Frontend local development with Docker and K3s

Table of Contents

- Agenda ... 1
- Requirements ... 2
  - Why those requirements ... 3
- Microservices Overview ... 4
  - Architecture ... 4
  - Automation ... 6
  - Configuration ... 15
  - Service Skeleton ... 17
  - Readiness Checklist ... 17
  - Documentation ... 20
  - Services ... 20
- Local Environment ... 21
  - K3s, Helm, and buildx preparations ... 21
  - Book Frontend ... 26
- What Next ... 34
  - Learning paths ... 34
  - Next workshops in the microservices series ... 34
  - Related Open Source project: ... 36
- Cleanup ... 36
- QnA ... 37

Solution which will be proposed during these workshops was created to simplify full process of developing applications. Main target of it is to reduce development time lost to play with the infrastructure in all teams: development, qa, and operations. For front-end development, it doesn’t matter whether we use microservices or a monolith, but this workshop shows how to try to solve the problems that microservices introduce to front-end development.

Agenda

- Start 10:00
- Section 1:
  - Architecture

  - Automation
  - Configuration
  - Service Skeleton
  - Readiness Checklist
  - Documentation
  - Services
- Coffee break 11:00 - 11:15
- Section 2:
  - Book Frontend (demo + practice)
- Coffee break 12:30 - 12:45
- Section 3:
  - What's Next
  - Cleanup
  - Q&A
- Ends between 13:00 and 13:30 - depends on Q&A session

Requirements

For this Workshop you will be working as all groups: Developers, QA, and Infrastructure - and you need this software to be installed:

- Languages
  - JavaScript (programming language used for our example project)
  - Ruby (documentation)
  - Python (automation and mocks)
- Services
  - Docker CLI, Docker Engine / Colima / Podman, Docker Compose / Podman Compose, Docker BuildX (containers)
- Tools
  - Brew (package management)
  - Ansible (configuration management)
  - docker-compose (container management)
  - kubectl (kubernetes cli)
  - kubectx (kubernetes cli extension)
  - helm (kubernetes package management)
  - k3d (lightweight kubernetes)
  - k9s (kubernetes management tool for cli)

You can find how to install this software here: https://local-workshops-requirements-guide-b9c7de.gitlab.io

Windows users should use WSL 2, or install VirtualBox, and install linux system in the VM.

In case if you will plan to experiment and fork our repositories to your own Gitlab group you will need a gitlab runner working on your laptop/machine connected to this personal group in the Gitlab. You will find documentation how to create it here: https://gitlab-runner-k3d-guide-1aca27.gitlab.io

Why those requirements

Like mentioned in this workshop you need all of this software installed but in the real life not.

JavaScript programing language will be used normally only by Developers. Testers and Infrastructure in most of the cases do not need it to be installed.

Ruby programing language in our case also will be used normally only by Developers and in this example they do not need to be Ruby programmers - tool we are using here for the documentation creation requires it. We expect developers to be responsible for documentation creation and maintenance.

Python programing language is used here by a Developers and QA. It will be easy to remove this need for the Developers just by serving doc not with using python but with using ruby. QA will need this language to work with our MOCK examples.

Brew tool is required in our setup only if you will choose this tool to install almost all required software.

Ansible is the configuration management tool will be used by all groups as our flow is based on it.

Docker Engine - For Linux and Windows users (WSL2) Docker Cli is part of the Docker package. Mac users needs to install a replacement like for example Colima. It is used by the Docker CLI to physically manage containers.

Colima - This is low weight Container Engine compatible with Docker Cli. In our case Mac users using it instead of the Docker Engine which currently is available on Mac only through paid Docker Desktop.

Podman - Podman is an open source container, pod, and container image management engine. Podman makes it easy to find, run, build, and share containers.

Docker Compose / Podman Compose - Easy to use simple orchestration tool for managing and configuring multiple containers.

Docker BuildX - This is a Docker CLI plugin for extended build capabilities with BuildKit.

All those tools:

kubectl - Kubernetes CLI will be used to check if our cluster works

kubectx - CLI tool extending kubeclt is used by our local deployment script to be sure that proper cluster and namespace is used.

helm - Package manager for Kubernetes used to deploy applications to our kubernetes cluster.

k3d - CLI wrapper for the K3s clusters which we will use to create out local kubernetes compatible K3s cluster

k9s - terminal based UI to interact with Kubernetes clusters

will be used mostly by the Infrastructure but also in many times by Developers and QA when working with Continuous Deployment.

Microservices Overview

Architecture

Our target backend architecture will be looking like this:

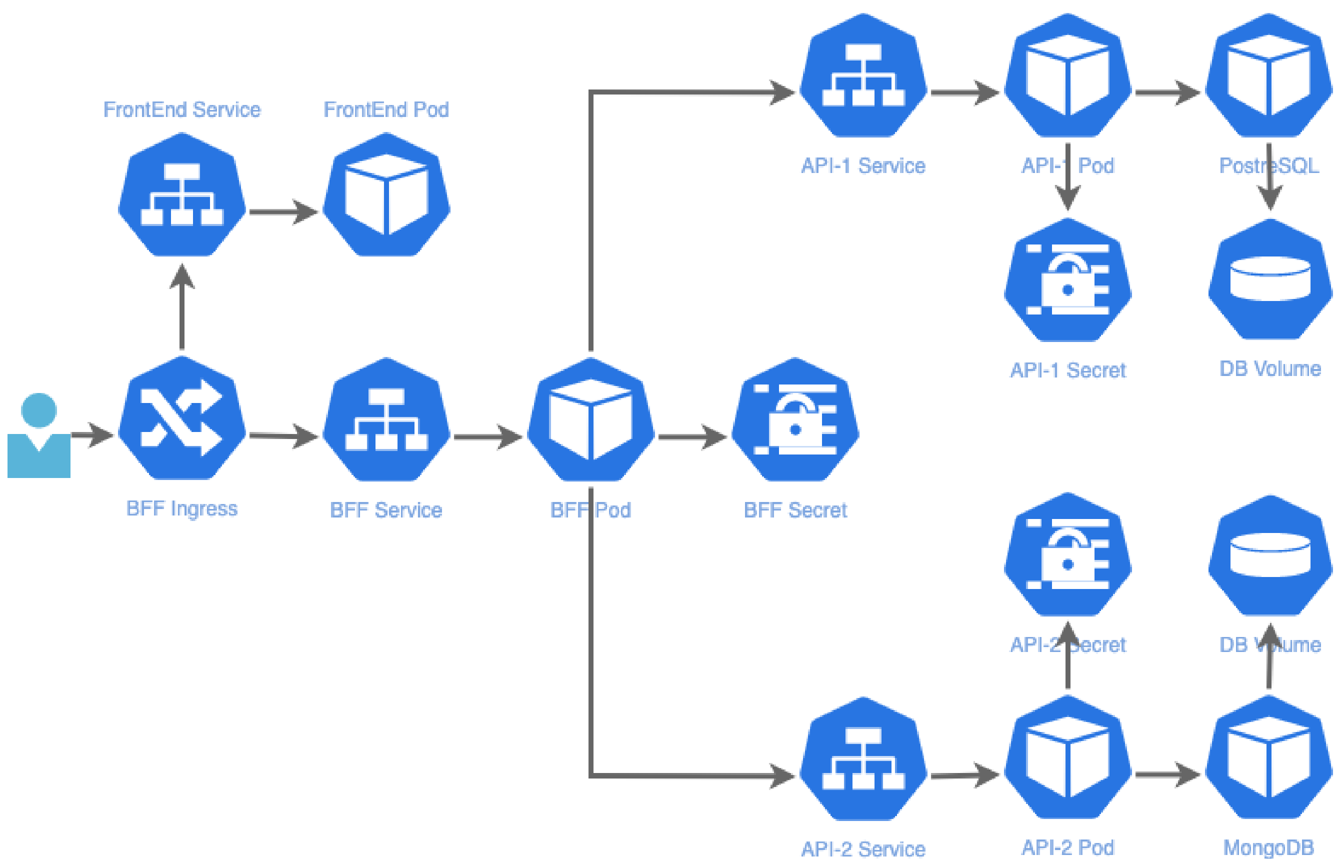

Figure 3: Target System Architecture

- Our FrontEnd Service will be simulated with Book Frontend.

We have three backend microservices prepared which we will be used by our deployed application:

- Our BFF Service will be simulated with Book API Gateway Service.
- Our API-1 Service will be simulated with Book List API Service.

- Our API-2 Service will be simulated with Book Admin API service

In the upper target system which will be available on the staging and production environments our frontend is talking through the Ingres with our BFF Service which is operating in the fully working environment.

Locally we will be working with the mocked version of this architecture which will be looking like this:

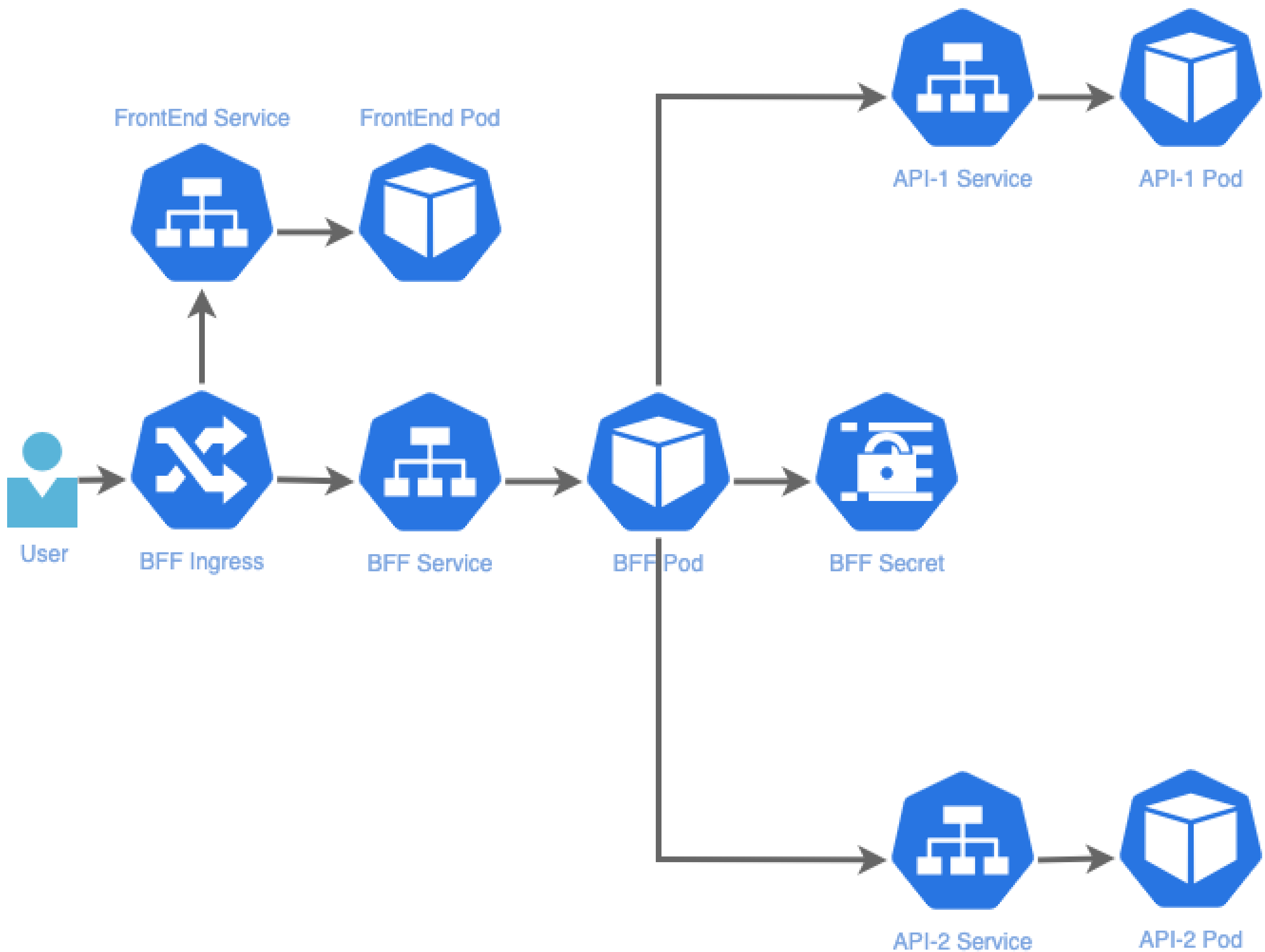

Figure 4: Local Mocked System Architecture

- API-1 is like mentioned before mocked with Book List Mockup API Service
- API-2 is also like mentioned before mocked with Book Admin Mockup API service

As you can see this will help our frontend team to test and work on the frontend services which depends on other services. With this design you do not need to recreate full architecture with all dependencies to just work.

Our mockup services are created using Python[1] language. In addition, for maximum simplicity we are using also Flask[2] micro framework. To be sure that all mocks are synced with the real services it is those services teams job to assure that mocks are synced with them.

Test doubles: Mocks, Stubs, and Fakes

```
For simplicity we are naming our Stubs as Mocks as this name is in most of the cases
```

```
used for Fakes and Stubs all the time.

Fake: Implements a shortcut to the external service, for example in memory database
instead real database. This method of mocking requires changes in the code.

Stub: It is a hard coded version of the external service and our examples are in the
reality Stubs.

Mock: We use mocks only to verify that access was made. Mocks do not hold data.
```

First question you should ask yourself is:

Do we really need to use microservices and why not to use monolith instead?

This question is directed at the back-end team, and you should ask them why?

Automation

DevOps methodology which is currently used in many projects requires us to automate everything we can. In this project we will be using many tools and practices connected with this practice.

Our example projects have a lot of scripts which automates our work. Please remember that to show this concept in the as simple as possible way we are not using more efficient workflows.

Another thing important to mention is that in a normal projects even if this simple method will be used there will be much fewer files. We have that many because we plan to show many ways of working.

Most common practice is to have one script which should just have many commands. This can be achieved very easily with tools like make[3], maven[4], gradle[5], and many others.

Let’s do a fast review of scripts, and other files created for automation we have in our examples.

Build

Let’s start with explaining our build related files which we are using to build our binaries and containers

scripts/build.sh

This script have inside command which is building our applications. It will be used during normal development by a developer to check if we can generate our frontend application with our scripts.

scripts/build-multi-stage-container.sh

This script is automating process of building docker image based on the multi-stage docker file. In our example this script is used for our K3s local cluster using multi-stage docker file: `multi-stage.Dockerfile`. This flow shows that our application can be built for desired architecture and operational system easily just by having a little more complex docker file.

scripts/build-single-stage-container.sh

This script is automating process of building docker image based on the single-stage docker file. It is the most basic and commonly know flow. We are using this script to check locally if our build process will NOT FAIL in the Continuous Integration pipeline and it uses file: `single-stage.Dockerfile`.

multi-stage.Dockerfile

Example of Dockerfile which is building service in the first stage and using generated binary in the second stage. In our flow this file is used by the `build-multi-stage-container.sh` and also by our docker compose file `compose-test.yml`.

scripts/multiplatform.build-multi-stage-container.sh

This script shows case when we wanted to build image for multiple architectures and operational systems. It is just a example which uses file: `multiplatform.multi-stage.Dockerfile`.

multiplatform.multi-stage.Dockerfile

Example of Docker file prepared to generate image prepared for multiple architectures and when modified also multiple operational systems. This file require Docker BuildX and may not work with other image build tools like for example Kaniko.

scripts/serve.sh

Script which is running our application in a development mode.

single-stage.Dockerfile

Example of Dockerfile which requires binary to be properly build prior and only copies it to the docker image.

Example responsibility table for it looks like this:

File name	Main responsibility	Helpers	Comment
build.sh	Developers	DevOps, Infrastructure	Developers will prepare build process of their app for requirements created by DevOps based on Infrastructure requirements

File name	Main responsibility	Helpers	Comment
build-multi-stage-container.sh	Developers	DevOps	Main responsibility is on Developers but DevOps sometimes needs to help with creation of this process. Everyone will use this script, and it needs to be bullet prof
build-single-stage-container.sh	Developers	DevOps	Main responsibility is on Developers but DevOps sometimes needs to help with creation of this process. Everyone will use this script, and it needs to be bullet prof
multi-stage.Dockerfile	Developers	DevOps	Main responsibility is on Developers but DevOps sometimes needs to help with creation of this process.
multiplatform.build-multi-stage-container.sh	Developers	DevOps	This is just an example of multiplatform version of multi-stage related files
multiplatform.multi-stage.Dockerfile	Developers	DevOps	This is just an example of multiplatform version of multi-stage related files
serve.sh	Developers	DevOps	DevOps can just help with it creation and maintenance
single-stage.Dockerfile	Developers	DevOps	Main responsibility is on Developers but DevOps sometimes needs to help with creation of this process.

Documentation

Documentation related scripts:

scripts/build-doc.sh

This script is created to automate process of creating our documentation which is based in our

examples on Asciidoctor[6].

scripts/serve-doc.sh

Serves our generated version of Asciidoctor documentation.

Example responsibility table for it looks like this:

File name	Main responsibility	Helpers	Comment
build-doc.sh	Technical Lead	Everyone	We are assuming that this is technical documentation created by everyone in the team and that PM at least will be checking that this documentation is alive
serve-doc.sh	Technical Lead	DevOps	DevOps can just help with it creation and maintenance

Our Documentation files will be stored in the `doc` folder.

Local Configuration

We will be discussing how our configuration works a little in the next section.

Local Configuration Management related scripts:

scripts/local-configure.sh

This file is used to generate all configuration files we will be using in the examples for all environments we will be playing with in our example. It is part of our Infrastructure and Configuration Management automation and will be explained in the next section.

ansible/local-playbook.yml

This is Ansible Playbook file used by `scripts/local-configure.sh`. It is part of our Infrastructure and Configuration Management automation and will be explained in the dedicated workshop session.

ansible/local-vault.yml

File with variables for Ansible with encrypted by ansible vault secrets. It is part of our Infrastructure and Configuration Management automation and will be explained in the dedicated workshop session.

ansible/requirements.yml

Ansible Galaxy requirements file. We are using it to show to ansible-galaxy from where we need ansible roles to be downloaded. This file is used by `local-configure.sh` and `gitlab-configure.sh` shell scripts. It is part of our Infrastructure and Configuration Management automation and will

be explained in the dedicated workshop session.

Example responsibility table for it looks like this:

File name	Main responsibility	Helpers	Comment
local-configure.sh	DevOps	Infrastructure, Developers, QA	DevOps will create and maintain this file but Developers, QA, and Infrastructure needs to understand how it works as it will be used by them a lot
local-playbook.yml	DevOps	Infrastructure, Developers, QA	DevOps will create and maintain this file but Developers, QA, and Infrastructure needs to understand how it works as it will be used by them a lot
local-vault.sh	DevOps	Infrastructure, Developers, QA	This file stores our configuration variables and secrets and is mainly managed by DevOps but Infrastructure, Developers, and QA will be maintaining it too.
requirements.yml	DevOps	Infrastructure, Developers	This file is maintained by DevOps but Developers, QA, and Infrastructure needs to understand how it works

Our Ansible roles will be downloaded to the `ansible/roles` folder. Those roles need to be created and maintained by DevOps and Developers with a help from Infrastructure.

We created our Configuration Management based on the Ansible for a good reason and a lot of things connected with it have meaning which is impossible to explain during this workshop and will be explained during the dedicated session for Infrastructure and DevOps.

In most of the cases you will need to use a different way to achieve automated configuration of your application. If your project do not know how to achieve

> multienvironment configuration you can use our example as a starting point to find something which will work for you.

Local Orchestration

Docker Compose related scripts:

compose-k3s.yml
: We are using this docker compose to serve our Swagger Editor and UI when only using K3s based local environment. This is not a proper way of doing this - we should have separate K3s based setup for this but there was no time to prepare it.

compose-local.yml
: We are using this docker compose file for local development of our service.

compose-mocked.yml
: We are using this docker compose file for local testing and checking if our app is deliverable using mocked api. In the normal flow this file will be the only file you need if you need to use mocks instead of full app.

compose-test.yml
: We are using this docker compose file for local testing and checking if our app is deliverable. In the normal flow this file will be the only file you need.

Example responsibility table for it looks like this:

File name	Main responsibility	Helpers	Comment
compose-k3s.yml	DevOps	Developers, Infrastructure	As this file is mainly for Infrastructure for testing DevOps with Developers will make it ready
compose-local.yml	DevOps	Developers	Since this file is intended primarily for developers, they need to help DevOps create it in a way that suits their needs
compose-mocked.yml	DevOps	Developers, QA	Since this file is intended primarily for QA, Developers need to help DevOps create it in a way that suits their needs

File name	Main responsibility	Helpers	Comment
compose-test.yml	DevOps	Developers, QA	Since this file is intended primarily for QA, Developers need to help DevOps create it in a way that suits their needs

Local Testing

Local Testing related scripts:

scripts/local-test.sh

This file is used to start our local integration/feature tests of our service. We should always run it before pushing to the remote repository. It is part of our Infrastructure and Configuration Management automation and will be explained in the dedicated workshop session.

Example responsibility table for it looks like this:

File name	Main responsibility	Helpers	Comment
local-test.sh	QA	Developers, DevOps	QA will be maintaining this file and Developers and DevOps will be using it a lot and needs good understanding how it works. Additionally, DevOps may give requirements for it to QA

Local Scanning

Local Scanning related scripts:

scripts/local-scan.sh

This file is used to start our local container security scanning of our service. We should always run it before pushing to the remote repository.

Example responsibility table for it looks like this:

File name	Main responsibility	Helpers	Comment
local-scan.sh	Security	Developers, DevOps	Security will be maintaining this file and Developers and DevOps will be using it a lot and needs good understanding how it works. However, DevOps may give requirements for it to Security

Local Deployment

Now we will check our deployment related scripts:

scripts/deploy-k3s.sh

This script is using image crated by `build-multi-stage-container.sh` and deploying it to our local K3s cluster. With this we can use latest local version of our application in the Kubernetes compatible environment.

Example responsibility table for it looks like this:

File name	Main responsibility	Helpers	Comment
deploy-k3s.sh:	DevOps	Infrastructure, Developers	Infrastructure will help with making this process as compatible as possible with the target production, Developers will help with making this environment running

Our Deployment to Staging and Production Environments is managed by the CI pipelines.

CI/CD

Now we will check our Gitlab CI related scripts:

scripts/gitlab-configure.sh

This file is used by Gitlab CI to generate configuration file used by our service in the prepared in it testing environment. It is part of our CI automation and will be explained in the dedicated workshop session.

ansible/gitlab-playbook.yml

This is Ansible Playbook file used by `scripts/gitlab-configure.sh` to generate only those

configuration files which are used by Gitlab CI. It is part of our CI automation and will be explained in the dedicated workshop session.

scripts/gitlab-test.sh

This file is used by Gitlab CI to start service tests which we are running in the CI pipeline. It is part of our CI automation and will be explained in the dedicated workshop session.

ansible/gitlab-vault.yml

In our case just a symlink to `local-vault.yml`.

Example responsibility table for it looks like this:

File name	Main responsibility	Helpers	Comment
gitlab-configure.sh	DevOps	Infrastructure	CI configuration need to be as similar as possible to the real infrastructure configuration and Infrastructure needs to help to achieve this.
gitlab-playbook.yml	DevOps	Infrastructure, Developers, QA	CI configuration need to be as similar as possible to the real infrastructure configuration and Infrastructure needs to help to achieve this. Because playbook is also generating application configuration Developers needs to help Devops with having this configuration in sync all the time and QA will be using it also in this case and needs to understand how this works.
gitlab-test.sh	QA	Developers, DevOps	QA need to manage their testing script with a help of Developers and DevOps

File name	Main responsibility	Helpers	Comment
gitlab-vault.yml	DevOps	Infrastructure, Developers, QA	In our case it is a symlink to `local-vault.yml`. Responsibility was explained there.

Summary

syWhen creating your own automation based on our examples please consider those changes:

- Normally you should have only one docker file in the project. We are showing three cases here, and we are ending with `multi-stage.Dockerfile`, `single-stage.Dockerfile` and `multiplatform.multi-stage.Dockerfile`. You should have just a `Dockerfile` and choose between multi-stage build process where all is done by docker for a single architecture and operational system, multi-stage build process for multiple architectures and operational system, or just a basic single-stage build process. Having one file called `Dockerfile` also simplifies other commands as this is a default file name for a Docker.
- Normally you should also have only one docker compose file or just use local K3s cluster with the help of K3d tool. When you have only one file you should name it `compose.yaml` which is preferred in your working directory or if you use older standard `docker-compose.yaml` or `docker-compose.yml` file names. This will simplify commands as there will be no need to target specific file.
- I strongly advise you to use K3d/K3s when your target environment is Kubernetes based instead of Docker Compose.
- In the project it is normal that Developer have also a DevOps role and sometimes also an Infrastructure role. When project is bigger fewer roles we have on one person, but this not means you do not need to know what people with other roles do. It is important as people can go on vacation or went sick and someone needs to take their roles for a period of time fast.

Configuration

Configuration values storing method shown in those examples is also not the best solution. Please remember that for an infrastructure related parts of the configuration, secrets, and things which changes often you should use tools like Vault[7], or Cloud related resources like for example [8].

For a simplicity of the examples our configuration for all environments will be stored in the project repository. To show how to have at least minimum security with this concept we will be using in this example Ansible Vault[9]. Problem is that for a simplicity we will not store this password in a secure way and for this **WARNING** bellow:

Our examples have password you need to use when running `scripts/local-configure.sh` in the `README.md` file.

> NEVER EVER DO THIS!!!

Ansible which will be used to create configuration files will be run with a prepared script `scripts/local-configure.sh` this script looks like this:

```
#!/usr/bin/env bash
set -e
rm -Rf ansible/roles/secrets.*
ansible-galaxy install -r ansible/requirements.yml -p ansible/roles
ansible-playbook --vault-id @prompt -i 'localhost,' --connection=local ansible/local-playbook.yml
```

Current goal of the Ansible playbook this script runs is to generate service and helm configuration files for all environments and store them in the `secrets` folder.

Ansible role used by the Ansible playbook is downloaded from the other location which can be checked in the `ansible/requirements.yml` file. This role is using in our examples values stored in the encrypted `ansible/local-vault.yml` file to generate mentioned upper files.

After using this script you will have those files in the mentioned `secrets` folder.

Our Services configuration files:

ci.env.yaml
: Service configuration which is used in the CI pipeline.

docker.env.yaml
: Service configuration which is also used in a local environment but for more complex testing - mainly by DevQA when they are working on integration tests and end-to-end tests. It is also something what can be used to check if service is deliverable.

k3s.env.yaml
: Service configuration which is used by local kubernetes called K3s. It will be used mainly by DevOps to check if service is deployable.

prod.env.yaml
: Service configuration for our production environment

local.env.yaml
: Service configuration which is used for a local service development where Developer is actively developing service.

staging.env.yaml
: Service configuration for our staging environment. Environment which is used before production deployment.

Our deployments Helm values files:

k3s-values.yaml

Values file used by our helm chart to deploy to our K3s local cluster.

prod-values.yaml

Values file used by our helm chart to deploy to our production cluster.

staging-values.yaml

Values file used by our helm chart to deploy to our staging cluster.

We will be analyzing those files during a Workshop.

Service Skeleton

This workshop also shows concept of skeleton service which have standardized files inside. You can have more than one skeleton if you also like to go more in dept with language and framework you are using. This concept isn't new as many frameworks are generating skeletons when creating new app, what we are adding to it is core documentation files, core CI files and automation. Your skeleton service should be created in a way that you should just clone skeleton create new project in the git server and start using it right away to create new service, or use special CLI tool which can use your prepared skeleton to generate new service like for example Cookie Cutter.

Other method is to use your framework project generator and merge result of new project generation process with a template which have files related to the DevOps process which generator do not create.

Our example of this solution is strongly simplified and available here: https://gitlab.com/devops-training-info/templates/services/generic/standard-service.

To start understanding how to use this template please check starting version of the README.md file. As you can see there are many sections which you will be modifying for your service.

`README.md` is also a file where you should store your **Production Readiness Checklist** we will mention in the next section and also **Changelog**.

This template example is also introducing in project documentation concept we will discuss later.

You are deciding what is in your template. In many cases developers were just copying previous project to achieve similar goal. We prefer templates as it is more clean solution.

Now please compare template with our service: https://gitlab.com/devops-training-info/examples/javascript/vuejs/book-frontend

Readiness Checklist

Each our example service is having in the `README.md` section called **Readiness checklist**. This is a list which should be based on your project **Non-Functional Requirements** list. Our checklist is based on https://github.com/kgoralski/microservice-production-readiness-checklist which is **Microservice**

Production Readiness Checklist project.

Storing it per service in readme guarantees that each requirement will be ticked with a comment in the git repository and all will be documented with in git repo.

For example in a start you will have record like this in it:

- [ ] **No shared database between different services** - a DB instance should only be used by one service exclusively.

when you will check that yes this is true and this requirement is set - you can just commit change in the repo looking like this:

- [x] **No shared database between different services** - a DB instance should only be used by one service exclusively.

with your commit message showing that you checked it and that from your perspective this task is done.

Section by Section example

General Rules - which are more architecture design related

```
#### General Rules

- [ ] **No shared database between different services** - a DB instance should only be
used by one service exclusively.
- [ ] **Not breaking the one-hop rule** - _By default, a service should not call
other services to respond to a request, except in exceptional circumstances._. The
exception can be for example Backend For Frontend (like GraphQL) which can compose and
aggregate data on top of other services.
- [ ] **Prefer APIs to Sharing SDKs**. Try to avoid using SDKs between the services,
it is not needed.
```

Documentation - which is connected with a planned standardisation of your project documentation

```
#### Documentation

- [ ] **README.md** - self-explanatory service name, how to run it locally and
domain/subdomain, bounded context described
- [ ] **Project documentation** - if possible should be kept with a code
- [ ] **Architecture docs / [C4 Model diagrams](https://c4model.com/)**
- [ ] **Development docs** - more detailed version of service development
documentation than **README.md** which will be used by new developers to start
development of the service and for other teams to cooperate with development team.
- [ ] **API Open Specification** file in root directory or other location known by
everyone: `openapi.yaml` file
- [ ] **API versioning** - if needed
```

Testing and Quality - which is a list created by cooperation between QA, Developers, and product owners

```
#### Testing and Quality

- [ ] **Linters** (with reports that can be exported to e.g. SonarQube)
- [ ] **Automatic code Formatter or code Format Checkers** (e.g. gofmt, ktfmt)
- [ ] **Test coverage above 70%** (use common sense, just getting to the required
number of coverage is not a goal here)
- [ ] **Functional/e2e/acceptance tests** in place
- [ ] **Load Tests** (at least basic ones) especially if higher traffic is expected
- [ ] **Contract Tests** are recommended if there is service-to-service communication
via HTTP (example: [PACT tests](https://docs.pact.io/))
```

Observability - which is in many cases ignored till there is to late and should be planned before project starting and created with cooperation with a Infrastructure team amd all representatives of the Development team.

```
#### Observability

- [ ] **Logging** in general https://12factor.net/logs
- [ ] **All logs are written to STDOUT / STDERR**.
- [ ] **Logs are written in JSON**.
- [ ] **No sensitive data is logged**
- [ ] **Monitoring**
- [ ] Integration with a monitoring platforms and Dashboards in place.
- [ ] Business metrics added to the dashboards
- [ ] **Tracing**
- [ ] **Distributed tracing configured**
- [ ] **Error tracking configured**
- [ ] **Alerts are configured**
```

Operations and Resiliency - which will allow you to enter production stage in the future.

```
#### Operations and Resiliency

- [ ] **Staging environment exists**
- [ ] There is **autoscaling** in place (based on CPU, memory, traffic,
events/messages e.g. HPA with K8S)
- [ ] **Graceful shutdown**: The application understands SIGTERM and other signals and
will gracefully shut down itself after processing the current task.
https://12factor.net/disposability
- [ ] **Configuration via environment**: All important configuration options are read
from the environment and the environment has higher priority over configuration files
(but lower than the command line arguments). https://12factor.net/config
- [ ] **Health Checks**: Readiness and Liveness probes
- [ ] **Define [SLO/SLI/SLA](https://cloud.google.com/blog/products/devops-sre/sre-
fundamentals-slis-slas-and-slos)**
```

```
- [ ] Build applications with **Multi-tenancy** in mind (sites, regions, users, etc.)
```

Security and Compliance - which is also in too many cases prepared to late and our production deployment without security can be blocked.

```
#### Security and Compliance

- [ ] If your service does need to be accessible through the public Internet
- [ ] **Authentication/Authorization** in place if needed / JWT / Cognito / Auth0
- [ ] Ensure it lives behind our Cloudfront **CDN** (and uses WAF if necessary)
- [ ] **Vulnerabilities scan check**
- [ ] **Does not violate any licenses**
- [ ] **GDPR** data not exposed (https://gdpr-info.eu/art-4-gdpr/)
- [ ] **PII data not logged or stored without any good reason** (ask your DPO) - [Best practices to avoid sending Personally Identifiable Information (PII)
](https://support.google.com/adsense/answer/6156630?hl=en), Check Data Retention Policies
```

Documentation

There is many ways you can create and store your documentation. In this workshop we will be showing concept of storing documentation with a code. To make possible to use this documentation in many ways we are using Asciidoctor which generating it from AsciiDoc files.

AsciiDoctor can generate it in many formats, most common is HTML and PDF. We are visualizing end result using Gitlab Pages. Generated version of the template documentation can be found here: https://standard-service-d001a2.gitlab.io.

This kind of the documentation should also explain all template files you have in a template.

In many cases you can use any Static Site Generator to generate your in project documentation. Asciidoctor solution we are showing allows also sending generated documentation to the Confluence and is just one of the simplest solutions we know.

Services

FrontEnd

Our FrontEnd service is created using Node.js[10] which is an open-source, cross-platform JavaScript runtime environment. We are also using Vue.Js[11] which An approachable, performant and versatile JavaScript framework for building web user interfaces.

BackEnd

Our BackEnd services are created using Go[12] Language.

- For a CLI part of our application Cobra[13] was used.
- For a configuration management part Viper[14] was used.
- For a REST service part of services Gorilla[15] web toolkit was used.
- For logging slog library was used.

You can check what commands and options each service have by using `--help` when running generated binary file.

Mockups

Like mentioned before our mockup services are created using Python language. In addition, for maximum simplicity we are using also Flask micro framework.

Local Environment

Now let's start practical part of this workshop.

In case there will be any problems with workshop instructions, and you will find yourself blocked somehow - than feel free to create issues ticket by using this link: https://gitlab.com/devops-training-info/documentation/microservices-frontend-local-development-guide/-/work_items and using button: **[New item]**. If you are on live workshop just ask a question.

K3s, Helm, and buildx preparations

K3s

To test if our service is ready for the deployment to the Kubernetes cluster we will use our local K3s installation.

Please use this command to create our **K3s** cluster:

```
k3d cluster create bookCluster --api-port 6443 --servers 1 --agents 3 -p "30700-30799:30700-30799@server:0"
```

You should have similar output to this one:

```
INFO[0000] Prep: Network
INFO[0000] Created network 'k3d-bookCluster'
INFO[0000] Created image volume k3d-bookCluster-images
INFO[0000] Starting new tools node...
INFO[0000] Pulling image 'ghcr.io/k3d-io/k3d-tools:5.8.3'
INFO[0001] Creating node 'k3d-bookCluster-server-0'
INFO[0002] Starting node 'k3d-bookCluster-tools'
INFO[0002] Pulling image 'docker.io/rancher/k3s:v1.31.5-k3s1'
```

```
INFO[0004] Creating node 'k3d-bookCluster-agent-0'
INFO[0004] Creating node 'k3d-bookCluster-agent-1'
INFO[0004] Creating node 'k3d-bookCluster-agent-2'
INFO[0004] Creating LoadBalancer 'k3d-bookCluster-serverlb'
INFO[0006] Pulling image 'ghcr.io/k3d-io/k3d-proxy:5.8.3'
INFO[0009] Using the k3d-tools node to gather environment information
INFO[0009] HostIP: using network gateway 192.168.96.1 address
INFO[0009] Starting cluster 'bookCluster'
INFO[0009] Starting servers...
INFO[0009] Starting node 'k3d-bookCluster-server-0'
INFO[0011] Starting agents...
INFO[0011] Starting node 'k3d-bookCluster-agent-1'
INFO[0011] Starting node 'k3d-bookCluster-agent-2'
INFO[0011] Starting node 'k3d-bookCluster-agent-0'
INFO[0020] Starting helpers...
INFO[0020] Starting node 'k3d-bookCluster-serverlb'
INFO[0027] Injecting records for hostAliases (incl. host.k3d.internal) and for 5
network members into CoreDNS configmap...
INFO[0029] Cluster 'bookCluster' created successfully!
INFO[0029] You can now use it like this:
kubectl cluster-info
```

Then you can check it with this command:

```
kubectl cluster-info
```

And you will have similar output to this:

```
Kubernetes control plane is running at https://0.0.0.0:6443
CoreDNS is running at https://0.0.0.0:6443/api/v1/namespaces/kube-
system/services/kube-dns:dns/proxy
Metrics-server is running at https://0.0.0.0:6443/api/v1/namespaces/kube-
system/services/https:metrics-server:https/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

let’s analise our creation command a little and our cluster itself. But before it to better understand it let start with some definitions:

```
A server node is defined as a host running the k3s server command, with control-plane
and datastore components managed by K3s. An agent node is defined as a host running
the k3s agent command, without any datastore or control-plane components. Both servers
and agents run the kubelet, container runtime, and CNI.
```

Starting with command:

- `k3d cluster create bookCluster` - this command will create cluster with the name bookCluster

and other parameters given after it will just help to create it in a way we wanted it to be created

- `--api-port 6443` - we are letting know to serve API on the standard port with will be used by our Portainer installation later. You can see this with `kubectl cluster-info` command.
- `--servers 1` - we are creating one server node
- `--agents 3` - we are creating three agent nodes
- `-p "30700-30799:30700-30799@server:0"` we are forwarding ports from 30700 to 30799 to the same set of ports in the server node. This is used by our NodePort setup later and Portainer we will install.

Let's see our nodes in details:

```
kubectl get nodes
```

Which will have similar output to this:

```
NAME                        STATUS   ROLES                  AGE     VERSION
k3d-bookcluster-agent-0     Ready    <none>                 3m18s   v1.31.5+k3s1
k3d-bookcluster-agent-1     Ready    <none>                 3m18s   v1.31.5+k3s1
k3d-bookcluster-agent-2     Ready    <none>                 3m16s   v1.31.5+k3s1
k3d-bookcluster-server-0    Ready    control-plane,master   3m26s   v1.31.5+k3s1
```

Now let's check how this look in docker:

```
docker ps
```

Which will have similar output to this:

```
CONTAINER ID   IMAGE                            COMMAND                  CREATED
STATUS                  PORTS
NAMES
ac51320054ee   ghcr.io/k3d-io/k3d-proxy:5.8.3   "/bin/sh -c nginx-pr…"   4 minutes
ago    Up 4 minutes            0.0.0.0:6443->6443/tcp, 80/tcp, 0.0.0.0:30700-30799-
>30700-30799/tcp, [::]:30700-30799->30700-30799/tcp   k3d-bookCluster-serverlb
9e175c325957   rancher/k3s:v1.31.5-k3s1         "/bin/k3d-entrypoint…"   4 minutes
ago    Up 4 minutes
k3d-bookCluster-agent-2
5d7c8c7a75fd   rancher/k3s:v1.31.5-k3s1         "/bin/k3d-entrypoint…"   4 minutes
ago    Up 4 minutes
k3d-bookCluster-agent-1
458d2bd68ad9   rancher/k3s:v1.31.5-k3s1         "/bin/k3d-entrypoint…"   4 minutes
ago    Up 4 minutes
k3d-bookCluster-agent-0
6fd8951e88a8   rancher/k3s:v1.31.5-k3s1         "/bin/k3d-entrypoint…"   4 minutes
ago    Up 4 minutes
```

```
k3d-bookCluster-server-0
```

Like you can see we have also special LB node `k3d-bookCluster-serverlb`.

You can always stop this cluster to save resources with:

```
k3d cluster stop bookCluster
```

And start it again with:

```
k3d cluster start bookCluster
```

Because you can have multiple clusters you can always change to this one with command like this:

```
kubectl config set-context k3d-bookCluster
```

or by using `kubectx` just:

```
kubectx k3d-bookCluster
```

Please create `services` namespace we will be using during this workshop:

```
kubectl create namespace services
```

Which will have output like this:

```
namespace/services created
```

Helm

We also need to have our helm repository registered if we didn't do this already:

```
helm repo add book-service
https://gitlab.com/api/v4/projects/81999726/packages/helm/stable
```

Which will have output like this:

```
"book-service" has been added to your repositories
```

Check which GitLab Runner versions you have access to:

```
helm search repo -l book-service/book-service
```

If you can't access the latest versions of book-service/book-service, update the chart with this command:

```
helm repo update book-service
```

Later in the workshop we will be using deployment script: `scripts/deploy-k3s.sh` which looks like this for `book-list` service.

```
#!/usr/bin/env bash
set -e
kubectx k3d-bookCluster ①
./build-multi-stage-container.sh ②
docker pull postgres:14-alpine
k3d image import -c bookCluster book-list:latest ③
k3d image import -c bookCluster postgres:14-alpine
helm upgrade -i -f secrets/k3s-values.yaml --namespace=services k3s-book-list book-
service/book-service --version 0.7.0 --dry-run --debug ④
helm upgrade -i -f secrets/k3s-values.yaml --namespace=services k3s-book-list book-
service/book-service --version 0.7.0 --wait ⑤
```

① We are switching to our local k3d-bookCluster which you need to install prior to this check

② We need to build our service image we plan to deploy

③ We need to import this service image to the cluster

④ In this line we are showing how to use `--dry-run` and `--debug` - you need to choose if this line will be run or add `#` in the start to not use it

⑤ In this line we are deploying for real to our cluster - you need to choose if this line will be run or add `#` in the start to not use it

Docker buildx

Check if BuildX is prepared for multi-arch build

```
docker buildx ls
```

In my case builder was not supporting multi-arch builds:

```
NAME/NODE     DRIVER/ENDPOINT   STATUS    BUILDKIT   PLATFORMS
default*      docker
\_ default    \_ default       running   v0.21.0    linux/amd64 (+4), linux/386
```

When your configuration is not supporting multi-arch just add it:

```
docker buildx create --name multiarch --driver docker-container --use
```

And check again

This configuration is not enough as you will have this message when build multi-platform images with it:

```
WARNING: No output specified with docker-container driver. Build result
will only remain in the build cache. To push result image into registry
use --push or to load image into docker use --load
```

We plan to show how to use buildah with Gitlab CI in the future to achieve multiplatform images in a more easy way.

Book Frontend

In case there will be any problems with this service code please create issues ticket by using this link: https://gitlab.com/devops-training-info/examples/javascript/vuejs/book-frontend/-/work_items and using button: **[New item]**. If you are on live workshop just ask a question.

You will need now to configure your local environment. For this please just use prepared shell script `./scripts/local-configure.sh` which look like this:

```
#!/usr/bin/env bash
set -e
rm -Rf ansible/roles/secrets.*
ansible-galaxy install -r ansible/requirements.yml -p ansible/roles
ansible-playbook --vault-id @prompt -i 'localhost,' --connection=local ansible/local-
playbook.yml
```

as you can see based on upper shell script ansible will ask you for a password.

You can find it in each project `README.md` file where it **SHOULD NOT BE !!!** and here. This password is: `ThisIsExamplePassword4U`

```
Vault password (default):
```

Next system will ask you for docker tags you like to use for the staging and production environments. You can just press `enter` in the both cases if you do not plan or can deploy to those environments from your laptop.

```
Version to deploy for the staging environment [latest]:
Version to deploy for the production environment [x.y.z]:
```

This just show you way to work if you like to play with deploying from your local environment but normally this is done with the CI/CD pipeline.

When script will end working you should see something like this:

```
PLAY RECAP
**************************************************************************************
**************************************************************************************
********************
localhost                  : ok=28   changed=0    unreachable=0    failed=0
skipped=20   rescued=0    ignored=0
```

and in the `secrets` folder you should have all configuration files you need.

This service have more detailed documentation which needs to be built. To do this you need to start `./scripts/build-doc.sh` script:

```
./scripts/build-doc.sh
```

This script is using Ruby and asciidoctor to generate project documentation. Your output will be looking like this:

```
Successfully installed webrick-1.9.2
Parsing documentation for webrick-1.9.2
Done installing documentation for webrick after 0 seconds
1 gem installed
Successfully installed bundler-4.0.11
Parsing documentation for bundler-4.0.11
Done installing documentation for bundler after 0 seconds
1 gem installed
In a future version of Bundler, running `bundle` without argument will no longer run
`bundle install`.
Instead, the `cli_help` command will be displayed. Please use `bundle install`
explicitly for scripts like CI/CD.
You can use the future behavior now with `bundle config set default_cli_command
cli_help --global`,
or you can continue to use the current behavior with `bundle config set
default_cli_command install --global`.
This message will be removed after a default_cli_command value is set.

...

Bundle complete! 6 Gemfile dependencies, 25 gems now installed.
Bundled gems are installed into `./.bundle/gems`
```

```
1 installed gem you directly depend on is looking for funding.
  Run `bundle fund` for details
```

If this success you can see this documentation in your browser with a help of `./scripts/serve-doc.sh` script:

```
./scripts/serve-doc.sh
```

which should give you this kind of output:

```
Serving HTTP on 127.0.0.1 port 8880 (http://127.0.0.1:8880/) ...
```

Just visit this link in the browser.

This is mostly useful when you are working on documentation itself as this documentation should be available in the Gitlab Pages for the project or by example Confluence and just should be generated by the CI and link to it should be located in the project `README.md` file.

Service Development and Continuous Integration

To show how normal development flow will look please start with creating service our api is using. To achieve this you can just run docker-compose like this:

```
docker-compose -f compose-local.yml up -d
```

you should have this type of output in the end:

```
[+] up 10/10
 ✔ Network book-frontend_book-frontend                Created
0.0s
 ✔ Container book-frontend-book-bff-1                 Started
0.2s
 ✔ Container book-frontend-swagger-editor-1           Started
0.3s
 ✔ Container book-frontend-swagger-ui-1               Started
0.4s
 ✔ Container book-frontend-book-admin-1               Started
0.3s
 ✔ Container book-frontend-redis-1                    Started
0.3s
 ✔ Container book-frontend-book-list-db-1             Started
0.3s
 ✔ Container book-frontend-book-admin-mongo-express-1 Started
0.3s
 ✔ Container book-frontend-book-admin-db-1            Started
0.3s
```

```
✔ Container book-frontend-book-list-1                Started
```

If there will be no problems you should have **Book Backend For Frontend** available on `localhost:8881`. This is API Gateway type of application we will be using for our Frontend Application.

You can check if this service is passing created tests by running script `local-test.sh`:

```
./scripts/local-test.sh
```

As this is a Vue application, and we do not need to build it to serve it just run `build.sh` script to check if we can create distribution correctly which will be required later for single stage docker generation example:

```
./scripts/build.sh
```

Our `serve.sh` script was prepared just for development serving here, as we need it now please run it:

```
./scripts/serve.sh
```

Your output should look similar to this:

```
  VITE v7.1.2  ready in 129 ms

  ➜  Local:   http://localhost:3000/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
```

You may ask how configuration works for our service.

We are using Pinia[16] which is intuitive store for Vue.js for storing our application configuration.

If you look into pinia file for the configuration with help of service: `src/services/ConfigService.js`:

```
import axios from 'axios'

const apiClient = axios.create({
  baseURL: '/',
  withCredentials: false,
  headers: {
    Accept: 'application/json',
    'Content-Type': 'application/json'
  }
})
```

```
export default {
  loadConfiguration() {
    return apiClient.get('config/local.env.json')
  }
}
```

you will see that we are just loading configuration by path where for our local environment `local.env.json` file looks like this:

```
{
    "api-url": "http://localhost:8881"
}
```

And we are using this service in the pinia `src/store/ConfigStore.js` store for a **ConfigStore**:

```
import { defineStore } from 'pinia'
import { ref } from 'vue'
import ConfigService from '@/services/ConfigService'

export const useConfigStore = defineStore(
  'ConfigStore',
  () => {
    const configuration = ref({
      apiUrl: ''
    })
    const error = ref('')

    async function loadConfiguration() {
      try {
        if (!configuration.value.apiUrl) {
          const res = await ConfigService.loadConfiguration()
          configuration.value.apiUrl = res.data['api-url']
        }
      } catch (e) {
        console.log(e)
      }
    }
    return { configuration, error, loadConfiguration }
  },
  {
    persist: {
      storage: localStorage
    }
  }
)
```

Additionally, we are using `piniaPluginPersistedstate` with a help of `pinia-plugin-persistedstate`

what you can see in this file. This persists our configuration in the `Local storage` by creating key: `ConfigStore` with value: `{"configuration":{"apiUrl":"http://localhost:8881"},"error":""}`.

This store is used when we need to access api endpoints. To better understand let's check our `src/services/UserService.js`:

```
import axios from 'axios'
import { useConfigStore } from '@/store/ConfigStore'

const apiClient = axios.create({
  withCredentials: false,
  headers: {
    Accept: 'application/json',
    'Content-Type': 'application/json'
  }
})

export default {
  async registerUser(user) {
    const configStore = useConfigStore() ①
    await configStore.loadConfiguration() ②
    return apiClient.post(configStore.configuration.apiUrl + '/book-admin/v1/admins',
user) ③
  },
  async loginUser(loginData) {
    const configStore = useConfigStore()
    await configStore.loadConfiguration()
    console.log(configStore.configuration.apiUrl)
    console.log(loginData)
    return apiClient.post(configStore.configuration.apiUrl + '/v1/login', loginData)
  }
}
```

① We are creating instance of a ConfigStore with a use of the `useConfigStore` function.

② We are loading configuration and waiting for it what is very fast when is cached in the `Local storage`

③ You can see here that we are using link to the API from the config store.

With this setup we can have different API links in each environment just by replacing our config.json file in the container.

For example our production config file looks like this:

```
{
    "api-url": "https://books-api.sandbox.gcp.cloud.devops-training.info"
}
```

Next what we can do is to visit Book Frontend in your browser and check if service is working

properly.

When you will finish your work please stop your serving application with `Ctrl` + `C`.

Please also turn off all containers when you will finish this section with command:

```
docker-compose -f compose-local.yml down
```

Service Development with Mocks

In many cases you will be not using full existing infrastructure but mocked one. In our example our Book BFF service can use mocks instead of real services. This will be very important in the next workshops which will focus on the local development for the FrontEnd services. We have special docker compose file prepared to show this flow.

```
docker-compose -f compose-mocked.yml up -d
```

In our example we are using the most dumb mocks possible what means there will be no reaction in them on changes done by UPDATE and/or DELETE type of operations. It is not important and in most of the cases enough to develop BFF app and FrontEnd aps which will be using this service.

Please start again your application with `serve.sh` script :

```
./scripts/serve.sh
```

Next what we can do is to visit Book Frontend in your browser and check if service is working properly.

When you will finish your work please stop your serving application with `Ctrl` + `C`.

Please also turn off all containers when you will finish this section with command:

```
docker-compose -f compose-mocked.yml down
```

Continuous Delivery and end-to-end testing

To be sure that our application is deliverable we need to check if docker containers we will end with are correct and will build with success in the CI.

Because in many cases we like to have testing, packaging service and building docker containers in separate jobs in the CI it is highly possible that we will be using single stage docker files.

To do that just run `build-single-stage-container.sh` script:

```
./scripts/build-single-stage-container.sh
```

to check if prepared Dockerfile `single-stage.Dockerfile` works correctly.

In other cases we will have multi-stage Dockerfile `multi-stage.Dockerfile`. Those files have multiple stages and only last stage is something what will end as the docker image. Stages before it are to prepare all required files for the last stage to ends with as minimal image as possible. In our example first stage is building binary of our service.

```
1 FROM registry.gitlab.com/devops-training-info/examples/docker/node-yarn:26.1.0 AS
  builder ①
2 WORKDIR /build
3 ADD . ./
4 COPY secrets/local.env.json public/config/local.env.json
5 RUN ./build.sh ②
6
7 FROM nginx:alpine ③
8 RUN mkdir /app
9 COPY --from=builder /build/dist /app ④
10 COPY nginx.conf /etc/nginx/nginx.conf ⑤
```

① Starting first stage we are naming here builder by taking as a source node:18 image

② We are building of our application using our prepared build script

③ Now we are starting our final stage, and we are using very small image nginx:alpine

④ We are copying our generated in the build stage application files to this one

⑤ We are also copying our prepared special version for our application NGiNX configuration

Our local environment which we will be using for end-to-end testing is based on this docker image and docker-compose.

Please first build your image using this shell command:

```
docker-compose -f compose-test.yml build
```

Now you can start it:

```
docker-compose -f compose-test.yml up
```

You will have now environment very useful for developing end-to-end tests which will be used in the CI or just to do manual testing like in our case.

Now just visit Book Frontend in your browser and check if service is working properly.

When you will finish please remember to stop your containers with `Ctrl` + `C`.

Continuous Deployment Testing

Just run our deployment script `deploy-k3s.sh`:

```
./scripts/deploy-k3s.sh
```

Depends on what you will choose - or you will see what will be sent to the cluster, or you will do a deployment, or both.

If you decided to deploy your application you can check how the deployment goes with the help of `k9s` tool:

```
k9s -n services
```

What Next

Learning paths

After this workshop you can to take those learning paths:

- A Cloud Guru Containers and Kubernetes related trainings - where you can learn more about Docker, Kubernetes, and Helm
- A Cloud Guru DevOps automation related trainings - where you can learn more about Configuration Management tools like by example Ansible and many type of Pipelines

Next workshops in the microservices series

Backend

For a Backend developers we have this workshop in the series:

- Backend local development with Docker and K3s for project using microservices architecture

Current agenda for the third edition looks like this:

- Start 10:00
- Section 1:
  - Architecture
  - Automation
  - Configuration
  - Service Skeleton
  - Readiness Checklist
  - Documentation
  - Services
  - Mockups

- Coffee break 11:00 - 11:15
- Section 2:
  - Book List API (demo)
- Coffee break 12:30 - 12:45
- Section 3:
  - Book Admin API (practice)
- Lunch 13:30 - 14:00
- Section 4:
  - Backend For Frontend (demo + practice)
- Coffee break 15:15 - 15:30
- Section 5:
  - Cleanup
  - What's Next
  - Q&A
- Ends between 16:00 and 17:00 - depends on Q&A session

DevOps Automator and QA

For a DevOps Automator and QA engineers we plan to release:

- Continuous Integration, and Continuous Delivery/Deployment for project using microservices architecture with a help of the GITLAB platform. Delivery/Deployment will be targeting prepared K8s cluster.

Current agenda for the first edition looks like this:

- Start 10:00
- Section 1:
  - Architecture Overview
  - Gitlab Overview
  - Frontend Service Pipeline Overview
  - Backend Services Pipelines Overview
  - Gitlab Container Registry Overview
  - Gitlab Pages Overview
  - Gitlab CI Overview
- Coffee break 11:00 - 11:15
- Section 2:
  - Your first pipeline (practice + explanation)
  - A complex pipeline (practice + explanation)

- Coffee break 12:10 - 12:25
- Section 3:
  - DevOps pipeline for our frontend microservice (demo)
  - DevOps pipeline for our frontend microservice (explanation)
- Section 4:
  - DevOps pipeline for our backend microservices (demo)
  - DevOps pipeline for our backend microservices (explanation)
- Lunch 13:45 - 14:15
- Section 5:
  - Simplified pipeline for our frontend app (explanation)
  - Simplified pipeline for our frontend app (practice)
- Coffee break 15:45 - 16:00
- Section 6:
  - What's Next
  - Cleanup
  - Q&A
- Ends between 16:15 and 16:30 - depends on Q&A session

Infrastructure

For Cloud Infrastructure Engineers and SysOps:

- Preparing and using Infrastructure required for the projects using microservices architecture with the help of the Kubernetes, Clouds, Ansible, and Terraform.

Like in the title we will be creating mainly Cloud Infrastructure with a help of Terraform and other mentioned tools. It is possible that also bare metal will be mentioned and example prepared.

Related Open Source project:

We have also a special OpenSource project MOB175:

- Preparing examples used in the workshops series connected to the microservices architecture.

Anyone who is interested can join in a free time and people who are currently without project can let us know if wanted to join this project. Please contact me to gain more details.

Cleanup

You can remove created K3d cluster created with this command:

```
k3d cluster delete bookCluster
```

You can clean up your docker engine with this command:

```
docker system prune -a --volumes
```

All tools installed with a help of the `brew` can be uninstalled with the command `brew uninstall` plus `tool name`.

```
brew uninstall tool-name
```

QnA

Waiting for Questions :)

This work is licensed under a Creative Commons Attribution 4.0 International License

[1] Python Language https://www.python.org/

[2] Flask micro framework https://flask.palletsprojects.com/

[3] Makefile tutorial https://makefiletutorial.com/

[4] Apache Maven https://maven.apache.org/what-is-maven.html

[5] Gradle https://gradle.org/

[6] Asciidoctor https://asciidoctor.org/

[7] HashiCorp Vault https://www.vaultproject.io/

[8] GCP Secret Manager https://cloud.google.com/security/products/secret-manager

[9] Ansible Vault documentation https://docs.ansible.com/ansible/latest/user_guide/vault.html

[10] Node JS https://nodejs.org/en

[11] Vue https://vuejs.org/

[12] Go Language https://go.dev/

[13] Cobra library https://github.com/spf13/cobra

[14] Viper library https://github.com/spf13/viper

[15] Gorilla web toolkit https://www.gorillatoolkit.org/

[16] Pinia https://pinia.vuejs.org/