COM.SE.140 Continuous Development and Deployment 03.12.2020

Tampereen yliopisto
Tampereen ammattikorkeakoulu

Devops Project Assignment
End Report

Sisällys

1. Description of the CI/CD Pipeline

As per the instructions of the project we also started to setup a local dockerized Gitlab + Gitlab CI + Gitlab Runner combo but after we had been working on the local virtualized environment we quickly realized that there is a better way: gitlab.com. Gitlab.com offers free of charge shared runners on their platform and it made it easy for us to work together efficiently. Also one of the optional requirements was implementing some sort of static analysis for the project. We implemented the whole project using .NET Core and C#. There is very little static code analysis tools available for .NET Core applications that could be run within a CI pipeline. .NET Core SDK has some built-in static analyzers which do provide warnings for mostly very obvious mistakes but they don’t do as much as something like SonarQube. Even though SonarQube’s support for C# is not on the same level as it is for other languages we still decided to use it. In order to do that our project had to be made open source to get free tier offering at sonarcloud.io. This is also one of the reasons for using Gitlab.com instead of the local setup.

Our CI/CD pipeline consists of 4 stages which are in order: build, test, analyze and deploy. The whole pipeline is documented as code in the .gitlab-ci.yml. The first phase build does the building of our application and a very basic short run to make sure that the build actually works. The very basic short run is the same that was defined in the message queue exercise and we thought it was a great smoke test to use in the build phase to see that the result of the build was actually usable to some degree. Our build works by running “docker-compose build --no-cache" and the execution of the test run can be done by running (assuming Unix OS) “TEST_RUN=True docker-compose up --abort-on-container-exit". This means that we are using docker-in-docker here.

Our 2nd stage is the test phase. In here we execute all our unit tests and if one of them fails we don’t move on to the next stage and the pipeline fails. This is the default operation of Gitlab CI. Executing the tests is done inside a standard .NET Core 3.1 container by running the command “dotnet test”. We have used NUnit unit testing framework to implement all the tests. The custom thing here is that we actually collect the test coverage information here by using necessary switches with the test command and store them as artifacts of the test stage. This way we don’t have to run the tests again in the analyze phase. The collection of test coverage also required that we added a coverlet.msbuild -NuGet package dependency to all of our test projects.

The 3rd stage is the analyze phase. As mentioned earlier in this document we are using SonarQube to conduct our static analysis since it is probably the best one out of the very limited pool of tools. It was also mentioned that C# support is not on the same level as other languages and it definitely shows in this stage. First of all, we are using a custom made Docker image that combines Java with .NET Core as Java is a requirement for the SonarScanner for MSBuild. Do note that SonarScanner and SonarScanner for MSBuild are two different things. SonarScanner for example has very good integration with SonarQube and there is a ready-made Docker container which you can start using without any hassle. SonarScanner for MSBuild can be installed as a global .NET tool and that's the way we chose to do it as it is also included in the custom image. Now to actually do the analyzing we first have to issue a begin command for the SonarScanner for it to start recording and after that we run our build command again. SonarScanner gets the required data through the build and after the build is finished we end the recording and the collected data is sent to sonarcloud.io. Here we also include the code coverage reports saved in the artifacts from the previous stage and required Gitlab plugin parameters by using correct switches.

The final stage is deploy. Here we deploy the service to a private server in external public cloud as per the optional requirement in the instructions. We use SSH to connect to the remote server and deploy it there using Git and Docker compose. This stage is the only one that is limited to master branch only as we don't want to do deploys anywhere else since this is our production environment. We had to utilize base64 encoding here because Gitlab CI environment variables didn't allow us to use the key in plain text directly.

2. Analysis of architecture

After completing the project both of us agree that the general architecture of the project is pretty good. We don't see any glaring issues with it. There are a lot of different areas that could use some optimization and cleaning of code but all in all we think that this meets the requirements presented in the project instructions. The architecture of most components is largely dictated by .NET Core way of doing things. We only had to get creative with the .NET Core console applications but even for those there was a lot of ready-made architecture solutions so we didn't have to reinvent the wheel.

The components built for the application were quite trivial and could have been made more generic to accept wider input and work with more similar scenarios and so that it would be easier to understand the functionality of these components. On the other hand this is a good thing that we didn't implement an overly abstracted system for simple tasks. This makes our system easier to work with when we are dealing with requirements like the current ones.

The tests we wrote were quite powerful and by using them it was very easy to implement the features without greater hassle. We used mocking library to mock any dependencies to be able to focus on single components when testing and not to be dependent on too many components at the same time. Also, we were able to ensure the functionality and expected usage of those mocked components during the tests which made the tests more reliable with TDD when actually implementing the tested features.

If we had to name one thing that we would focus our efforts on it would definitely be error handling. We didn't really implement too much error handling because the whole system is run within a single docker network and usually it is only run locally. But if this system was to be used in a long running production situation we definitely should add more error handling and more tests for it.

3. Instructions for the examiner

To get our application up and running one has to simply have Docker compose installed and run **docker-compose up** in the root directory of our project. This will deploy our whole project into Docker containers locally. If running tests is required, we recommend installing .NET Core SDK 3.1 or using a Gitlab CI and Runner to run the tests for you.

We have not defined a way to execute the test locally with Docker because we haven't needed that sort of feature. To run the tests locally just run **dotnet test** in the root directory of the project. If you are using Visual Studio, you can load the Solution file and execute the tests using the Visual Studio Unit testing tools, also, which is visually much more pleasant way to test the project, but of course requires Visual Studio with correct .NET tools to be installed.

When changing the system states using PUT requests to the API gateway /state endpoint, there is a special requirement to use "JSON" format for the request body, in other words, the request should have "Content-type"-parameter set to "application/json". The request body payload should be just plain JSON string, e.g. "INIT", "PAUSED", "SHUTDOWN" or "RUNNING", with quotes included.

Example request for changing service state to PAUSED with cURL: `curl -X PUT -H "content-type: application/json" -d "\"PAUSED\"" localhost:8081/state`

A remote version of the application is running in the public cloud at https://devops.peltonet.com/. Be careful! If you send the SHUTDOWN state to our service there is absolutely nothing you can do to get it back up again as the instructions clearly stated that **all** containers should be powered off so this also includes ApiGateway. To get it back up running again, would require another deployment using the pipeline. Our repository, source code and collaboration history can be found here: https://gitlab.com/kapseliboi/devops-project

and our SonarQube analysis report page can be found here: https://sonarcloud.io/dashboard?id=kapseliboi_devops-project.

Implemented features / features to test (italic means optional):

- Continuous integration (.gitlab-ci.yml)
- Automated testing framework
- Changes to the application (API Gateway)
- *Static analysis (SonarQube)*
- *Deployment to external cloud (available through URL)*
- *Node-statistic –endpoint (API Gateway)*
- *Separate monitoring / logging service (Seq)*
- This End Report (EndReport.pdf)

In order to test the automated testing framework you need to have .NET Core SDK 3.1 installed as described earlier. ApiGateway can be tested by just using Docker compose to run the system. The static analysis can be hard to test. It would require us to invite you to our project and to give examiner rights to run tests. We think that this is pretty much unnecessary because you have the whole history of our test runs available to you but if the examiner requires access to our repository with more rights then please contact us. Our separate monitoring / logging service can be troubleshooted at URL https://logger.peltonet.com and locally at http://localhost:5341/.

4. Example runs of failing and passing

Our whole pipeline and commit history is fully and publicly available at https://gitlab.com/kapseliboi/devops-project/. If you require something else, please let us know.

5. Main learnings and worst difficulties

Person 1: For me the main learnings definitely came from writing unit tests with C#. I hadn't done that before to this degree and it is definitely something that I can directly utilize on work projects. Learning to setup the CI/CD pipeline was also a valuable lesson even if I wouldn't set one up ever at work but it gave me more understanding of how it works which makes it easier for me to debug a CI/CD pipeline if necessary.

One very surprising thing was the fact that support for C# in SonarQube was awful. The analyzer works just fine, but there is no integration with Gitlab CI at all when comparing to other programming languages. At the same time this was probably one of the worst difficulties that came my way during the project. It took a lot of Googling around to get it working.

I think I also made the right choice when it comes to choosing a pair for this project. My pair has a lot more experience on working with C# and .NET world so I had the opportunity to learn a lot of valuable lessons about that topic as well and that I did.

Person 2: This was actually the first project where I used TDD approach, and that made it difficult first to begin implementing the tests and after that write the implementation. This was hard, because it was not easy to ensure that the tests would be correctly written and required exessive planning of the upcoming features beforehand and making quite broad decisions about the architecture and implementation.

Also, it required to be confident with writing unit tests and required to write quite errorless code as it was not possible to verify the functionality against a working implementation. I was surprised how slowly I gained process, compared to the usual

I also occurred additional completely project-unrelated problems with my working equipment, which significantly slowed down the working efficiency at the very final metres. My computer had serious heating problems, thus become unusable and some of the project code left unpushed and had to be re-written. The project could have otherwise been returned even on schedule. This issue was discussed with the course staff and the issue was understood.

My partner was very skilled with building the pipelines and working with the CI process which made it easier to collaborate and understand the CI

6. Amount of effort

Both of us used approximately 50-60 hours into this project. The project took greatly more effort than we estimated at the beginning and the amount of work was surprisingly high, even though not even all possible features were implemented.

7. Description of individual roles

- focused on building tests for the existing functionality. This required a lot of refactoring of the original services to make them testable. - also handled the development of the Gitlab CI pipeline including the SonarQube integration and wrote most of the end report. - only created two new features: one to ApiGateway and that was the node-statistic –endpoint, and the other one was Seq logging service. - handled the ApiGateway implementation and implementing of the state's effects to all services. - also configured our server to accept the SSH connection and to have Docker compose available for running the application complex. - also configured a reverse proxy through our server to use a certificate and HTTPS connection and a simple subdomain for accessing the gateway service and also configured proxy for the logging service (seq). Overall, the workload in the team was quite well balanced.