you prefer that) as you normally would in Xamarin.Forms and .NET MAUI. An example of a **BlazorWebView** can be seen below.

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
              xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
              xmlns:b="clr-namespace:Microsoft.AspNetCore.Components.WebView.Maui;
              assembly=Microsoft.AspNetCore.Components.WebView.Maui"
              xmlns:local="clr-namespace:BlazorWebViewDemo"
              x:Class="BlazorWebViewDemo.MainPage"
              BackgroundColor="{DynamicResource PageBackgroundColor}">
    <b:BlazorWebView HostPage="wwwroot/index.html">
        <b:BlazorWebView.RootComponents>
            <b:RootComponent Selector="app" ComponentType="{x:Type local:Main}" />
        </b:BlazorWebView.RootComponents>
    </b:BlazorWebView>
</ContentPage>
```

Transitioning from Xamarin.Forms to .NET MAUI

If you have an existing Xamarin.Forms application, you might be wondering how hard the transition is going to be. The team working on this wants you to switch to .NET MAUI as soon as possible so they will do everything to make that transition as easy as possible.

Also, what is good to know is that, while the complete architecture has changed, the outer APIs are still largely the same. Your code will still be C# and still work, and your UI code or XAML is 99% the same between Forms and .NET MAUI. There will be some properties that are renamed, but not much.

How hard the exact conversion is going to be depends on the complexity of your app. If you have a lot of custom renderers, the transition might be harder. If you have been using a lot of third-party libraries, you must make sure that those libraries are compatible with .NET MAUI.

Upgrade Assistant

Already introduced earlier to .NET is the upgrade assistant. This tool will help you convert .NET solutions to a newer version. It's not specific to .NET MAUI, but it will support it. The biggest change between Xamarin.Forms and .NET MAUI is that you must use different namespaces. Replacing all of those is not hard, but not fun either. Luckily for us it's something that can be easily automated. That is exactly one of the things the upgrade assistant does.

Besides that, it will also convert your csproj file structure, detect if those third-party libraries have issues, and some other things. You can find more information about the upgrade assistant on the GitHub repository: <https://github.com/dotnet/upgrade-assistant>.

Custom Renderers

If you do have a lot of custom renderers, don't worry. The .NET MAUI team has also implemented a so-called compatibility layer (or compat for short). By adding this to your .NET MAUI app (whether it's new or a converted Xamarin.Forms app), you can easily reuse your existing renderers almost as-is.

Here again, you will have to rename some namespaces and maybe some other minor things, but you can reuse them without too much hassle. While this is great for a quick and easy transition of your app, ideally you would want to rewrite those renderers to the new handler architecture. But this will give you some extra time to do so while still being able to release new versions of your app.

In Closing

The big question after learning all of this is: **when can I play with it?**

The good news is: **right now!** There have been already a good number of previews released, each one adding more details of the actual implementation of the final product.

A word of caution though; there are a lot of moving parts now. As of this writing, there is also a Visual Studio 2022 preview, .NET 6 is still a preview, .NET MAUI is still a preview, the single-project approach is still being worked on, and WinUI has been in preview while developing .NET MAUI.

Editorial Note: .NET 6 released on Nov 8th, 2021. Read more here:

<https://devblogs.microsoft.com/dotnet/announcing-net-6/>

While you can try this out today, you must keep in mind that it's all in preview and the experience is a bit rough in terms of manual work. Of course, all of this will be resolved when we reach a general availability state.

In regard to the release, the first target date would be together with .NET 6 in November 2021. However, the development of .NET MAUI will take a bit longer and is now pushed back to the second quarter of 2022. .NET MAUI will still be available as preview during this time.

If you are working with Xamarin.Forms today, don't worry! The support windows for Xamarin.Forms is also pushed back in line with the release date of .NET MAUI.

More information on the delay can be found in this blog post: <https://devblogs.microsoft.com/dotnet/update-on-dotnet-maui/>.

You can find all the code and documentation to get you up and running at the .NET MAUI repository over on GitHub: <https://github.com/dotnet/maui>.

Useful Links

- SkiaSharp: <https://github.com/mono/SkiaSharp>
- Microsoft.Maui.Graphics: <https://github.com/dotnet/Microsoft.Maui.Graphics>
- Comet: <https://github.com/dotnet/Comet>
- WinUI: <https://microsoft.github.io/microsoft-ui-xaml/>
- Xamarin Forms to .NET MAUI Upgrade tool: <https://github.com/dotnet/upgrade-assistant>

If you want to stay up to date or learn in more detail about the features I have described, please have a look at my YouTube channel as well: <https://www.youtube.com/c/GeraldVersluis/>. Don't forget to subscribe and Happy Learning!

• • • • •



Gerald Versluis

Author

Gerald Versluis (@jfversluis) is a software engineer on the .NET MAUI team at Microsoft. Not only does he likes to code but also spreading some knowledge-as well as gaining - is part of his daytime job by speaking, creating videos (<https://youtube.com/GeraldVersluis>) and writing blogs (<https://blog.verslu.is>) or articles. Twitter: @jfversluis Website: <https://jfversluis.dev>



Technical Review

Damir Arh



Editorial Review

Suprotim Agarwal

Subodh Sohoni



AZURE PIPELINES CI / CD FOR DEVELOPERS

Introduction

Developers in a DevOps team have certain expectations from Continuous Integration / Continuous Deployment (CI / CD) pipelines. In this article, I will list these expectations and provide information on how Azure Pipelines, which is a part of Azure DevOps, fulfills these expectations.

During this discussion, I will also enumerate the responsibilities and constraints that developers have to follow for successful implementation of CI / CD in Azure DevOps. Although I may give some code examples using C#, the concepts and tools are applicable for any language used for writing code.



What do developers expect when they are part of DevOps Team?

DevOps team is a team of people with various capabilities that take the responsibility of development, testing, deployment and monitoring of an application or a product, under development.

When developers work as team members in a DevOps team, they are *collectively* responsible for the development of code.

Each developer writes code which may use or may depend upon code written by some other team member. A basic and essential service for such code integration is *Version Control*.

Over and above the services provided by version control, each developer may have the following expectations from the code developed by others:

1. Code should be compiled before it is shared for use or should compile without any issue.
2. Code should adhere to the agreed standards.
3. Code as a unit must not fail. Unit tests should be created and run to pass on that code.
4. Before it is shared, code should be reviewed by a competent authority to ensure that it does not have any obvious bugs.
5. All code dependencies should be checked to match these expectations.
6. If possible, it should be ensured that the code passes any functional tests created.

Each developer should ensure that the code that he / she has created and is going to share with other developers in the team, should meet these expectations.

In Azure DevOps, the services that help developers fulfil the above-mentioned expectations are **Build and Release**.

Build Pipelines for Continuous Integration (CI)

A build pipeline is a set of ordered activities that are run to create deployable output. These ordered set of activities are provided to us by a template in Azure Pipelines when a pipeline is created. We can make changes to those activities as desired.

At the core of the build pipelines, is the compilation process. Usually, this pipeline gets the code from the version control repository and compiles it as per the details provided in the project file or the solution file. Before the compilation, it has to go through some checks. These checks are as follows:

1. All the capabilities as mentioned in pipeline definition are met by the agent where the pipeline is going to run. A capability can be, for example, a specific version of .NET Framework present or JDK 8 installed. These capabilities can be system capabilities or user defined capabilities.
2. During the project compilation, code of detected dependencies needs to be compiled before the dependent code. If there are any project dependencies, then that code is either available in the same repository of version control, or it may also be available in another repository that is accessible and

included in the pipeline definition.

3. If there are packages like NuGet included in the project, these packages are to be downloaded to the agent where the compilation is to take place.
4. All the necessary tools like compilers, software for code analysis and unit testing are installed.

After all these checks are passed, the build pipeline compiles the projects in a Solution. If a single project (.CSPROJ) is selected, then it compiles only that project and its dependencies. All the other projects in the solution are ignored.

Once the compilation process is completed successfully (without any compilation errors), the pipeline performs quality checks as per the configuration done for that pipeline.

The following quality checks are possible:

1. **Static Code Analysis:** This part of the process ensures that the compiled code adheres to the rules that are specified as the standard within the organization. A tool for running such an analysis is installed along with Visual Studio. In the configuration of the project (.CSPROJ or .VBPROJ), we can specify a set of rules that should be used to evaluate the code. These rules are clubbed together to form *named rulesets*.

Following are the default rulesets:

- a. Basic Correctness Rules: Includes Managed Recommended Rules (for .NET) plus rules for logic errors and framework usage.
- b. Extended Correctness Rules: Basic Correctness Rules + some more rules for logic errors and framework usage.
- c. Managed Minimum Rules: Includes only four rules for critical managed code (.NET) problems.
- d. Managed Recommended Rules: Extends the Managed Minimum Rules ruleset with some more critical managed code problems.
- e. Basic Design Guidelines Rules: Includes Managed Recommended Rules plus rules for ensuring code is easy to read, understand, and maintain
- f. Extended Design Guidelines Rules: Includes Basic Design Guideline Rules (which includes Managed Recommended Rules) plus more maintainability rules that focus on naming
- g. Globalization Rules: Includes rules for globalization problems.
- h. Native Minimum Rules: Includes rules for critical problems in native code (C++)
- i. Native Recommended Rules: Includes Native Minimum Rules plus more rules for critical problems in native code.
- j. Mixed Minimum Rules: Includes rules for critical problems in C++ code for CLR
- k. Mixed Recommended Rules: Includes Mixed Minimum Rules plus more rules for critical problems in C++ code for CLR
- l. Security Rules: Includes rules for finding security vulnerabilities.

m. All Rules: Contains all available managed and C++ rules.

If Static Code Analysis is not enabled for a project or a solution, we can enable it externally from the pipeline definition. This is done by providing a value of “true” to msbuild argument RunCodeAnalysis (‘/p:RunCodeAnalysis=true’). In such a case, we cannot specify the ruleset to be used. By default, it is taken as Managed Minimum Ruleset.

By default, all rulesets treat non-compliance as warnings, so the build will always pass even if there are non-compliances in the code against some of the rules.

The standard rulesets that are provided in the box are not editable but we can create custom ruleset based upon those standard rulesets, where we can toggle the specific rules to be enabled or not.

We can also convert some of the compliance issues into errors so that the build fails when the pipeline is run. A custom ruleset, which is an extension of existing (out of box) ruleset, is stored in a file with extension ‘ruleset’ and that file is then stored under the version control. After creating such a custom ruleset, we can assign that to various projects.

In addition to Microsoft Static Code Analysis Tool, we can also configure build to run the code analysis using third party tools like “SonarQube / SonarCloud” or “ReSharper”.

2. Unit Testing:

There is a bit of confusion around Unit Testing in many organizations. These organizations consider unit testing as running an application manually to observe errors in the functionality.

It is not so.

A unit test is an automated test that executes a method written by the developer, and then compares the output that is generated (for example, return value), against the expected value of the output. If these values match or are as expected, then the test passes, otherwise it fails.

Unit test is code written using a language similar to the language of development.

Microsoft provides a framework for creating and running Unit Tests. These unit tests can be run either in Visual Studio or using MsTest.exe.

Build pipeline can leverage MsTest.exe to run the same unit tests that are part of the solution. It has an out of the box task named Visual Studio Test (VsTest v2.0) to run these tests.

Unit tests are searched and found using a mini match pattern like “***test*.dll”. This pattern specifies “Any DLL inside or under any folder below the working folder that has a word ‘test’ embedded in its name”. If unit test project is created using standard naming conventions, then all the unit tests that are created by the developer and pushed into the repo, adhere to this pattern.

Test results of these unit tests are shown as part of the build results. If any of the Unit Test fails when this task is run, then that build is also set to failed.

3. Code Metrics:

Every developer experiences pain while deciphering code that is written by someone else. These unpleasant encounters usually happen while doing enhancements or bug fixes in that code. Those developers who have felt the pain of cryptic code that is not easy to maintain, desires that the code he / she creates is easily maintainable in future.

A tool named Code Metrics gives an idea of the maintainability of code. Code Metrics provides a numerical value called Maintainability Index by analyzing the code. Value of Maintainability Index should be more than 60 for a well maintainable code. Higher the value of this index, more maintainable is the code.

This value is based upon analysis of code based upon following parameters:

- a. Length of code: We should try to maintain the length of every method that we write, to as minimum as possible. If the code of a method becomes lengthy, then it becomes difficult to understand and make any modifications to it. Length of the code is categorized into line of executable code and lines of source code.
- b. Depth of inheritance from *Object* class: In .NET, every class is directly or indirectly derived from an Object class. Every level of inheritance adds a little bit of complexity to the code by using overloading and overriding. Certain level of depth of inheritance is good and expected, but if this depth becomes too high, the complexity of the code increases. The code then becomes difficult to make any modifications to.
- c. Cyclomatic Complexity: It is a quantitative measure of the number of linearly independent paths through a program's source code. Higher the value of this parameter, lesser is the maintainability of that code.
- d. Class Coupling: This is a measure of how the classes are dependent on each other. If the coupling is tight, any change in one class will necessitate a change in another dependent class and a domino effect happens. This reduces the maintainability of the code.

Code Metrics can be calculated either in Visual Studio or while being built by [Azure Pipelines](#). We will focus on the latter case of computing the Code Metrics as part of the pipeline.

To enable a pipeline to run Code Metrics, we need to prepare a project for the same. In this preparation, we add a NuGet package to the project. This NuGet package is Microsoft.CodAnalysis.Metrics. After adding the package, we can commit and push the code to the repository on Azure Repos (another part of Azure DevOps). While we create the build pipeline, we will provide two MsBuild arguments to enable the Code Metrics to be computed. The MsBuild arguments to be given are:

```
'/t:Metrics /p:MetricsOutputFile=$(Build.ArtifactStagingDirectory)/  
CodeMetricsResults.xml'.
```

We will create the pipeline using a UI based wizard and or add the following snippet of code in the YAML pipeline definition (if you are using it).

```
- task: VSBuild@1  
  displayName: 'Build solution **\*.sln'  
  inputs:  
    msbuildArgs: '/t:Metrics /p:MetricsOutputFile=$(Build.  
      ArtifactStagingDirectory)/CodeMetricsResults.xml'  
    platform: '$(BuildPlatform)'  
    configuration: '$(BuildConfiguration)'
```

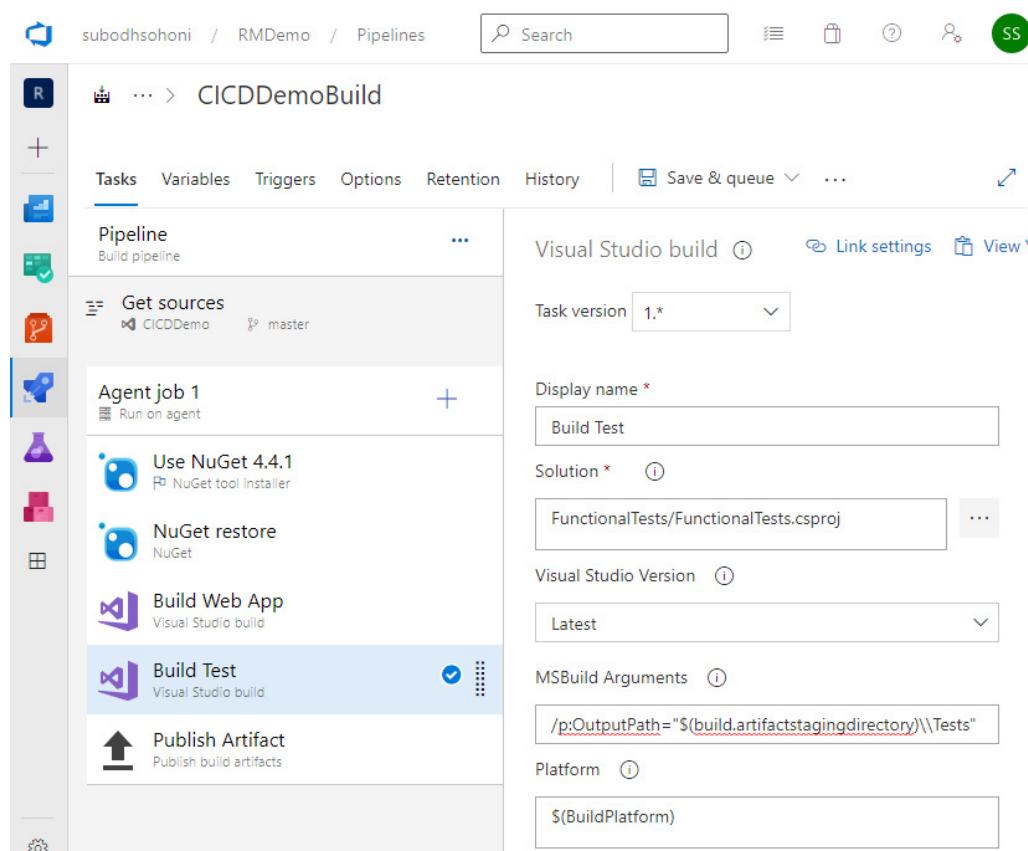
- 4. Security Code Analysis:** Developers should be writing code that is secure. It should not have vulnerabilities. Code should be analyzed for SQL Injection, LDAP Injections, Cross Site Scripting, Cryptography weakness etc. There are many tools that can be integrated into the build pipelines. The most popular tool is SonarQube (On-premises) and SonarCloud (Cloud based). There are tasks available in the Visual Studio Marketplace for including any of these tools in the build pipeline.

Create the build pipeline for Continuous Integration (CI)

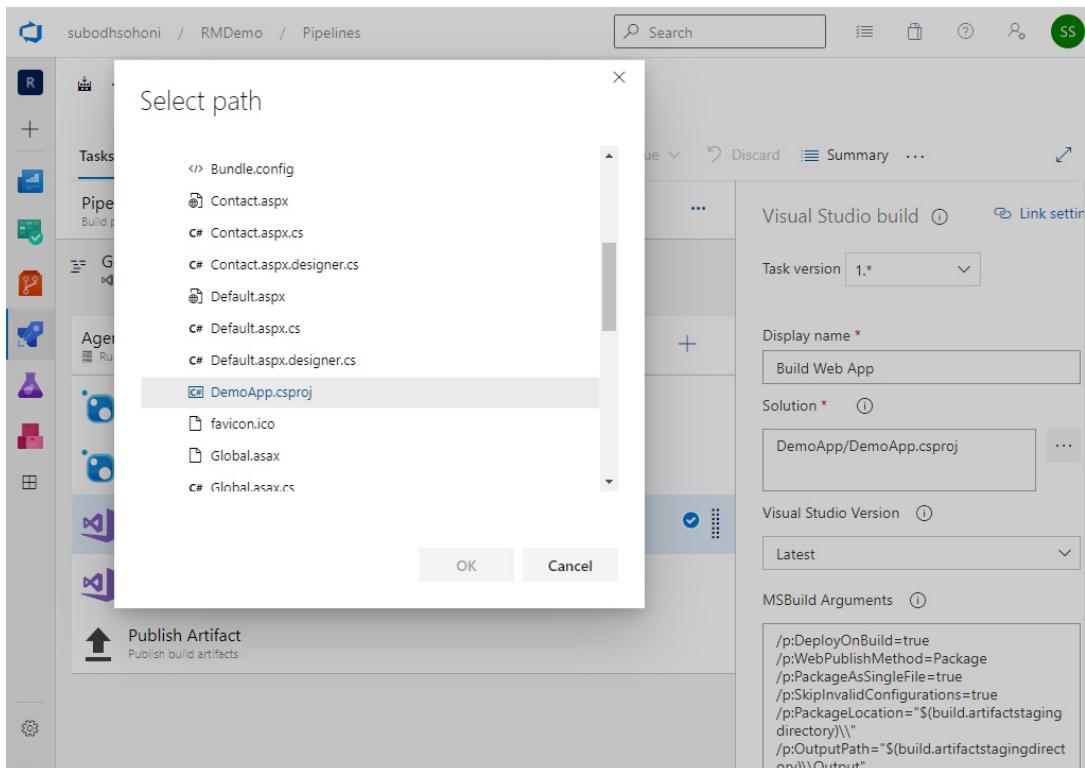
In this example, we will create a new ASP.NET Web Application project that will be built and deployed. As a first step in this walkthrough, we'll create a new project in Visual Studio 2019. For this project, we'll select the template of ASP.NET Web Application (.NET Framework) – C#. I selected Web Forms model, but you can select other models like MVC etc.

After making necessary changes in the code of the project, we will Commit and Push the code to Azure Repo. For knowing more about Azure Repos using Git in Azure DevOps, please read the article [Git in Azure DevOps](#) written by Gouri Sohoni.

Now we can create our build pipeline that will build a deployable package of our ASP.NET web application project. For more information about Azure Pipelines read my article [Demystifying Pipelines in Azure DevOps \(Build and Deployment\)](#). We will target the .CSPROJ instead of the entire solution to be built. To do so, in the Pipeline section of the pipeline that we are creating, click the hyperlink of “Unlink All”.

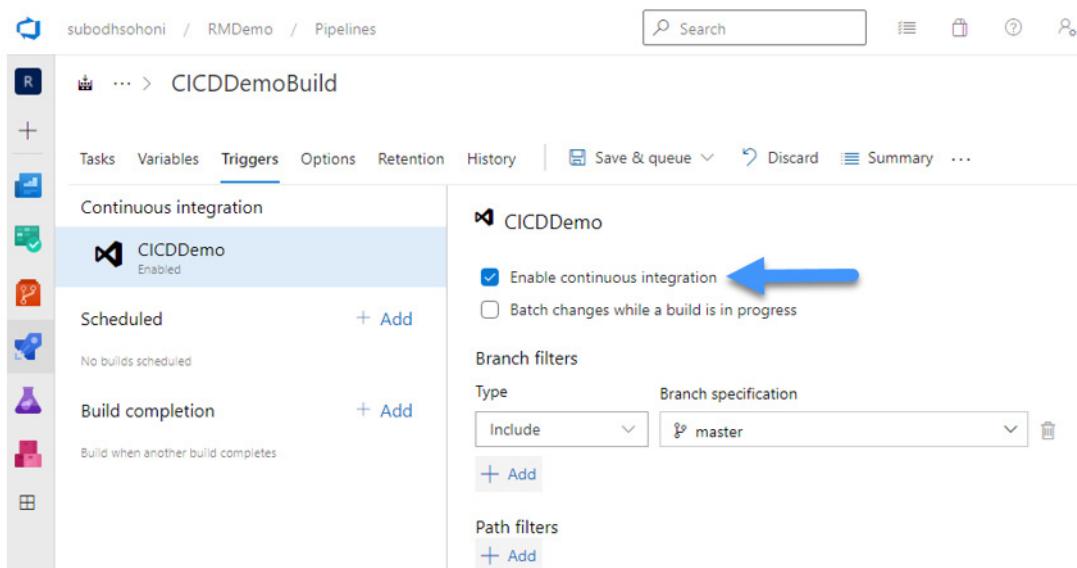


In the *Tasks* blade of VS Build, select the .CSPROJ file of your project by drilling down in the solution folder.



Add an Argument under MSBuild Arguments - `/p:OutputPath="$(build.artifactstagingdirectory)\\Output"`. This argument is required only when we are building a project and not the complete solution.

Under the Triggers section, enable the Continuous Integration (CI) trigger so that every time the code is pushed in the repository, it will trigger this build pipeline. CI is a very important part of the agile development process. It gives immediate feedback to the developer about quality and buildability of the code that is added to version control.



Now you can Save and Queue the build. It will create an artifact that contains the .ZIP file which is a deployable package of the web application.

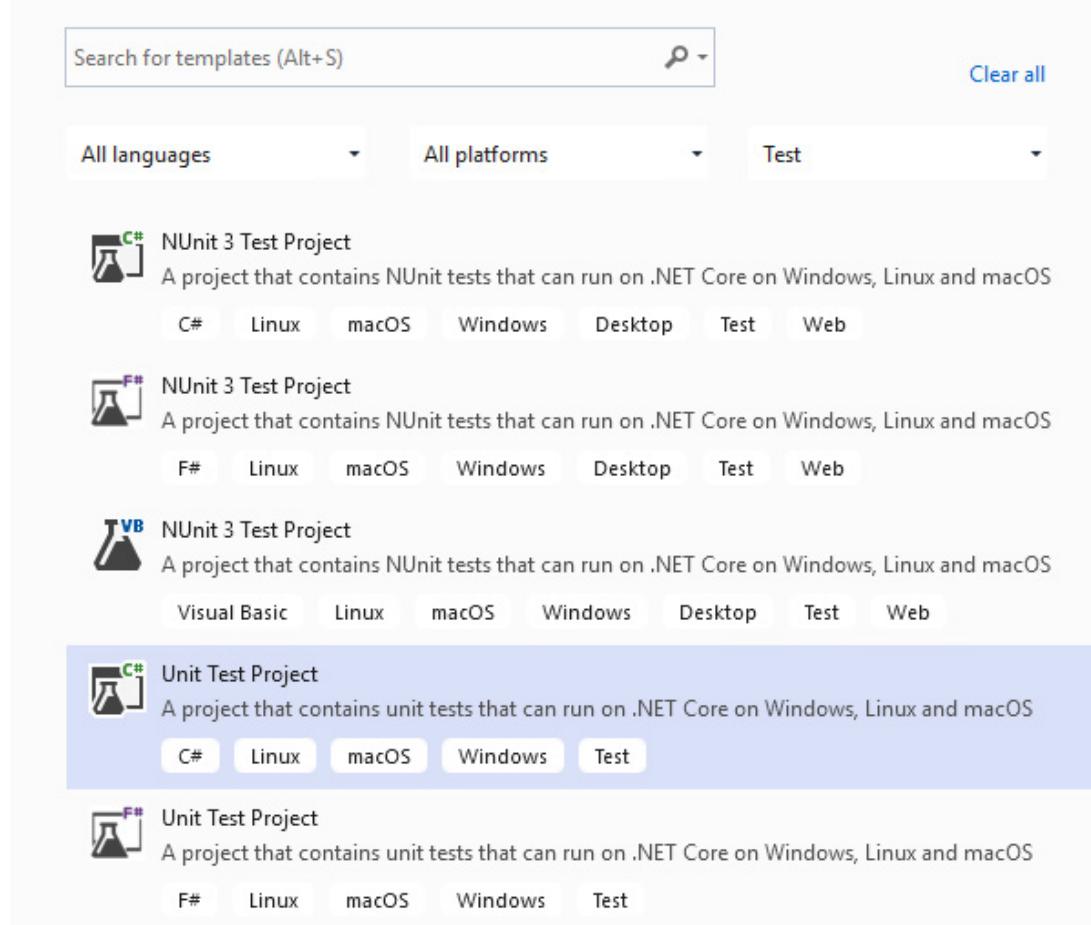
Automated Functional Testing using Selenium as part of Continuous Deployment (CD)

When the CI build is created and passed, we would like to automatically deploy the created application and run some functional tests on it. These functional automated tests are written using an appropriate framework. In this example we are going to use Selenium.

Running such functional tests, which may also be called a Regression Test Suite, ensures that there is no bug that has come into being while we modified the code of the application. This part of the process where the application is automatically deployed and then tested is called the [Continuous Deployment \(CD\)](#). For this article, we are going to focus on running the functional tests automatically when the application is deployed to Development or Testing stage.

To begin with, we need to write code for Automated tests. As an example, we will create a Selenium test to run it against a web application that we are creating.

In Visual Studio, add a project of the type Unit Test Project – C# to the same solution where your web application project exists.



For this project, add the following NuGet packages:

1. Selenium.WebDriver
2. Selenium.Support
3. Selenium.WebDriver.ChromeDriver – if you want to run the selenium test in Chrome browser

4. Selenium.WebDriver.GeckoDriver - if you want to run the selenium test in Mozilla
5. Selenium.WebDriver.MicrosoftDriver – if you want to run the selenium test in Microsoft Edge.
6. Microsoft.Edge.SeleniumTools – If you want to run the selenium test in Microsoft Edge Chromium.

Now we can create code for the test. Here's a sample code for the test where it opens a web page in Chrome browser and navigates to the home page of the application (To check that deployed application is running) before closing that instance of the browser:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using OpenQA.Selenium;
using OpenQA.Selenium.Chrome;
using System;

namespace SeleniumTests
{
    [TestClass]
    public class FunctionalTestsClass
    {
        static IWebDriver chromeDriver;
        [TestMethod]
        public void SeleniumUITest()
        {
            chromeDriver = new ChromeDriver(Environment.
GetEnvironmentVariable("ChromeWebDriver"));
            chromeDriver.Navigate().GoToUrl("https://demowebappss.azurewebsites.
net/");
            chromeDriver.Close();
        }
    }
}
```

Now, you can Save, Commit and Push the code to the repository on Azure DevOps.

The next activity is to build the code of the Test project to create binaries that will be used by release management to run the test. The test project Build can be done as part of the main solution build pipeline or can also be done using a separate build pipeline for itself.

We will follow a convention of adding the test project build along with the solution build. It keeps the test assemblies in the same artifact to make it easy to search and find.

Let us add another task of the type Visual Studio Build in the pipeline that we have already created. You can rename the Display name of the task to indicate that it is going to compile the test project. Select the .CSPROJ of the test project to build as the target. In the text box for MSBuild Arguments add the argument: /p:OutputPath="\$(build.artifactstagingdirectory)\\Tests".

The screenshot shows the Azure Pipelines interface for creating a build pipeline. The pipeline is named 'CICDDemoBuild'. It includes a 'Get sources' task for the 'CICDDemo' repository, a 'Build Web App' task, and a 'Build Test' task which is currently selected. The 'Build Test' task is configured to use 'Visual Studio build' with 'Latest' version and specific MSBuild arguments. The pipeline also includes 'NuGet restore' and 'Publish Artifact' tasks.

When we Save and Queue the build now, it will create the binary (.DLL) of the test assembly. We can now run this test as part of the release pipeline.

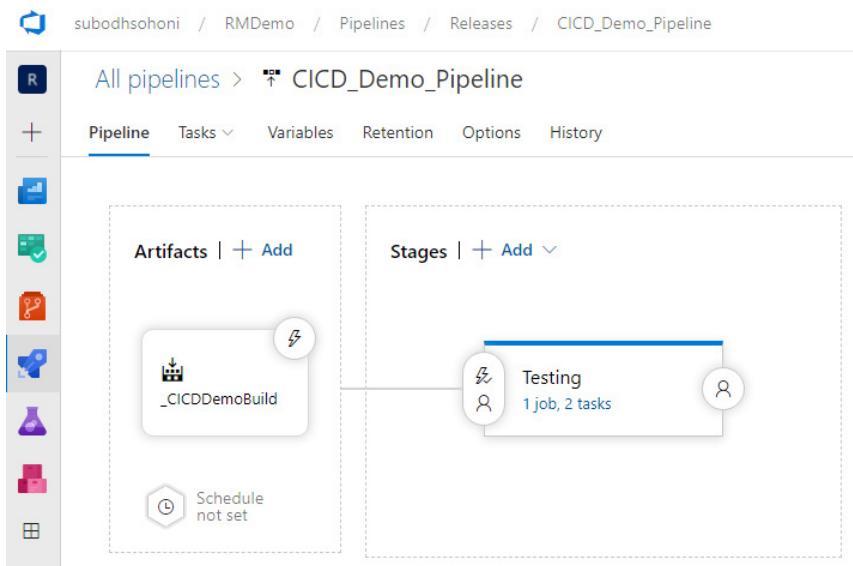
Release Pipeline for Continuous Deployment (CD)

Besides build pipelines, Azure Pipelines also covers Release Pipelines which are used for:

- Defining the stages through which the application may pass before getting deployed to production e.g., Development – Testing – UAT – Staging – Production.
- Provisioning the prerequisites of the application in various environments which are used in these stages.
- Deployment of the application to a stage when it is approved for that.
- Running the functional automated tests.

Let us now see how to create a release pipeline which will deploy the built application to the Test environment, which we assume is the first stage in our pipeline. After deploying the application, we will also run functional tests using Selenium.

When we create the release pipeline, a stage is automatically created for us. Let's rename that to "Testing". Now add the artifact that we had built. It contains the built application and may also contain the built assembly of the functional test. If the functional test is built using a separate build pipeline, then we also add the artifact that was created by that build pipeline. In this example, I am assuming it to be a single artifact that contains functional test along with the application.



In the first task of the stage, we will deploy the application to an Azure Web App Service. For more details about deployment of ASP.NET web application to Azure Web App Service, please refer to Platform as a Service (PaaS) section of the article [Developing Cloud apps in .NET](#) written by Damir Arh. We will add a task for testing in the same stage.

The task to be added for testing is Visual Studio Test (V2.0). We don't need to make any changes in the parameters of this task.

The screenshot shows the 'Tasks' configuration screen for the 'Testing' stage. The 'VsTest - testAssemblies' task is selected. The configuration includes: 'Run on agent' (selected), 'Task version' (2.*), 'Display name' (VsTest - testAssemblies), 'Test selection' (Test assemblies), 'Select tests using' (Test assemblies), 'Test files' (*.test*.dll, !**\TestAdapter.dll, !**\obj**), 'Search folder' (\$System.DefaultWorkingDirectory), and 'Test results folder' (\$(Agent.TempDirectory)\TestResults).

In the task, the build engine will search and find any assemblies under the artifact that has a word "test" embedded in it. When it finds those assemblies, it will run tests written in those. Results of the tests are stored under the specified folder and reported through the pipeline summary.

The screenshot shows the Azure DevOps Pipeline Testing summary for the CICD_Demo_Pipeline under Release-2. It displays the following metrics:

- Total tests: 1
- Pass percentage: 100%
- Run duration: 35s 667ms
- Tests not reported: 0
- Test status breakdown: 1 Passed, 0 Failed, 0 Others

After the application deployment, when the functional test passes, the developer knows that the code changes that they have made has not broken the application.

Summary

In this article, we have explored the significance of Continuous Integration – Continuous Deployment, to developers.

We saw various tools that can be used to ensure code quality during the build. We have also seen how we can run regression tests as part of the deployment process.

CI/CD ensures that developers get immediate feedback about quality and buildability of the code, as well as to know that there are no obvious bugs that have come into being during the recent code changes made by the developer.



Subodh Sohoni

Author



Subodh is a Trainer and consultant on Azure DevOps and Scrum. He has an experience of over 33 years in team management, training, consulting, sales, production, software development and deployment. He is an engineer from Pune University and has done his post -graduation from IIT, Madras. He is a Microsoft Most Valuable Professional (MVP) Developer Technologies (Azure DevOps), Microsoft Certified Trainer (MCT), Microsoft Certified Azure DevOps Engineer Expert, Professional Scrum Developer and Professional Scrum Master (II). He has conducted more than 300 corporate trainings on Microsoft technologies in India, USA, Malaysia, Australia, New Zealand, Singapore, UAE, Philippines and Sri Lanka. He has also completed over 50 consulting assignments - some of which included entire Azure DevOps implementation for the organizations.

He has authored over 90 tutorials on Azure DevOps, Scrum, TFS and VS ALM which are published on www.dotnetcurry.com. Subodh is a regular speaker at Microsoft events including Partner Leadership Conclave. You can connect with him on [LinkedIn](#).



Technical Review
Gouri Sohoni



Editorial Review
Suprotim Agarwal



Damir Arh

CREATING AND CONSUMING A C# SOURCE GENERATOR (.NET 5 AND C# 9)

In this article, I explain how source generators work and give examples of using existing source generators and creating new ones.

```
self.debug = debug
self.logger = logging.getLogger(__name__)
if path:
    self.file = open(os.path.join(path, 'fingerprint.log'), 'a')
    self.file.seek(0)
    self.fingerprints.update(self._load_fingerprints())
    self._write_fingerprints()

@classmethod
def from_settings(cls, settings):
    debug = settings.getbool("DEBUG")
    return cls(job_dir(settings))

def request_seen(self, request):
    fp = self.request_fingerprint(request)
    if fp in self.fingerprints:
        return True
    self.fingerprints.add(fp)
    if self.file:
        self.file.write(fp + os.linesep)
```

Source generators are a new metaprogramming feature in C# introduced in the .NET 5 and C# 9 timeframe.

Metaprogramming is a general name for programming approaches that take other programs or their source code as input to modify or extend them. It is mostly used to reduce the amount of plumbing or repetitive code that needs to be written manually.

Before source generators, there were some ways to use metaprogramming with C#:

- [Reflection](#) is probably the best known. It is the ability of the .NET runtime to examine programs at runtime. It is commonly used to inspect [attributes](#) applied to symbols in our code (e.g., to entity classes to describe validation rules or object-relational mappings), but also allows us to dynamically access type members at runtime or determine available types (e.g., for registration with dependency injection).
- [T4 \(Text Template Transformation Toolkit\) templates](#) were originally introduced in the early days of Entity Framework as a tool to generate code from the entity model. Although more versatile than that, they never caught on. To some extent, this might be because of the limited built-in support in Visual Studio.
- IL weaving is a technique for modifying the assemblies generated from the source code at the end of the build process. It is not directly supported by the C# compiler or by Visual Studio, but there are third-party tools such as [Fody](#) and [PostSharp](#) for it. It is most commonly used to reduce repetitive boilerplate code, for example when implementing the [INotifyPropertyChanged](#) interface.

Although source generators do not directly replace any of the above approaches, they can be a good alternative for many tasks for which they are commonly used.

Understanding source generators

Source generators are a feature of the [Roslyn C# compiler](#) and as such perform compile-time metaprogramming.

A source generator receives a model of the existing source code (and optionally other files) as input and generates additional source code as output. The generated code is included in the compilation process and becomes an integral part of the compiled assembly.

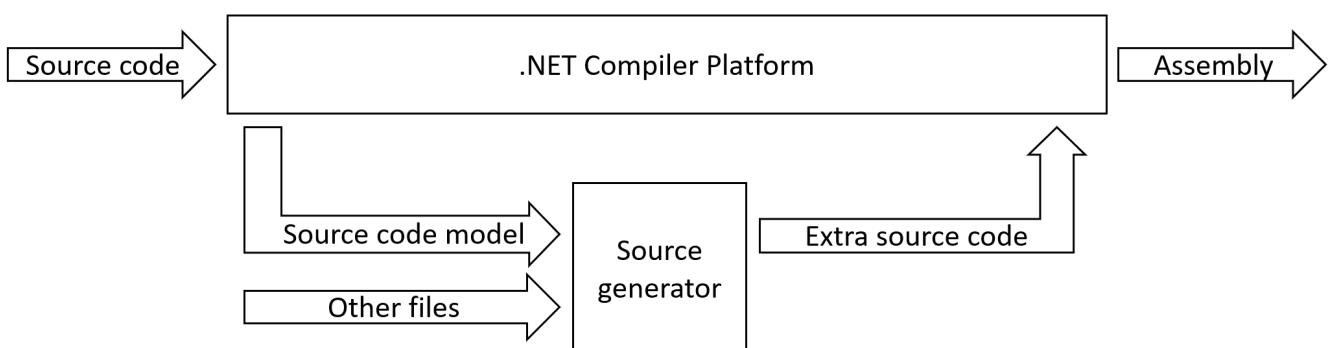


Figure 1: Inputs and outputs of a source generator

The generated source code usually depends on the inputs. For example, for each type that meets certain criteria, additional code is generated to extend its functionality. This hopefully saves developers time in the long run, as they don't have to write and maintain this code.

However, there are some limitations to source generators that can make them useless in certain scenarios:

- They can only generate additional source code, but cannot modify existing code. This makes it impossible, for example, to generate plumbing code for the implementation of the [INotifyPropertyChanged](#) interface, as is usually done by IL weaving tools.

- There's no way to specify the order in which multiple source generators are applied to the same source code. Source code generated by one source generator isn't included in the input of other generators. This reduces the risk of multiple generators conflicting, but makes it impossible to further augment code from one generator by another one.

Because source generators are tightly integrated into the Roslyn compiler platform, code editors can treat the generated code like any other source code. Developers can review and debug it; they just can't change it.

Using a source generator

As an extension of the Roslyn C# compiler, source generators are similar in many ways to their predecessors - **diagnostic analyzers**. They depend on the Roslyn platform SDK and are distributed as NuGet packages.

You can learn more about diagnostic analyzers in my previous DotNetCurry Magazine articles: [Diagnostic Analyzers in Visual Studio 2015](#) and [Create Your First Diagnostic Analyzer in Visual Studio 2015](#).

Just like any other NuGet package, they are distributed through the [NuGet Gallery](#). Unfortunately, there is no easy way to limit the NuGet Gallery search to source generators. This makes it nearly impossible to get an overview of the currently available source generators.

If you are looking for a specific source generator, your best bet is to check the [C# Source Generators GitHub repository](#). To my knowledge, this is currently the most complete list of source generators. It also includes some links to documentation and other resources to help you get started. A shorter, more curated list is also included in the [Awesome Roslyn](#) GitHub repository.

The list of source generators is not very long yet, so it's best to browse it to see what's available. To give you an idea of what to expect, I'll list some of them here as well:

- Many source generators replace run-time reflection with compile-time code generation in various scenarios: serialization ([JsonSrcGen](#), [StackXML](#)), dependency injection ([StrongInject](#), [Jab](#)), object mapping ([MapTo](#)), reading assembly metadata ([ThisAssembly](#)), etc.
- Others focus on generating general boilerplate code: [WrapperValueObject](#) Generator.Equals, [Cloneable](#), etc.

I chose [MapTo](#) as an example of using a source generator in your own project. It is similar to the [AutoMapper](#) library and simplifies the task of mapping from one type of object to another (entity object, data transfer object, domain object, etc.).

This means that you typically have a pair of classes with similar, if not the same, properties:

```
public class Person
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

```

public class PersonDto
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

```

Instead of writing the mapping code between the two types by hand, you can install the [MapTo NuGet package](#) in your project and add an attribute to the target class:

```

[MapFrom(typeof(Person))]
public partial class PersonDto
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

```

You must also mark the class as `partial`, which is a common requirement for many source generators because it allows additional code for the same class to be placed in a different source code file (created by the source generator). There are other attributes you can use to further customize the mapping, but that's not the focus of this article, and you can read about them in the [MapTo source generator documentation](#) if you decide to use it.

The important thing is that the source generator will create the mapping code for you based on this attribute, and expose it in several ways. One of them is the mapping extension method for the source class, which allows you to write code like this:

```
var personDtos = personEntities.Select(person => person.ToPersonDto());
```

Of course, you'll need to read the documentation to find out what code is generated for you. However, the compiler platform, and thus the code editor, are fully aware of the generated code, which means you can see the generated methods in IntelliSense popups just like any other method:

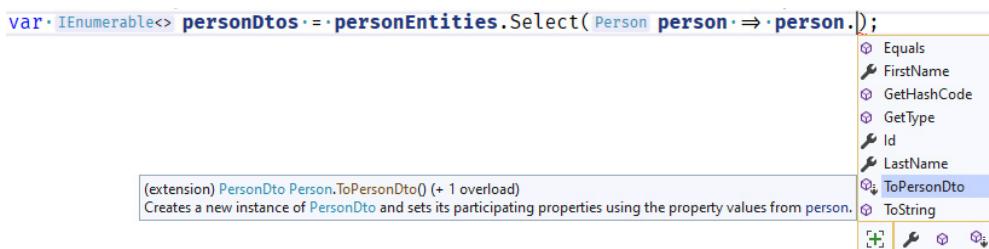


Figure 2: Generated extension method in IntelliSense popup

You can also navigate to the method implementation to see the generated source code:

This file is auto-generated by the generator 'MapTo.MapToGenerator' and cannot be edited.

```

Miscellaneous Files -> SrcGenClient.Api.Dto.PersonToPersonDtoExtensions.cs -> ToPersonDto(Person person)

43     ....
44     ....
45     ....
46     ....
47     ....
48     ....
49     ....
50     ....
51     ....
52     ....
53     ....
54     ....
55     ....
56

```

The screenshot shows a code editor window with the title bar "Miscellaneous Files -> SrcGenClient.Api.Dto.PersonToPersonDtoExtensions.cs -> ToPersonDto(Person person)". The code itself is mostly empty, with some XML comments and a single line of code: `return person == null ? null : new PersonDto(person);`. A red dot on the left margin indicates the current line of code.

Figure 3: Generated source code in Visual Studio 2019

The code editor window is clearly marked to contain auto-generated code that cannot be edited. However, you can still set breakpoints and debug it like any other code you have written.

The behavior described is fully supported in the latest versions of Visual Studio 2019 and JetBrains Rider. In Visual Studio Code, I could not get the debugger to stop at the breakpoints set in the generated code, but everything else (IntelliSense, navigate to implementation, read-only source code) worked the same. In Visual Studio for Mac, the method showed up in IntelliSense, but none of the other functions worked.

Creating a source generator

The next step after using an existing source generator is to create one yourself that meets your code generation needs.

A project template for a source generator in Visual Studio 2019 is not available yet, but the closest approximation I could find is a [template project on GitHub](#) created by a member of the Roslyn team. You can download its code as a starting point.

But you can also start with an empty class library project. That's the approach I'll take in this article.

When creating the class library, make sure you use .NET Standard 2.0 as the target framework. You will also need to install the following two NuGet packages: [Microsoft.CodeAnalysis.Analyzers](#) and [Microsoft.CodeAnalysis.CSharp](#).

Now you are ready to write your first source generator. To do so, implement the `ISourceGenerator` interface and annotate the class with the `Generator` attribute. The interface declares two methods:

- `Initialize` is called first and allows you to register hooks to get information about various syntax elements in the input code as it is processed. This part is completely optional and you can leave the method empty if you do not need it.
- `Execute` is the method responsible for code generation. It must add at least one source code file, otherwise you will get a warning when the compiler executes your source generator.

Let us start with a simple source generator that creates a read-only property with the time of the latest build:

```
[Generator]
public class BuildInfoGenerator : ISourceGenerator
{
    public void Execute(GeneratorExecutionContext context)
    {
        var buildTime = DateTime.Now;

        string source = $@"
using System;

namespace Build
{{{
    public static class Info
    {{
        public static DateTime Timestamp {{ get; }} = new DateTime({{buildTime.
Ticks}});
    }
}}
```

```

        }
    }
};

        context.AddSource("BuildInfo", source);
    }

    public void Initialize(GeneratorInitializationContext context)
    {
    }
}

```

As you can see, the source code is contained in the class as a literal string. The indentation of the code is important because this is exactly how it will look to the developer using your source generator when they view it.

Note that I use a verbatim interpolated string to make it easier to format the code and insert dynamic values into the code. However, as the size and complexity of the generated code increases, this may not be sufficient. You should look at [the IndentedTextWriter class](#) to help you properly indent the generated code.

The generated code is added to the compilation using the `AddSource` method. Multiple source code files can be added this way. For each one of them, a unique name must be specified as the first method parameter.

To test the source generator, you can create another project in the same solution and reference the source generator project in it. You must manually modify the generated reference in the project file by adding the `OutputItemType` and `ReferenceOutputAssembly` attributes to the `ProjectReference` element:

```

<ItemGroup>
    <ProjectReference Include="..\BuildInfo.Generator\BuildInfo.Generator.csproj"
        OutputItemType="Analyzer"
        ReferenceOutputAssembly="false" />
</ItemGroup>

```

Your program can now use the code generated by the source generator in your code:

```

static void Main(string[] args)
{
    Console.WriteLine($"Timestamp: {Build.Info.Timestamp}");
}

```

If it does not work as expected, make sure that you are using the latest version of Visual Studio. You may need to restart Visual Studio after adding the project reference.

Using other NuGet packages for code generation

As the source generator becomes more complex, it becomes more likely that you will want to use code from other NuGet packages in its code. Of course, you can install NuGet packages in your source generator to use in its code. The project will also compile without errors.

However, if you try to use the generator from another project, it will fail to load the referenced assemblies. To fix this, you need to add more entries to the project file of your source generator:

- Add the `GeneratePathProperty` and `PrivateAssets` attributes to the `PackageReference` element of the NuGet package your source generator needs (`CliWrap` in my case). This informs the consuming project that it does not need this package reference to compile, and creates an MSBuild property with the path to the NuGet package installation location which you are going to use in the next step.

```
<ItemGroup>
  <PackageReference Include="CliWrap"
    Version="3.3.2"
    GeneratePathProperty="true"
    PrivateAssets="all" />
</ItemGroup>
```

- Specify the assembly or assemblies that your source generator needs as dependencies at runtime. In the code snippet below, `$(PKGCliWrap)` is an MSBuild property that was generated as a result of the `GeneratePathProperty` attribute above. It contains the path to the `CliWrap` package folder on your machine, e.g. `c:\Users\Damir\.nuget\packages\cliwrap\3.3.2`.

```
<PropertyGroup>
  <GetTargetPathDependsOn>$(GetTargetPathDependsOn);GetDependencyTargetPaths</GetTargetPathDependsOn>
</PropertyGroup>

<Target Name="GetDependencyTargetPaths">
  <ItemGroup>
    <TargetPathWithTargetPlatformMoniker Include="$(PKGCliWrap)\lib\netstandard2.0\CliWrap.dll" IncludeRuntimeDependency="false" />
  </ItemGroup>
</Target>
```

With these changes, you can safely use the NuGet package in your source generator and be sure that it will run without errors.

I used the `CliWrap` NuGet package to call the Git executable to get the SHA hash of the current Git commit at build time:

```
private async Task<string> GetGitSha(string path)
{
    var stdOutBuffer = new StringBuilder();
    var stdErrBuffer = new StringBuilder();
    try
    {
        await Cli.Wrap("git")
            .WithArguments("rev-parse --short HEAD")
            .WithWorkingDirectory(path)
            .WithStandardOutputPipe(PipeTarget.ToStringBuilder(stdOutBuffer))
            .WithStandardErrorPipe(PipeTarget.ToStringBuilder(stdErrBuffer))
            .ExecuteAsync();
        return stdOutBuffer.ToString().Trim();
    }
    catch
    {
        return "N/A";
    }
}
```

For this to work, the working directory for the Git executable must be set to the project folder. I could not

find the project folder path in the context passed from the compiler to the source generator, so I used this as an opportunity to use another feature of the source generator API: accessing selected properties from the consumer project file. Any `CompilerVisibleProperty` declared in it becomes accessible to the source generator:

```
<ItemGroup>
  <CompilerVisibleProperty Include="MSBuildProjectDirectory" />
</ItemGroup>
```

`MSBuildProjectDirectory` is a built-in MSBuild property that contains the path to the project directory. The XML above makes it accessible to the source generators. It can be accessed and used with the following code:

```
context.AnalyzerConfigOptions.GlobalOptions.TryGetValue(
  "build_property.MSBuildProjectDirectory",
  out var path);
```

I then pass the path to the `GetGitSha` method and use its return value in the generated code:

```
var gitSha = this.GetGitSha(path).Result;

string source = $@"
using System;

namespace Build
{
    public static class Info
    {
        public static DateTime Timestamp {{ get; }} = new DateTime({buildTime.
Ticks});
        public static string GitSha {{ get; }} = "{$gitSha}";
    }
}
```

Since no SHA is provided in case the `CompilerVisibleProperty` element is missing from the consumer project file, it is a good idea to add a diagnostic to inform the developer of the problem. It works the same way as in diagnostic analyzers:

- First, create a descriptor. Usually as a private static field:

```
private static readonly DiagnosticDescriptor MissingPropertyWarning = new
DiagnosticDescriptor(
  id: "BUILDINFOGEN001",
  title: "Missing CompilerVisibleProperty",
  messageFormat: "Generator requires MSBuildProjectDirectory
CompilerVisibleProperty to get Git SHA",
  category: "BuildInfoGenerator",
  defaultSeverity: DiagnosticSeverity.Warning,
  isEnabledByDefault: true);
```

- Then, when needed, you can report the diagnostic from the `Execute` method of your source generator:

```
if (!context.AnalyzerConfigOptions.GlobalOptions.TryGetValue(
  $"build_property.MSBuildProjectDirectory",
  out var path))
```

```

{
    context.ReportDiagnostic(
        Diagnostic.Create(MissingPropertyWarning, Location.None));
}

```

This now also allows you to print the Git commit SHA from the consuming project:

```

static void Main(string[] args)
{
    Console.WriteLine($"Timestamp: {Build.Info.Timestamp}");
    Console.WriteLine($"Git SHA: {Build.Info.GitSha}");
}

```

If you have trouble getting everything to work, you can check the full source code of this source generator in [my GitHub repository](#).

The techniques described above are only a small subset of what you will most likely need to know to create a useful source generator. The best source to learn more about it is the [Source Generators Cookbook](#), which is written and regularly updated by the Roslyn development team.

Publishing a source generator

To use the source generator from other projects (not in the same solution as the source generator project) and share it with other developers, you need to create a NuGet package for it.

To do this, you must add at least the following properties to the source generator project file:

```

<PropertyGroup>
    <GeneratePackageOnBuild>true</GeneratePackageOnBuild>
    <IncludeBuildOutput>false</IncludeBuildOutput>
</PropertyGroup>

```

This enables NuGet package generation on build and specifies that the source generator assembly should not be included in the NuGet package in the default location, where it would be treated as a regular dependency by the consuming project.

Instead, the source generator assembly must be placed in the `analyzers/dotnet/cs` folder, where it will be recognized as a diagnostic analyzer (or specifically as a source generator based on the class in the assembly):

```

<ItemGroup>
    <None Include="$(OutputPath)\$(AssemblyName).dll"
        Pack="true"
        PackagePath="analyzers/dotnet/cs"
        Visible="false" />
</ItemGroup>

```

All other assemblies used by the source generator must be included in the same folder. In my case, this is the `CliWrap` assembly:

```

<ItemGroup>
    <None Include="$(PKGCliWrap)\lib\netstandard2.0\CliWrap.dll"
        Pack="true"

```

```
    PackagePath="analyzers/dotnet/cs"
    Visible="false" />
</ItemGroup>
```

Before publishing the generated NuGet file to the gallery, you should also add the NuGet metadata properties like **PackageId**, **Version**, **Authors**, **Company** and **others** to better describe your package.

Conclusion

Source generators are a great addition to the existing selection of metaprogramming techniques in C#. This article explored how they work and their limitations. I explained how to use a source generator in your own project, how to view the generated code, and how to debug it.

This article also discussed the process of creating and testing your own source generator and provided examples of several common development techniques, including referencing other NuGet packages, reading project properties, and creating diagnostics.

• • • • •



Damir Arh
Author



Damir Arh has many years of experience with software development; from complex enterprise software projects to modern consumer-oriented mobile applications. Although he has working knowledge of many different languages, his favorite one remains C#. He is a proponent of test-driven development, continuous integration, and continuous deployment. He shares his knowledge by speaking at local user groups and conferences, blogging, and writing articles. He is an awarded Microsoft MVP for Developer Technologies since 2012. You can find his blog at <https://www.damirscorner.com/> Twitter: @DamirArh <https://twitter.com/DamirArh>.



Technical Review
Yacoub Massad



Editorial Review
Suprotim Agarwal

Daniel Jimenez Garcia



SERVER-SIDE JAVASCRIPT FOR .NET DEVELOPERS II

WEB FRAMEWORKS, EXPRESS AND FASTIFY

Node.js wouldn't have been as successful as it is if everyone creating a web application was directly using its HTTP module.

Building on the base library, a number of frameworks appeared over the years. They provided developers with features such as routing or error handling, making the task to build a web server, more approachable.

In this article, we will take a look at two specially selected frameworks – Express and Fastify.

Express has been the most popular framework and has been used for years, and for good reasons! Even though its aging compared to the newer frameworks, there are two reasons why I cannot ignore it.

The first reason is that it remains widely used. And the second is that it influenced the newer frameworks that came after it. Knowing Express, with its strengths and shortcomings, will make it easier to pick other frameworks!

The second framework is Fastify, a young framework designed with first-class support for modern JavaScript features such as `async/await` or ES modules. It has been built from the ground up, extracting the good ideas from earlier frameworks such as Express and hapi, fixing some of their shortcomings. To top it off, it offers top-notch performance and developer

experience.

If you were to start creating web applications and services with Node.js, these are the two frameworks I would recommend getting started with. But that doesn't mean the likes of [hapi](#) or [koa](#), which I won't be able to cover, aren't worth a look!

You can find the examples discussed through the article on [GitHub](#).

For the most part I will leave TypeScript outside of the article. That's not to say these frameworks do not support TypeScript, in fact Fastify has first-class TypeScript support. If you are a TypeScript user, you should have no trouble finding tutorials for using TypeScript with these frameworks.

Express.js, a battle tested framework

Let's begin by looking at [Express.js](#), one of the first and arguably the most popular Node.js web framework during the last few years. Compared with a framework such as ASP.NET Core, Express is a much more lightweight and minimalistic framework, that leaves plenty up to the developer and/or the community.

If you run into trouble or just want to follow along by browsing code, check the “express-example” project on GitHub.

First steps with Express

To begin with, initialize a new empty Node project like we did in the getting started section of Part 1 of this series (Page 6) by running `npm init` on a new folder. Then simply install express:

```
npm install --save express
```

As in the first article of the series, let's also install nodemon with `npm install --save-dev nodemon`. Then add the following script to `package.json` so you can run the project with `npm run dev`:

```
"dev": "nodemon --ignore 'test/**/*' ./index.js",
```

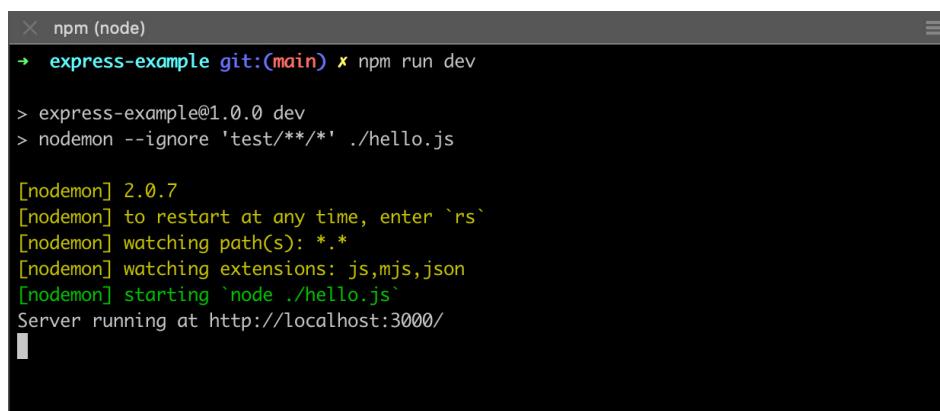
Let's now create the `index.js` file, writing the simplest Express application:

```
const express = require('express');
const process = require('process');

const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send(`Hello world!
  The current time is ${ new Date() }
  and I am running on the ${ process.platform } platform
  `);
});

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

A screenshot of a terminal window titled 'npm (node)'. The command 'express-example git:(main) x npm run dev' is entered. The output shows the execution of 'nodemon --ignore 'test/**/*' ./hello.js'. Nodemon starts and logs the message '[nodemon] 2.0.7', followed by '[nodemon] to restart at any time, enter `rs`', '[nodemon] watching path(s): *.*', '[nodemon] watching extensions: js,mjs,json', '[nodemon] starting `node ./hello.js`', and finally 'Server running at http://localhost:3000/'.

```
npm (node)
→ express-example git:(main) x npm run dev

> express-example@1.0.0 dev
> nodemon --ignore 'test/**/*' ./hello.js

[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node ./hello.js`
Server running at http://localhost:3000/
```

Figure 1, running the hello world Express application with nodemon

Looking at the code, this looks very similar to the example in the first article using the `http` module. However, notice the introduction of the `app` object, and the definition of a handler function for a specific route `'/'` and HTTP verb, `GET`. Also note how by using `res.send`, express is setting the status code and the content type header for us.

Given that functionality is almost the same, we can test it using almost the exact same integration test we used in the first article (Page 6). First, we need to refactor the code a little bit, so the creation of the *express application* is separated from binding the HTTP server to a port. Create a file `/server/app.js` and move the initialization of the app object:

```
const express = require('express');
const process = require('process');

const app = express();

app.get('/', (req, res) => {
  res.send(`Hello world!
  The current time is ${ new Date() }
  and I am running on the ${ process.platform } platform
  `);
});

module.exports = app;
```

Next update the `index.js` file to simply import the app module:

```
const app = require('./server/app');
const port = 3000;
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

Then install Jest and supertest:

```
npm install --save-dev jest supertest
```

...and create a `/test/integration/server.spec.js` test file. Its contents are almost identical to the one we used during the first article. The only differences are how the server is started/closed (since we use an Express app rather than Node's `http` module) and the expected Content-Type header (since we haven't overridden the default value set by Express):

```
const { beforeAll, afterAll } = require("@jest/globals");
const supertest = require('supertest');
const server = require('../..../server/app.js');

describe('the server', () => {
  let request;
  let listener;
  beforeAll(() => {
    listener = server.listen(0); // start server on any available port
    request = supertest(listener);
  });
  afterAll(done => {
    listener.close(done);
  });
});
```

```

test('GET / returns a helloworld plaintext', async () => {
  const res = await request
    .get('/')
    .expect('Content-Type', 'text/html; charset=utf-8')
    .expect(200);

  expect(res.text).toMatch(/Hello world!\s+The current time is .*\s+and I am
running on the .* platform/);
});
});

```

Finally copy from the first article the script `npm run test:integration` into package.json, and you will be able to test your express application.

```
"test:integration": "jest --silent \"test/integration/.*\\".spec\\.js\""
```

```

..press-example (-zsh)
→ express-example git:(main) ✘ npm run test:integration
> express-example@1.0.0 test:integration
> jest --silent "test/integration/.*.spec.js"

PASS | test/integration/server.spec.js

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.225 s
→ express-example git:(main) ✘

```

Figure 2, running integration tests for an Express application

Organizing your routes and endpoints

The `app.get('/', ...)` route handler gives us a hint at how Express allows you to organize your application using `routing`, the most important feature Express provides.

The express application object exposes a method for each HTTP verb, such as `app.get`, `app.post`, etc. Each of these methods lets you define one or more handler functions, which are invoked in order:

```

app.get('/', 
  (req, res, next) => {
    console.log('I am invoked before the next handler!');
    next();
},
  (req, res) => {
    res.send('Hello world!');
});

```

Express is designed around the idea of associating a request path such as `'/'` with a number of request handling functions. Each of these handlers can either return a response back to the client or call the next handler. Don't be surprised if this sounds very similar to the [ASP.NET Core](#) request pipeline and middleware!

You can keep adding routes to the root express `app` object. For example, you could define a CRUD API for

customers such as:

```
app.get('/customer', (req, res) => { ... });
app.get('/customer/:id', (req, res) => { ... });
app.post('/customer', (req, res) => { ... });
```

Sometimes you might want to further organize your application and group several of these routes into its own file, which can then be imported by the root application. In these cases, you can use express routers to bundle a group of routes. For example, create a new file `customer.js` with a simple CRUD API:

```
const express = require('express')
const router = express.Router()

let customers = [
  {id: '111', name: 'Foo'},
  {id: '222', name: 'Bar'}
];

router.get('/', (req, res) => {
  res.send(customers);
});

router.get('/:id', (req, res) => {
  const customer = customers.find(c => c.id == req.params.id);
  if (!customer) res.sendStatus(404);
  res.send(customer);
});

router.post('/', (req, res) => {
  customers = [...customers, req.body];
  res.sendStatus(201);
});

module.exports = router;
```

...and then add it to your root express application. In the `app.js` file:

```
const express = require('express');
const customer = require('./customer');

const app = express();
app.use(express.json());
app.use('/customer', customer);
```

Note we need to include a middleware to parse incoming request body into the `req.body` object. This is what the line `app.use(express.json());` does. More on middleware in the next section.

When you structure your express application in this way, each of these route files plays a similar role as Controller classes in ASP.NET Core. And the path where each API endpoint is exposed, like `GET /customer/:id`, is defined by methods such as `app.use` or `router.get`.

Sharing behavior and data via middleware

We saw that Express works by associating request handler functions with a specific path and HTTP verb. Each request handler is a function like:

```
(req, res, next) => {
  // do something, optionally return a response to the client
  // if not returning a response, then call next handler
  next();
}
```

These functions can be added anywhere in the request pipeline, including globally, for any request. Generic-purpose handlers that are added globally or for multiple requests, are typically known as *middleware*.

Creating your own middleware is straightforward, simply create a function like the one above. For example, a simplistic middleware to log incoming requests can be implemented like:

```
const loggingMiddleware = (req, res, next) => {
  console.log(`${req.method.toUpperCase()}: ${req.baseUrl}${req.path}`);
  next();
}
app.use(loggingMiddleware);
```

Note this is just an example. In real projects, you will use a logging framework compatible with Express like [pino](#) or [winston](#).

The order in which each middleware and route is defined, matters. i.e., if you were to add this middleware after the line `app.use('/customer', customer);` then you won't see customer requests logged to the console.

You are not limited to use middleware globally. Express allows any number of middleware functions to be added any time you call methods such as `app.use`, `app.get`, `router.get`, etc. For example, you could add your logging middleware exclusively for the customer routes, but not the rest, by adding it explicitly in the same `app.use` that mounts the customer routes:

```
// do not add logging globally
// app.use(loggingMiddleware);
// add exclusively for customer routes
app.use('/customer', loggingMiddleware, customer);
```

The idea of request handler functions that can be added either to specific routes or globally is what makes Express simple, yet powerful to work with. It is how Express can remain lightweight and unopinionated, leaving up to the users and community to create and use the middleware they want.

You might also be wondering how one middleware can share or prepare some data, so another middleware can use it down the line?

The answer is that the request object (i.e. the `req` parameter of the functions) acts as the **request context**, where you can add any additional properties you need. A common example is authentication middleware extracting the user from the request and saving it as `req.user`:

```
const authMiddleware = (req, res, next) => {
  // extract user information from the request cookies/header
  req.user = extractUserFromRequest(req);
  // now any handler function after this point can use req.user
  next();
};
```

```

const authMiddleware = (req, res, next) => {
  // extract user information from the request cookies/header
  req.user = extractUserFromRequest(req);
  // now any handler function after this point can use req.user
  next();
};

```

For example, a naive `/whomai` route could simply send back the `req.user` object previously added by the auth middleware:

```

app.use(authMiddleware);
app.get('/whoami', (req, res) => {
  return res.send(req.user);
});

```

In the end, Express is a minimalistic framework, which doesn't provide many features beyond the routing and request pipeline. Using Express typically means researching and evaluating the existing official and community-driven middleware, adding functionality which isn't available out of the box. For example:

- We already saw `express.json()` while creating the sample REST API. It provides a middleware function to parse JSON from the request body into `req.body`, so the next middleware/handlers can consume it.
 - You can serve `static files` using an official middleware. For example, serve files located in a `/public` folder inside the server using:
- ```
app.use(express.static('./public'));
```
- Similar to the JSON middleware, there are other popular community middleware to parse parts of a request, like `cookies` or `uploaded files`.
  - Implement authentication using one of the most popular middleware for Express: `Passport`. There are many authentication strategies ready to use from, like local username & password, social sites (google, twitter, facebook, ...), auth0, okta and more.
  - When it comes to security, `helmet` provides you a great starting point to hardening your server. There are also popular specialized middlewares to implement `CSRF protection` and `CORS`.

## Error handling

Handling errors in Express is, rather unsurprisingly, done through middleware functions. These ones are a bit special though:

- They should always declare four arguments, rather than the usual three. The extra argument is the error being handled.
- You should add them last, after all the other `app.use` statements. This is so the error handler can catch an error that happened in any of the other handlers executed for the request.

For example, an extremely simplistic error handler could simply log to the console:

```

const errorHandler = (err, req, res, next) => {
 console.log(err);
 next(err);
}

```

```
};
app.use(errorHandler);
```

As you can imagine, this is just an example for illustrative purposes. Express has a default error handler built-in, which does log the error to the console and returns a 500 status. It even includes the stack trace when the NODE\_ENV environment variable is other than production.

If you understood middleware functions earlier, error handling feels like a natural extension. There is one major caveat to be aware of though, and that is how errors in asynchronous code behave.

By default, express will be able to catch errors thrown synchronously, and pass them onto the error handlers (if any). For example, the following error is automatically caught and will be logged by our error handler:

```
app.get('/sync-error', (req, res) => {
 throw new Error('Something happened');
});
```

However, the same thing does not happen in asynchronous request handlers, i.e. the ones that use Promises or async/await. An error like the following one ends being handled by the registered error handlers:

```
app.get('/sync-error', (req, res) => {
 throw new Error('Test the error handler');
});
```

While an error like the following will cause the process to crash due to an unhandled promise rejection:

```
const doSomethingAsync = () => {
 return new Promise((resolve, reject) =>
 setTimeout(() => reject('Test the error handler'), 1000));
};
app.get('/async-error-crash', async (req, res) => {
 await doSomethingAsync();
});
```

To avoid this problem, when using Express version 4 or lower, you **must always catch errors in asynchronous handlers** and call next. For example, when using async/await, you need to add a try catch as in:

```
app.get('/async-error', async (req, res, next) => {
 try{
 await doSomethingAsync();
 }catch (err){
 next(err);
 }
});
```

*Note this is meant to be fixed once Express version 5 is released. From that version, errors thrown in async functions or rejected promises will be automatically caught and passed to error handlers. The problem is that version 5 has been in alpha state for more than 6 years!*

# Fastify.js, a performant and modern framework

There is no denying Express was the most successful Node.js framework during many years and inspired many great projects and libraries. However, it is also a framework designed in the early days of Node.js, before Promises, async/await, HTTP2, TypeScript, ES6 and many other features that came over the years.

Fastify is a relatively young web framework, actively maintained and with a thriving community. It has been built to leverage modern JavaScript features, avoid the mistakes in past frameworks like Express and offer excellent performance.

If you look at the [state of JS 2020](#) survey, it ranks very high on interest and satisfaction. You can also see it's a young framework, real usage is still quite low compared to Express.

*If you run into trouble or just want to follow along by browsing code, check the “fastify-example” project in GitHub. Additionally, I recommend checking [this example](#) that one of the lead maintainers of Fastify put together.*

## First steps with Fastify

Let's get started in a similar way to what we did with Express. Create a new folder, run npm init and install the usual dependencies plus Fastify:

```
npm install --save fastify
npm install --save-dev nodemon jest supertest
```

Then add the same `npm run dev` script to package.json that we added during the first article of the series (Page 6):

```
"dev": "nodemon --ignore 'test/**/*' ./index.js",
```

*Once we take a look at the fastify-cli by the end of the article, we will see a better way to run the project.*

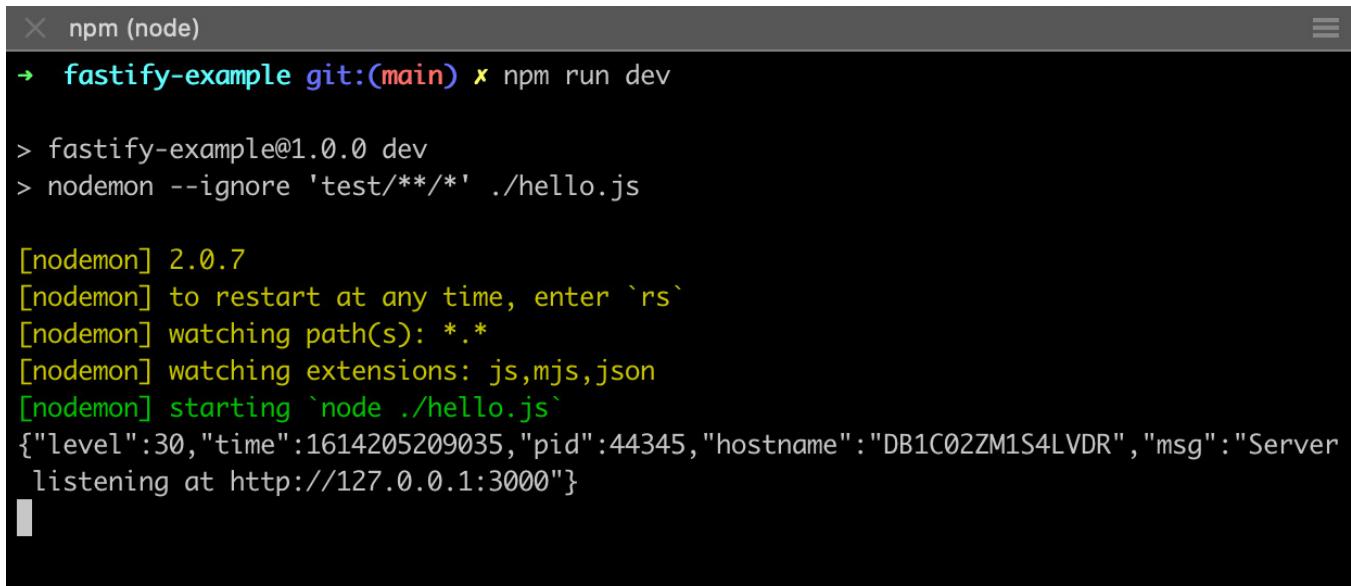
Finally create an index.js file with our simple hello world web server:

```
const fastify = require('fastify')({ logger: true });
const process = require('process');
const port = 3000;

fastify.get('/', async (request, reply) => {
 return `Hello world!
 The current time is ${ new Date() }
 and I am running on the ${ process.platform } platform`;
});

const start = async () => {
 try {
 await fastify.listen(port);
 } catch (err) {
 fastify.log.error(err);
 process.exit(1);
 }
}
start();
```

Once you have finished with the setup, start it with `npm run dev` and open `http://localhost:3000` in your browser.



```
npm (node)
→ fastify-example git:(main) ✘ npm run dev

> fastify-example@1.0.0 dev
> nodemon --ignore 'test/**/*' ./hello.js

[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node ./hello.js`
{"level":30,"time":1614205209035,"pid":44345,"hostname":"DB1C02ZM1S4LVDR","msg":"Server listening at http://127.0.0.1:3000"}
```

Figure 3, running a Fastify application with nodemon

The code above is very similar to the initial example we saw with Express. That's because Fastify takes note of frameworks that came before. However, note how its API now leverages `async/await` rather than callbacks. This is the first of many differences we'll see!

Let's finish this section by showing how similar the integration testing with `supertest` is (You should have already installed `Jest` and `supertest`, if not `npm install them`). Let's do the usual refactor to separate the app definition from opening the actual server port. Add a new file `/server/app.js` with these contents:

```
const process = require('process');

module.exports = async function(fastify, opts){
 fastify.get('/', async (request, reply) => {
 return `Hello world!
 The current time is ${ new Date() }
 and I am running on the ${ process.platform } platform`;
 });
};
```

*Don't worry if this doesn't make sense, we will come back and look at why the route is wrapped in an initialization function.*

And update `index.js` so it just imports the defined Fastify application:

```
const fastify = require('fastify')({ logger: true });
const app = require('./server/app');

fastify.register(app);

const port = 3000;
const start = async () => {
 try {
 await fastify.listen(port);
 } catch (err) {
```

```

 fastify.log.error(err);
 process.exit(1);
}
start();

```

Now define an integration test `/test/integration/server.spec.js`. This is very similar to the tests you have seen so far, which comes to show how the basic ideas/techniques can be ported across frameworks. The main difference is how Fastify is started/stopped in the `beforeAll` and `afterAll` hooks:

```

const { beforeAll, afterAll } = require("@jest/globals");
const supertest = require('supertest');
const fastify = require('fastify')({ logger: false });
const app = require('../server/app.js');

describe('the server', () => {
 let request;
 beforeAll(async () => {
 fastify.register(app);
 await fastify.ready();
 request = supertest(fastify.server);
 });
 afterAll(done => {
 fastify.close(done);
 });

 test('GET / returns a helloworld plaintext', async () => {
 const res = await request
 .get('/')
 .expect('Content-Type', 'text/plain; charset=utf-8')
 .expect(200);

 expect(res.text).toMatch(/Hello world!\s+The current time is .*\s+and I am
running on the .* platform/);
 });
});

```

Finally copy the same `test:integration` script to your `package.json`:

```
"test:integration": "jest \"test/integration/.*\\.spec\\.js\""
```

Then run `npm run test:integration` to validate that your hello world server behaves as it should.

```

→ fastify-example git:(main) ✘ npm run test:integration
> fastify-example@1.0.0 test:integration
> jest --silent "test/integration/.*\\.spec\\.js"

PASS test/integration/server.spec.js

Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 1.737 s
→ fastify-example git:(main) ✘

```

Figure 4, running integration tests for a Fastify application

## Organizing your routes with plugins

Fastify provides a mechanism to organize and encapsulate behavior without having to add routes to the root `fastify` object. Previously in the article, we saw with Express you would normally organize your business routes using a *router*.

In Fastify you organize your routes by using a *plugin*. For now, you can see a plugin as a way of creating a child Fastify instance which gets its own context where routes can be defined. Not too different from an Express router! For now, consider a *plugin* as a way to group routes. In fact, you have already seen the first plugin in the `app.js` file!

Let's see another example. The following `/server/customer.js` plugin implements the exact same functionality as the Express router we saw earlier in Part 1 (Page 6):

```
let customers = [
 {id: '111', name: 'Foo'},
 {id: '222', name: 'Bar'}
];

const customerRoutes = async function (fastify, opts) {

 fastify.get('/', (req, reply) => {
 return customers;
 });

 fastify.get('/:id', (req, reply) => {
 const customer = customers.find(c => c.id === req.params.id);
 if (!customer) res.sendStatus(404);
 return customer;
 });

 fastify.post('/', (req, reply) => {
 customers = [...customers, req.body];
 reply.statusCode = 201;
 reply.send();
 });
};

module.exports = customerRoutes;
```

You can then register a plugin in a similar way as you would register middleware/routers in Express. In the `app.js` file, include the necessary lines to import the customer plugin and register it:

```
const customer = require('./customer');

module.exports = async function(fastify, opts){
 fastify.register(customer, { prefix: '/customer' });
 ...
}
```

Overall, a plugin that implements business behavior, like the simple example above, plays a very similar role as a traditional controller class in ASP.NET (or the new minimal APIs). Compared to Express, it doesn't look much different from the route we saw. However, note subtle improvements like:

- You don't need to explicitly call `res.send`, you can just use a normal return statement. Fastify will return the object in the response body and automatically set the content header.
- The plugin is created *inside* an async function. This lets you control exactly when an instance of this plugin is created. Being a function, it can be called multiple times which allows you to create multiple instances, each with its own independent state. You can also run any initialization code, including async code like establishing a connection to databases and other external services.
- This means a plugin is a *function* that receives a Fastify instance and gets its own context whereby it can define specific request handlers.

Let's now take a look at what else can plugins do for us other than grouping routes.

### *Everything is a plugin! Plugins define routes, hooks and decorators within their own boundary context*

When studying **Express**, it becomes clear everything is a middleware, where a **middleware** is any function with signature `function(req, res, next)`. Any common or shared behaviour can be implemented as one of these middleware functions and then be added to the request pipeline with methods like `app.use`, `app.get`, `router.get`, etc.

In **Fastify**, everything is a **plugin**. A **plugin** defines an encapsulation context, a *child Fastify instance* if you want, where behaviour can be defined. We have already seen how that behaviour can be added by defining routes, like the CRUD example above, but that's not the only element a plugin can contain.

Fastify's design is a bit more complex than Express, and defines several elements that a plugin can contain:

- **Routes**. This is the element we have seen so far with our CRUD customers example. It is Fastify's way of associating an endpoint (i.e. an HTTP method and path like GET /customers), with a specific handler function.
- **Hooks**. Fastify defines a very explicit `request lifecycle`, and allows you to execute code at any one of these stages by adding a hook function to your plugin. Behaviour shared for multiple routes, like authentication or logging, needs to be implemented with a *hook* since routes apply to a single endpoint.
- **Decorators**. When you want to share some data across the different hooks and routes, you don't directly modify the request object as in the case of Express. Instead, you create a *decorator* - Fastify's way of adding extra properties to the request object. Like in Express, the request object is also the *request context*, but in Fastify you don't modify it directly.
- **Error handlers**. Each plugin can define its own error handling function, invoked in case of errors during one of the other elements like hooks or routes. Note there can only be one handler defined (or no handler, and inherit the root one)

There is one final important concept about plugins, and that is the **encapsulation boundary**. What this means is that routes, hooks and decorators inside a plugin are only executed for requests that match any of the routes also defined in that plugin!

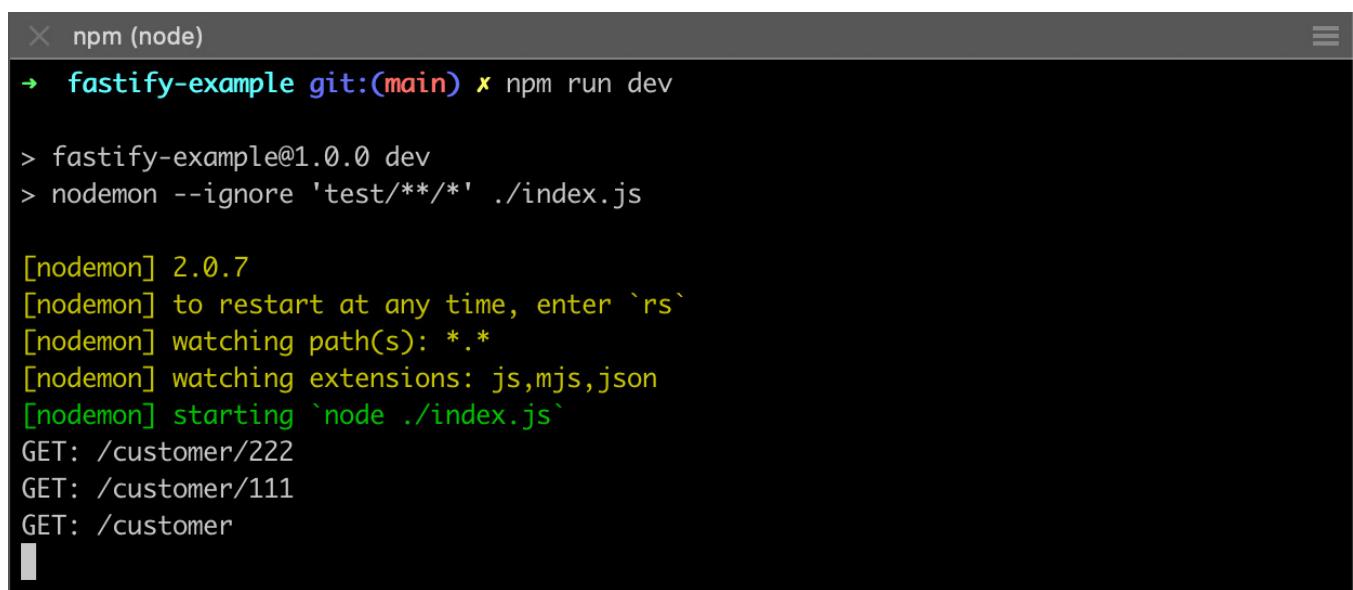
## Hooks and boundary context

This might begin to sound complicated, it's time to see some examples! To begin with, let's ignore for a moment that Fastify has a robust logging module built in. Update index.js and pass the option `logger: false` when initializing the `fastify` object.

Now imagine we wanted to write our own plugin that logs every request. According to Fastify's design, we would use a **hook** for the **onRequest lifecycle event** so we catch any request. Update the existing `customer.js` plugin and add a hook like:

```
const customerRoutes = async function (fastify, opts) {
 fastify.addHook('onRequest', (request, reply, done) => {
 console.log(` ${request.method.toUpperCase()}: ${request.url}`);
 done();
 });
 ...
};
```

Now open some URLs in the browser like `http://localhost:3000/customer/111` and `http://localhost:3000/customer/222` and note how they are logged to the console. However, if you reload the root URL `http://localhost:3000/`, note how nothing gets logged.



```
npm (node)
→ fastify-example git:(main) ✘ npm run dev

> fastify-example@1.0.0 dev
> nodemon --ignore 'test/**/*' ./index.js

[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node ./index.js`
GET: /customer/222
GET: /customer/111
GET: /customer
```

Figure 5, the hook is scoped to the plugin. Only plugin routes are logged

That makes sense. Since the hook is defined within the customer plugin, it is only executed for the routes defined by the customer plugin. It is not executed for routes outside that plugin, like the `fastify`.

`get('/', ...)` defined directly in `app.js`.

Remove those changes. Now let's consider how could we add the hook for **every request**. One way could be having to add the hook on every single plugin, but as you can imagine, that would be tedious. Therefore, your first instinct might be to add something like this:

```
const logging = async (fastifyInstance, opts) => {
 fastifyInstance.addHook('onRequest', (request, reply, done) => {
```

```

 console.log(` ${request.method.toUpperCase()}: ${request.url}`);
 done();
 });
};

fastify.register(logging);

```

This **won't** work, since you have now added the hook as part of a new plugin with no defined routes. And the hook only applies to routes of that plugin, which has none.

What you need is for the hook to be added to the *same boundary context where you are calling `fastify`.*

`register`, in this case the root context, so it applies globally.

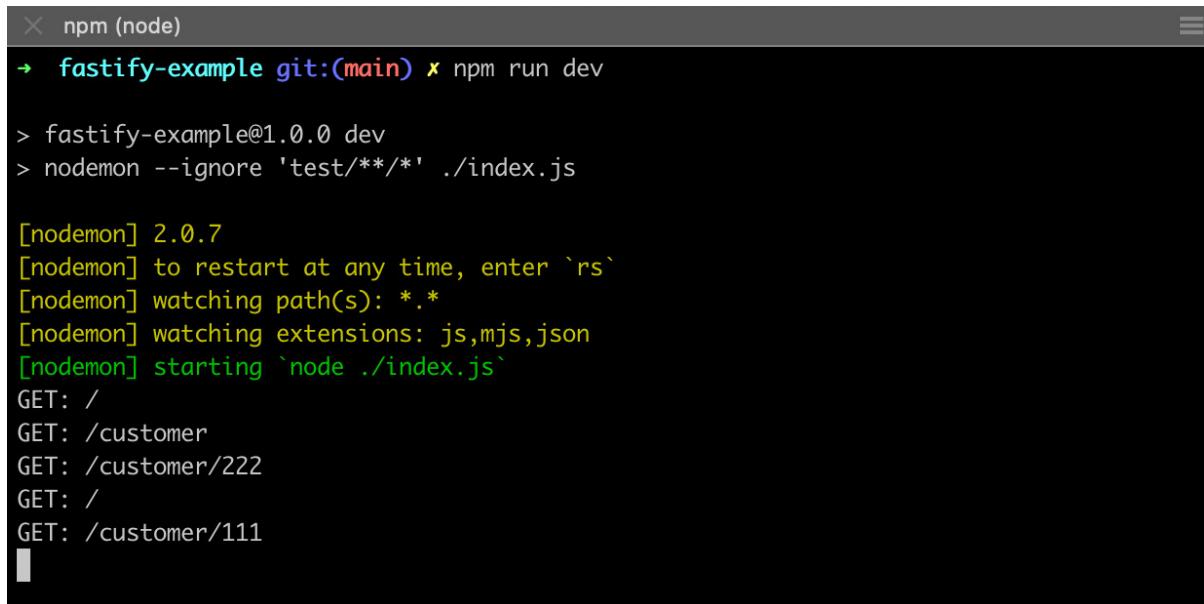
To cover this scenario, Fastify provides the `fastify-plugin` utility. Install it with `npm i --save fastify-plugin`, and then update the code above with:

```

const fp = require('fastify-plugin');
...
fastify.register(fp(logging));

```

Notice now how the hook is executed for every request, including `http://localhost:3000/!`



```

× npm (node)
→ fastify-example git:(main) ✘ npm run dev

> fastify-example@1.0.0 dev
> nodemon --ignore 'test/**/*' ./index.js

[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node ./index.js`
GET: /
GET: /customer
GET: /customer/222
GET: /
GET: /customer/111

```

Figure 6, the hook is added globally and therefore all requests are logged

## Decorators

Let's now take a look at the **decorators**. Let's imagine you want to add the `user` object to the request context, so routes can check which user is making the request. You would then create a plugin with a **request decorator** as follows:

```

const userContext = async (fastifyInstance, opts) => {
 fastifyInstance.decorateRequest('user', null);
 fastifyInstance.addHook('onRequest', (request, reply, done) => {
 request.user = { name: 'Sofia' };
 done();
 });
};

```

```
fastify.register(fp(userContext));
```

Note how we are using **fastify-plugin** again, so the decorator and hook are applied to every request. It is also important pointing out that the decorator is initialized empty, and it's **combined with an onRequest hook** which sets the value, so **every request has its own user instance!**

To test if this works as expected, add another route like follows. Then load <http://localhost:3000/whoami> either in the browser or with tools like curl/postman:

```
fastify.get('/whoami', async (request, reply) => {
 return request.user;
});
```

**Request** decorators are not the only decorator type available. You can also define **server** decorators and **response** decorators. For example, server decorators are typically used to initialize connections to external services, like databases. See the [official docs](#).

*If you are considering Fastify, I recommend reading through the plugins guide in their [official docs](#). You might also want to check the plugins created by the [community](#), since there are already existing plugins for features such as authentication, security hardening, CORS, CSRF and more.*

## Taking advantage of Fastify

So far, Fastify might feel like a step up in complexity over Express, with no clear benefits. **That cannot be further from the truth!**

In the following sections, in addition to the fastest JavaScript server framework, and a healthy open-source community, we will see some of the advantages you get by using Fastify,

### Async code

To begin with, Fastify was designed with first-class support for `async/await`, and Promises. This means any time you define a plugin, a route handler or a hook, you can use either `async/await` or Promises, as well as callbacks if you are so inclined!

Imagine we have an asynchronous function, for example one that loads the user from the database. We will simulate it with this function:

```
const loadDataFromDb = function(){
 return new Promise((resolve, reject) =>
 setTimeout(() => resolve({foo: 42, bar: 'baz'}), 1000));
};
```

Fastify then supports an `async/await` function as a route handler. This way you can write code like:

```
fastify.get('/async-await', async (request, reply) => {
 const data = await loadDataFromDb();
 return data;
});
```

If you prefer, you can also work directly with promises. Fastify handles them out of the box if you return a

promise from the route handler, returning to the client the value that the promise resolves to. The route we just saw, but now with promises becomes the following:

```
fastify.get('/async-promises', (request, reply) => {
 return loadDataFromDb()
 // I've kept the "then" callback just to show it is a Promise
 .then(data => data);
});
```

Sometimes you might have to use an API based around callbacks. Worry not, that's also a coding style supported by Fastify, although you will then have to explicitly call `reply.send` yourself:

```
const loadDataFromDbCallback = done => {
 const error = null;
 setTimeout(() => done(error, {foo: 42, bar: 'baz'}), 1000);
};

fastify.get('/async-callback', (request, reply) => {
 loadDataFromDbCallback((err, data) => {
 if (err) reply.send(err);
 reply.send(data);
 });
});
```

Of course, you shouldn't mix all these styles within the same application, it's recommended that you pick one! And you should be careful not to both return a value/promise *and* call `reply.send` as part of the same route:

```
fastify.get('/dont-do-this', (request, reply) => {
 // this both returns a promise AND calls reply.send
 // the first one executed, {a:1}, is sent to the client. The other is discarded
 return loadDataFromDb()
 .then(data => reply.send({a: 1}))
 .then(() => ({b: 2}));
});
```

*You have the same choices when writing hooks, except when using callbacks in a hook you use a `done` parameter rather than `reply.send`. See the [official docs](#).*

## Error handling

Alongside the first-class support for `async/await` and Promises, comes built-in support for handling errors raised during Promises and `async/await` functions.

If you remember, in Express you had to manually catch errors and call `next(err)`. In Fastify, a rejected Promise is correctly handled by the framework. Therefore, if we update our `loadDataFromDb` function so it throws an error:

```
const loadDataFromDb = () => {
 return new Promise((resolve, reject) =>
 setTimeout(() => reject(new Error('Something happened')), 1000));
};
```

...then both sample routes defined above using `async/await` and Promises will automatically catch the error and invoke the defined error handler.

Whenever an error occurs, Fastify has a built-in error handling logic. The error will be caught, logged to the console and a 500 error returned to the user. You can however define your own error handling function using `fastify.setErrorHandler` as in:

```
fastify.setErrorHandler(function (error, request, reply) {
 console.log(`Found error: ${error}`);
 reply.status(500).send(error)
});
```

There can only be one error handling function defined. This means, if you call `fastify.setErrorHandler` for a second time, the first error handler is ignored. However, you can define a specific error handler inside a plugin, which will be invoked for errors in hooks/routes defined by that plugin.

## Logging

Fastify comes with a powerful and performant logger using `pino`, see the [official docs](#). When creating the Fastify root instance, you can enable or disable the logger as:

```
const fastify = require('fastify')({ logger: true });
```

The logger object is available as both `fastify.log` and `req.log` and it offers an API based on abstract logging.

This means there are several log severity levels: trace, debug, info, warn, error and fatal (in severity order). For each of these levels there is a method like `fastify.log.debug` or `req.log.info`. Any of these methods accept:

- Just a message: `req.log.info('Hello world')`
- A message with interpolation values: `req.log.info('Hello %s', 'world')`
- A message with extra properties to be added to the JSON log entry: `req.log.info({user: 'Lara'}, 'Hello world')`
- All of the above

If you simply enable logging, you will get the default options. The logger will write each entry to the console as a JSON message and has the log level set as info (i.e., messages logged with lower severities like debug and trace will be ignored). It also automatically writes an entry to the log for every single request handled by the server.

Note that every log entry collects a lot of metadata, and the actual message is in the `msg` property. See for example a trace after starting the server and loading the `http://localhost:3000` route:

```
{"level":30,"time":1614121547286,"pid":19720,"hostname":"...","msg":"Server listening at http://127.0.0.1:3000"}
{"level":30,"time":1614121549408,"pid":19720,"hostname":"...","reqId":1,"req": {"method": "GET", "url": "/", "hostname": "localhost:3000", "remoteAddress": "127-0.0.1", "remotePort": 53094}, "msg": "incoming request"}
{"level":30,"time":1614121549416,"pid":19720,"hostname":"...","reqId":1,"res": {"statusCode": 200}, "responseTime": 6.9483460038900375, "msg": "request completed"}
```

*Note at this point you might want to comment/remove any sample code defined earlier that includes `console.log`*

*statements.*

The main reason why you want to use `req.log` instead of `fastify.log` when logging inside request handlers and hooks is to associate those entries with the same request metadata. Among those, there is a request id logged as `reqId`, which Fastify either autogenerated as an incremental integer or reads from the existing `request-id` header. This ensures that all the messages logged while serving the same request can be correlated, since they will have the same `reqId` in the logs.

Anyone who has ever troubleshooted an issue through the logs can vouch how important this is! It's great to know that Fastify gets it right by default. For example, you can include a log statement in a route handler:

```
fastify.get('/', async (req, reply) => {
 req.log.info('hello logger!');
 return `Hello world!`;
});
```

Note how that log statement and the default "incoming request" and "request completed" messages all share the same `reqId`:

```
{"level":30,"time":1614121991658,"pid":21986,"hostname":"...","reqId":1,"req": {"method":"GET","url":"/","hostname":"localhost:3000","remoteAddress":"127.0.0.1","remotePort":53094}, "msg":"incoming request"}
{"level":30,"time":1614121991663,"pid":21986,"hostname":"...","reqId":1,"msg":"hello logger!"}
{"level":30,"time":1614121991666,"pid":21986,"hostname":"...","reqId":1,"res":{"statusCode":200}, "responseTime":6.9483460038900375, "msg":"request completed"}
```

In case you need to configure how the logger works, you can pass any of the options `pino` accepts when creating the `fastify` instance as in `require('fastify')({ logger: { ... } })` instead of simply a Boolean.

Combined with the default `pino` serializers defined by Fastify for the request, response and error objects, you can easily control what gets logged.

For example, you could raise the log level from the default of `info` to `warn`, and you could omit (i.e. `redact`) certain headers and manually specify which properties from the request should be logged:

```
const fastify = require('fastify')({
 logger: {
 level: 'trace',
 redact: ['hostname', 'req.headers.authorization'],
 serializers: {
 req (request) {
 return {
 method: request.method,
 url: request.url,
 }
 }
 }
 }
});
```

Note how some properties are now redacted, and how the `req` object only logs the properties included in

the serializer:

```
{"level":30,"time":1614122820158,"pid":25970,"hostname":
"[Redacted]","reqId":1,"req":{"method":"GET","url":"/"},"msg":"incoming request"}
{ "level":30, "time":1614122820160, "pid":25970, "hostname":
"[Redacted]", "reqId":1, "msg": "hello logger!" }
{ "level":30, "time":1614122820165, "pid":25970, "hostname":
"[Redacted]", "reqId":1, "res":{"statusCode":200}, "responseTime":6.573631003499031,
"msg": "request completed" }
```

You can actually pass any options that *pino* accepts, see the [pino docs](#).

## Schemas: Request validation and Response serialization

So far, we have only seen the shorthand way of defining routes. If you remember from the CRUD example:

```
fastify.post('/', (req, reply) => {
 customers = [...customers, req.body];
 reply.statusCode = 201;
 reply.send();
});
```

...however since Fastify allows several options to be defined, there is a more general way of defining routes. To begin with, you can use the general `fastify.route` method, where the following code defines an equivalent route:

```
fastify.route({
 method: 'POST',
 path: '/',
 handler: (req, reply) => {
 customers = [...customers, req.body];
 reply.statusCode = 201;
 reply.send();
 }
});
```

The interesting part is that Fastify has built-in support for declaring the schema that your request/response objects will conform to. These schemas can be defined via additional options when defining a route. For example, we can define the properties that should be found in the request body when creating a new customer as in:

```
fastify.route({
 method: 'POST',
 path: '/',
 schema: {
 body: {
 type: 'object',
 properties: {
 id: { type: 'string' },
 name: { type: 'string' }
 },
 required: ['id', 'name']
 }
 },
 handler: (req, reply) => {
```

```
 ...
});
```

As per the [official docs](#), schemas are defined using the Ajv library, which lets you use a range of validations other than required values like min/max value, min/max length, range of values and more. If necessary, it can also be extended with custom behavior and there is already a range of plugins written by the community.

If the request does not conform to this schema and its validations, Fastify automatically returns a 400-BadRequest error. See the following integration test:

```
test('POST /customer returns 400 when missing required property', async () => {
 const res = await request
 .post('/customer')
 .send({it: 'is missing required properties'})
 .expect(400);

 expect(res.body).toMatchObject({
 error: 'Bad Request',
 message: "body should have required property 'id'"});
});
```

Similarly, it is possible to define the schema of the responses. Rather than validating incoming requests, there are two main purposes for this:

- allowing Fastify to optimize the JSON serialization achieving greater throughput
- ensuring other properties are not exposed by default

For example, if you define a route like:

```
fastify.route({
 method: 'GET',
 path: '/response-serialization',
 schema: {
 response: {
 '2xx': {
 name: { type: 'string' }
 }
 }
 },
 handler: (req, reply) => {
 return {name: 'Sofia', someSensitiveProperty:'foo', anotherProperty: 'bar'};
 }
});
```

*Note the schema is associated with the response in case of a specific status code or range of status codes is returned.*

The actual value that the client will receive is:

```
{"name": "Sofia"}
```

To top it off, this makes publishing OpenAPI schemas and documentation a simple exercise. Declare request/response schemas alongside your Fastify routes, then install the plugin [fastify-swagger](#)!

Hopefully this shows how Fastify has been designed to be more than *just* a routing mechanism. It is more complex than Express, but it also requires less gluing of different libraries to get a decent starting point for your application.

## Autoregister plugins

Since everything in Fastify is handled as a plugin, it is common to organize your application as a series of plugin files, which get imported and registered by the root [Fastify](#) instance.

```
const myPlugin = require('./my-plugin');
...
fastify.register(myPlugin);
```

This can become quite tedious after a while, so the Fastify team has created a utility named [fastify-autoload](#). With this utility, you can place all your plugins into a folder (or various folders) and register them all in a single call. For example, if you create your plugins inside a folder `/plugins`, then you could register them all as:

```
const autoload = require('fastify-autoload');
...
app.register(autoload, {
 dir: path.join(__dirname, 'plugins')
});
```

See the [docs](#) on how `fastify-autoload` deals with prefixes, recursive folders and inclusion/exclusion templates.

It is worth noting that the community has adopted a pattern whereby plugins are typically divided into two different folders at the project root:

- The `/plugins` folder is where common behavior that applies to all requests such as authentication gets defined.
- The `/routes` folder is where your actual application business logic is defined.

This is exemplified in the excellent [sample project](#) created by one of Fastify's lead maintainers.

## Fastify CLI

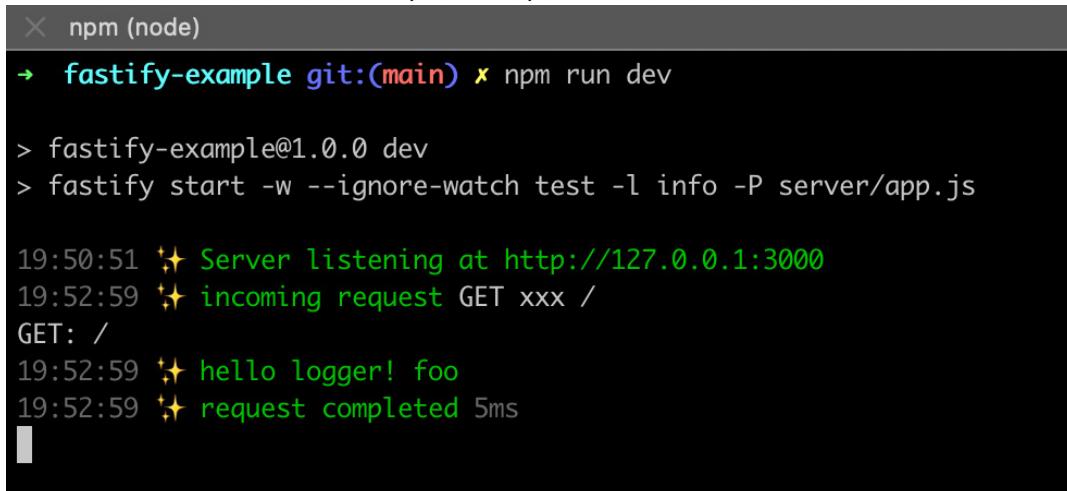
To finish the overview of Fastify, let's look at the official CLI [fastify-cli](#) and what it can do for us. Begin by installing it as `npm install --save-dev fastify-cli`.

One of the main use cases of the CLI is to start the application. This means we can just use fastify's CLI to start the application rather than nodemon. Let's try that, update the dev script in `package.json` as:

```
"dev": "fastify start -w --ignore-watch test -l info -P server/app.js",
```

This executes the “start” command of the CLI, pointing it to our app.js root plugin. Remember that app.js is a plugin, it exports a function like async function(fastify, opts)!

It’s also setting up the logging level and enable pretty-logs, which are not useful for production where you want JSON, but make the development experience nicer:



```
npm (node)
→ fastify-example git:(main) ✘ npm run dev

> fastify-example@1.0.0 dev
> fastify start -w --ignore-watch test -l info -P server/app.js

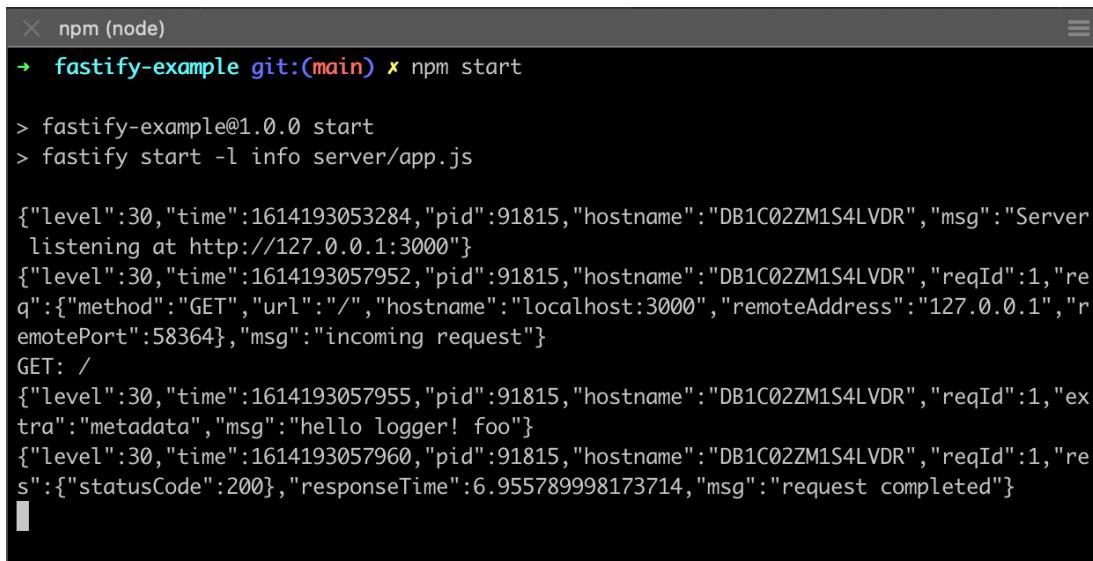
19:50:51 ✨ Server listening at http://127.0.0.1:3000
19:52:59 ✨ incoming request GET xxx /
GET: /
19:52:59 ✨ hello logger! foo
19:52:59 ✨ request completed 5ms
```

Figure 7, running a fastify project in development with fastify-cli

We can also add a command to run the application in production, which means we don’t even need the index.js file!

```
"start": "fastify start -l info server/app.js",
```

Note how this defaults the logs to the JSON format. If you make any changes to the code, the application will not automatically reload:



```
npm (node)
→ fastify-example git:(main) ✘ npm start

> fastify-example@1.0.0 start
> fastify start -l info server/app.js

{"level":30,"time":1614193053284,"pid":91815,"hostname":"DB1C02ZM1S4LVDR","msg":"Server listening at http://127.0.0.1:3000"}
{"level":30,"time":1614193057952,"pid":91815,"hostname":"DB1C02ZM1S4LVDR","reqId":1,"req":{"method":"GET","url":"/","hostname":"localhost:3000","remoteAddress":"127.0.0.1","remotePort":58364},"msg":"incoming request"}
GET: /
{"level":30,"time":1614193057955,"pid":91815,"hostname":"DB1C02ZM1S4LVDR","reqId":1,"extra":{"metadata":true},"msg":"hello logger! foo"}
{"level":30,"time":1614193057960,"pid":91815,"hostname":"DB1C02ZM1S4LVDR","reqId":1,"res":{"statusCode":200}, "responseTime":6.955789998173714,"msg":"request completed"}
```

Figure 8, running a fastify project in production with fastify-cli

Something that might not be immediately obvious is that fastify-cli can start a server that hosts any Fastify plugin you point it to. You could for example start a server that only mounts the customer plugin as in:

```
"dev:customer": "fastify start -w --ignore-watch test -l info -P server/customer.js",
```

This way, you could very easily start a server that only exposes a subset of the routes/functionality!

For example, imagine you are working on a new API for your project. You could then use the CLI to start a server hosting the plugin file you are working on, and use a tool like curl or postman to send requests to routes defined by that new plugin.

If your project follows a design approach whereby routes are divided in self-contained independent plugins, or *services*, this is a neat bonus point!

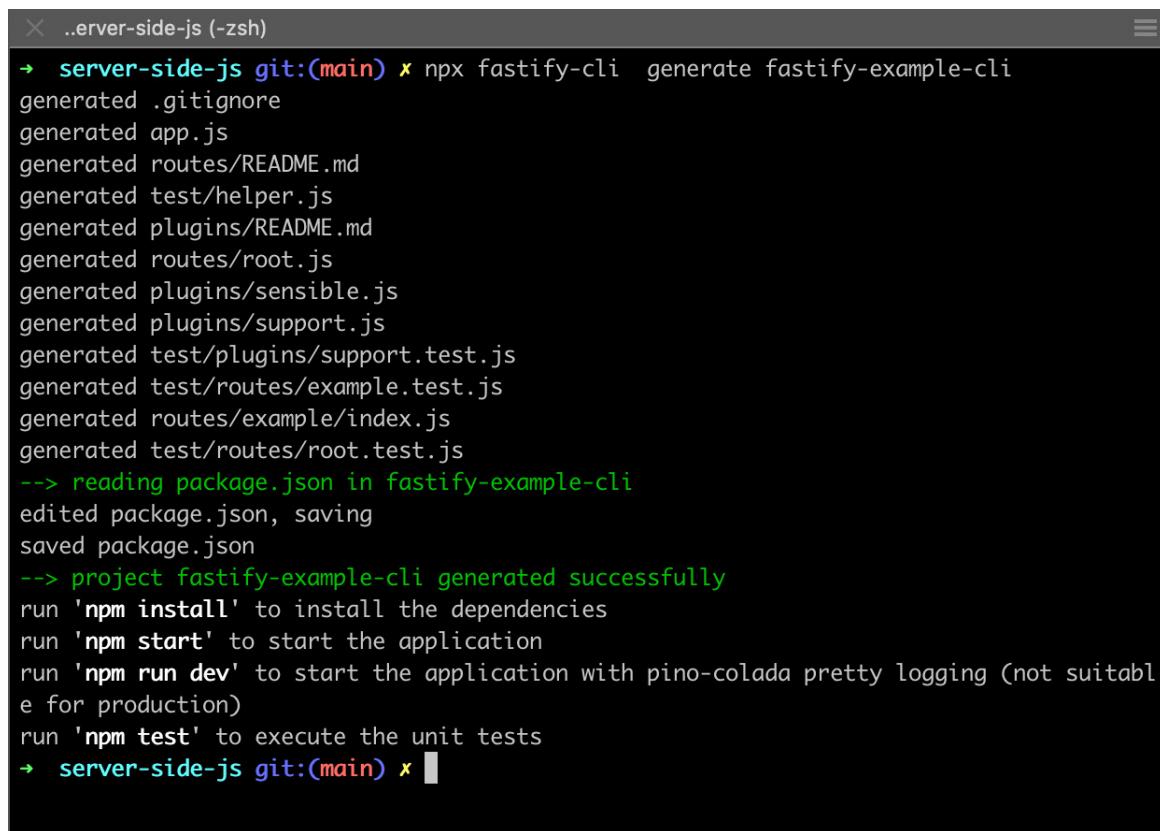
The second use case of the fastify-cli is to initialize a new project from scratch!

You will get a project with the basic structure in place:

- An app.js file with the root plugin
- start and dev scripts in package.json using fastify-cli
- the /plugin and /routes folder with fastify-autoload preconfigured
- integration tests
- optionally preconfigure the project to use TypeScript

Give it a try and generate a new project with:

```
npx fastify-cli generate my-awesome-project
```



A terminal window titled 'server-side-js (-zsh)' showing the output of the command 'npx fastify-cli generate fastify-example-cli'. The terminal shows the creation of various files and folders, reading package.json, editing package.json, and providing instructions for running the application. The terminal window has a dark background with light-colored text.

```
server-side-js git:(main) ✘ npx fastify-cli generate fastify-example-cli
generated .gitignore
generated app.js
generated routes/README.md
generated test/helper.js
generated plugins/README.md
generated routes/root.js
generated plugins/sensible.js
generated plugins/support.js
generated test/plugins/support.test.js
generated test/routes/example.test.js
generated routes/example/index.js
generated test/routes/root.test.js
--> reading package.json in fastify-example-cli
edited package.json, saving
saved package.json
--> project fastify-example-cli generated successfully
run 'npm install' to install the dependencies
run 'npm start' to start the application
run 'npm run dev' to start the application with pino-colada pretty logging (not suitable for production)
run 'npm test' to execute the unit tests
→ server-side-js git:(main) ✘
```

Figure 9, generating a Fastify project using fastify-cli

You can navigate to the generated my-awesome-project folder, run npm install and start the application with npm run dev.

Using fastify-cli is the easiest way to get a new project started with the recommended setup in place.

## Conclusion

Express and Fastify are two great frameworks for building web servers, APIs and services using Node.js. Express is easy to understand and get into, has a huge user base and has accumulated an impressive number of plugins over the years that help you with features outside of its scope. It has been hugely influential in the Node.js developer community, and its success meant even Microsoft took note when designing ASP.NET Core (among other successful projects outside C# ecosystem).

But Express also shows it's aging in certain ways, and that's where newer frameworks come into play. Of those, I recommend starting with Fastify. It's beautifully designed and engineered around the best that JavaScript and Node.js offer today, providing an excellent developer experience. In addition, it supports a gentle transition from Express that doesn't require mastering all its concepts at once.

In a future article (and the final part of this series), we will take a look at two more frameworks that go beyond the scope of web servers. These frameworks are considered application frameworks and aim to solve many typical problems that developers face building applications. Stay tuned!

• • • • • •

### Daniel Jimenez Garcia

Author

*Programmer, writer, mentor, architect, problem solver and knowledge sharer. In his 15+ years of experience, Daniel has lead teams building scalable systems, driven transformational changes to increase teams and developers' productivity, delivered working quality software, worked across varied industries and faced many challenges.*



*His DNC articles have garnered over 2 million reads. He has also published libraries, contributed to open-source projects, created many sample projects, reviewed books, answered many stack overflow questions and spoken at events.*

*He remains interested in many areas and technologies such as: Cloud architecture, DevSecOps, containers, Kubernetes, Node.js, Python, Vue, .NET Core, Go, Rust, Terraform, API and framework design, code quality, developer productivity and automated testing.*

Twitter: [https://twitter.com/Dani\\_djg](https://twitter.com/Dani_djg)

Github: <https://github.com/DaniJG>

Stack Overflow: <https://stackoverflow.com/users/1836935/daniel-j-g>

Personal site: <https://danijg.github.io/>



Technical Review

Damir Arh



Editorial Review

Suprotim Agarwal



et curry.com

**Want this  
magazine  
delivered  
to your inbox ?**

**Subscribe here**

**[www.dotnetcurry.com/magazine/](http://www.dotnetcurry.com/magazine/)**

\* No spam policy