# Docker Developer Workflows

Demoing VS Code, on a Mac, debugging a .NET Core Worker role

Steve Lasker

[Steve.Lasker@Microsoft.com](mailto:Steve.Lasker@Microsoft.com)

<http://blogs.msdn.com/SteveLasker>

# Demo environment setup & prereqs

[VS Code for Mac](http://aka.ms/dockertoolsforvstscode)

[Docker for Mac](http://beta.docker.com/docs)

Repo: <https://markruss.visualstudio.com/VotingApp>

SQL Image is sourced from: private-repo.microsoft.com

See [Travis Wright](mailto:twright@microsoft.com) for access

# Demo Reset

Clear all running containers

**docker rm -f $(docker ps -a -q)**

**docker rmi -f $(docker images -q)**

**docker pull node**

**docker pull microsoft/dotnet:1.0.0-rc2-core**

**docker pull FROM microsoft/dotnet:1.0.0-preview1**

## Pre Demo Steps – Optimizations

# Demo Reset

Browse to: <https://azuredevex.visualstudio.com/>

Set the **polyglot-web-node** and **polyglot-api-dotnet** repos to the **before-docker** branch.

# TODO

* Set the theme for the vscode terminal

# Pollyglot Demo

Setup:

Review slide of the app we’re going to build

* Node.js Web Front End
* .NET Core Web API

Each app will be developed individually, but separate teams. Each team must be able to test their component individually. They may want to test it as a collection of containers at the same time. All locally, prior to checking in – the inner loop

After they check it in, the CI system will build each image individually, and if tests pass, do a single deploy using docker compose to our production environment. Or, any environment you’d like.

For tools, we’ll use:

VS Code, on the Mac for both our Node.js and .NET Core app

Github, with VS Code github integration

Visual Studio Team Services for CI/CD to the Azure Container Service using Swarm

Let’s get started

# Open Node.js Project

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| Open GitHub, point to the 3 polyglot demos | We’ll start in GitHub and see our 3 projects |
| Open **polyglot-web-node** in terminal cd web-node | Lets get to the root of our web front end |
| Code . | From the root of our project, we’ll open VS Code  The root is important in VS Code if we want to use some of the integrations, such as source code control, or in our case, the debugging experience  VS Code is directory based. It doesn’t need a “project”, but it does look for a .vscode folder to find editor extensions and settings |
| Run under VS Code – on the local machine |  |
| Yo docker | We’ll add the docker assets with yo docker. This will scaffold out the default files we need to run this app within a docker container, including |
| docker build -t stevelasker/web-node . |  |
|  |  |

# VS Code

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
|  | Before we get started with our Node Project, we can see a few things about VS Code |
| Point to Explorer | We have the file explorer, which is just the files on disk |
| Search | We can search through our very large project |
| Git | We have great git integration to view, label and sync our changes |
| Git toolbar – below left | We can see the branch and sync our code right here |
| Editor toolbar – below right | We have some customizations we can make to our editor down here |
| Click Spaces… | And, if we click the toolbar, we can see that the whole editor is really a command line driven tool. |
| Click JSON, or whatever editor is active | We can see a set of editors that may be available for each file. And we can choose different ones, or even change the default with our settings file. |
| Code🡪Preferences🡪Workspace Settings | Notice we have 3 levels of settings.   1. The settings that ship with VS Code. 2. Setting per project 3. Settings per user   This gives me the ability to customiz things at many levels. In my case, I’ve added the dockerfile.debug mapping to our docker language service extension. This is a bug for files that don’t have a default extension.  Seems the linux world is fine in black and white. I prefer to live on the other side of the rainbow. But, that is a choice you have with VS Code, and that’s the point |
| View🡪Toggle Integrated Terminal | We even have an integrated terminal in VS Code, which can be really nice when needing to see commands, or run docker commands |

# Back to Node.js

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| Open server.js | Our Node project is a pretty basic angular app. It loads   * index.html page * uses app.js to call a Hello API * defined in server.js |
| Terminal | We want to run our project, but, do we want to run it on our Mac? Seems pretty normal. But, are we really running and testing as it would run when in production?  Don’t we really want to run it in a container? |
| Yo docker | From the root of our project, we’ll use our yeoman generator for docker. By calling **yo docker**, we can see the languages we currently support.  We’ll select Node.js of course |
| Port: 3000 ***{default}***  Image name: stevelasker/web-node {dockerhub username}/web-node  Service name: web-node ***{default}***  Project: webnode ***{default}*** | We’ll configure some options, and since we’re running in a container, we’re going to want to run this on port 80. Now, this is the container port, not the public port of our app.  As we’ll see here in a moment, we’re really just setting up the defaults |
| Switch back to VS Code | As we look at the files added in VS Code, we’ll see several. It may seem like a lot at first, especially when compared to our app that has a single file, but let’s review these a bit |
| Dockerfile | This is the definition of how we’ll build our image, when we want to deploy it.  It has the most basic for what we need in node. |
| Dockerfile.debug | This dockerfile is the definition for what we need while doing local debugging.  It has:   * Nodemon * Debugger port exposed * ENTRYPOINT that chooses which debugger to use. One will start debugging, right from the first line, capturing our startup The other attaches to the server, after startup, and will hit breakpoints we’ve set in our page. But, would miss the startup code as it attaches, and it turns out that code can run quickly. |
| Docker-compose.yml | The compose files are interesting, and we’ll talk more about these in a moment. |
| dockerTask.sh / ps1 | In a command line environment, we need a set of commands to execute our intentions. Rather than build magic into our extension and VS Code, we wanted to give you the control, but without having to stitch everything together  Earlier, I mentioned earlier about seting up the defaults.  What we’re doing here is scaffolding out the defaults, but these are all your files to edit, modify, enhance. You may need to change the flow, or use newer experimental tools. |
| .vscode/tasks.json | Here we have a collection of tasks that can be executed. If we look at the OSX collection of tasks, we can see various calls to our dockerTask.sh script. Each with different parameters.  For instance, we can build the image, which only builds it. It doesn’t run it  Or, we can compose and composeForDebug.  Compose is related to starting the app, and just running it  However, if we wish to compose and debug it, for now, this is a two-step process.  First, we’ll composeForDebug, which as you can see will use the docker-compose.debug.yml file. It will build the image and issue a docker-**compose –f docker-compose.debug.yml up** command.  This we’ll start the app, in the container, with the debugger |
| [⌘] [P]  task composeForDebug | We’ll open the command palette and launch the **task composeForDebug**  This will call our script and compose up the containers |
| ^ `  docker ps | If we open the terminal window, here in VS Code, we can see our running container |
| Start debugging | However, we haven’t yet started our node process. Since we may want to debug the startup process, we’ve intentionally not yet started the node process, as we want to do that under the debugger  As we start the debugger, we see our first line in server.js hit a breakpoint. Even though we didn’t actually set one. |
| Browse <http://localhost:3000> | We can now open the browser to see our app. Which is now running in a Linux docker container |
| Click [Repeat it again] | We can see an undefined response from our api-dot net call.  Lets debug what’s going on here |
| Set a breakpoint server.js res.send('From api-dotnet: ' + body); | If we set a breakpoint, we can inspect what’s happening here |
| Click [Repeat it again] | As we make the API call again, we can see our breakpoint hit |
|  | Hover over the error line to see that our API isn’t yet available  Of course, we haven’t built it yet, much less have it running |

# Environment Variables

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| **var apiPort = process.env.APIPORT || 5000** | Notice our code is making a reference to an environment variable named APIPORT. If not found, it will default to 5000  That’s good to know in the code, but nowhere is that expressed outside the code. In the container world, this is pretty important, as we don’t want others to know what’s in the container, just what they need to configure it.  Now, we all know that Node, .NET and other languages that have local execution environments use various ports to avoid local web server conflicts.  But, let make this externally configurable |
| Open **docker-compose.debug.yml**  Add:  **environment:**  **- REMOTE\_DEBUGGING**  **- APIPORT=5000**  Add to **docker-compose.yml as** well | We’ll add that port, and its default value to our docker-compose files. We want to add it to both the debug and release versions. |
| In server.js – highlight  **var port = process.env.PORT || 3000;** | We can see our web server was also defaulted to port 3000, based on an environment variable named PORT |
| **environment:**  **- REMOTE\_DEBUGGING**  **- APIPORT=5000**  **- HTTPPORT=3000** | We’ll add that one as well |
| Task composeForDebug | To make sure we did everything correct, we’ll retest our container and our code |
| Start debugging  <http://Localhost:3000> | Refresh the page, hit the button  Yup, everything is working good, except we need an API to call |

Change master to use HTTPPORT and APIPORT

# Demo API w/.NET Core

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| Open api-dotnet in vs code | Now that we have our front end, we’ll need the back end API to communicate with.  This will be a separate container, as we may need to scale it independently from the front end, and we happen to be using different stacks. .NET Core and Node.js on the front end.  This is just one of the many benefits of using containers and microservices. |
| Yo docker | Not surprisingly, we’ll use yo docker again. This time we’ll chose .NET Core as our language |
| Port: 5000  Image name: stevelasker/web-node {dockerhub username}/api-dotnet  Service name: api-dotnet ***{default}***  Project: apidotnet ***{default}*** | We’ll configure the parameters |
| Dockerfile | Here we have the production image definition in .NET  Notice we have a fairly clean dockerfile, that includes just the output of our compiled app.  This is a difference than Node.js. Node, as an interpretive langage just places its code in the container, restores and goes.  .NET is similar to Go, in that we compile the language, to get an efficient binary, ready for running. |
| Dockerfile.debug | This dockerfile is the definition for what we need while doing local debugging.  It has:   * The CLR Debugger   And the ENTRYPOINT that can either starts with, or without debugging. If we start with debugging, we tell the container to just run, as we need to start the app under the debugger in .NET. This is how we capture the startup process.  What you see missing here is the equivalent of nodemon. For .NET, it’s called dotnetwatch.  This is something we didn’t get finished for DockerCon, but it’s coming. |
| dockerTask.sh | Similar to Node, we have a dockerTask that “orchestrates” the local development debugging |
| Start debugging | With .NET Core, we’ve done a bit more work getting the container started for you. All you need to do is hit F5, or start debugging.  This will build the docker image, and compose up our container(s)  We see the browser actually launch as well. This is also what you’re seeing as an integrated debugging experience, where we’ll launch the browser. We may not know where to launch it, but we can fix that. |
| <http://localhost:5000/api/hello> | Before we do, lets just call into our API  Voila, we have an API running in our container. |
| Open .vscode/launch.json  "launchBrowser": {  "enabled": true,  "args": "${auto-detect-url}**/api/hello**", | To startup our API, we can just change the url to be opened. |
| Recycle the debugger | If we restart, we can see our API come right up |
| Set breakpoint:  **var machineName = Environment.MachineName;** | We can set a breakpoint in our controller |
| Refresh the page | And we can now step through various memory variables. Here, we’re just showing a few examples of subtle code that becomes platform specific. This was just one of the examples one of our customers expressed caused them a great deal of pain. |

# Inner Loop, w/Multiple Containers

Now, lets take a pause here for a moment. In this case, we would likely have two different teams developing these containers individually. Not because they’re written in two different languages. But, rather in a reasonably sized company, there are many different teams working on different projects. And, because they’re different teams, with different leaders, they choose different languages and stacks, because they can.

Now, the .NET Core team needs to test their API with the updated front end the other team is building. How can they do that? Should they have to get their source, the right Node version, and blah, blah, blah?

Or, how about they just run their container?

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| Open docker-compose.debug.yml | I said to wait for how we might use these compose files. You saw we were already using them, but you might have thought that was a little overkill.  Turns out, it’s easier to start with compose, than add it later on. And, how many of you think you’ll only be developing with single containers? |
| Add  services:  web-node:  image: stevelasker/web-node  ports:  - "80:3000"  environment:  - APIPORT=5000  - HTTPPORT=3000 | We’re going to add our web-node image, and notice we’re going to reconfigure it’s default port. We’re going to use port 80, as any good webserver should.  Again, this is the beauty of containers. We can change the external behavior, regardless of the internal behavior. Assuming it was setup to be externalized. |
| Start debugging | As we start our debugging session, we’ll see the .NET Core container build, the images be instanced, and we have our… web API come up. Well, that’s because we told VS Code to do that. |
| <http://localhost> | If we simply go back to the root URL, we can see our Node front end, call our .NET Core Web API, and our breakpoint hit. |
| launch.json  "launchBrowser": {  "enabled": true,  "args": "http://localhost", | To set the proper startup, we’ll change our launch args to be http://localhost, and we’re good to go |

# Optimized Production Images

Let’s take a moment and talk about the difference between development and production images

We have a competing set of priorities here. Or, more like we have some reality that we need to deal with.

During development, our priorities are:

During production, or priorities are:

It would be nice if these could not require differences, but they do, at least for today.

REPOSITORY TAG IMAGE ID CREATED SIZE

stevelasker/web-node latest 1ce1b6900796 4 seconds ago 663.4 MB

node latest 9121f2a78909 32 hours ago 659.4 MB

stevelasker/api-dotnet debug 474598746dd6 5 minutes ago 888.6 MB

microsoft/dotnet 1.0.0-preview1 1707bb9a2d73 9 days ago 585.1 MB

stevelasker/api-dotnet latest 183a82e06e42 3 minutes ago 247.2 MB

microsoft/dotnet 1.0.0-rc2-core 137095b10869 9 days ago 242.3 MB

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| Terminal docker images | If we look at our image sizes on disk, we can see that our dot net image is slightly larger than our node image. |
| ./dockerTask.sh build release | However, let’s see what happens when we build a release optimized image |
| Docker images | We can see a pretty big drop in size as we no longer have all the SDKs, debugger and other components we need for development, but not for production.  This will also impact our startup perf, container density per host and network traffic as your container orchestration system moves images across its network to spin up on various hosts |
|  | And, this is just the start. We’ve only recently started looking at .NET in containers. We have a bunch of work we’re planning that should make this even better. |

# CI/CD Container Workflow

Ok, we’ve done a bunch of work locally. We’ve developed and debugged our front end app in Node.js and our API in .NET Core. We’ve prepped them both for running in production. We even tested them working together, even if we weren’t the same team developing both.

Now we’re ready to check-in. How’s this going to work?

|  |  |
| --- | --- |
| **Demo step** | **Talk track & notes** |
| <https://azuredevex.visualstudio.com/Polyglot> | We’re going to open our Visual Studio Team Services build configuration  Here we have a full dashboard for what’s happening in our various teams.  Review various tiles  Here we can see a quick visual result of our various builds and deploys are going  We can see we have 3 different projects   * Web-node * API-dotnet * deploy   We can also see our deploy tile has more activity, and that’s because it does a deploy for each validated image build |
| Click web-node Tile | We can first look at our Node build steps  Here we see a historical list of successful, or not so successful builds |
| Click Edit | If we go into the Build Definition, we can see what we’ve got |
|  | We see a collection of docker build steps, with some testing in the middle  What this outlines is we’re building our image, and pushing the build number tagged image  We then run a set of tests, to make sure the image is actually working. Like most of us, the tests are disabled, but we only have 40 minutes here, so…  Assuming the deep testing passes, we re-tag the image latest and push it to our registry. |
| Point to trigger deployment | If we look at this last step, it’s the way we’re triggering deployment of the third project. This is how we’re using either one of the two repos to tell the Release definition to run. |
| Click api-dotnet 🡪Edit | Now, let’s look at the api-dotnet build definition. |
|  | These are largely the same. The only minor difference is we first do a dotnet restore and publish. Publish is a bit of a misnomer.  This first restores all our packages from our public repo, Nuget.org. This is where anyone can contribute to packages for developers to use, and is a growing vibrant ecosystem.  Next, we compile and assemble all the content that will be placed in the runtime container. This is the “publish” step. Publish implicitly does dotnet build for us.  Once we have the content to place in the container, we resume our normal docker image build and validation steps.  With the last step to trigger our common deployment. |