IT Infrastructures / Cloud Computing

Pre Requirements for Workshops **2**
Basics **2**
Methods 2
Design Thinking 2
Wireframes 3
Software Development Life Cycle 4
Scrum as agile framework 4
Product vs. Project 5
Proof Of Concept (POC), Prototype and Minimum Viable Product (MVP) 6
Architecture Concepts 7
Monoliths 7
Scaling 9
Load Balancer 10
1 Hands On: First Cloud Server 11
2 Hands On: Raw Server With Scaling 14
Deployment Strategies 19
Networking 21
Hands On: Raw Server With Scaling - Improvement 24
Infrastructure as Code (IaC) 24
Hands On: Really Simple IaC 25
CI / CD Process 27
Hands On - Repo with Code Commit 28
Container with Amazon Elastic Container Registry 33
Hands-On: 1/4 Install Docker 33
Hands-On: 2/4 Configure a simple Website in Docker 34
Hands-On: 3/4 Register Your Container 35
Hands-On: 4/4 Use Docker on another machine 38
Serverless 38
Microservices 39
Definition of Microservices 39
Scalability 40
Hands On: First Microservice 40
Hands On: First Serverless Website 41
Hands On: A pipeline for the website 41
Application Program Interfaces (API) 42
API Calls 43
Hands on: Serverless Infrastructure - Microservices 44
Hands on: Serverless Infrastructure - API Gateway 46
Hands on: Serverless Infrastructure - Error Handling 49
Hands on: Serverless Infrastructure - Webpage 50

Hands on: Serverless Infrastructure - More dynamic 54
Messaging System 59
Notification 60
Hands on: Simple Notification 60
Asynchronous Calls 63
Async calls 64
More Details 65
Queuing 65
On-Prem vs. Cloud 65
Cloud Computing 65
Cloud Computing Compliance Controls Catalogue (C5) 66
ISO 27001 67
Beyond the MVP - production as a challenge 68
Factors of success 68
Ingredients for success 69
Projects 70

Pre Requirements for Workshops

There are some parts where a hand on example could be found. Therefore you'll need:

- AWS Account
- CLI with SCP (on Windows Putty and WinSCP should be installed)
- Draw.IO for drawing infrastructures
- Postman for REST calls

Basics

The following chapter covers some aspects and methods which are seen as essential for successfully developing a product.

Methods

Design Thinking

The main goal of design thinking is to put the user in the focus of all activities. The HPI says: "Design Thinkers step into the end user's shoes"[1]. This the user is not only interviewed, but observed in all her or his behaviors.

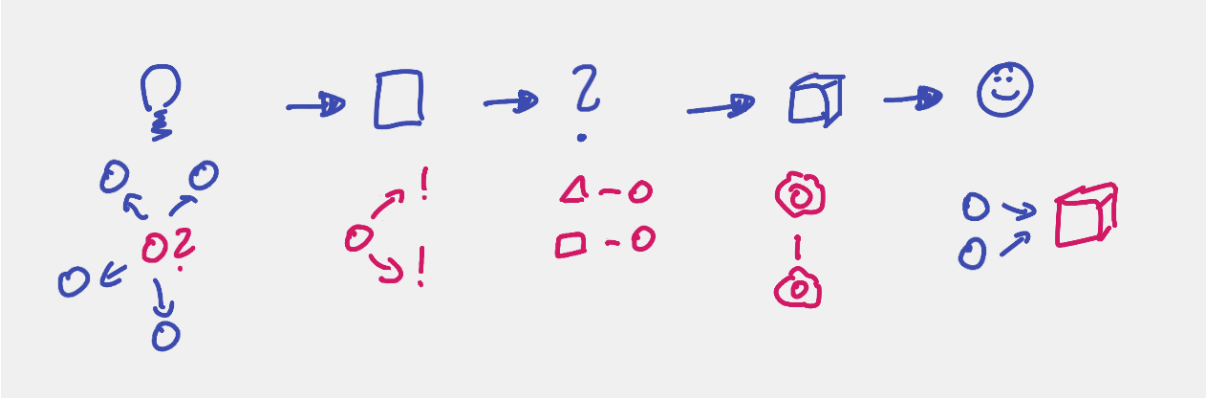
[1] https://hpi-academy.de/en/design-thinking/what-is-design-thinking/

The success factors for design thinking are “the collaborative interaction of multi-disciplinary and decision-capable teams, flexible workspace for collaborative work and a workflow that follows the design thinking process.” [2]

As a software engineer we feel reminded of scrum or similar agile methods. And with a deep look into the design thinking process, we’ll see that we are right. The setting is similar to agile frameworks where the design thinking process starts a little bit before the development cycle.

XD Ideas from Adobe describes the phases in design thinking with a reference to HPI as follows:

- Empathize. Understand the problem of the user for whom you are designing.
- Define. Form a problem statement.
- Ideate. Generate creative solutions to this problem.
- Prototyp. Build a tangible representation of this solution.
- Test. Validate this solution with your target audience.[3]

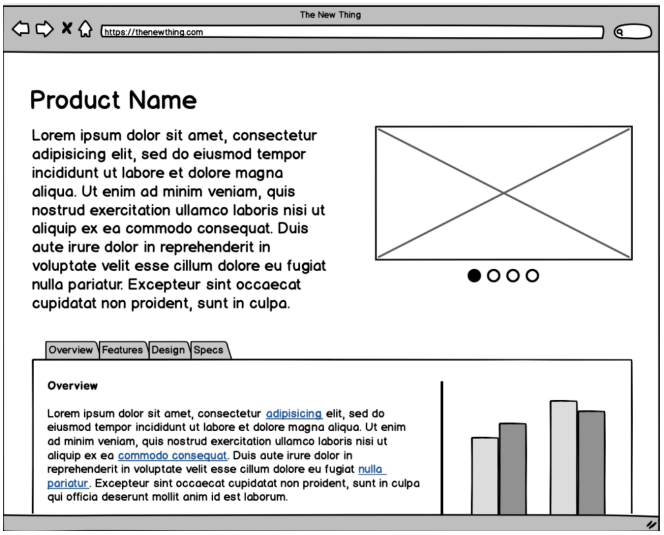


At the end of a design thinking session we have a tested prototype (could be a paper one) which perfectly fits the users problem. This prototype can be swapped in a development cycle where we create a real working product. Actually there could be a loop back to the design thinking process, especially when it comes to the point that not all problems are solved when testing the real product.

Wireframes

“A wireframe is a schematic or blueprint that is useful for helping you, your programmers and designers think and communicate about the structure of the software or website you're building.”[4]

[2] https://hpi-academy.de/en/design-thinking/what-is-design-thinking/
[3] https://xd.adobe.com/ideas/principles/design-systems/design-thinking-process/
[4] https://balsamiq.com/learn/articles/what-are-wireframes/



By using a wireframe you can create an UI prototype and use it for testing purposes.

Software Development Life Cycle

The software development life cycle contains different phases and can be archived by different frameworks like Kanban, Scrum, Waterfall, DevOps.

When you take a closer look at the frameworks you will see that there are huge differences. Common to all is that, in the end, a working product is developed and accessible by the users.

Scrum as agile framework

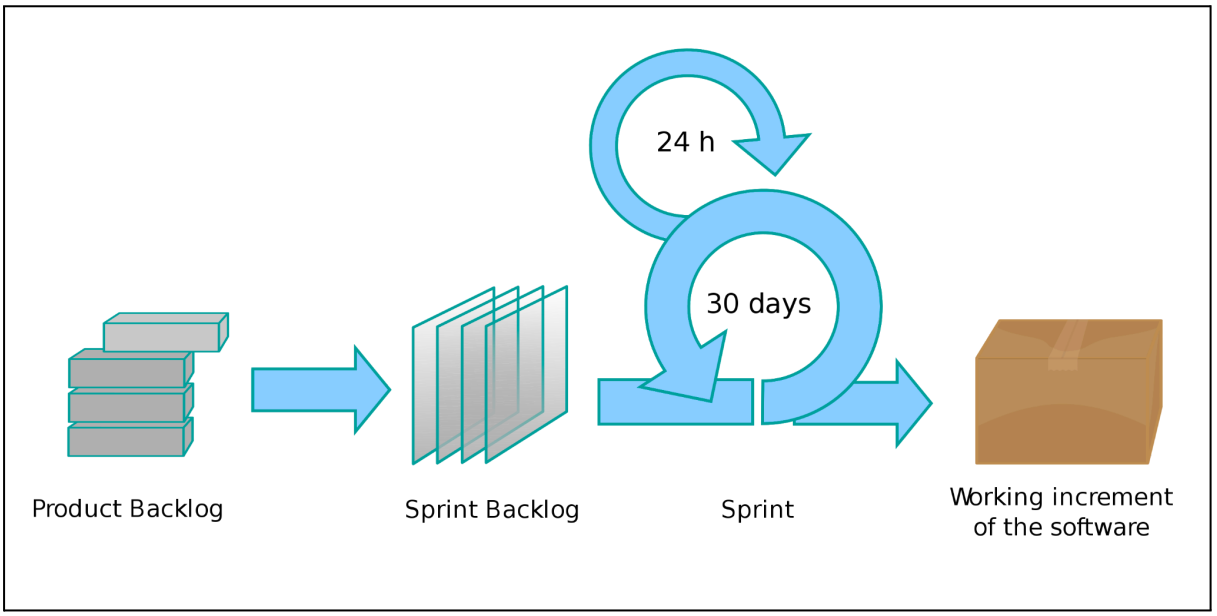
Scrum is a representative of an agile framework. Scrum is defined completely in the Scrum Guide[5] by Ken Schwaber and Jeff Sutherland. Scrum has different artifacts, like a Backlog, a Sprint Backlog etc.. In the backlog there is everything defined which should be part of the product. Parts of the tickets and stories could or should be an output of the design thinking process.

The backlog is defined by the product owner. That's the guy who is responsible for the success of the product. Thus the product owner (should or) could also be a participant of the design thinking process.

Due to the agil character the backlog is never really ready, but the important things should be completely defined. The product owner decides which parts of the backlog should be part of the next sprint, i.e. what should be developed from the developer team in the next sprint.

[5] https://www.scrum.org/ and https://www.scrum.org/resources/scrum-guide

Because of the outstanding role of the project team, the team decides what could be implemented during the next sprint. All the chosen items are going straight into the sprint backlog. At the end of the sprint we have an artefact (piece of software), which makes the product a little bit better.
Of course the whole process is a little bit more complex than described here, but we can see a correlation between scrum and design thinking. At the end of a sprint and to decide what should be developed next or what should be part of the product backlog, a design sprint could be started. Thus we find out which fancy extensions are useful and these insights could be cast into tickets of the backlog.



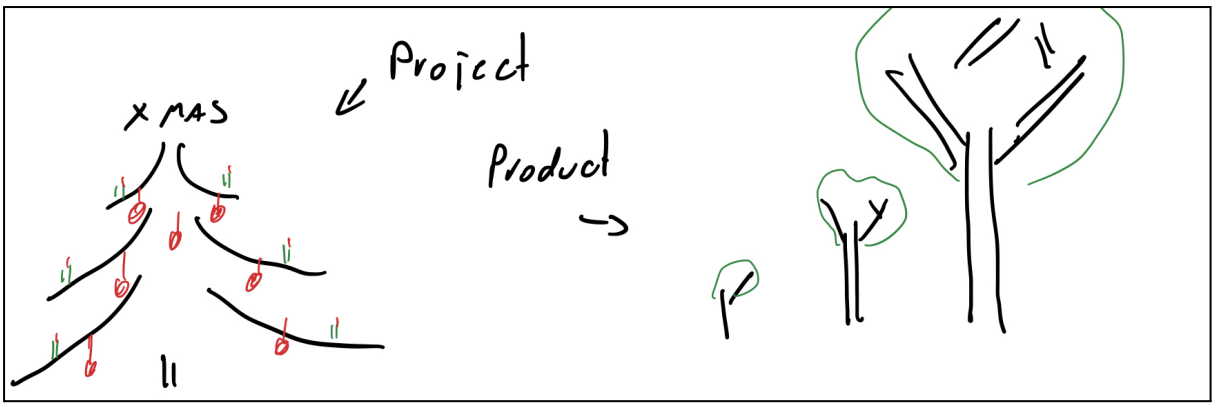
[6] Scrum

Product vs. Project

Even if there is only one agile framework listed, we can see that there is a lot of talk about products but nothing about projects. Why is that?

Let's think about a project as a xmas tree. We have a defined period of time by which the tree needs to be decorated. That can be seen as a product launch, the go live, etc.. Afterwards the xmas tree needs to be shiny for a few days - that's when everybody has an eye on it. That's the time the project is presented to the public or the management.

[6] https://no.wikipedia.org/wiki/Scrum#/media/Fil:Scrum_process.svg



After that period, the project budget runs out, the project manager is leaving the project and everything is handled a different way. Back to the xmas tree, think about Knut - the Danish way to say goodbye to the tree.


[7]

In contrast a product is something like a tree in a forest. You start to select a seed (design thinking) and then you take care of the small plant, like watering, building a fence against animals, etc.. That can be seen as an improvement to the product and so we are in an agile framework like scrum. In this scenario there is no project manager, it's about a guy with a deep passion for what should be developed. That's the product owner. He or she want's to get a big and stable tree out of the seed.

Proof Of Concept (POC), Prototype and Minimum Viable Product (MVP)

Proof Of Concept (POC), Prototype and Minimum Viable Product (MVP) are playing together or better they are going hand in hand. A POC can be created after the ideation phase of a design thinking process. The POC helps to find out if the envisaged technologies really help to fulfill the product goal. For example it can be found out which database technology or cloud strategy should be used. A POC helps to find out if a product can be delivered and if the goals can be fulfilled.

[7] https://www.vivawest.de/blog/detail/deutschland-feiert-knut

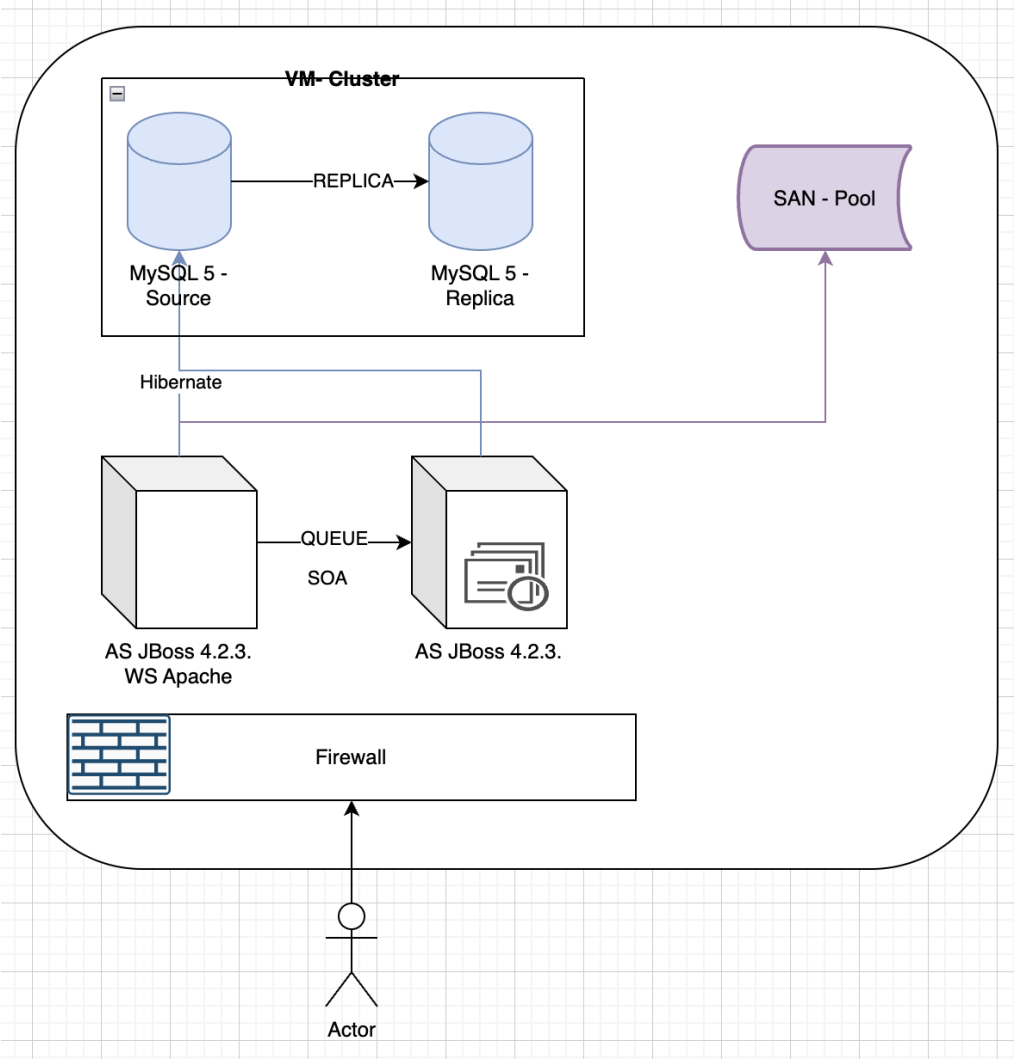
By creating a prototype you go one step further. With a prototype you can validate the product idea and show it to an audience. Even if it’s common sense, that a prototype is something technically, it doesn't have to be. In the context of a design thinking process a prototype can also be a better click dummy, e.g. a webpage with some simple interactions or just a set of wireframes.

The definition of an MVP stays a little bit clearer. A MVP is a working version of a product with some functionalities. The MVP is testable and from the insights of a MVP you can start to create the real product. In practice an MVP is often understood as the first version of the real product and is sometimes also brought to production. But that’s not how it should be.

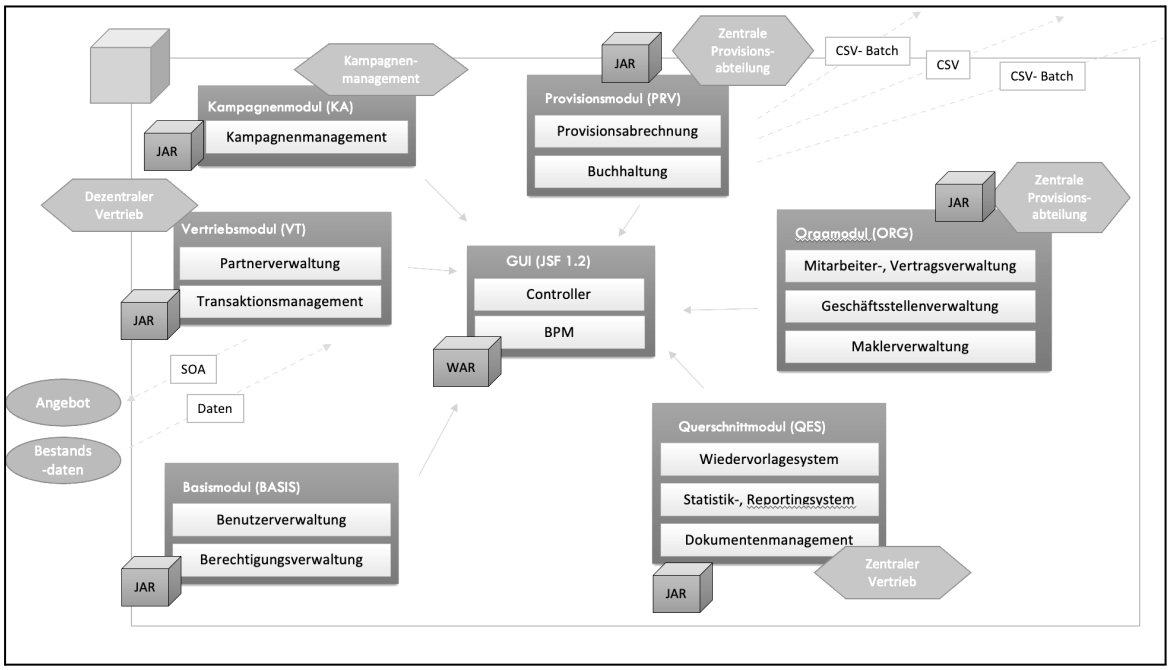
Architecture Concepts

Monoliths

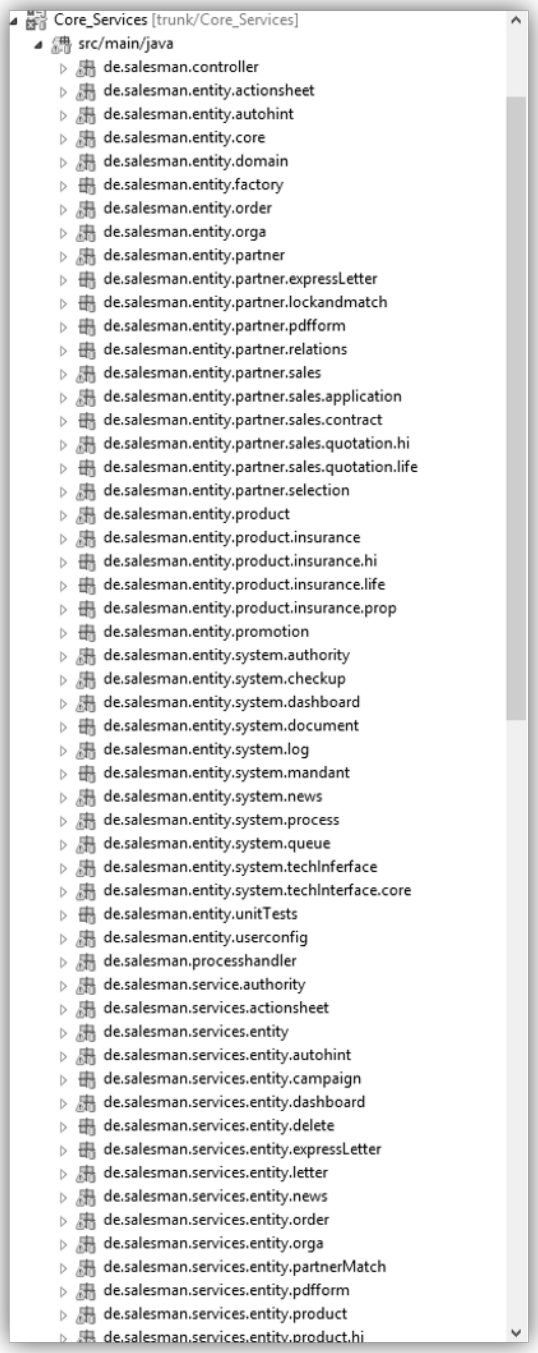
In contrast to a microservice architecture in a monolithic system everything is in one system. Let’s take a look at a (old) java application. It’s a CRM system designed in 2010:



We basically can see two servers (JBoss), a DB cluster and a file system. On the server there are a bunch of JAR files installed.



Inside each jar file there are a bundle of external libraries of which the module depends. The programming itself depends on a high amount of packages and classes. The figure shows an excerpt of only one package.

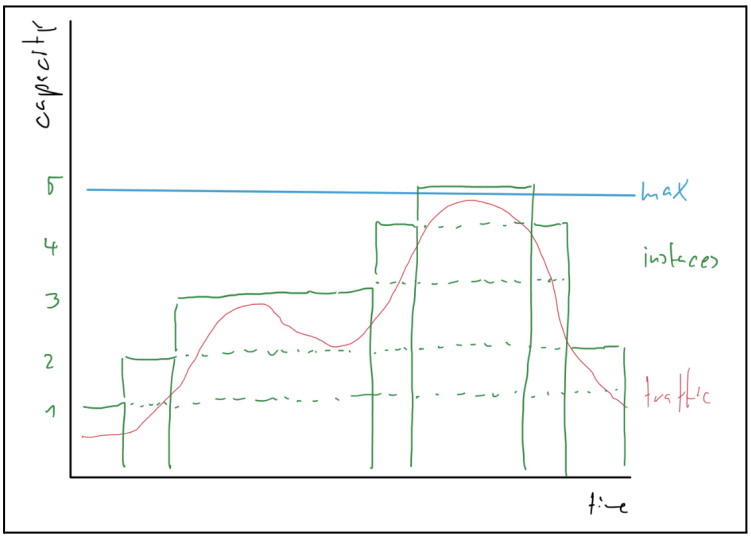


If there is only one thing to change it could lead into a huge hustle. The deployment could fail, because of some unseen changes in the dependencies. Moreover there could be something disrupt due to the changes, which leads to a system failure.

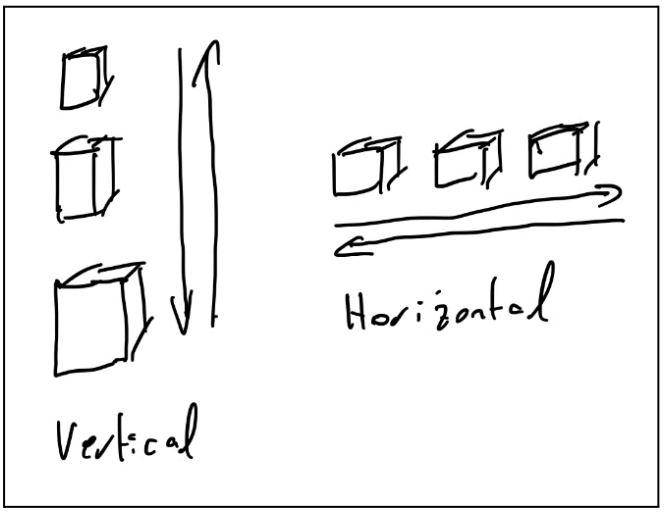
Scaling

What's scaling and why do we need that? Without scaling we need to choose capacities (or instances) which are able to serve the maximum amount (blue in the figure) of traffic (red) possible. That could be during a day at a certain time or even worse xmas or black friday. All our resources need to be of a size which handles the traffic for this peak. That's not cost efficient nor ecologically. So when it comes to scaling, scaling shouldn't be done once a while when the resources have not enough power, it should be done all the time. More (or

bigger instances (capacity) when the traffic increases and less instances (capacity) when the traffic slows down.



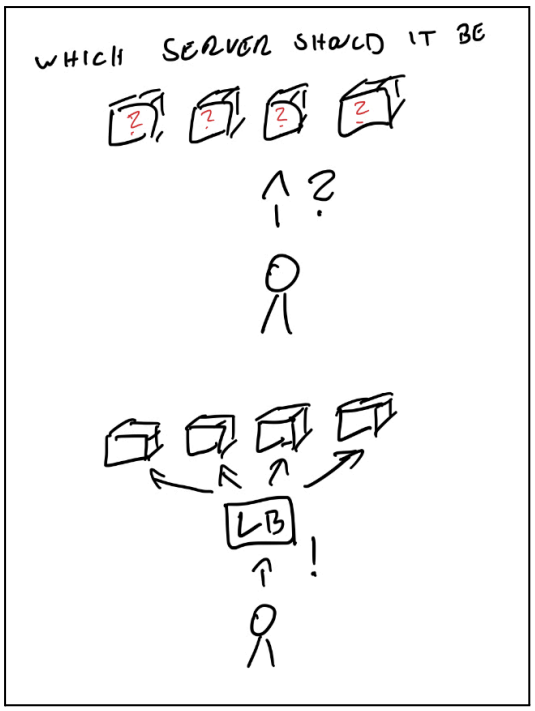
When it comes to scaling you can distinguish between vertical and horizontal scaling. Vertical scaling means that you set up a machine with more power, whereas horizontal means that you use two, three and more machines to scale. You can scale in (less power) and scale out (more power).



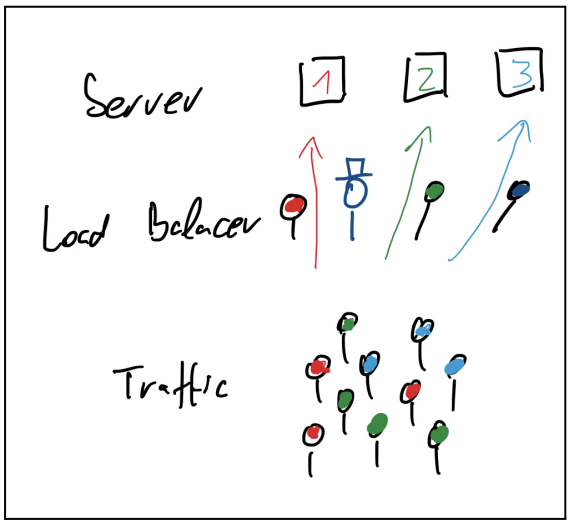
You can scale a monolith or a microservice in the same way. Usually it’s more common to have a dedicated real metal server or a virtual machine to host a monolithic system. Here vertical scaling becomes quite easy. To scale the system horizontally is more sophisticated.

Load Balancer

The second figure shows that you need something in between the servers and the user who tries to access the application. That's usually a load balancer.



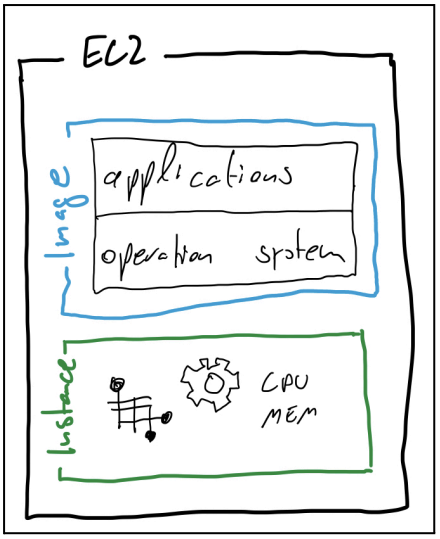
A load balancer works like a guy in the supermarket to guide everyone in the line to the fastest checkout.



To archive this a lot of infrastructure is needed. That’s not only bare metal, it's also know how in server infrastructure. This leads directly into the next chapter.

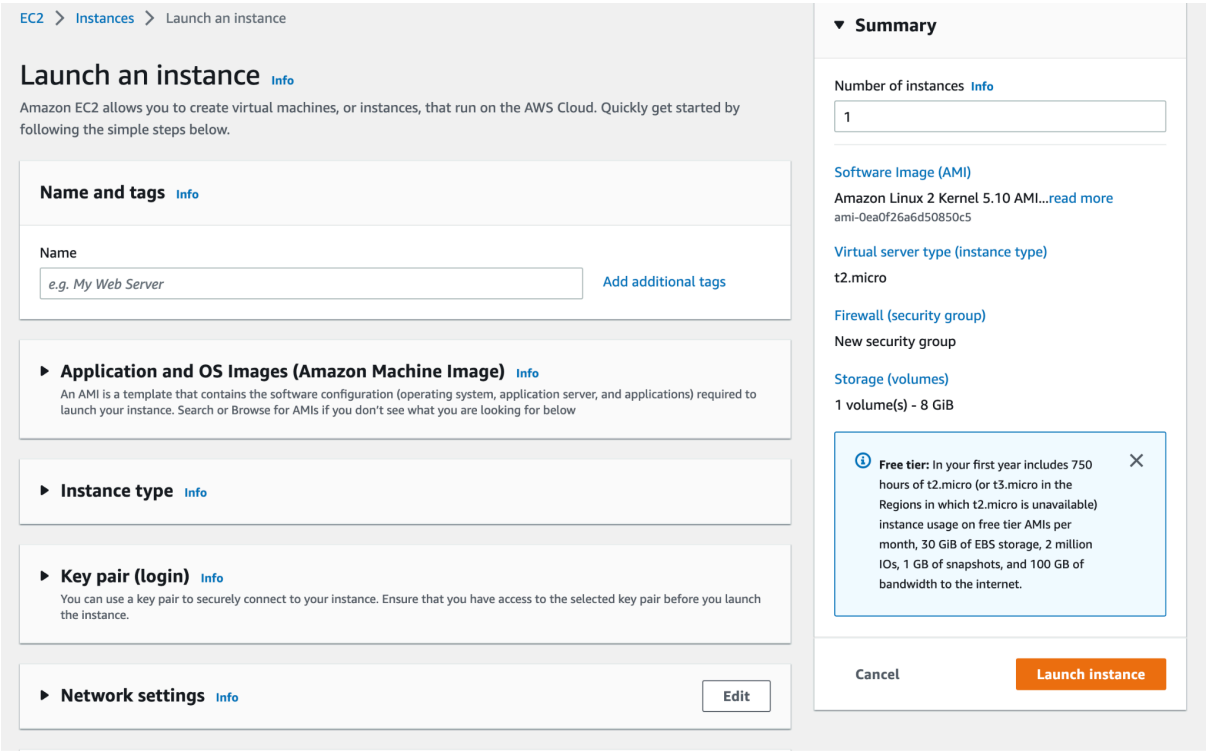
1 Hands On: First Cloud Server

We want to create a server on an EC2 instance. The EC2 instance is a virtual compute server. Such an instance relies on different components. They are:

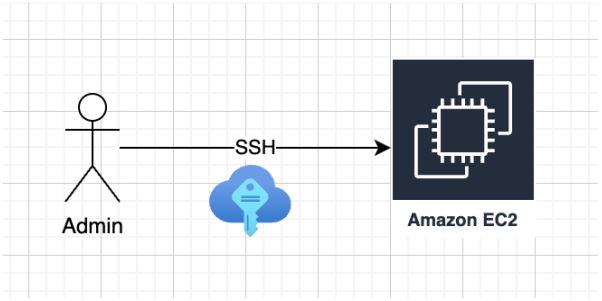


To make it a little bit more exciting we want to run a simple WordPress (application) on the server. To archive this we need to find an image which either contains an already installed WordPress system or a clean image with only an operating system installed. In the last case we need to install a WordPress system afterwards.

First go to the AWS console and open the EC2 services. Here you can launch an instance.

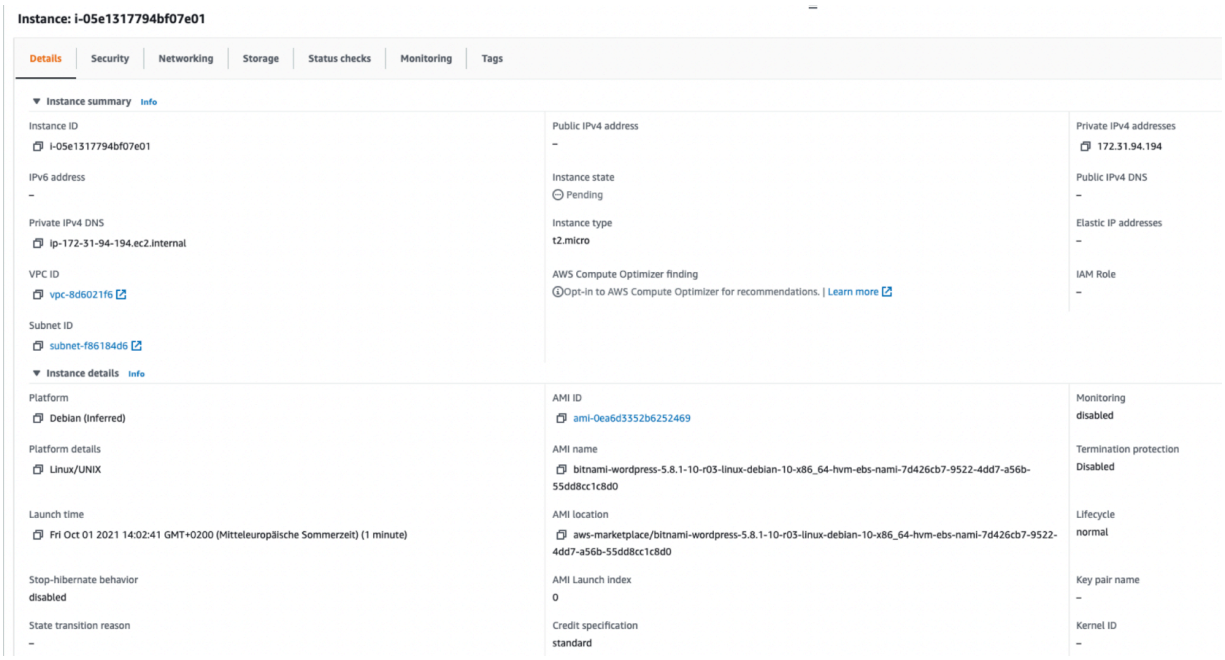


Launch your WP system. Select an image with WordPress (recommendation is: WordPress Certified by Bitnami and Automattic) If you like to access the instance via SCP a SSH key is needed. Thus you should create a key pair. For this tutorial it's not needed.



Additionally you need to choose an instance type (use t2.micro). The instance type defines the power of your instance, i.e. RAM, CPU etc.. ⇒ Launch the instance.

After launching the instance you can find some information about the instance here:

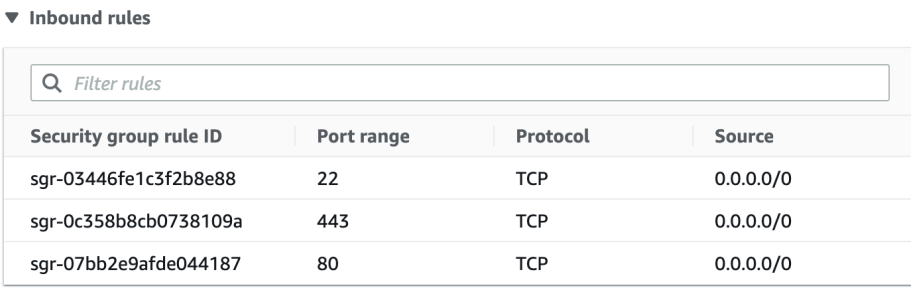


The important information are:

- The public IP address.
- The VPC
- The Security Group (tab Security)

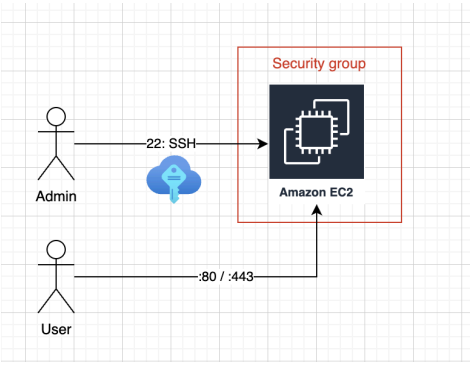
⇒ Add this IP address to your browser and see if it's working.

If you can't visit your WordPress "Hello World", it's probably up to the security group. Have a look at the following figure.



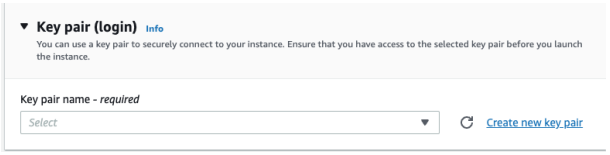
You can see that port 22, 443 and 80 is open to the public (0.0.0.0/0). In your case that's probably not the case. So you should add the port 80 for html to the list. If it's working, try it the other way around and remove the port range.

A more correct infrastructure would now look as follow:



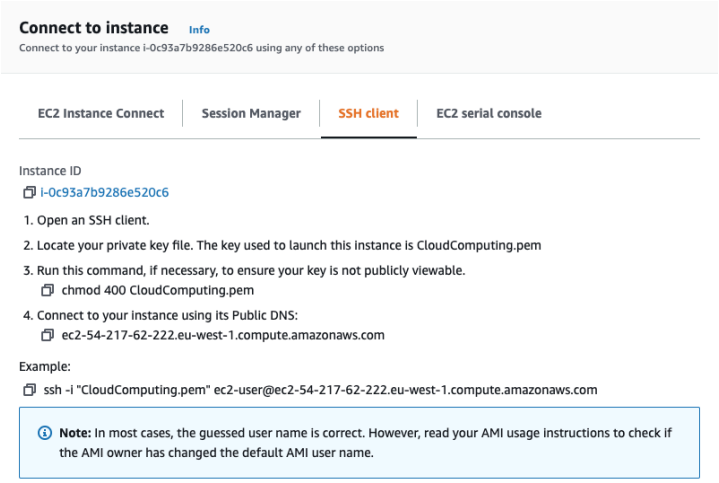
2 Hands On: Raw Server With Scaling

⇒ **Launch** a raw instance with only an operation system. Remember to create a key pair, because you need to access that instance later on. If you're using PuTTY[8] to access your instance use the ppk file format. For Mac and Linux clients, use the pem format.

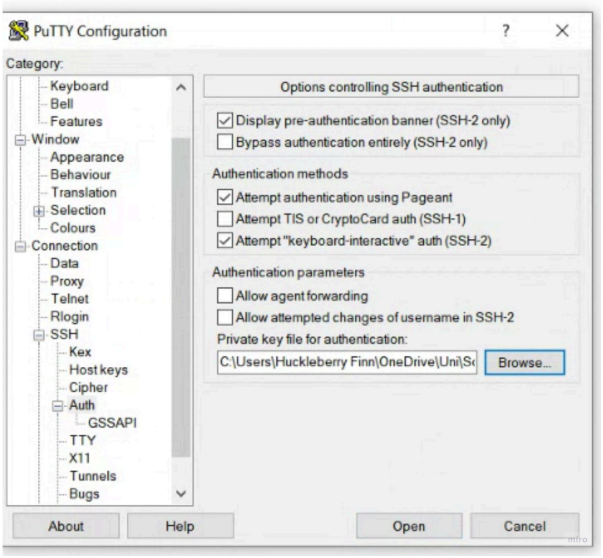


⇒ **Connect** to your Instance. You find the necessary information under actions → connect. Follow the instructions and be aware of step 3! That is important.

[8] https://www.putty.org/

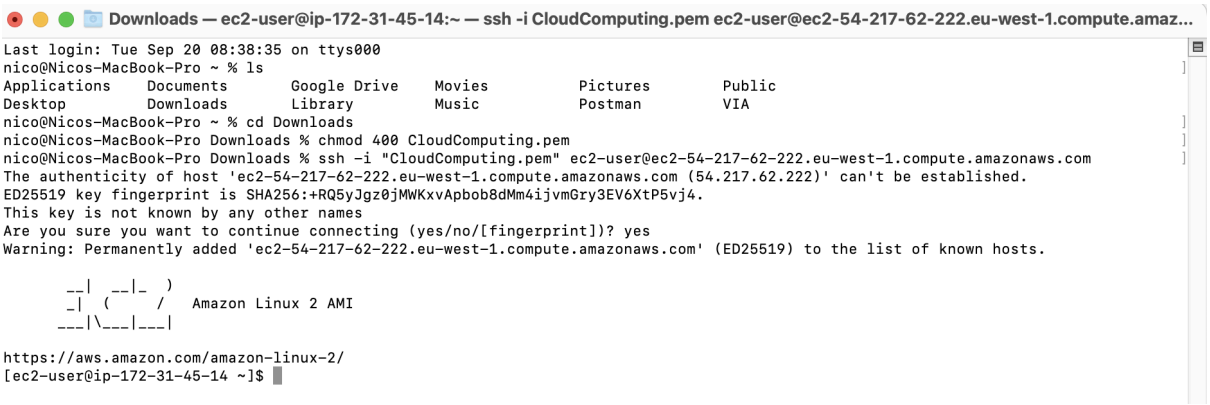


If you are using PuTTY to connect the following settings could help:



Use as user-name: ec2-user (in PuTTY).

After you’re connected, it should look like this:

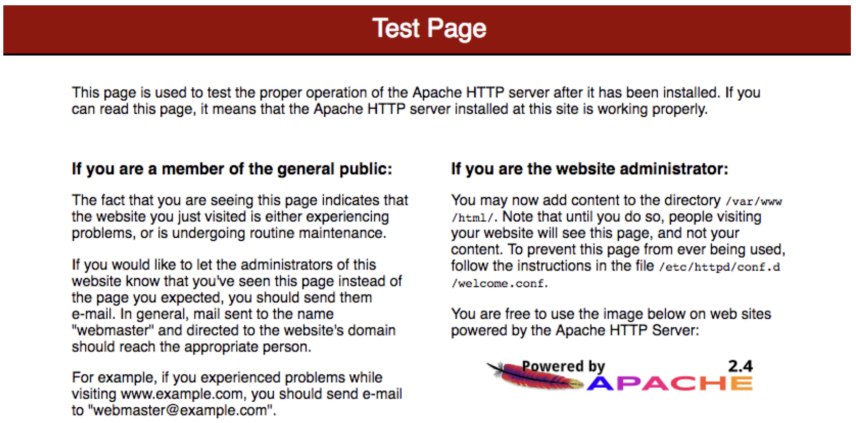


⇒ **Install the Web Server.** Until now there is no Web Server as application installed on your instance. To do that, follow that tutorial here: [Tutorial: Installieren eines LAMP-Webservers auf Amazon Linux 2] (installation of maria-db isn’t needed).

In the tutorial you connect to your instance and basically typing this commands:

```
sudo yum update -y
sudo yum install -y httpd
sudo systemctl start httpd
sudo systemctl enable httpd
sudo systemctl is-enabled httpd
```

⇒ **See the results.** After that a Web Server is installed. When you try to visit the website by using the public IP of the server, you'll find out that you need to add a rule for port 80 to the security group. Afterwards you should see that screen:



!!! PORT 80 HTTP !!!

⇒ **You can stop here (step 1.8) with the tutorial (not the hands on).** Optionally you can try to install a WordPress system by yourself. To do so, follow the instructions here: [Tutorial: Hosten eines WordPress-Blogs auf Amazon Linux 2]

⇒ **Add some stuff to the server.** We want to create a simple index.html file. Therefore connect to the server again and navigate to this path and create an index.html file:

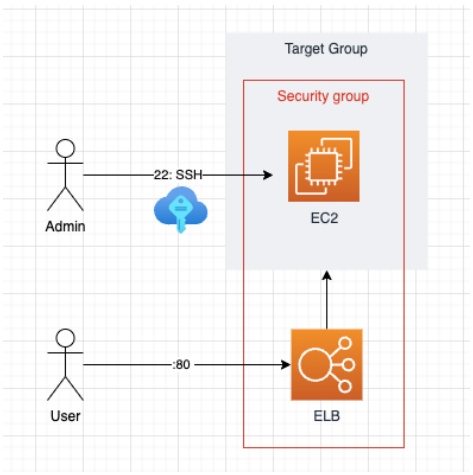
```
cd /var/www/html
sudo vi index.html
```

Now press <<i>> and insert the follow text:

```
<html>
<body>
HALLO
</body>
</html>
```

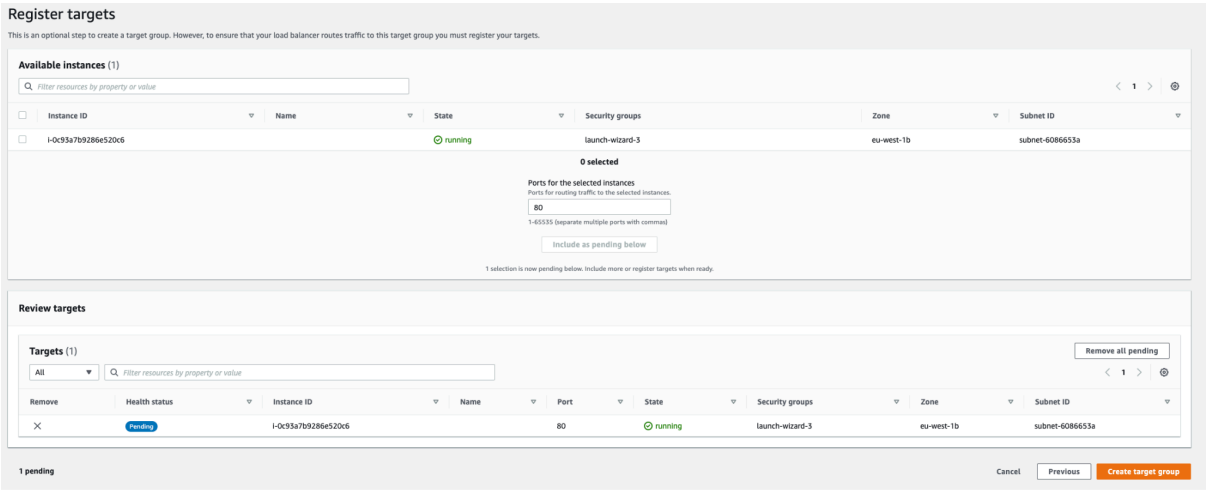
Press <<ESC>> to exit the insert mode and type <<:wq>> <<ENTER>> for write and quit. When you now open the link again, you should see your html file instead of the default apache page.

⇒ **Implement a Load Balancer** to balance the load. In AWS it's called ELB (Application Load Balancer). The result should look like this:



As you can see in the figure, you need a target group. In the target group all instances are registered which should be connected to the load balancer. That's because the load balancer is only responsible for a defined set of instances. You don't want all traffic balanced with one load balancer. Think about an airport where you have the luggage drop off and the bulky luggage. If there would be only one guy (load balancer) he or she would send some customers to the bulky luggage and some to the normal one without thinking. Thus the guy with the surfboard will be in front of the normal drop off whereas the one luggage 23kg guy will be at the desk of the bulky luggage. Both desks are therefore not on one target group and should be addressed by different load balancers.

⇒ **Create a target group**. On your way to create the load balancer you'll be asked for a target group. You can do so in a different tab. Don't forget to register your instance as a target in your new group.



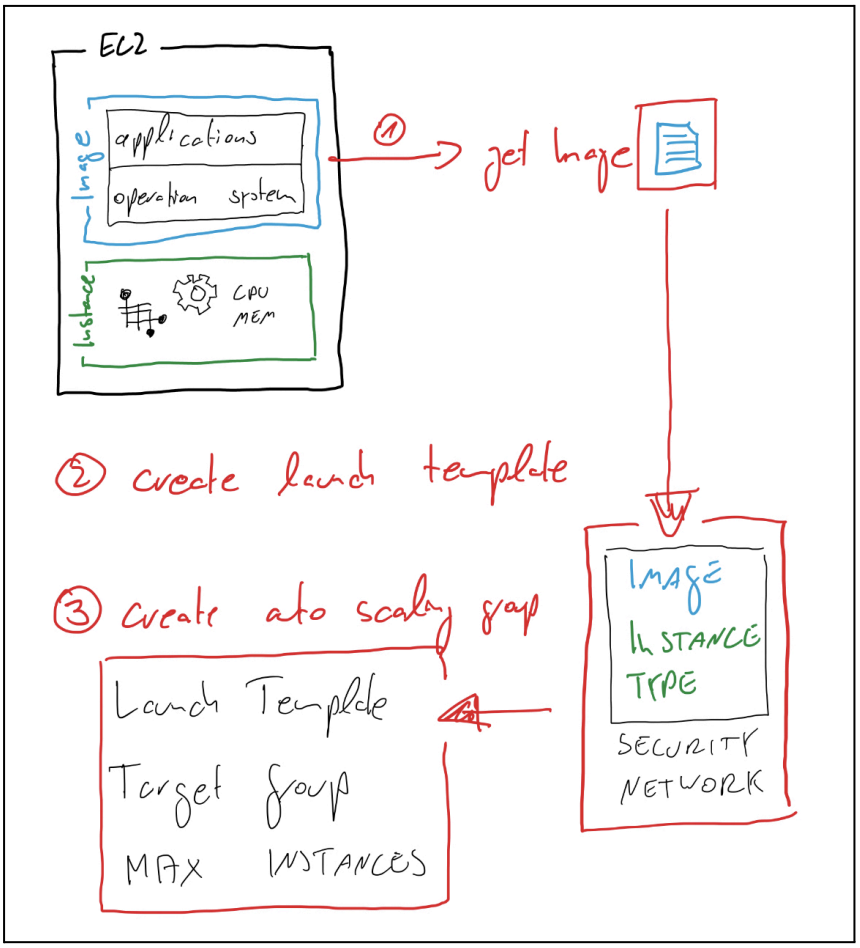
After the Load Balancer was provisioned, you can use the DNS and call it directly.

You should see the same output you saw, when you called the EC2 directly.

⇒ **Draw a better infrastructure.** When you have a look at the infrastructure you find, that's not correct. Access to the instance itself is still possible via port 80. Thus you should restrict it. The user should only be able to access the instance via the ELB. What can you do?[9]

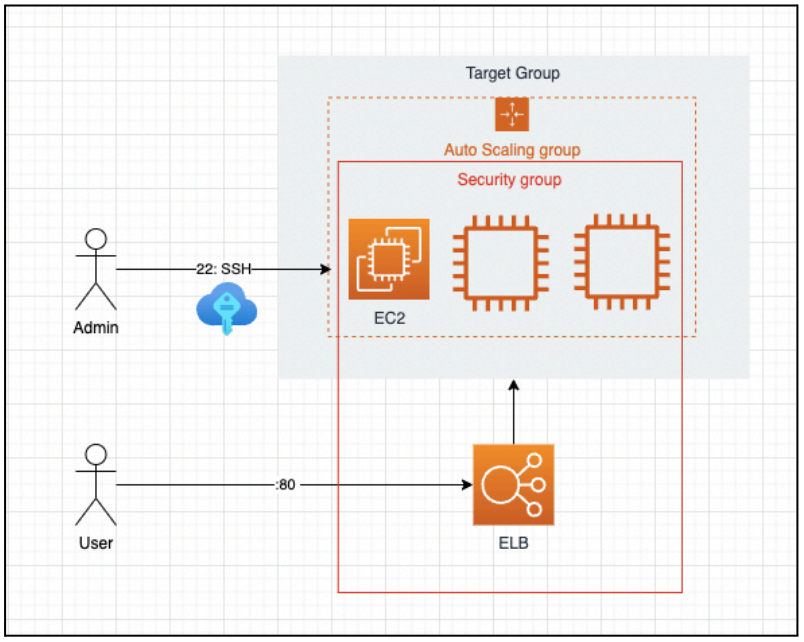
⇒ **Implement autoscaling.** Now that we have a load balancer we can go for a horizontal scaling. That means we can set up more instances. Each of these instances must be in the same target group.

Here are the things we need to do.



[9] Hint: Think about the security groups.

(1) First we need to create an image. This is done by actions → image → create image. This image is the base for our scaling.
(2) Next we need to create a launch template. This is THE configuration for our autoscaling. In the launch template you define which image should be used, which security group should be setted and which instance type should be selected. That are more or less exactly the same information you need to insert when launching an instance on your own (like in the first hands on example).
(3) Now we need to create the auto scaling group. Here we define which template should be used (the one we just created) and to which target group the new instances should be added. Furthermore we define how many instances should be created or the rules, when a new instance should be created (scaling). As a target group we use the target group of our load balancer. So we archive this:



Deployment Strategies

Beside the scaling we have seen one deployment strategy in the example above: Prebaking. That means we have a ready to use Image with all necessary apps and updated (prebaked) and can use this for an deployment.

Alternatively we could automate the process of installation after choosing a raw image. Hereby we put everything into a script and after provisioning the instance (with the raw image) our script will be performed. That's called bootstrapping. The disadvantage is that this takes much more time. On the other hand, we have an updated operating system. Side effect is that this could be unstable.

See the following figure for a rough overview.

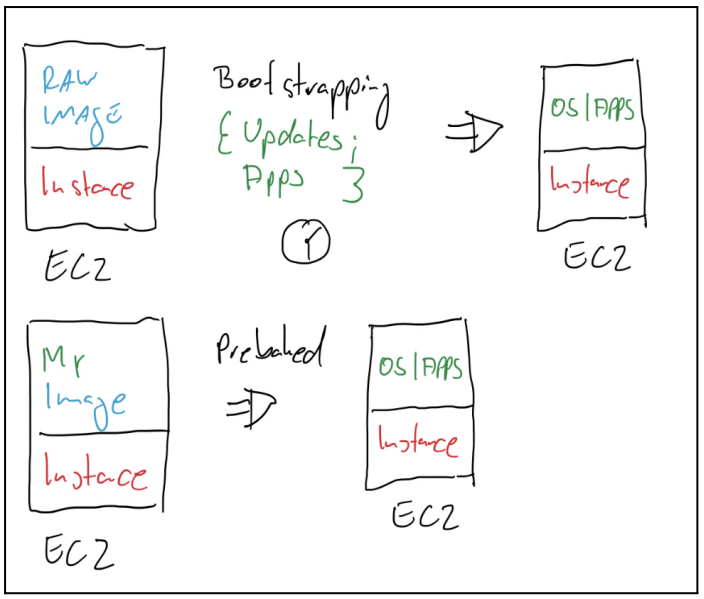


Fig.: Bootstrapping / Prebaking

Our scenario could also be set up as a so-called blue/green deployment. Here we are setting up a new environment (maybe based on our current image) - green. Here we are starting to install and deploy everything needed for our update. After testing the new environment we simply switch from the running environment - blue - to the new one - green. This could be done with a load balancer or launch configuration. An easy way to go is to add a new image (our green environment) to the launch configuration. By terminating the old running instances (blue), new instances (green) will be launched. Thus the switch is performed more or less automatically.

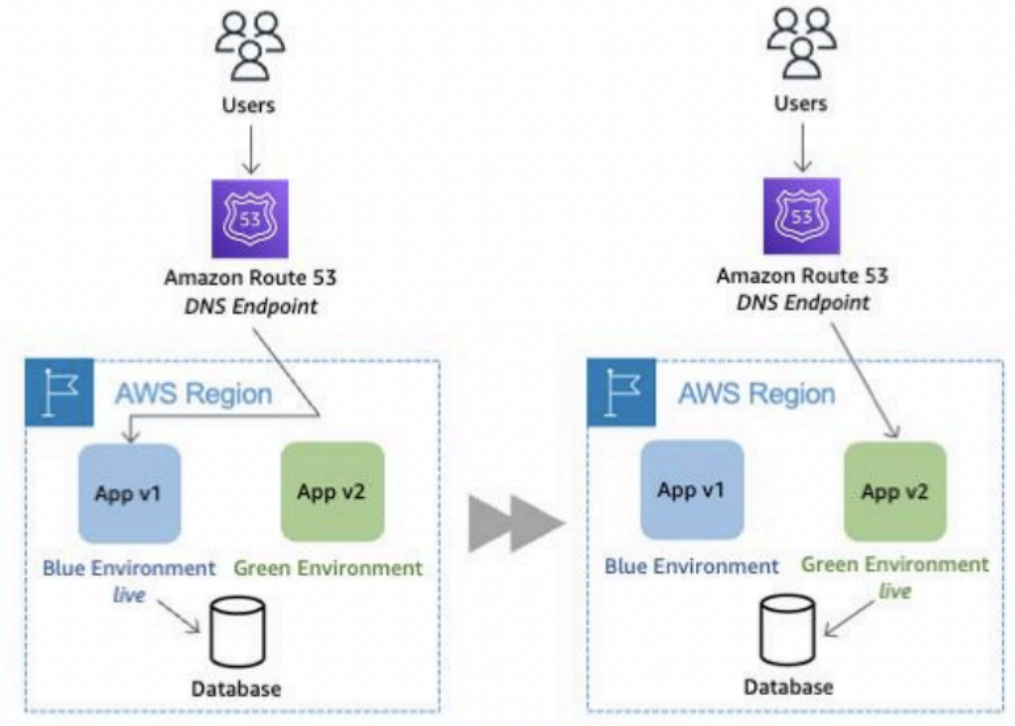
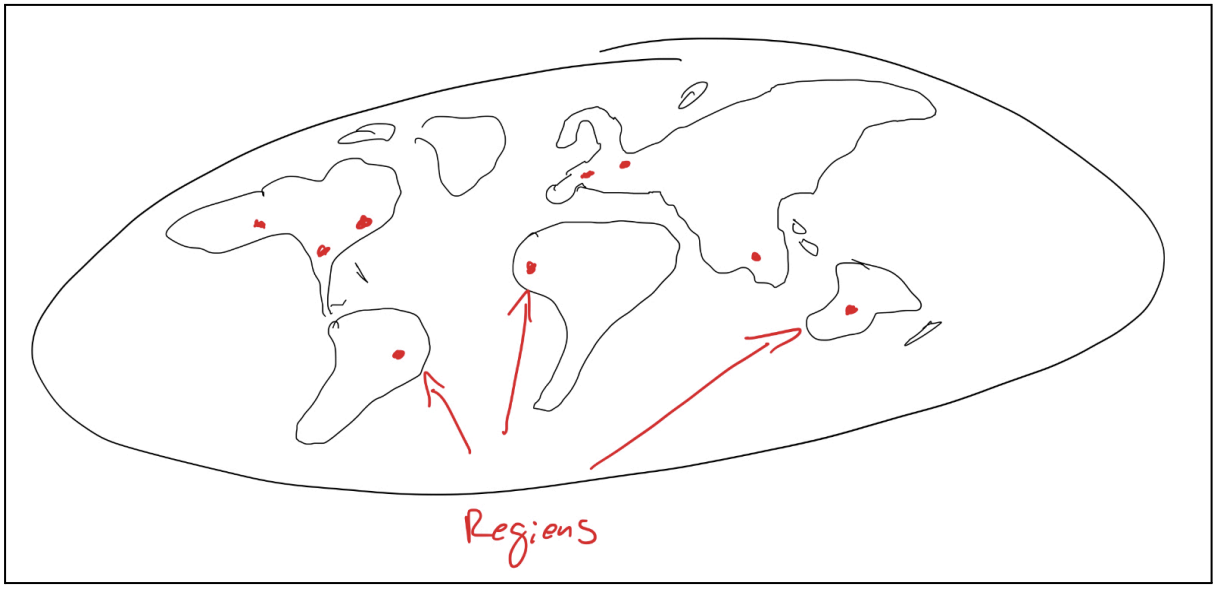


Fig: blue/green deployment [10]

Networking

A region is the geographic location of a cloud data center (see the following figure). Different regions offer different service qualities in terms of latency, solutions portfolios, and costs.[11]



In AWS you can choose and switch the regions on the right hand side in the menu.

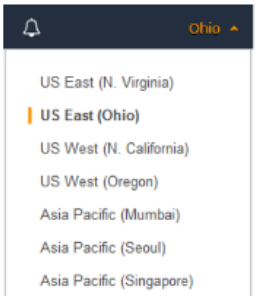


Fig.: Menu for choosing the region

Every region has its own code. This code could be found in the links and in your programs when implementing something.

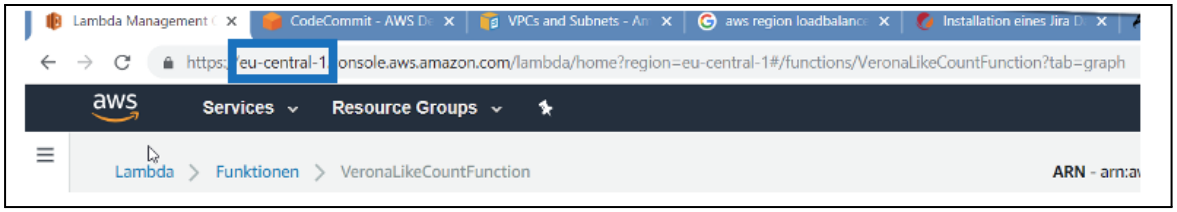


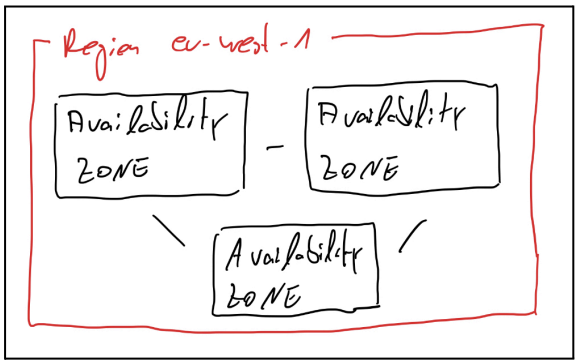
Fig.: Region in the links

[10] https://d1.awsstatic.com/whitepapers/AWS_Blue_Green_Deployments.pdf
[11] https://www.bmc.com/blogs/cloud-availability-regions-zones/

Code	Name
us-east-1	USA Ost (Nord-Virginia)
us-east-2	USA Ost (Ohio)
us-west-1	USA West (Nordkalifornien)
us-west-2	USA West (Oregon)
ca-central-1	Kanada (Zentral)
eu-central-1	EU (Frankfurt)
eu-west-1	EU (Irland)
eu-west-2	EU (London)

Fig.: Region codes in AWS

In a region there are availability zones. These zones refer to isolated data centres within a single region. Each availability zone includes multiple data centres. So when choosing a region for your EC2 instance you also need to choose the availability zone.



By using more availability zones you can reach a certain state of reliability. To achieve this you can use for instance a load balancer in combination with auto scaling and a target group in different availability zones. If you define as minimum 1 instance in the autoscaling and one data centre (availability zone) went down, your instance is launched in another data centre. And due to the load balancer, the load is balanced to the new EC2 instance in the new region.

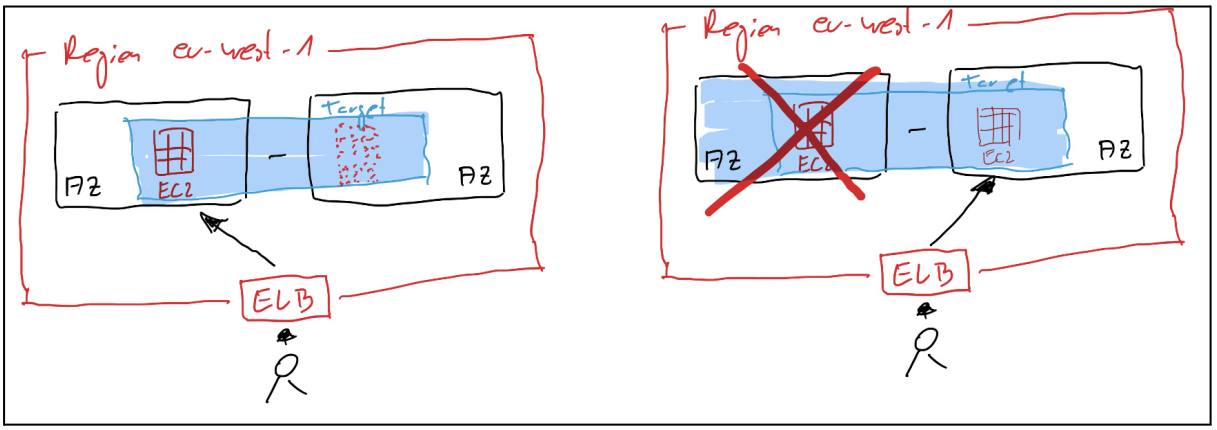
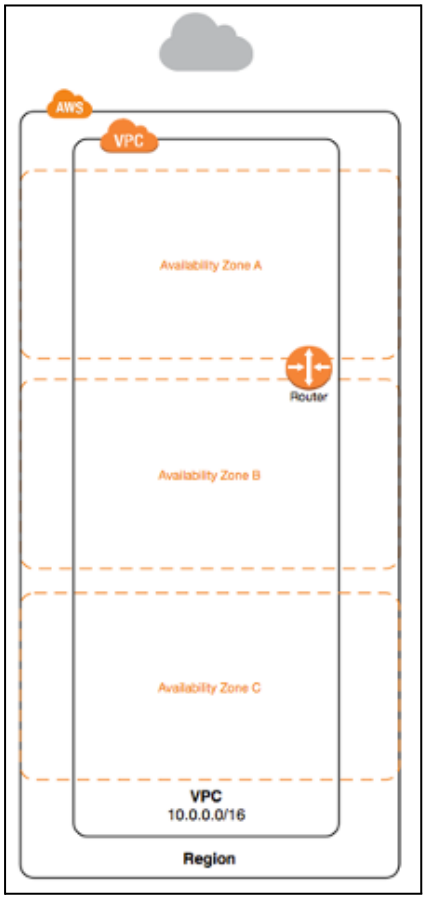


Fig.: Target group in different availability zones

To gain IP addresses so that your instance could be called internally (privat) or externally (public) you need a virtual private cloud network (VPC). The vpc is dedicated to a region in all availability zones. The VPC holds a set of possible IP addresses for your services.



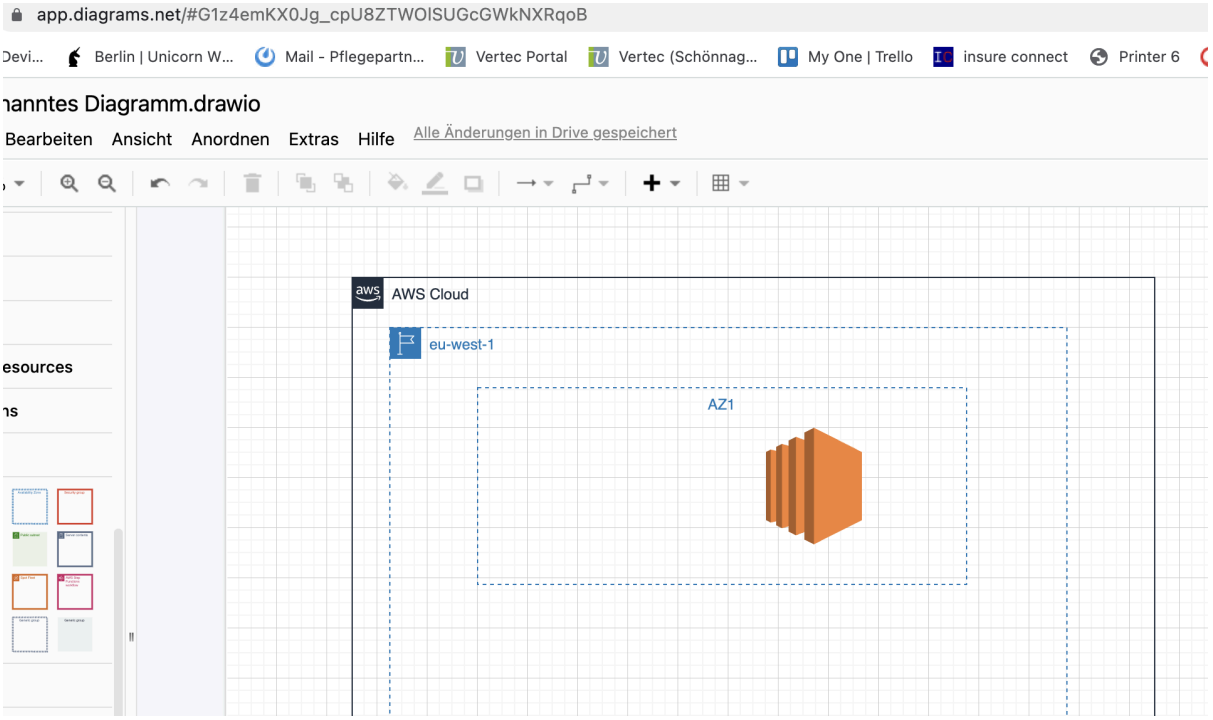
Additionally the VPC holds a routing table like this:

Destination	Target
0.0.0.0/0	igw-8e097ee9
172.31.0.0/16	local

Here we can see that all internet traffic (0.0.0.0/0) is routed over an internet gateway, which is managing the internet traffic.

But how do an instance know in which availability zone it should be deployed? Here it comes to subnets. The subnet belongs exactly to one availability zone. The subnet defines a subset of IP addresses which could be delegated to services within the subnet. Furthermore a subnet could be public or private.

===> FIGURE WITH SG; TG, ETC ….



Hands On: Raw Server With Scaling - Improvement

⇒ **Improve the infrastructure.** Now that you know everything about VPC, Subnet, Availability Zones, Regions and Security Groups, please draw your current infrastructure. Check your infrastructure afterwards and improve it to make it more secure.

Infrastructure as Code (IaC)

“Infrastructure as Code (IaC) is the managing and provisioning of infrastructure through code instead of through manual processes.”[12] By that way you can describe and program what you intend to do. So everything is reusable, i.g. You can clone your infrastructure.

[12] https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac

Imagine you want to try out a new feature or you want to launch an integration environment. In the old world you had to find out all configurations of the server, all settings of the infrastructure by yourself. In the worst case scenario, there would have been an admin in your company who did some really special stuff you don't understand. Maybe you're missing something, like the database server on a different instance.

When everything is coded instead of manually clicked, the infrastructure is described like 100%. It's a perfect documentation and a perfect base for setting up the same infrastructure again.

By using a repository like GIT you can even check who did which changes over time. Imagine there is a big issue one day. An admin takes over and does some fancy stuff. Everything is working fine the next day. But then, at the end of the month there is one service which only runs once a month and this service is not working (like the monthly bills). The administrator is on vacation and nobody could remember what happened between this and the last month.

When everything that was done would have been coded and accessible via a repository we could easily find out. Maybe we could just roll back the infrastructure and try another fix.

To avoid things like that from the first step, we could work with approvals. So you can't create a new instance until somebody approves the pull request.

The description could be done in different ways. In AWS you could use cloud formation or you use an agnostic tool like terraform.

Hands On: Really Simple IaC

Assume we want to launch an instance based on our image. We just need a yaml file like this:

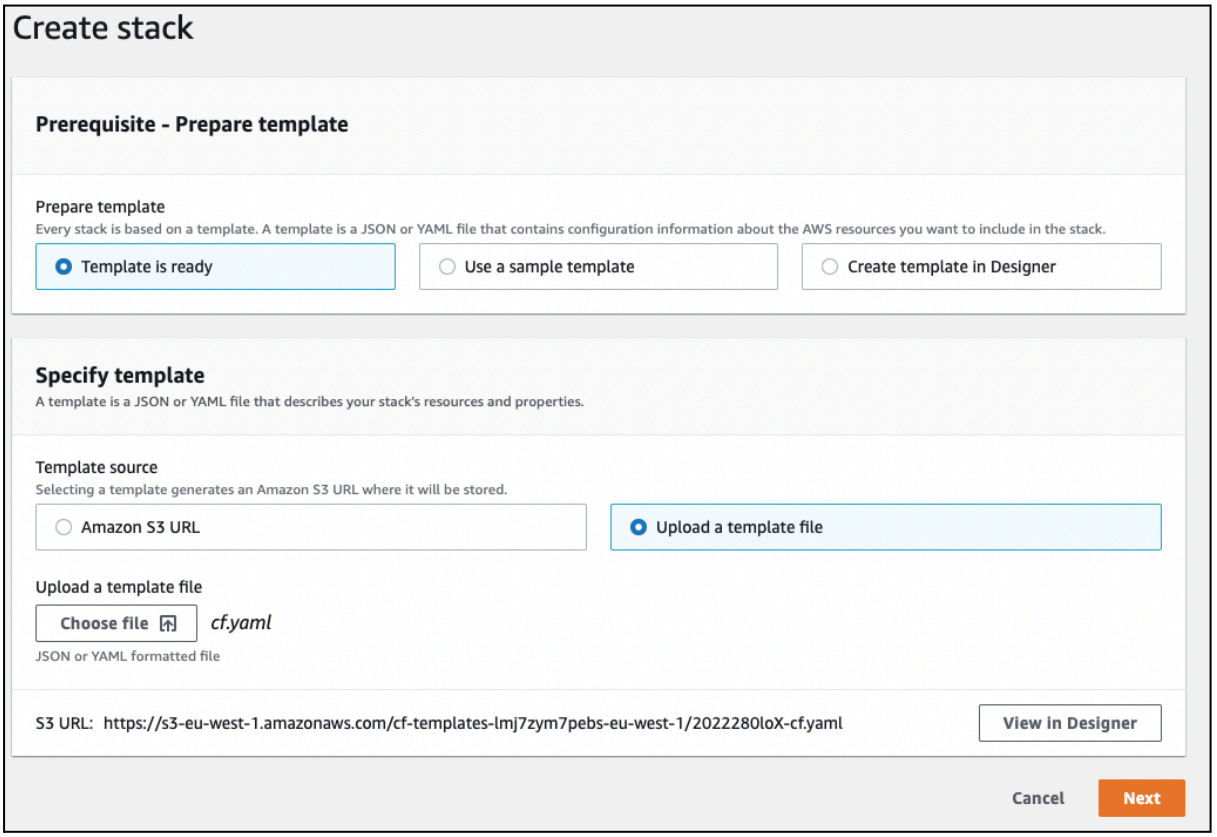
```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
(1)HTWInstance:
(2)Type: "AWS::EC2::Instance"
(2)Properties:
(3)ImageId: "ami-06106653b82b7fae1"
(3)InstanceType: "t2.micro"
```

Fig.: YAML IaC

This file needs to be uploaded and executed. Therefore we are using the service cloud formation.

We can also use a prompt like this:
use the folder /Users/nico/Documents/Uni/VSCode/htwlesson4 and create a iac script for aws cloud formation in yaml. a ec2 instance with following attributes shoould be lauched: ami ami-03026aa8227f2f9c3 security group: sg-0be14659a6379fe7a type: t2.micro region: eu-west-1

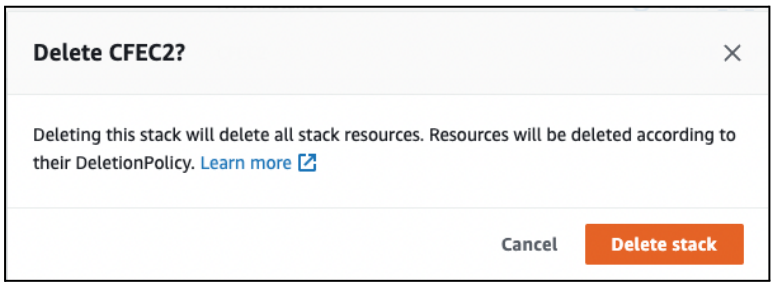
⇒ **Create a yaml IaC file** for your instance and execute it by using cloud formation.



After executing the script you'll see to things. First you find in cloud formation, that the stack is executed:

When open the EC2 console you'll find out that a new instance is created.

⇒ **Delete the stack**. Imagine that this is not only one EC2 instance, but a whole bunch of services, like the load balancer, autoscaling etc.. Now, that you are just using this for testing reasons you want to destroy the infrastructure. In an old world you needed to find out which ressources you had provisioned. In the IaC world you just need to delete the stack.



On the EC2 console you can see that the instance will be terminated.

CI / CD Process

Now that we have our infrastructure in a code - which's great, we are not really done. How to work collaboratively together, how can we bring our code to production, what's with versioning? The answer could be CI/CD. CI/CD is a method to frequently (continuously) deliver artifacts to customers. Concrete CI means continuous integration, which means built, tested, and merge changes to a shared repository. Continuous delivery describes the upload to a repository (like GitHub), where the artifact can be deployed to a live production environment by the operations team. Continuous deployment can refer to automatically releasing artifacts from the repository to production.[13]

[13] https://www.redhat.com/en/topics/devops/what-is-ci-cd

14

When we think about our infrastructure we can improve our process of creating an instance. Or better: we start to create a process.

Hands On - Repo with GIT

First thing to do is to create a repository. Therefore we are using GitHub.

⇒ Use GitHub

- Create an account (if you don't have one)
- Create a new repository in GIT
- Use Codespace for easy editing
- Add a new file: cf.yaml
- Commit and Update the changes

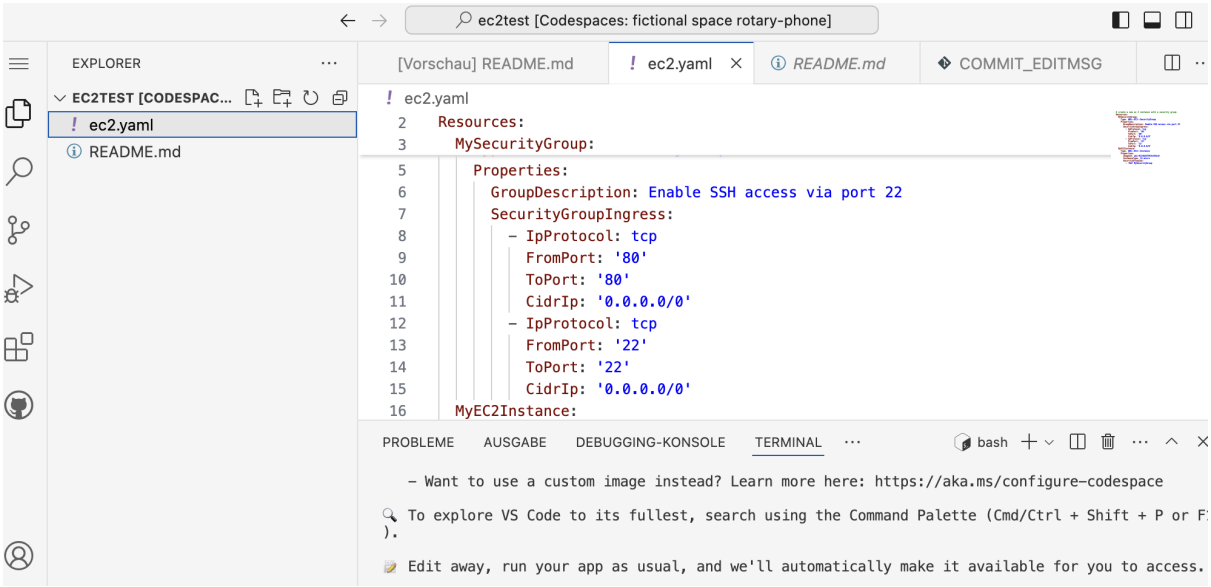
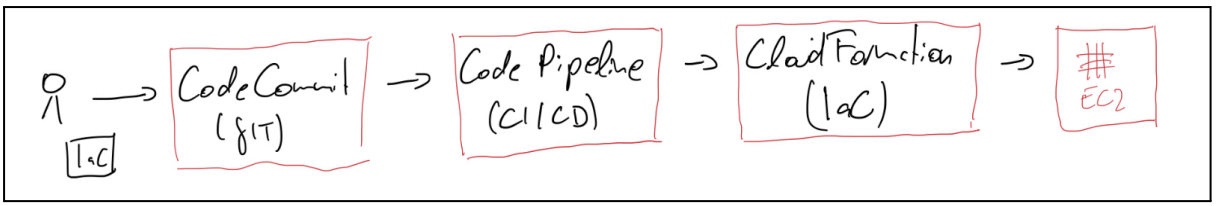


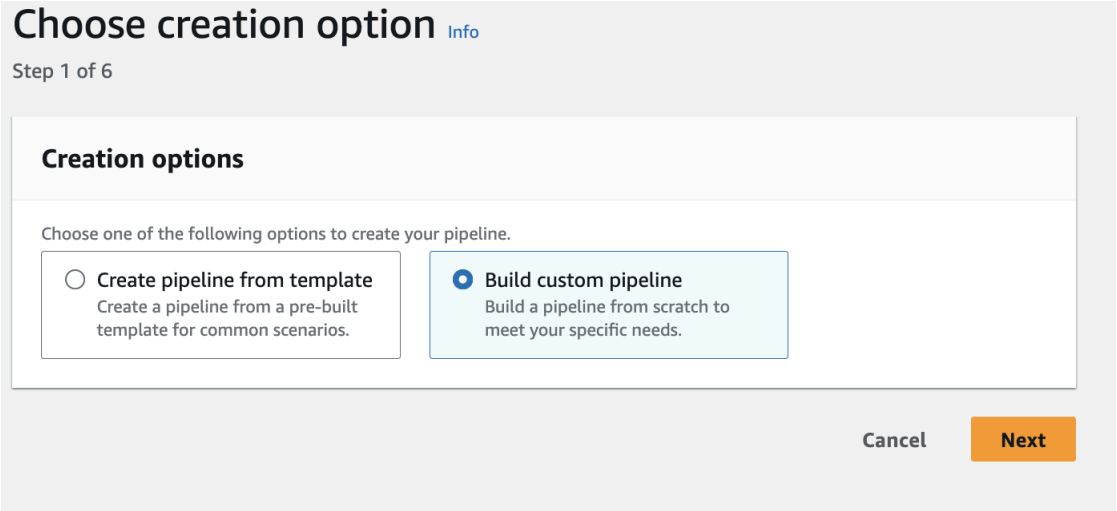
Fig.: repository with our file

When you have a look at the next figure you learn that there are still some steps left until we can create an EC2 instance via a pipeline out of our IaC file.

[14] https://www.redhat.com/en/topics/devops/what-is-ci-cd

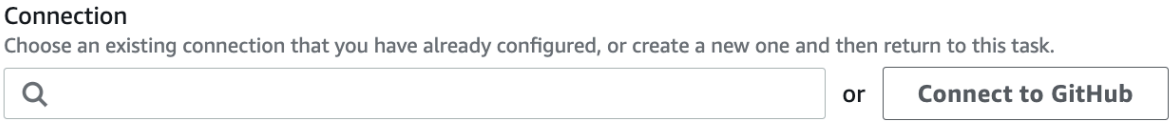


⇒ Create the pipeline. Therefore we go to the service CodePipeline and start to create it. Go with: Build Custom Pipeline.

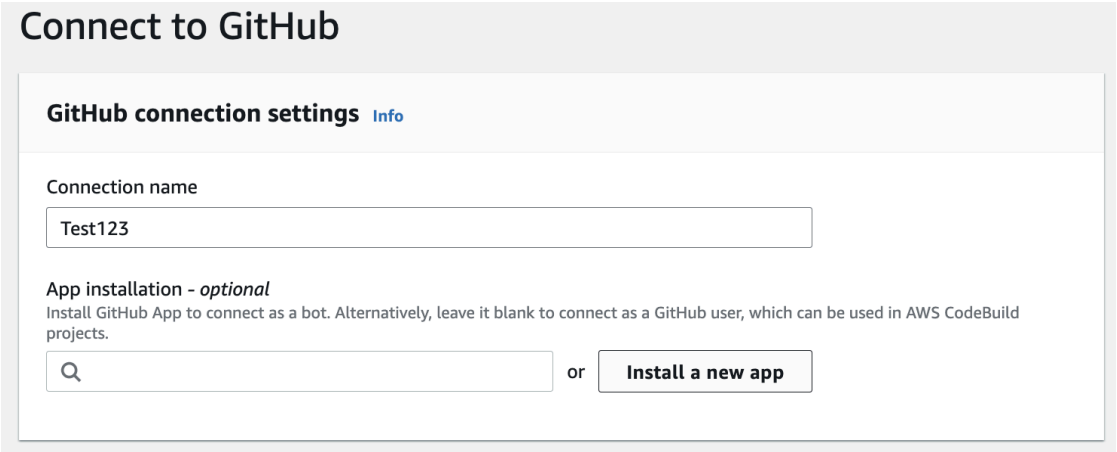


Afterwards select “create a new service role” and type in a name. The service role allows CodePipeline to run the pipeline.

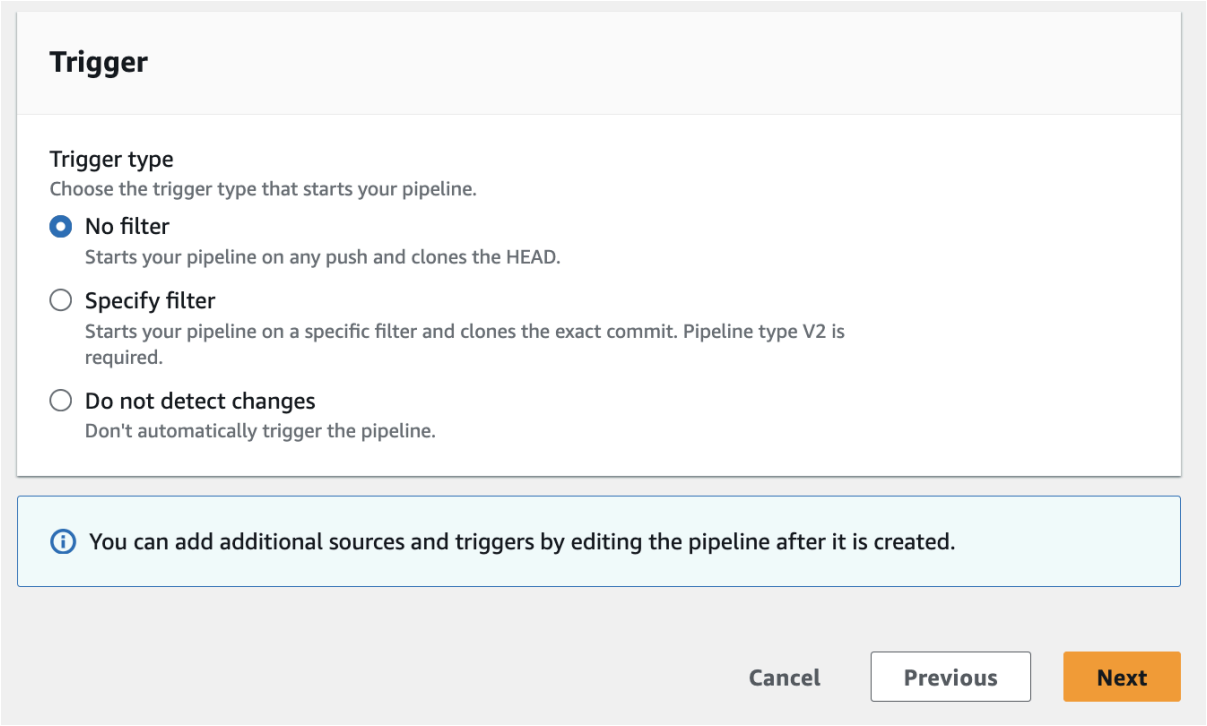
Use GitHub (Version 2) as a source. You need to create a new connector.



Click on “Connect to GitHub” and then create the connector.



Choose your repository and in trigger, select: **No Filter**.



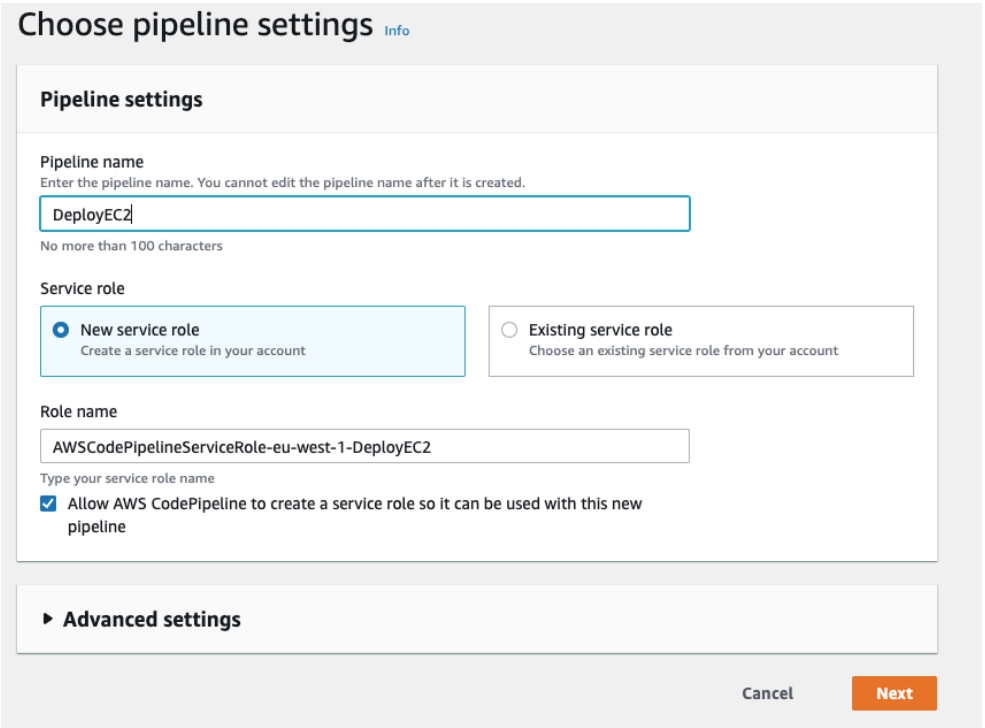


Fig.: pipeline settings

In the next steps you can just use the default values. As source select <repository>. So the pipeline is listening to the repository if something changes (a new commit to the master) the CI/CD pipeline will start. But what is the pipeline exactly doing?

The pipeline shouldn’t do a build. A build could be part of a pipeline when you want to do a npm build (React, nodeJS etc.) or a maven build (Java). After the build there is an artifact produced which could be the base for a deployment. In our case we already have the base for the deployment. Our IaC file just needs to be deployed with CloudFormation.

The following figure shows the deployment step.

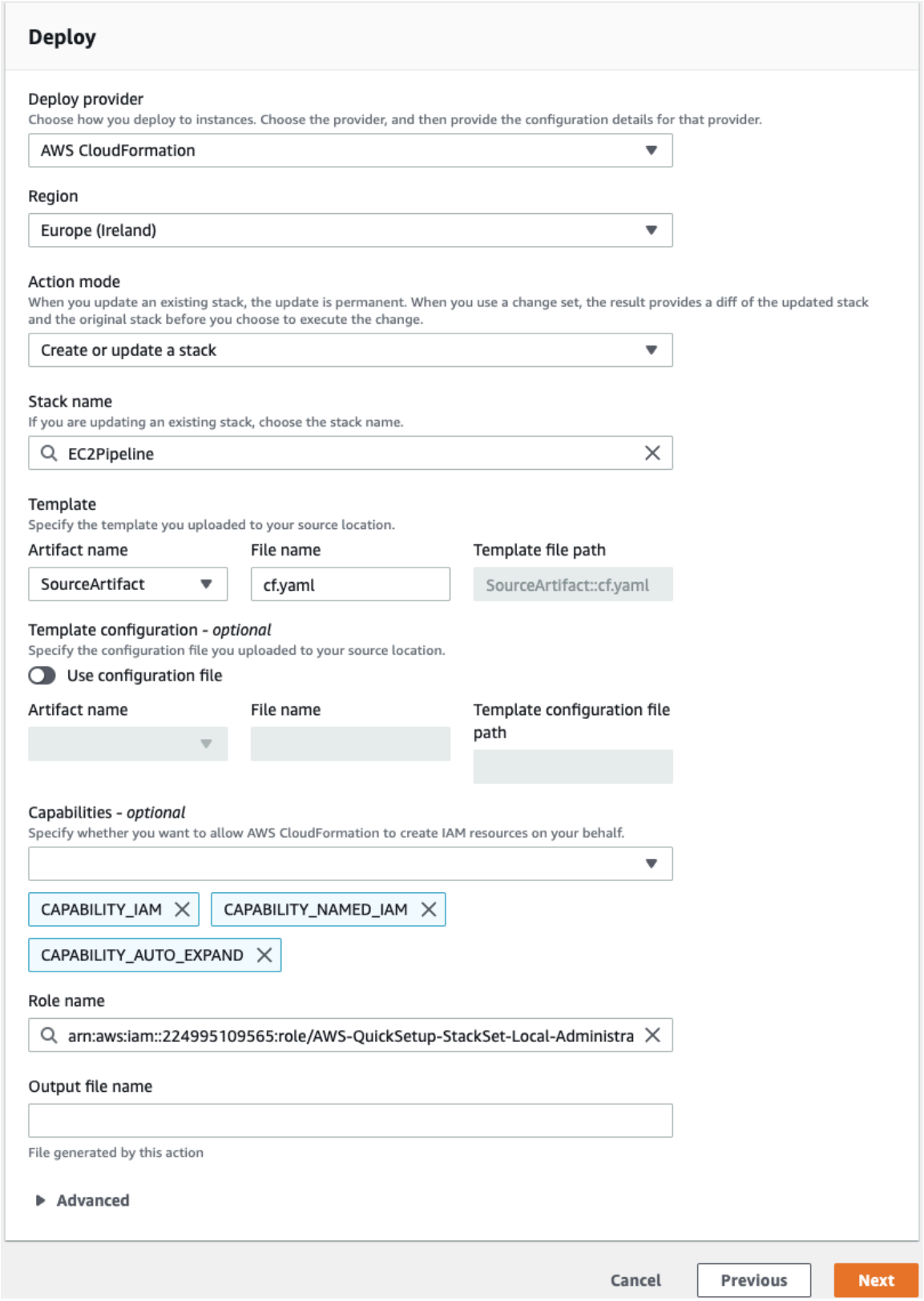


Fig.: Deployment step

⇒ Unfortunately we're facing a problem here. Which role should we choose and why? The deployment step itself needs a role to perform actions. That are:

- Perform CloudFormation
- Launch an EC2 Instance

⇒ Create a role. We need to open a new tab and create a role. Our role needs to be trusted by a service (CloudFormation) and have the above mentioned policies.

Fig.: a role with the trusted service and the use case for CloudFormation

⇒ Add the following two policies to the role.

Now you can start over again with the configuration of the deploy step in our CodePipeline.

If everything is set up correctly. You should see the following pipeline.

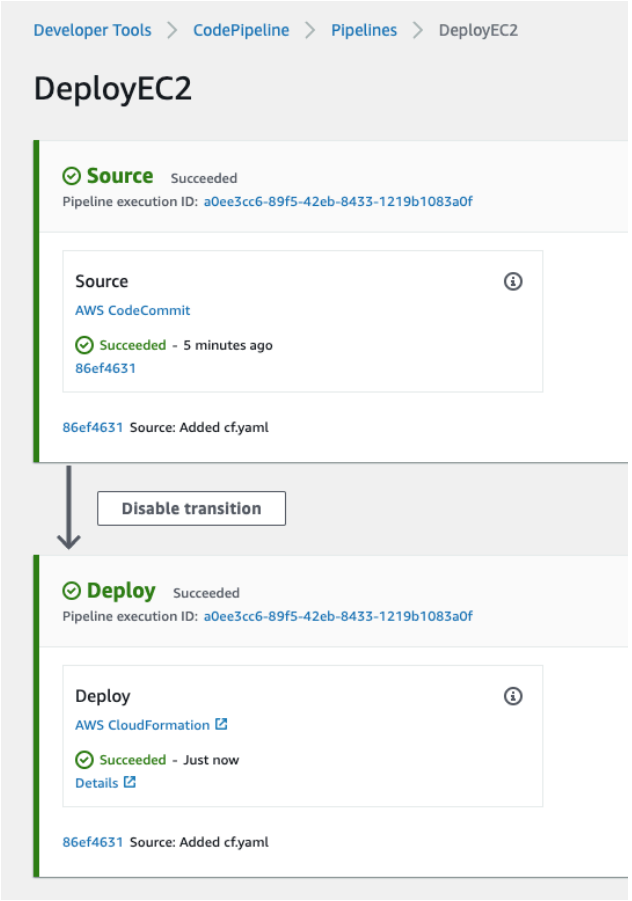
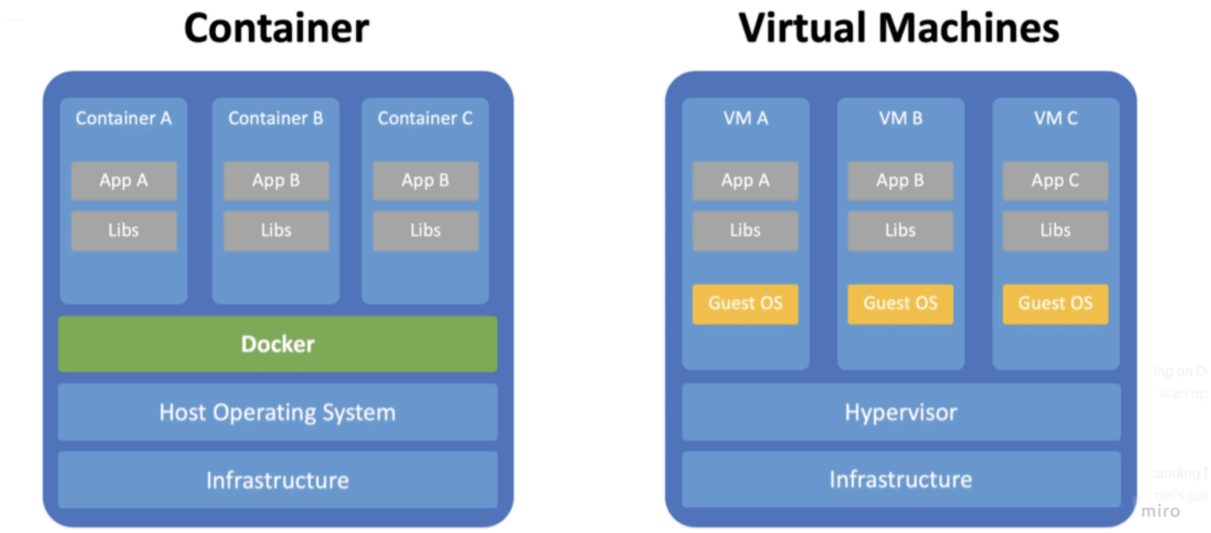


Fig.: correct running pipeline

After running the pipeline take a look at the EC2 instance. See if your instance was started. Afterwards destroy everything. You can do this in different ways. Worst solution is just to terminate the EC2 instance. A better way would be to destroy the stack in CloudFormation and the best solution would be to commit a new change to your CodeCommit repository with nothing inside. So you can trace in the version control, what's happened.

Container with Amazon Elastic Container Registry

EC2 Instances are virtual machines, containers can be run inside a virtual machine. They are way more lightweight than a virtual machine. You can run different Containers on one virtual (or physical) machine.



https://medium.com/codingthesmartway-com-blog/docker-beginners-guide-part-1-images-containers-6f3507fffc98https://miro.com/app/board/o9J_l_URb6M=/

Hands-On: 1/4 Install Docker [15]

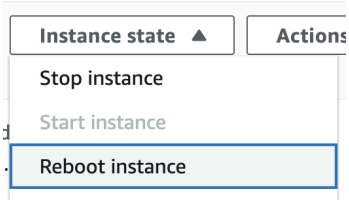
⇒ Launch an EC2 Instance with an AMI of AWS Linux 2023.
⇒ Connect to your instance (doesn't matter how)
⇒ Update Linux, install Docker and add the ec2-user to the docker group so you can execute Docker commands without using sudo:

```
sudo yum update -y
sudo yum install docker
sudo service docker start
sudo usermod -a -G docker ec2-user
newgrp docker
docker info
```

Check if <nano> is installed. It's easier to use than vi. Can be installed with <sudo yum install nano>

[15] https://docs.aws.amazon.com/AmazonECR/latest/userguide/getting-started-cli.html

⇒ Reboot your engine



⇒ See the docker info to confirm, that everything is in place.

```
docker info
```

Output should be something like:

```
[ec2-user@ip-172-31-22-253 ~]$ docker info
Client:
 Version:    25.0.3
 Context:    default
 Debug Mode: false
 Plugins:
  buildx: Docker Buildx (Docker Inc.)
    Version:  v0.0.0+unknown
    Path:     /usr/libexec/docker/cli-plugins/docker-buildx

Server:
 Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 0
 Server Version: 25.0.3
 Storage Driver: overlay2
```

Hands-On: 2/4 Configure a simple Website in Docker

⇒ Your EC2 instance should be up and running

⇒ Create a file named Docker

```
touch Dockerfile
```

⇒ Add content (manifest) to the file. Can be done by using <nano>.
We're telling the docker file that it should be based (used) on amazon linux. Afterwards we are installing a web server and exposing the port 80. And we are adding a really basic html file as index.html. Like:

```
nano Dockerfile
```

And now: Copy&Paste the following part into the file:

```
FROM public.ecr.aws/amazonlinux/amazonlinux:latest
# Install dependencies
RUN yum update -y && \
 yum install -y httpd
# Install apache and write hello world message
RUN echo '<html><body>Hello HTW!</body></html>' > /var/www/html/index.html
# Configure apache
RUN echo 'mkdir -p /var/run/httpd' >> /root/run_apache.sh && \
 echo 'mkdir -p /var/lock/httpd' >> /root/run_apache.sh && \
 echo '/usr/sbin/httpd -D FOREGROUND' >> /root/run_apache.sh && \
 chmod 755 /root/run_apache.sh
EXPOSE 80
CMD /root/run_apache.sh
```

And now: Command+O for saving the file. And then Command+X for leaving the editor.

⇒ Build the Dockerfile and list all builded containers.

```
docker build -t hello-htw .
docker images
```

⇒ Start the docker. You're now mapping your container to the port 80 of the server. In this case you're connecting 80 to 80.

```
docker run -t -i -p 80:80 hello-htw
```

8080 (Docker- Container - Expose) : 80 (Ec2 Instance)

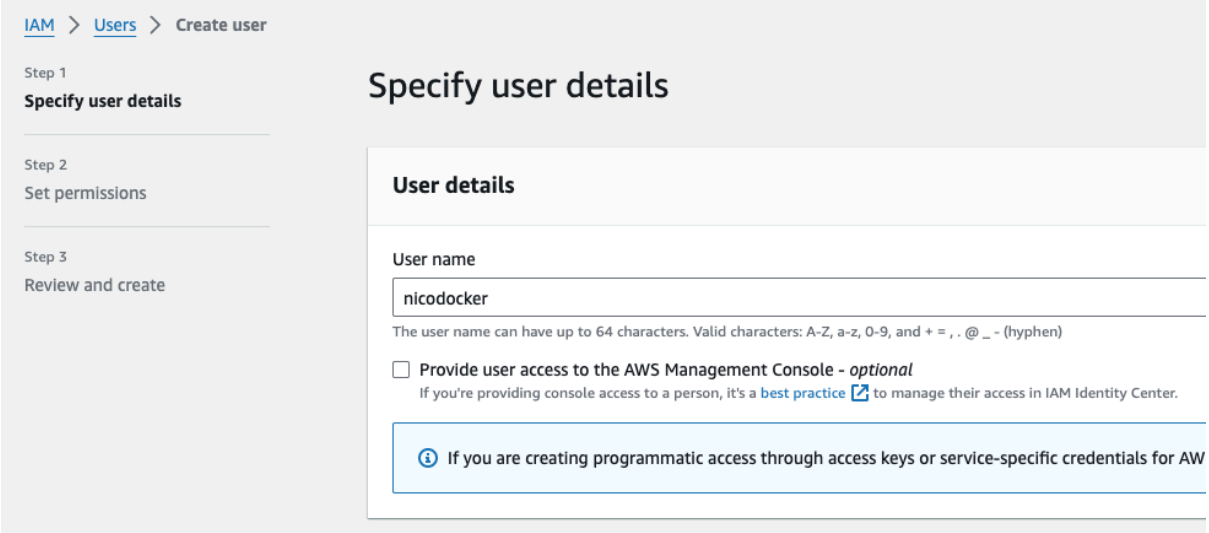
⇒ Test your website by calling your instance in a browser, e.g.:
http://<you instance>.eu-west-1.compute.amazonaws.com/

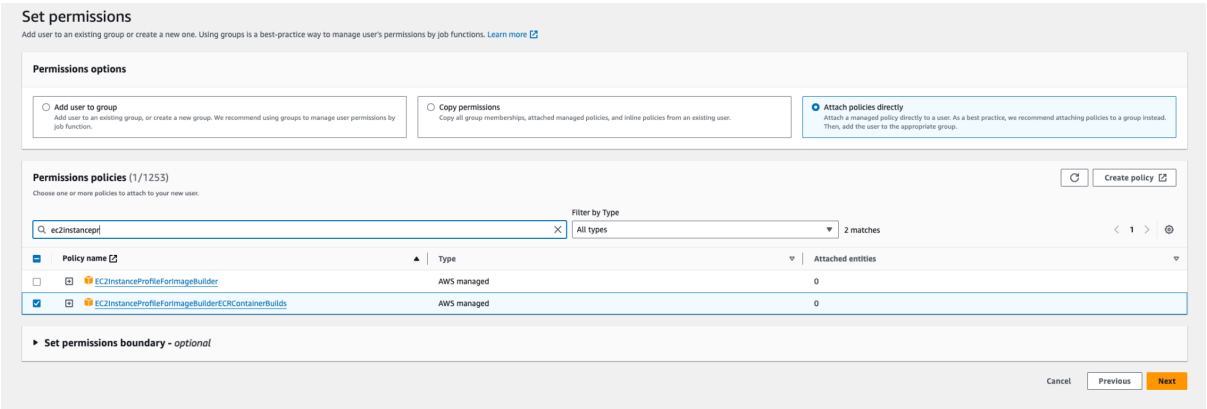
Quit with <Control + C>

Hands-On: 3/4 Register Your Container

A container should be registered in a public repository. Therefor we need several steps

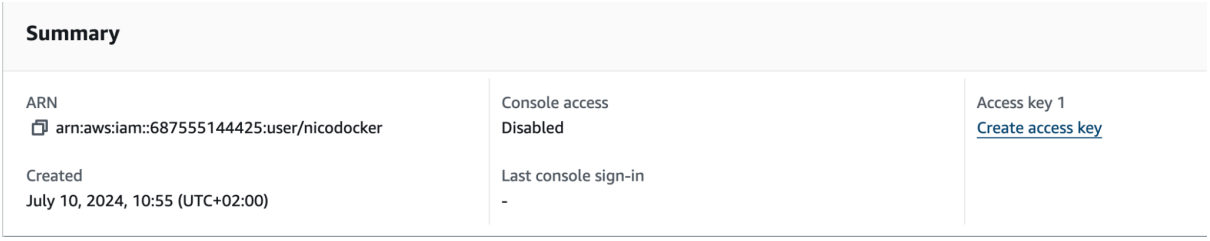
⇒ create a technical user in IAM and set the policies

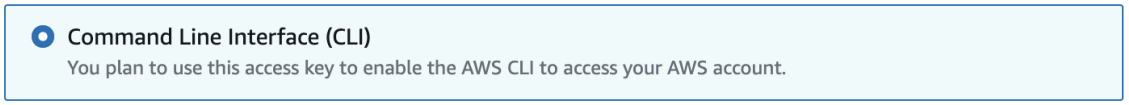




If you haven't chosen the right policy, you find out later. Maybe you need to add / create an inline policy. Where All Elastic Container Registry actions (ecr:*) are allowed.

⇒ We need to create an access key which we need later on:





⇒ Connect to your instance (if not already connected)

```
aws ecr create-repository --repository-name hello-oev --region eu-west-1
```

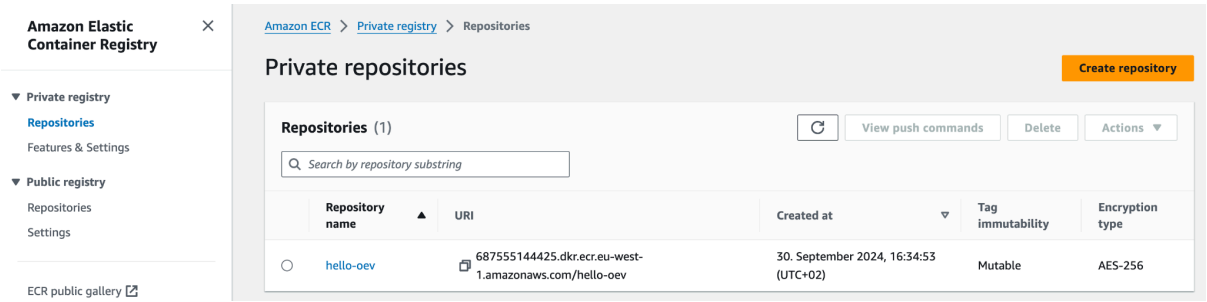
⇒ You're facing an error message, that you need to configure the credentials. Therefore we created the user in the step before.

```
aws configure
```

Your CLI should look something like:

```
[ec2-user@ip-172-31-11-246 ~]$ aws ecr create-repository --repository-name hello-oev --region eu-west-1
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:eu-west-1:687555144425:repository/hello-oev",
        "registryId": "687555144425",
        "repositoryName": "hello-oev",
        "repositoryUri": "687555144425.dkr.ecr.eu-west-1.amazonaws.com/hello-oev",
        "createdAt": "2024-09-30T14:34:53.846000+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": false
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
[ec2-user@ip-172-31-11-246 ~]$
```

Now, you repository can be found in the ECR service:



Next you need to push your Image to the repository, thus, everybody can use your awesome docker container in his/ her environment!

⇒ tag and push your image to the repository

```
docker tag <YOUR IMAGE> <YOUR REPOSITORY URL>
```

```
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
hello-htw    latest    8886f85c8671   2 hours ago   281MB
[ec2-user@ip-172-31-23-242 ~]$ docker tag 8886f85c8671 559050216210.dkr.ecr.eu-west-1.amazonaws.com/hello-proit-nico
[ec2-user@ip-172-31-23-242 ~]$ docker images
REPOSITORY                                                  TAG       IMAGE ID       CREATED       SIZE
hello-htw                                                   latest    8886f85c8671   2 hours ago   281MB
559050216210.dkr.ecr.eu-west-1.amazonaws.com/hello-proit-nico   latest    8886f85c8671   2 hours ago   281MB
```

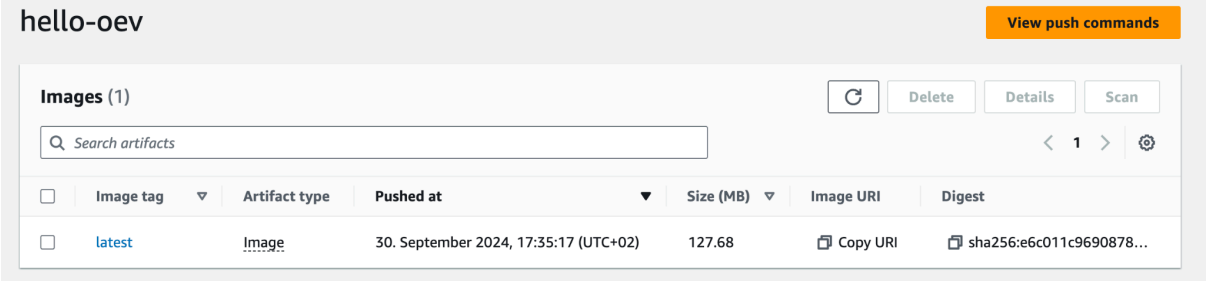
By calling docker images again, you can see that a new image is created because of the tagging. Now there is a reference between your image and the repository, i.e. one Docker Image is "local" the other one is from your global ECR - Repository.

```
[ec2-user@ip-172-31-11-246 ~]$ docker images
REPOSITORY                                          TAG       IMAGE ID       CREATED             SIZE
687555144425.dkr.ecr.eu-west-1.amazonaws.com/hello-oev   latest    64c45179363f   About an hour ago   276MB
hello-oev                                           latest    64c45179363f   About an hour ago   276MB
```

⇒ To push it really to your ECR repository, we need to do so. First we need to login into ECR from our instance by using basic authentication. Afterwards we are pushing the repository.

```
aws ecr get-login-password --region eu-west-1 | docker login --username
AWS --password-stdin <YOUR ACCOUNT ID>.dkr.ecr.eu-west-1.amazonaw
s.com

docker push <YOUR REPOSITORY URL>
```

Now our ECR should look like this:



Hands-On: 4/4 Use Docker on another machine

Now we can use our Docker file wherever we want. To do so, we need to make the same steps as before:

- Launching an EC2 instance
- Connecting to the EC2
- Installing Docker
- Authenticate Docker

Afterwards we can pull the image and run it

```
docker pull
687555144425.dkr.ecr.eu-west-1.amazonaws.com/hello-oev:latest


docker run -d -p 80:80
687555144425.dkr.ecr.eu-west-1.amazonaws.com/hello-oev:latest
```

Serverless

“Serverless computing is a method of providing backend services on an as-used basis. A serverless provider allows users to write and deploy code without the hassle of worrying about the underlying infrastructure. A company that gets backend services from a serverless vendor is charged based on their computation and do not have to reserve and pay for a fixed

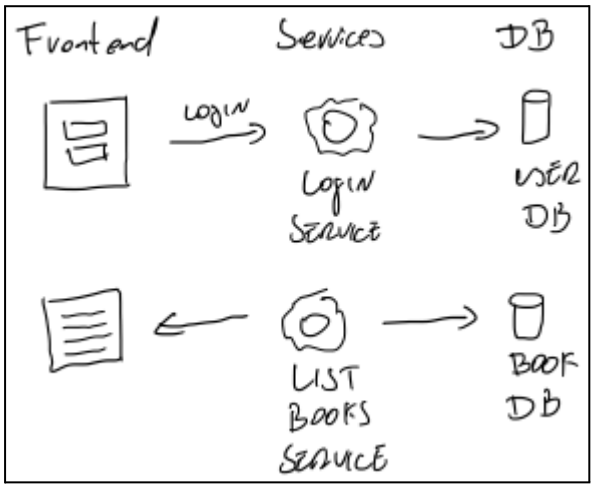
amount of bandwidth or number of servers, as the service is auto-scaling. Note that despite the name serverless, physical servers are still used but developers do not need to be aware of them."[16]

Serverless and microservices are a perfect match. So let's have a look at microservices.

Microservices

"Microservices architecture (often shortened to microservices) refers to an architectural style for developing applications. Microservices allow a large application to be separated into smaller independent parts, with each part having its own realm of responsibility."[17]

Let's think about a bookstore. We have a lot of services like: login, show all books, add new books, order a book, do the payment etc.. Created as a microservice architecture it would look like:



Definition of Microservices

There is no "one" definition of microservices. There is a definition at microsoft which states the following:

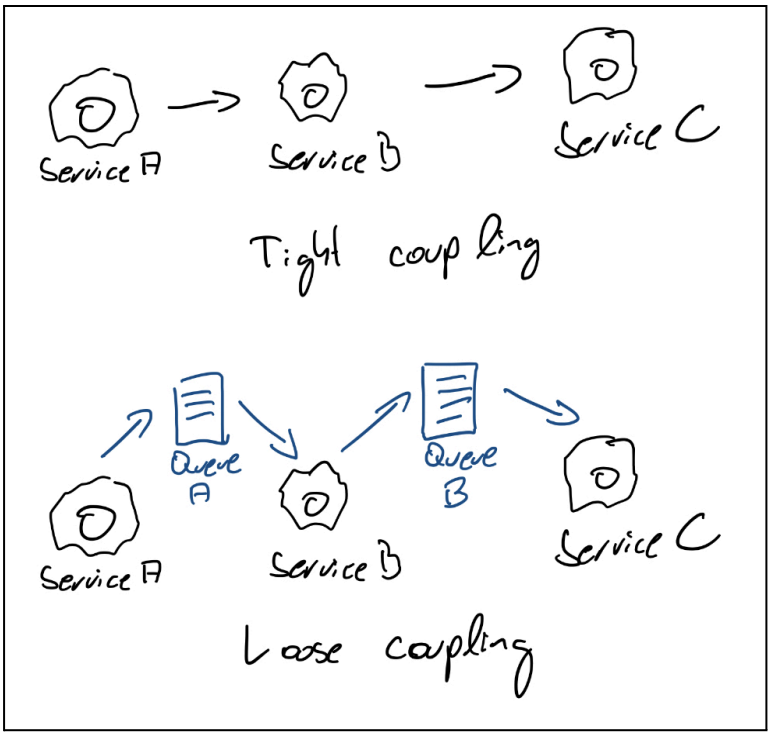
- "Microservices are small, independent, and loosely coupled. A single small team of developers can write and maintain a service.
- Each service is a separate codebase, which can be managed by a small development team.
- Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.
- Services are responsible for persisting their own data or external state. This differs from the traditional model, where a separate data layer handles data persistence.
- Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services.

[16] https://www.cloudflare.com/learning/serverless/what-is-serverless/
[17] https://cloud.google.com/learn/what-is-microservices-architecture

- Supports polyglot programming. For example, services don't need to share the same technology stack, libraries, or frameworks."[18]

The definition doesn't clearly state the difference between a microservice and a microservice infrastructure. Thus some aspects are mixed up. But due to the fact that a microservice without a microservice architecture doesn't really make sense, we keep the definition.

One aspect is the coupling of microservices. The first figure shows a tight coupling of microservices. Here evere microservice is calling directly another service. Let's imagine something went wrong in Service B and we want to fix it, we have a direct dependency on Service A and Service C. In the worst case the whole system crashes. The next figure shows us a better way to interact with microservices. Here we use a queue instead of direct calls.



Scalability

A Microservice could be scaled quite easily. Due to the fact that a microservice stands for itself and it's independent, it could be scaled for itself. By that way you can scale every microservice exactly how you need it.

In contrast, a monolith (installed on an EC2 instance) can only be scaled at once. You can only scale the whole system. Imagine there is just one huge data warehouse calculation running once a year inside the system. You need to scale everything to perform the issue.

Hands On: First Microservice

In our last hands-ons we were really close to real infrastructures. We used the infrastructure as a service (IaaS). We still had to install applications and update the operating system. Now

[18] https://learn.microsoft.com/en-us/azure/architecture/microservices/

we're going to use services on a higher layer. Here we don't have to care about updates and installations. This layer is called functions as a service (FaaS). In our first example we are using plain functions. In AWS they are called Lambda Functions.

⇒ **Implement a Lambda Service.** Go through the following tutorial: Run a Serverless "Hello, World!" with AWS Lambda. Instead of the python blueprint you can use nodejs or other programming languages as well. There is nothing special about this service, but you can see that you can easily implement programs without the need of setting up an instance with an operating system and installing Java, nodejs or python on it.

Hands On: First Serverless Website

Now we want to host a new Website. Remember the raw server hands-on examples? Here was the need to:

- Launch an EC2 instance
- Install an apache web server by connecting via scp
- Configure security groups
- Create an load balancer, a scaling group and a launch configuration for scaling

And all this effort just for a "Hello World". Serverless keeps all this much simpler. We are using S3 (simple storage system). This can also be seen as a special function (FaaS). We just need to perform some simple steps like: create a bucket, upload a file and make some configurations and then we have a highly scalable, highly resistant infrastructure.

⇒ **Use this tutorial:** Tutorial: Configuring a static website on Amazon S3

⇒ **Draw the infrastructure and compare it to the infrastructure of the raw server with scaling.**

Hands On: A pipeline for the website

Create a new pipeline to "upload" the website to your S3 bucket instead of simply uploading it. Use CodeCommit as a repository. In CodeCommit you should upload your html file:

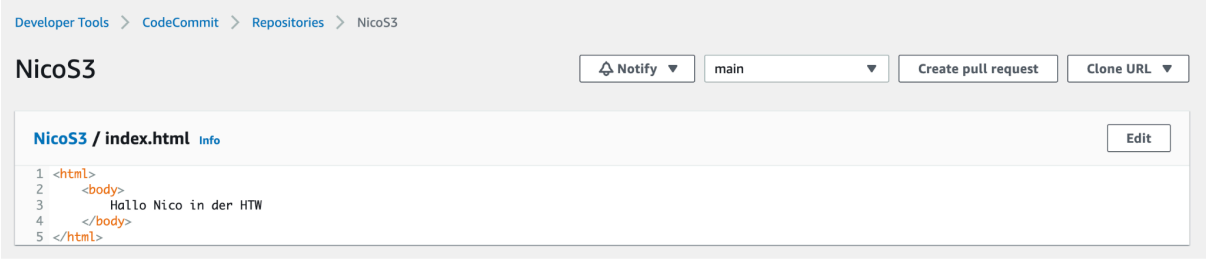


Fig.: CodeCommit with a index.html in the repository

Now create the pipeline and skip the build stage again.

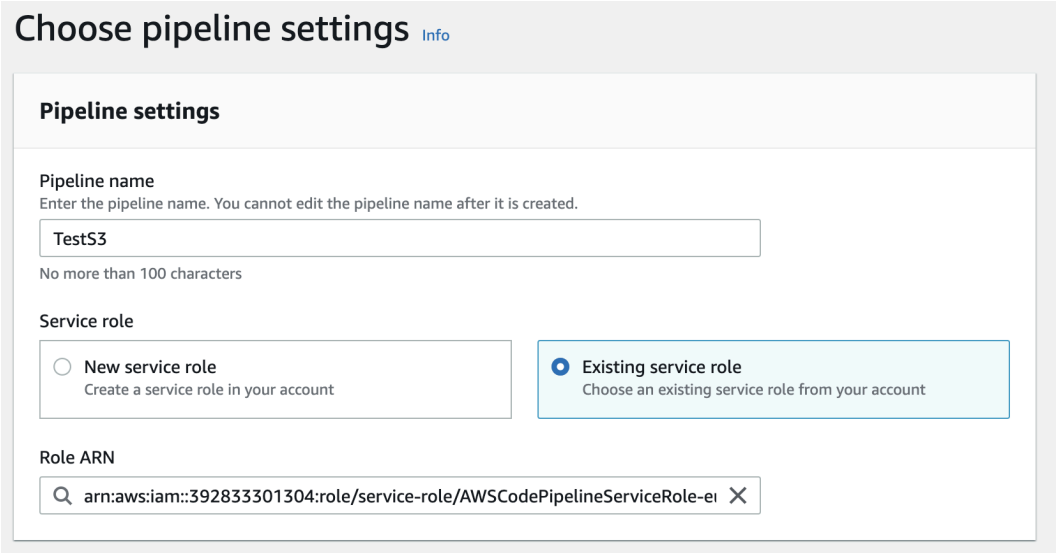


Fig. Create a pipeline

Use S3 in the deployment phase as a provider.

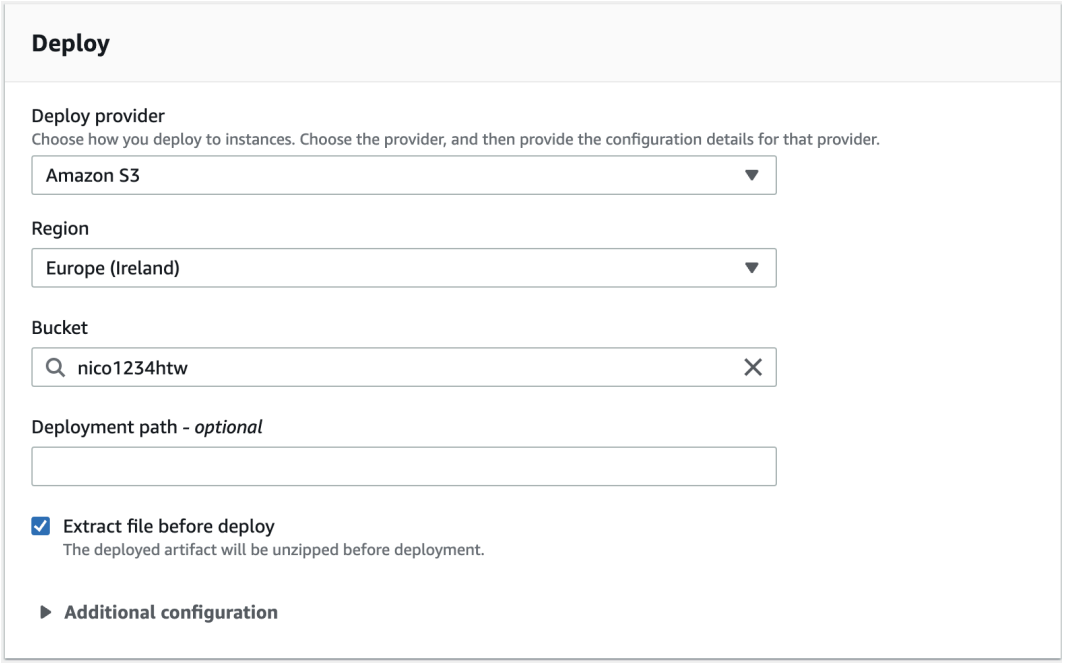
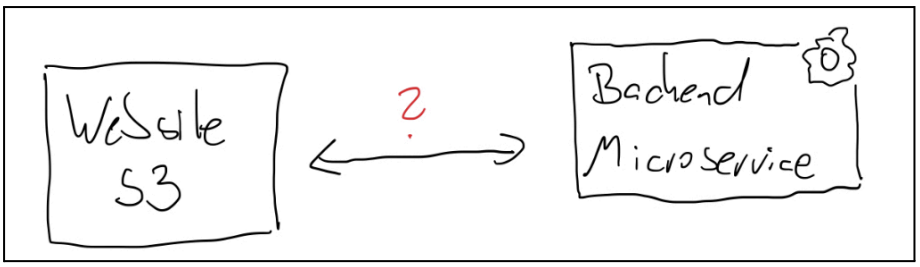


Fig. deploy stage in the pipeline

Perform the pipeline. You should see the index.html file deployed in your S3 bucket.

Application Program Interfaces (API)

Interfaces are needed between services / functions or between a frontend and a backend (see the question mark in the following figure).



There are some possibilities to set up an interface. In a microservice architecture the interface could be a message system or an API call.

API Calls

The most common way to perform API calls is via a RESTful interface. Here usually REST over HTTP with a JSON payload is used.

Fig. Schema of a REST call

We can use all HTML request methods like POST, GET, DELETE and perform a http request with a JSON payload. For example we send a request like http://mybookstore.com/listbooks?name=john as a result we get a json file like:

```
{
   "books":[
      {
         "name":"John Grisham",
         "title":"The Partner"
      },
      {
         "name":"John Grisham",
         "title":"The Pelican Brief"
      }
   ]
}
```

We don't call the microservice directly but via an API gateway. Thus you can separate the business logic in the microservice from the gateway. That helps for example when you're

changing the microservice. You can create a new version of a microservice and later switch the API to the new one. Thus the API is in some way like the load balancer or routing service in our server based infrastructure.

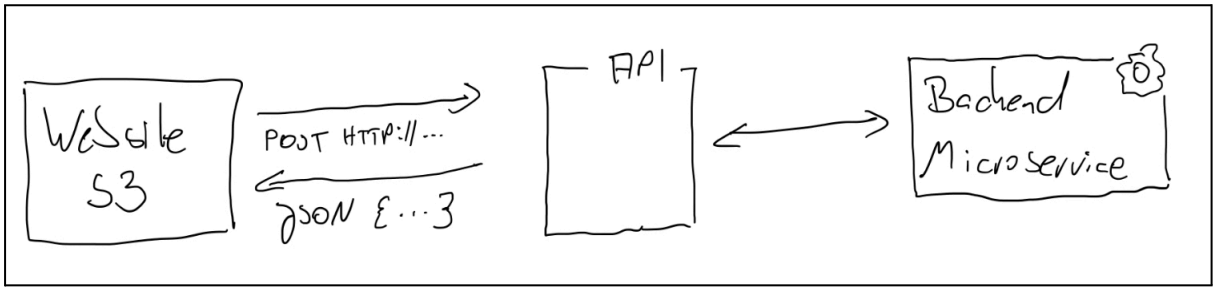
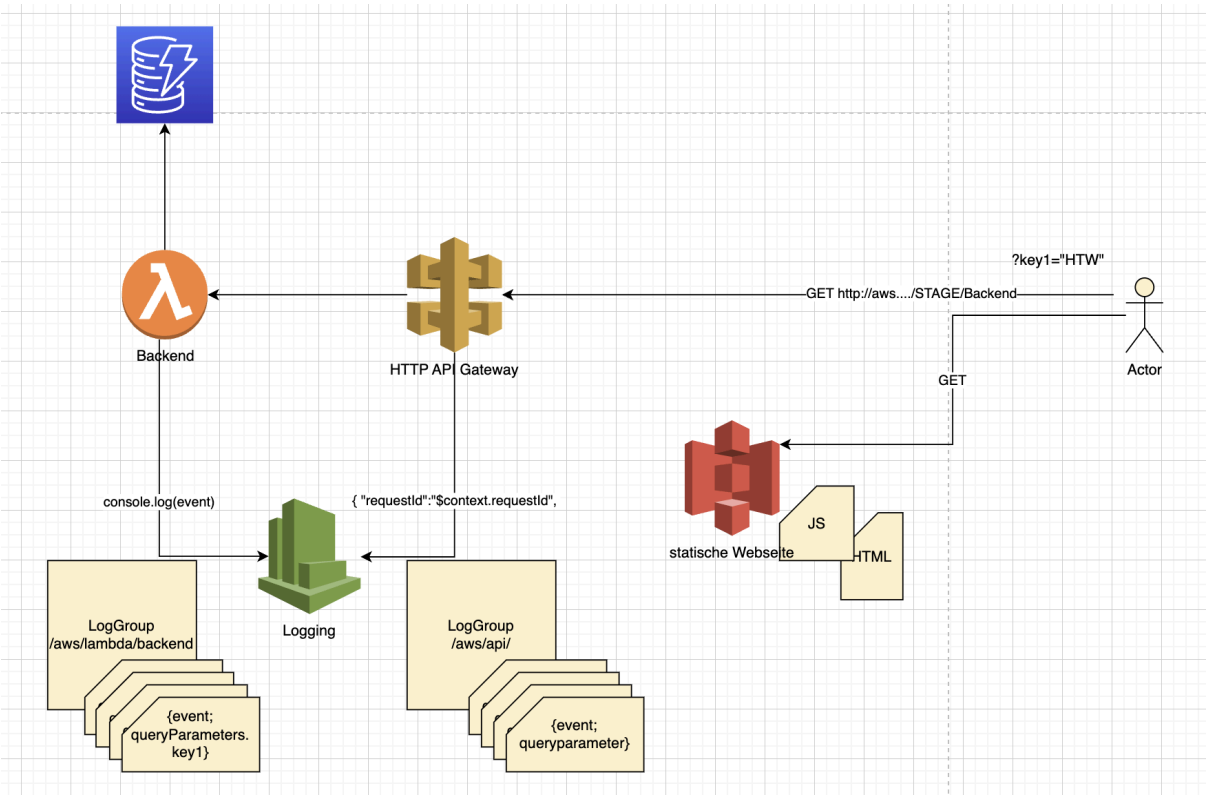


Fig. A Rest call with an API gateway

Hands on: Serverless Infrastructure - Microservices

Now that we learned about microservices and API gateways we start with our first complex example.

The overall infrastructure will look like this:



A lambda function as we learned before, is a (small) code snippet in any programming language that can be executed. It can be executed synchronously or asynchronously.

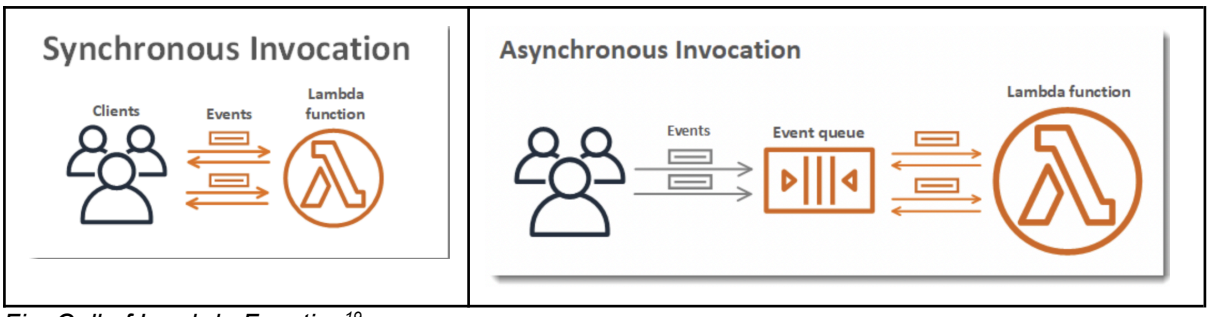
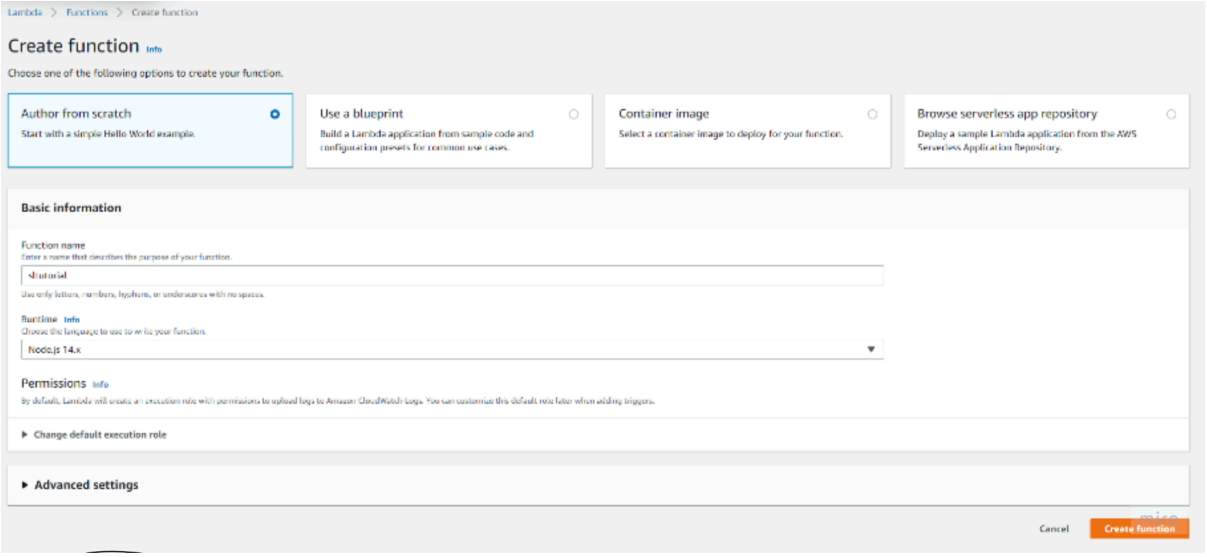


Fig. Call of Lambda Function[19]

⇒ **Create Lambda Function with Python from scratch**. Go to the Lambda services and create the function. You can also use a different programming language.



Add the following code:

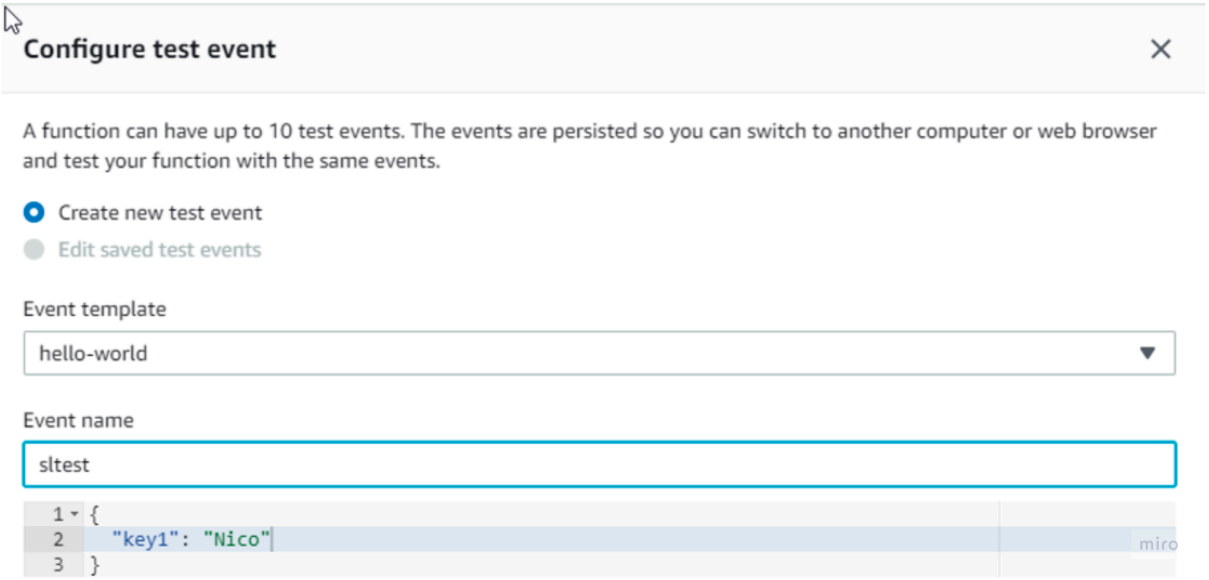
```
import json

def lambda_handler(event, context):
    # Retrieve 'key1' from the event, or set a default
    key1_value = event.get("key1", "")

    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda! {key1_value}')
    }
```

The function is called asynchronously. It returns a response with the attributes: statusCode and body. As you probably can imagine, these attributes match for an html request. When you perform a call like this through a browser, the browser will expect a statusCode - 200

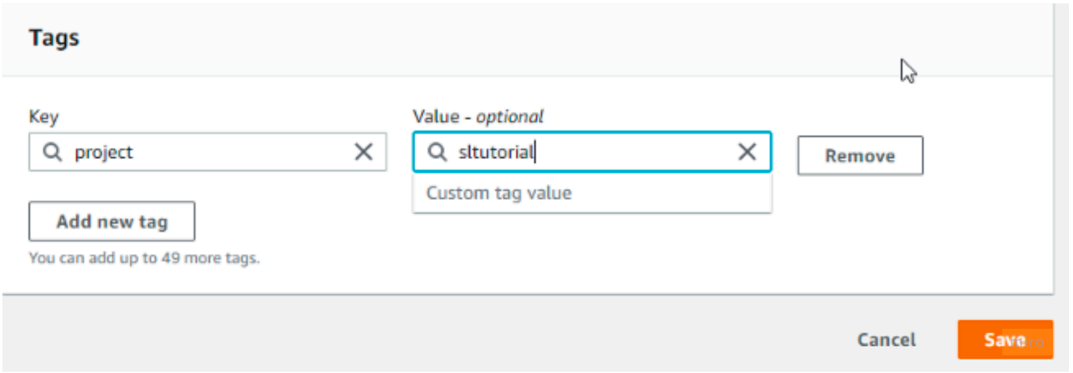
[19] https://docs.aws.amazon.com/lambda/latest/dg/lambda-invocation.html

(means everything is fine) and a body. The body is a simple text, but it could also be a complete HTML page.

⇒ **Configure the test**. This can be used to test the release. It is important that the function has been deployed beforehand. As you can see in the source code above and in the test, there is a parameter called <<key1>>. Later on we'll use this parameter to hand over values from another service or the webpage.



⇒ **Use tags to find the function**. Tags can be found everywhere. They are a great way to identify projects and resources. You can use them for example to identify all resources which belong to a certain project or you create a tag with the name of a responsible person for a service.



Hands on: Serverless Infrastructure - API Gateway

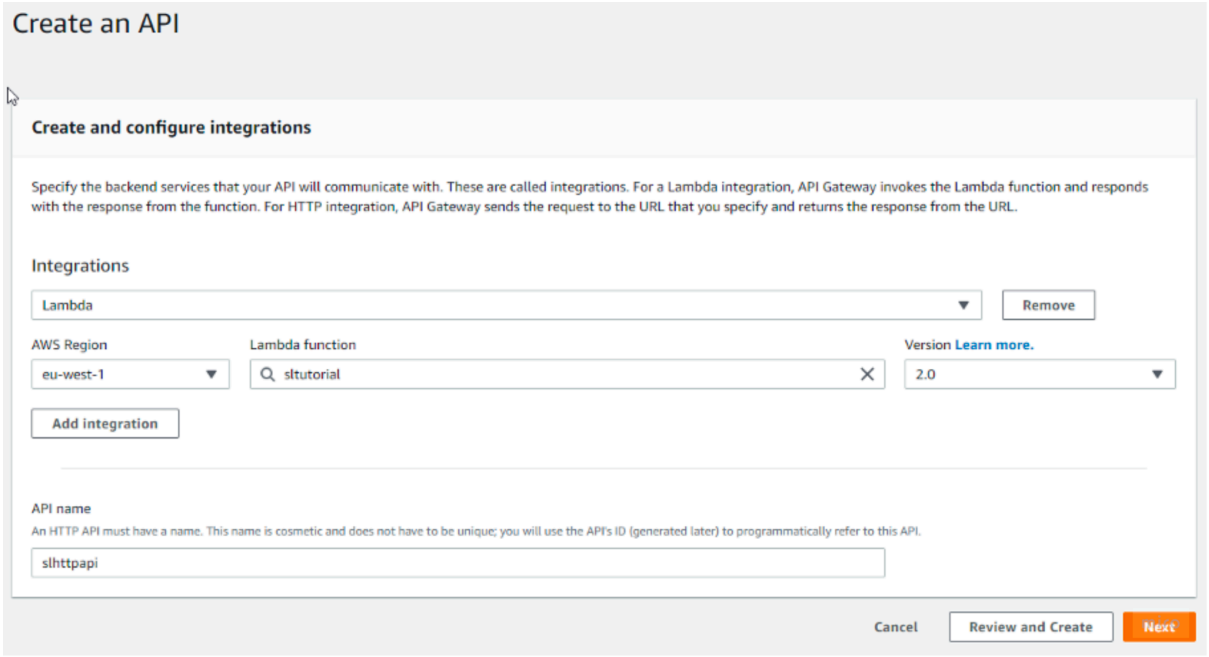
Now we need a way to call our microservice from the outside. Thus we need to add an API gateway. There are a lot of ways to achieve this. We can start from our service or we can go to API Gateways in the service menu and start to create the gateway here.

There are 3 types of API Gateways.[20]

[20] https://aws.amazon.com/de/api-gateway/

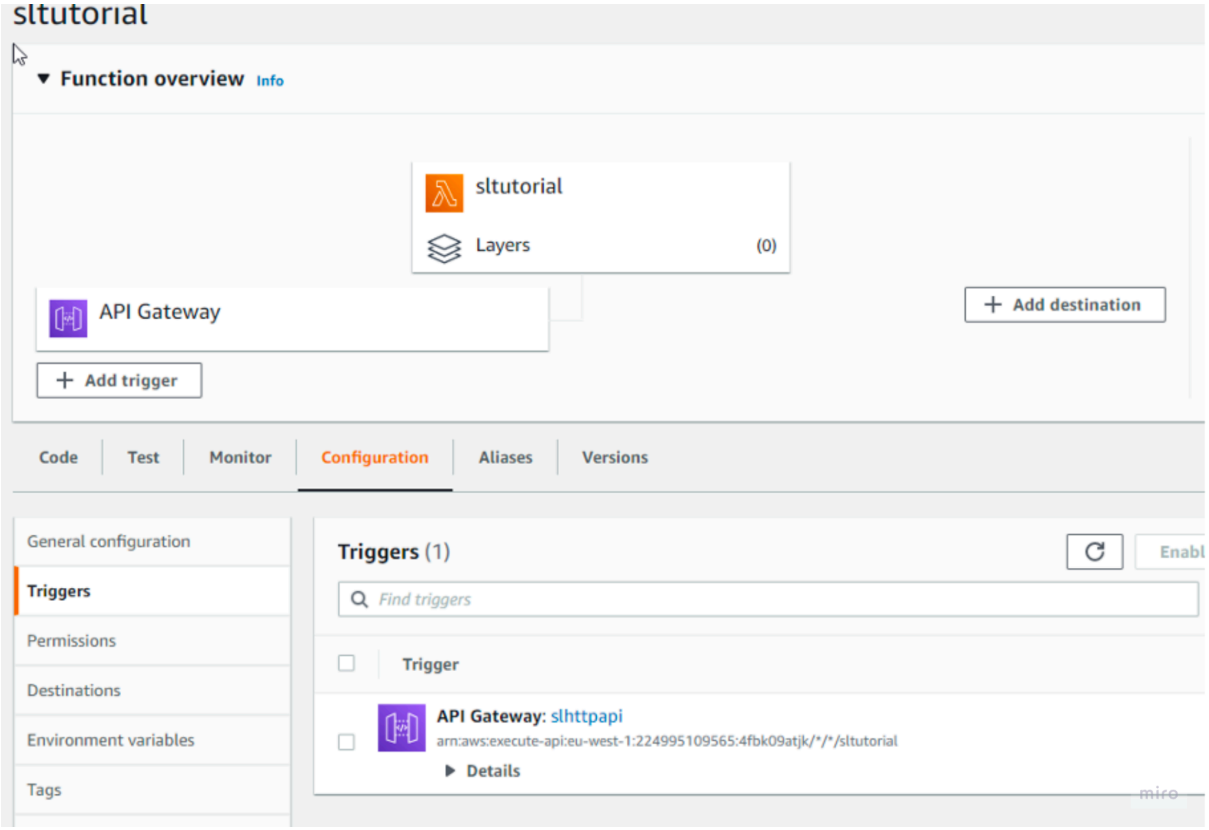
- WebSocket for permanent connection between backend and frontend.
- REST for connecting various services.
- HTTP for RESTful only for Lambda[21].

⇒ **Create a REST API gateway directly as a trigger**. We only want to access the lambda with GET. Select the stage name as desired.



In the result there was a trigger added (see the next figure).

[21] https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api.html



When we analyze the API url we can find some details. The first part of the link identifies the domain. The domain could also be replaced by an DNS registered for example in Route53. The next part is the stage followed by the service. The stage could be used for staging (prod, dev) or versioning (v1, v2). See the following figure:

Domain: https://4fbk09atjk.execute-api.eu-west-1.amazonaws.com

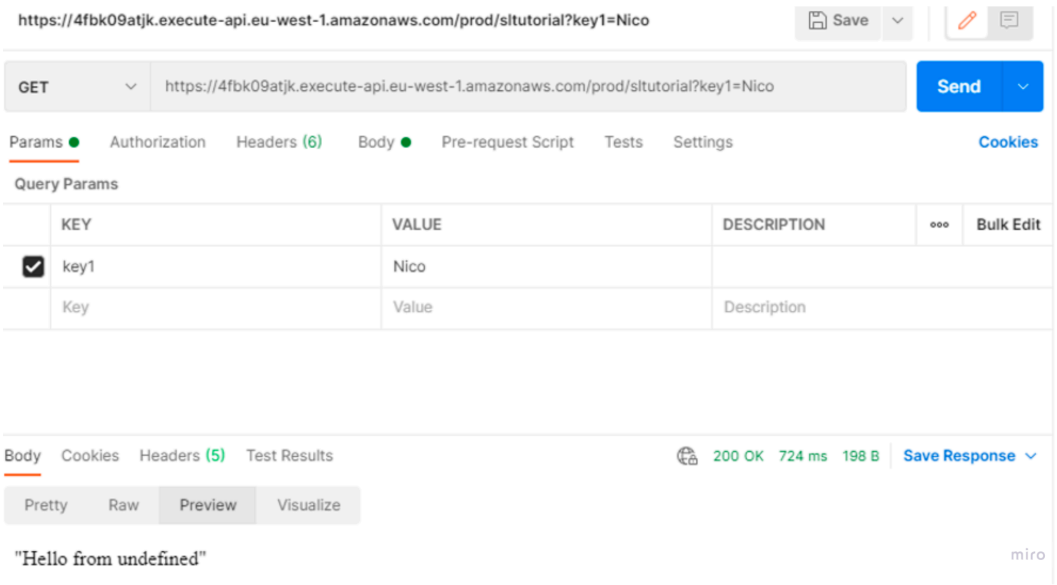
Stage : /prod

Service : /sltutorial

https://4fbk09atjk.execute-api.eu-west-1.amazonaws.com/prod/sltutorial

Our API gateway is working as a proxy. Thus we just route everything from the internet to the microservice and the other way around. It's actually a good idea to write a mapping between the API and the microservice. Doing this, makes your microservice more autark. On the other hand, it's more effort to configure the service. As often, it's a tradeoff.

⇒ **Use Postman to execute the request**. In postman you can set the query parameter and define the method. In our case we're using GET. Now it's time for our first test. The result should be similar to the figure. As a response you get the status code 200. This is only the status code you set manually in your microservice.



Hands on: Serverless Infrastructure - Error Handling

We handed over a parameter *key1*, but obviously it wasn't taken into account, because the result is: "Hello from *undefined*".

⇒ **Check all logs in the lambda function**.
These logs can be found in the Lambda Service → Monitoring → View Logs in Cloud Watch. You find something like:

```
queryStringParameters: { key1: 'Nico' },
multiValueQueryStringParameters: { key1: [ 'Nico' ] },
requestContext: {
  accountId: '687555144425',
```

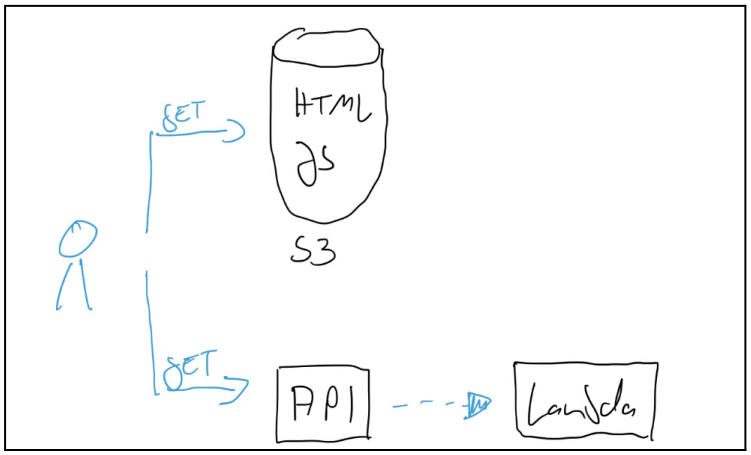
When you compare this to your output, you see that you make no use of *queryStingParameter*. Thus there is the need to change the code. Please note that the test no longer works afterwards. We would have to adapt the test here as well.

⇒ **Change the code and the test**. Afterwards you should see the right output.

```
query_params = event.get("queryStringParameters", {})
key1_value = query_params.get("key1")

[...]

 'body': json.dumps(f'Hello from Lambda! {key1_value}')
```

Hands on: Serverless Infrastructure - Webpage

We want to set up a simple status website (like in one of the first hands-ons). The overall goal should be like this:



We have a Website in HTML and JS, after clicking a link, we call our API-Interface and then the Lambda is called.

⇒ **Option 1) Build a simple HTML page** that has an input form (name) and then outputs the name again. For this, the code can be pulled from GitHub or you download the directory from the GIT **in a local folder** and then simply change/adapt it locally.

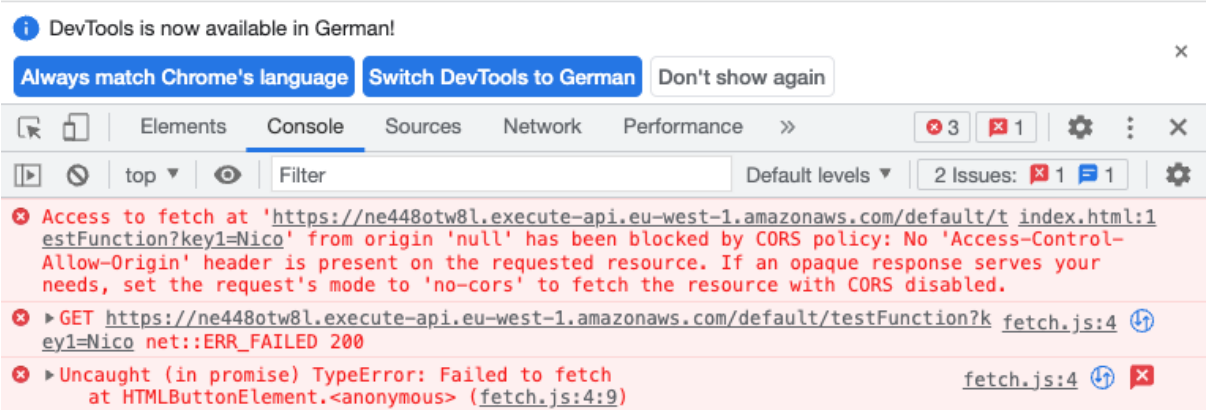
```
% Git clone https://github.com/nschoennagel/website
```

⇒ **Change the URL for your API-Call**.

⇒ **Option 2) Build a simple HTML page with ChatGPT Support**
Let your index.html be created automatically - for example by using a prompt like this:

> create a html which I can place on a AWS S3 bucket. It should be a formular with name and email. I want to send the input fields via an API gateway to a lambda function. Send the request with a get request and hand over the parameters in the query parameters. Call the email para1. use this endpoint of my API: https://5f4hosat20.execute-api.eu-west-1.amazonaws.com/default/nicostore

⇒ **Test your HTML page locally.** Open your index.html. An error will occur, after pressing the button.

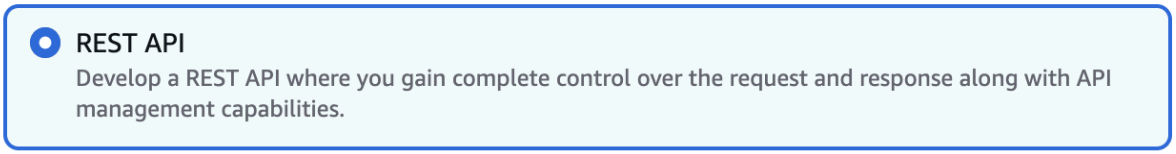


There are CORS errors. The reason is that you try to access your API gateway from a different domain. Our browser is clever enough to recognize this and is throwing an error. The error states: “No Access-Control-Allow-Origin”. Thus we define an exception, that the call of our functions is allowed from everywhere.

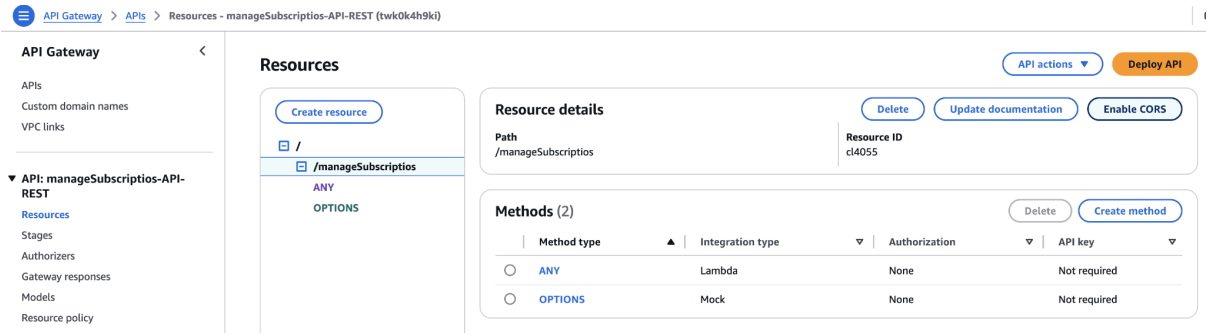
⇒ 1) Add CORS to your lambda (just in case)

```
headers: {
"Access-Control-Allow-Origin": "*",
},
```

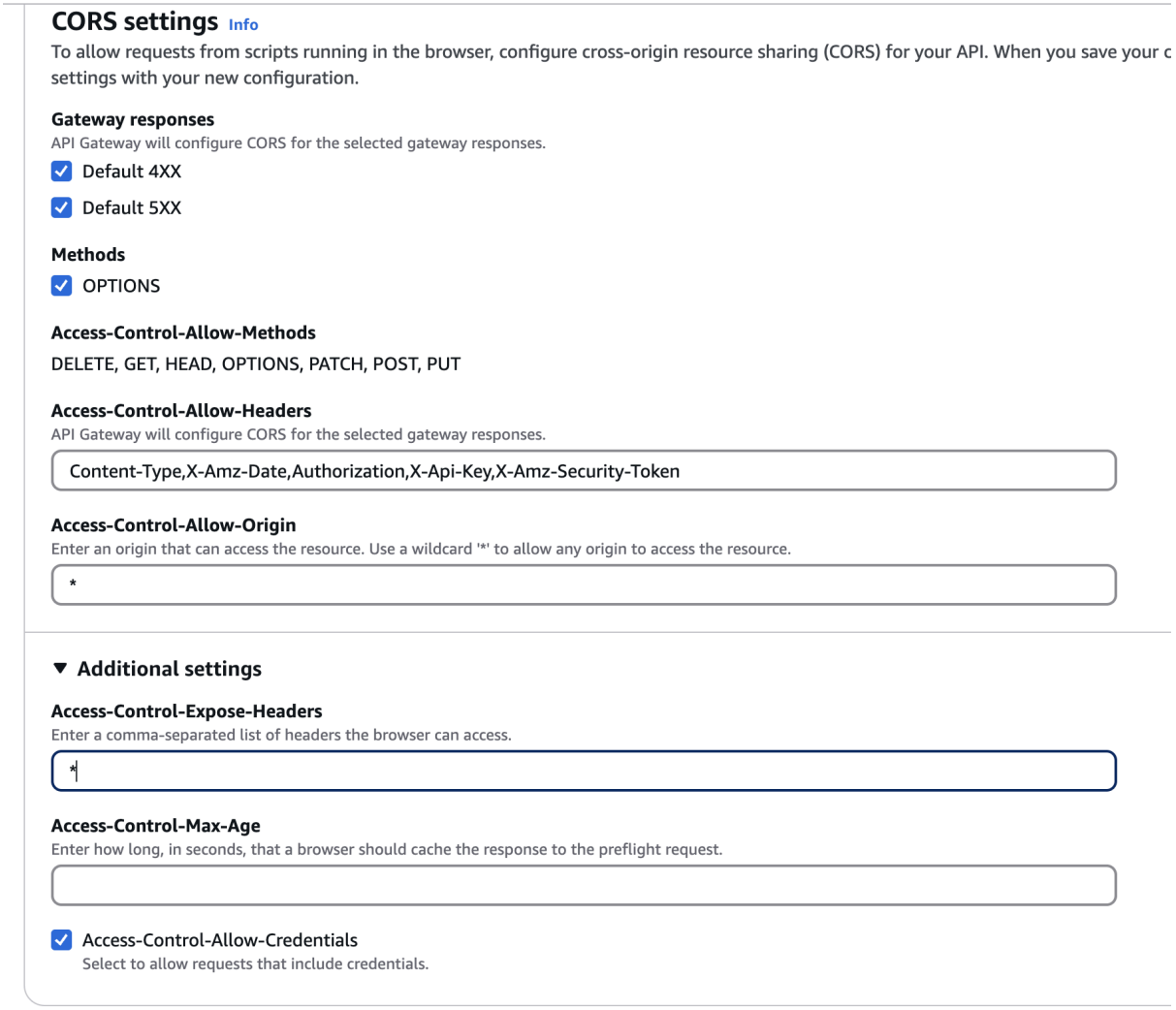
⇒ 2) Adapt the API Gateway
Remember, that you should have used an REST API!



Open the API Settings:



Enable CORS in the setting of the API like this: (Attention: unfold the Additional Settings)



Now, when opening the index.html, you should see the following result:

⇒ **Upload everything to S3**. Therefore you need to create a public S3 bucket. Remember to make the bucket accessible as a web page.

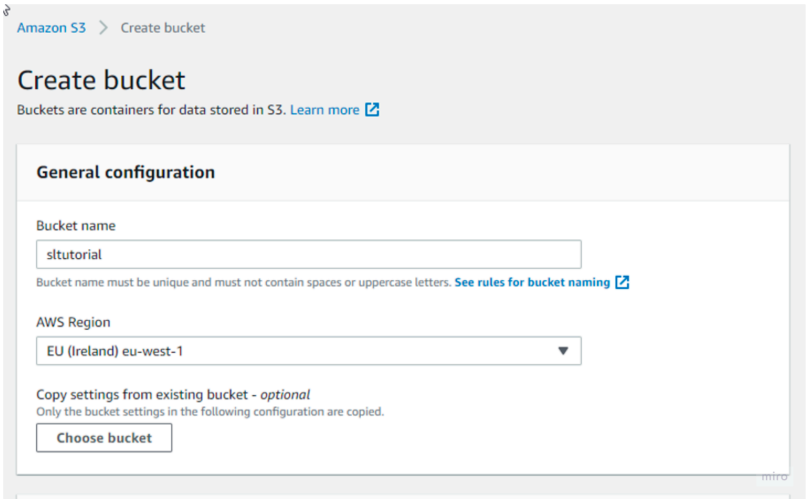


Fig. Create a S3 bucket

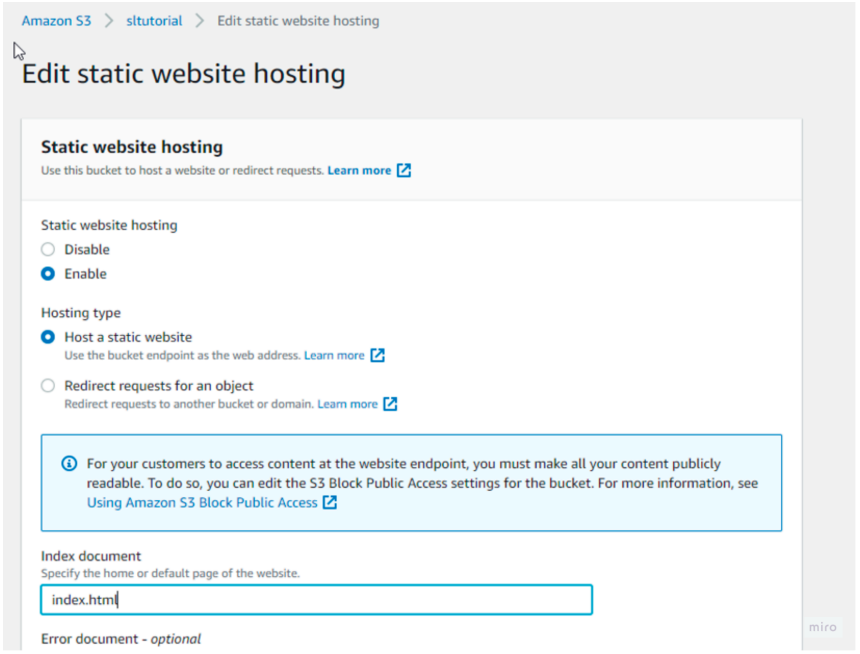


Fig. Make the bucket a static website

You also need to edit the rights to allow access to your website. As a policy you can use this one. Here you say, that a read access (GET) is allowed for everybody. The settings can be make in *Permissions → Bucket policy*.

```
{
  "Version": "2012-10-17",
  "Id": "Policy1616691635283",
  "Statement": [
  {
    "Sid": "Stmt1616691630099",
    "Effect": "Allow",
    "Principal": "*",
    "Action": "s3:GetObject",
```

```
    "Resource": "arn:aws:s3:::<<YOUR BUCKET>>/*"
  }
  ]
}
```

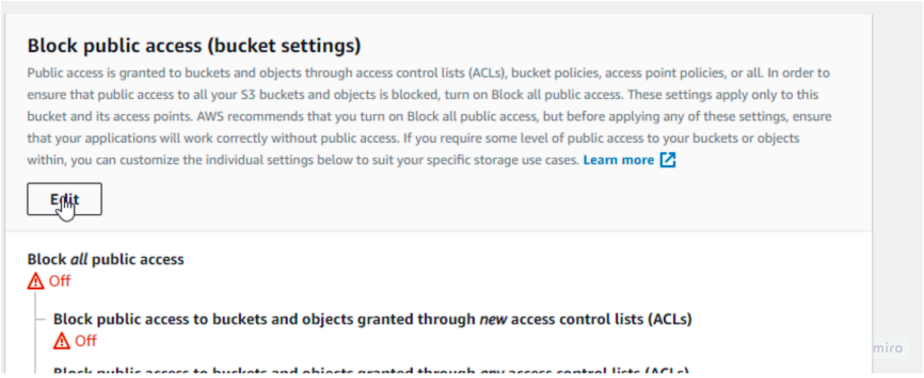
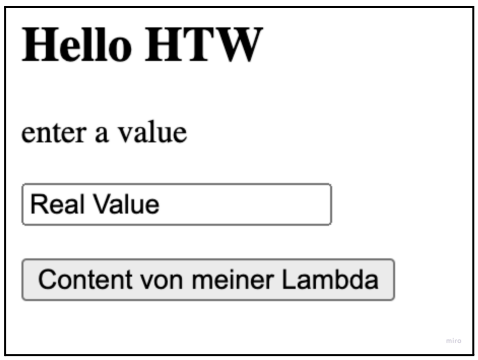


Fig. allow access to the page

⇒ **Test your page.**

Hands on: Serverless Infrastructure - More dynamic

We want to dynamically add a parameter to our webpage and get the result back. It could be look somehow like the following figure:



To archive this you need to add a formula and input to your html and adopt your JS as well. See the following two screenshots for inspiration.

```
window.onload = function() {
   document.getElementById('button').addEventListener('click', function () {
       let val = document.querySelector('input[id="myvalue"]').value;
       console.log(val);
       fetch('https://<<YOUR URL>>?key1='+val)
       .then((response) => {
           return response.text();
       })
       .then((myContent) => {
```

```
            document.querySelector('.content').innerHTML = myContent;
        });

    }, false);
}
```

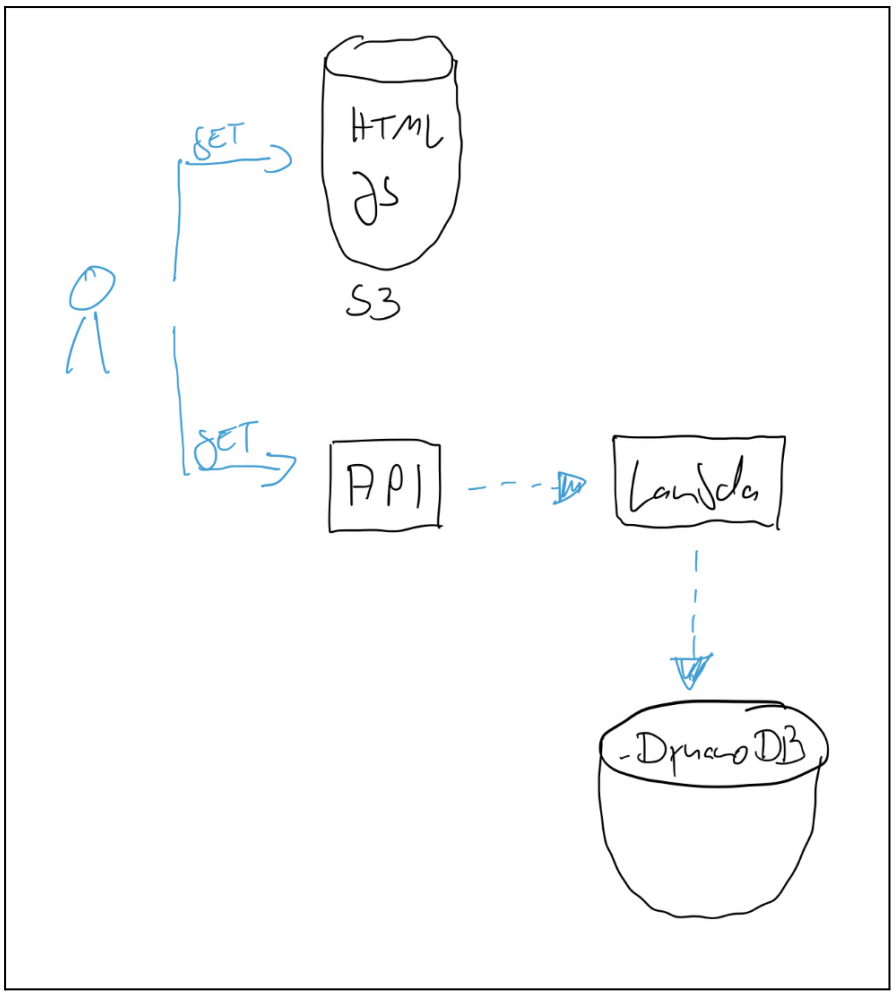
```
<html>
    <head>
        <script src="fetch.js"></script>
    </head>
    <body>
        <h2>Hello HTW</h2>

        <p>enter a value</p>
        <input id="myvalue" type="text"/>

        <p class="content"></p>

        <button id="button">Content von meiner Lambda</button>

    </body>
</html>
```

Hands on: Serverless Infrastructure - Add a database

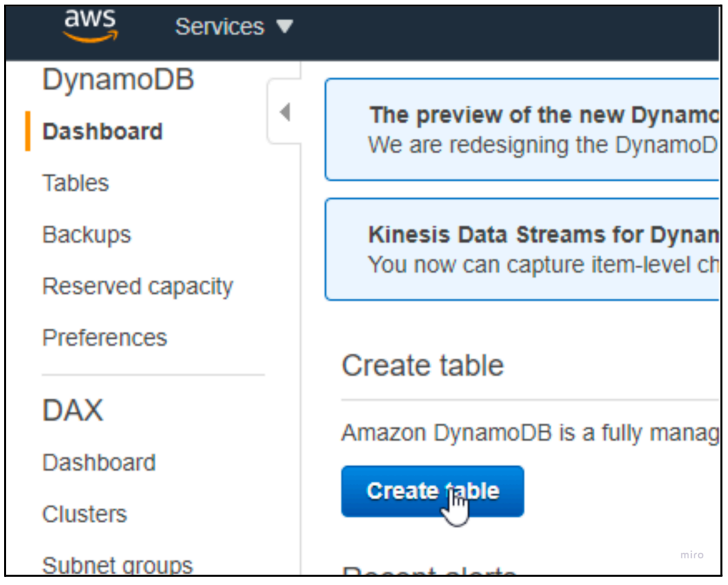
A DynamoDB is an object-oriented database like MongoDB. “An object-oriented database (OOD) is a database system that can work with complex data objects — that is, objects that mirror those used in object-oriented programming languages.”[22]

Due to its kind, the DynamoDB could be easily connected to and used in microservices.

[22] https://www.mongodb.com/databases/what-is-an-object-oriented-database



⇒ **Create a database**: Find more information about DynamoDB right here: [Amazon DynamoDB Examples - AWS SDK for JavaScript](#) (just to gain more informations)

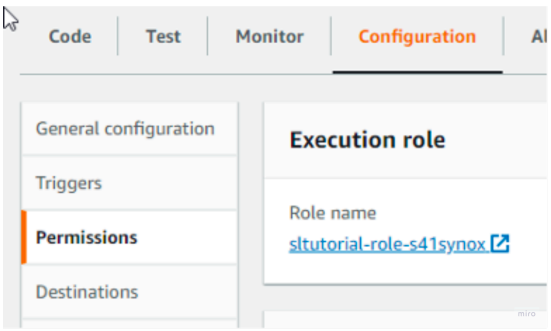


⇒ **Integrate your database with your lambda function**. Try it yourself with the link above. Here is some advice anyway, you can also find the source code here or check the following figure: https://github.com/nschoennagel/Microservice *(nodeJS)*

Here in Python:

```python
import json
import boto3

# Initialize the DynamoDB client
dynamodb = boto3.resource('dynamodb')
table_name = '<<YOUR TABLE NAME>>'
table = dynamodb.Table(table_name)

def lambda_handler(event, context):

    # print out event
    print("Received event:", json.dumps(event, indent=2))

    # Extract data from query string parameters
    query_params = event.get("queryStringParameters", {})
    email = query_params.get("key1")

    # Prepare the data to be stored
    item = {
        'key': key1,        # Primary Key
    }

    try:
        # Store data in DynamoDB
        table.put_item(Item=item)
        return {
            'statusCode': 200,
            'body': json.dumps(f'Successfully stored data for {email}')
        }
    except Exception as e:
        # Catch any other exceptions
        return {
            'statusCode': 500,
            'body': json.dumps(f'Unexpected error: {str(e)}')
        }


    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

⇒ **Change the policy**. Now that our function should write to a database, we need to give the right rights to this function. That could be done by the roles. Go to the current role:



When viewing the role, you'll see that the permission for the database is missing. We can simply add this by creating a new policy and attach the new policy to the role.

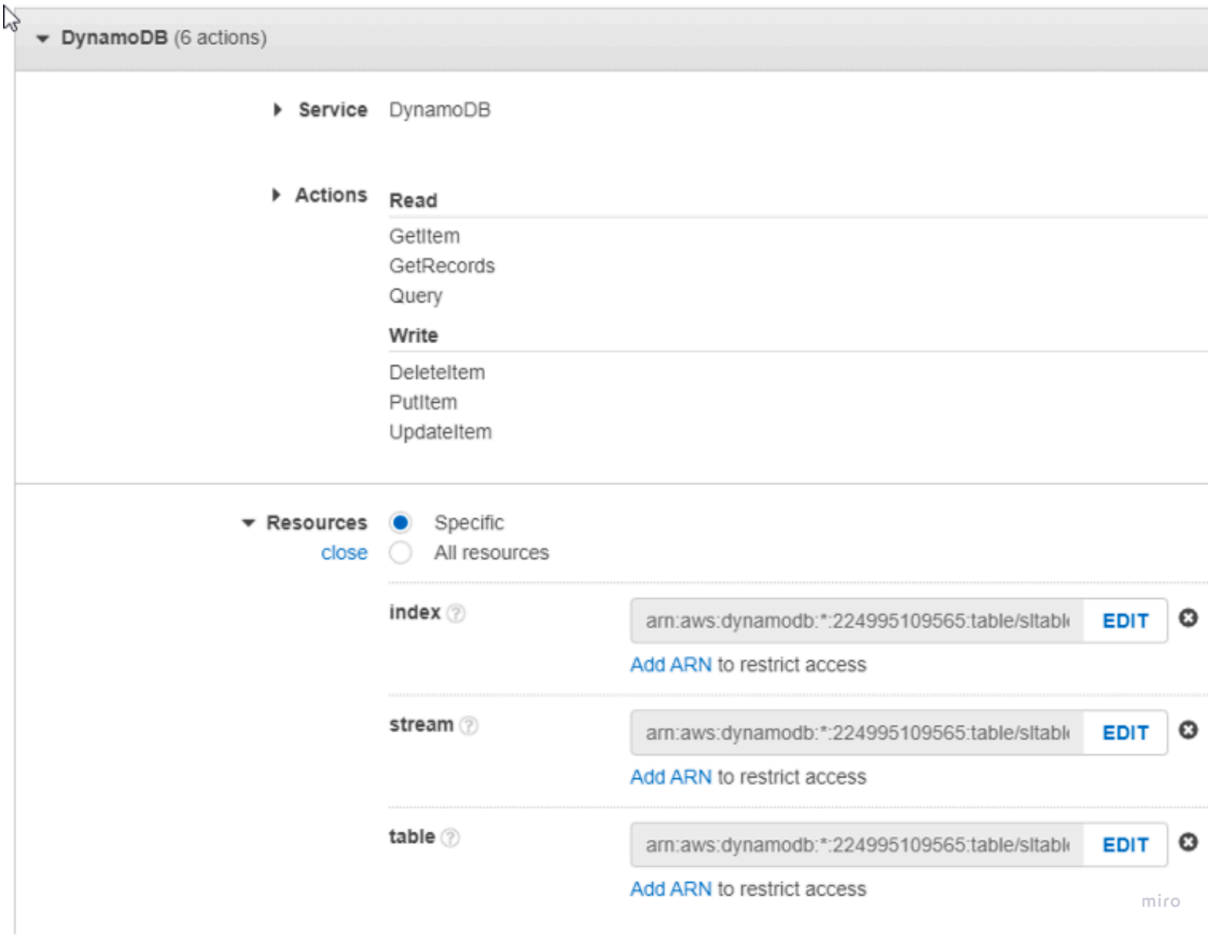


Fig: create a new policy

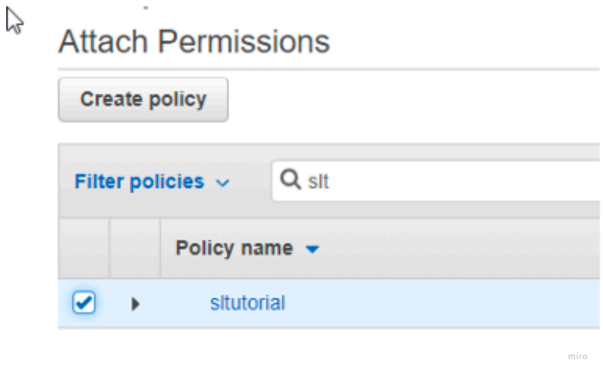
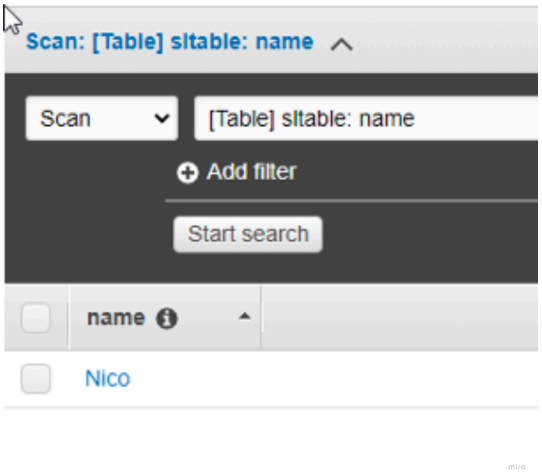


Fig: Add the policy to the role of our function

⇒ **Test.** When calling the website and clicking the link, the parameter should be added to our dynamoDB like this:



Messaging System

Messaging systems can be used for loose coupling. Instead of directly calling service B from service A, we send an event or a message to a queue. Another service is listening to the queue and after a new message is placed, the service starts to run.

By doing that, not only one service could listen to the message, a new service can also listen to it and thus starts a parallel stream.

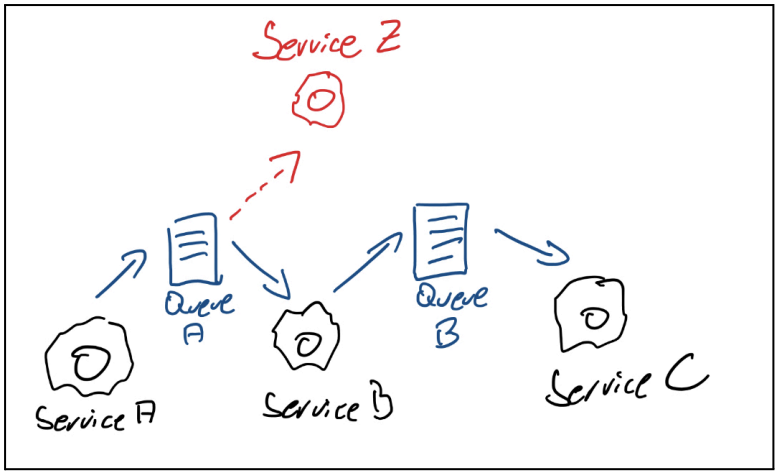


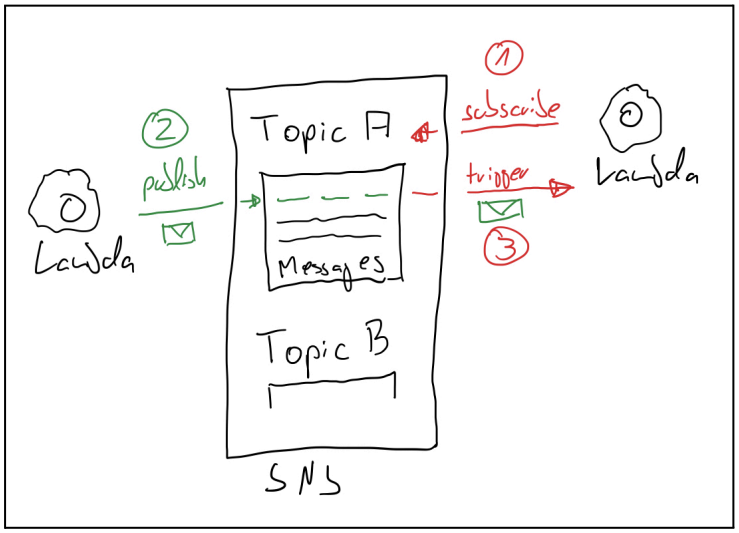
Fig.: Loose coupling with messages

Notification

One way to deal with messages is to send notifications. Here we just send a notification to a queue (as shown in the figure above).

Hands on: Simple Notification

To send a message we use the Simple Notification Service (SNS).



In SNS we can create topics, that's similar to a queue. After creating a topic you can subscribe to that topic with different services. In the figure above a Lambda function subscribes to topic A. Whenever a message is published to the topic, every subscriber will receive a notification with the message.

⇒ **Create a new topic in SNS**. As a subscriber use your email address or your phone number. Instead of a function, we subscribe with an email address or SMS to the topic.

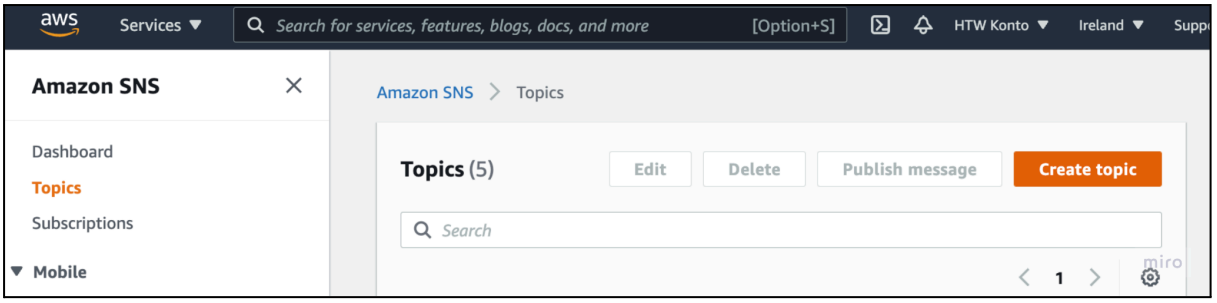


Fig. create a new SNS topic

⇒ **Write a lambda function and publish something to your topic**.

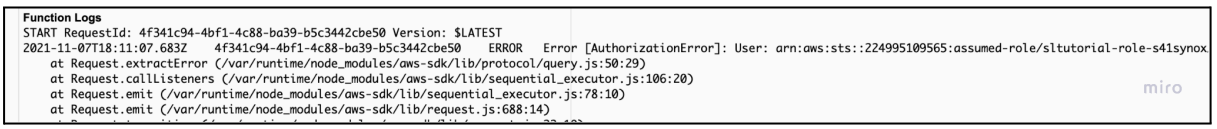
You need to import boto3:

```
import boto3
```

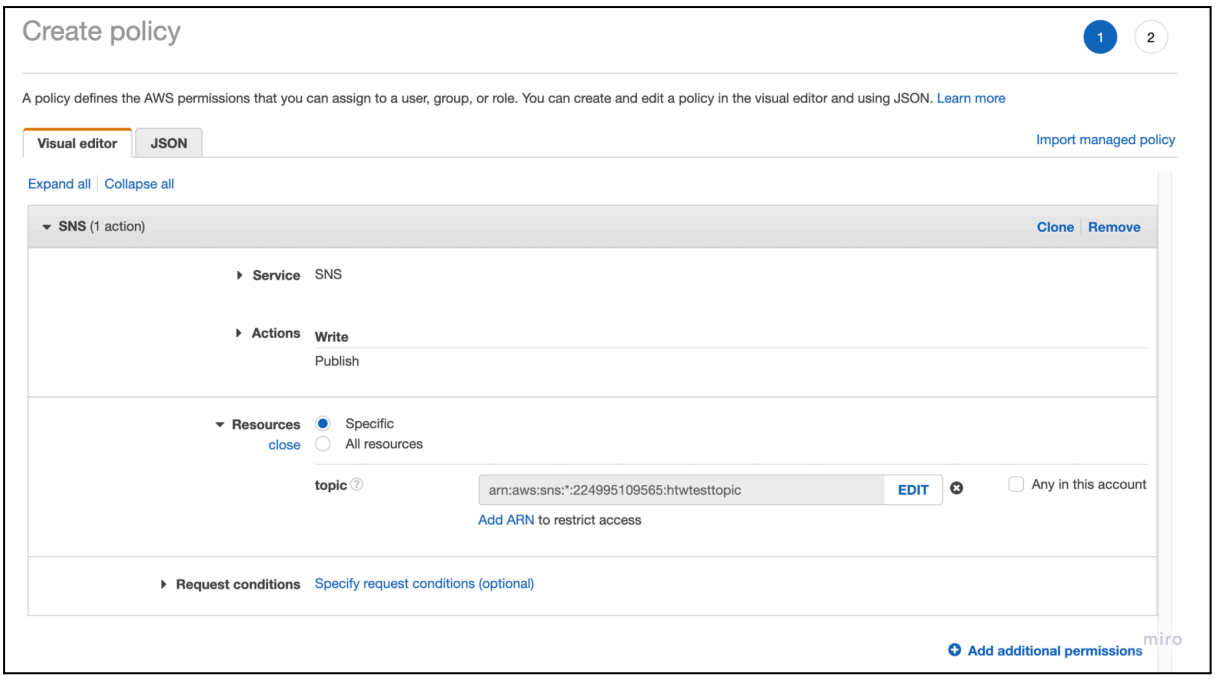
The publishing could be done by these lines of code:

```
// write to an sns topic

// Create publish parameters
   sns_client = boto3.client('sns')

   message = "Your message here"
   topic_arn = "arn:aws:sns:eu-west-1:559050216210:nicosnsv4"

   response = sns_client.publish(
       TopicArn=topic_arn,
       Message=message
   )
// --
```

⇒ **Test everything.** You'll find out that permissions are missing. You can use the logs (in your lambda) for this.



You need to add the policy to send messages to a SNS topic in your lambda role. Therefore find the role of the lambda function. Afterwards go to IAM and edit the role by adding the following policy:



⇒ **Call the function and see if you receive an email / sms**.

⇒ **Write a new Lambda Function which receives the message**
When a lambda function receives a message, it's working as a trigger to the function. The message of the topic is sent to the function and could be read.

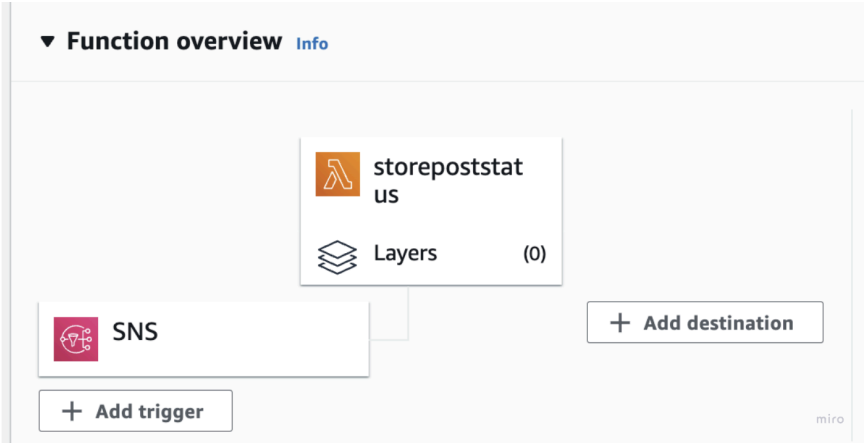


Fig: SNS as trigger for Lambda

Following lines could be used to receive a message:

```
import json

def lambda_handler(event, context):
    # TODO implement

    msg = event['Records'][0]['Sns']['Message']

    print(msg)
    print(json.dumps(event))
```

```
...
```

Fig.: source code for receiving messages

⇒ Separate the lambda function in smaller pieces and draw the infrastructure

That how your infrastructure could look like:

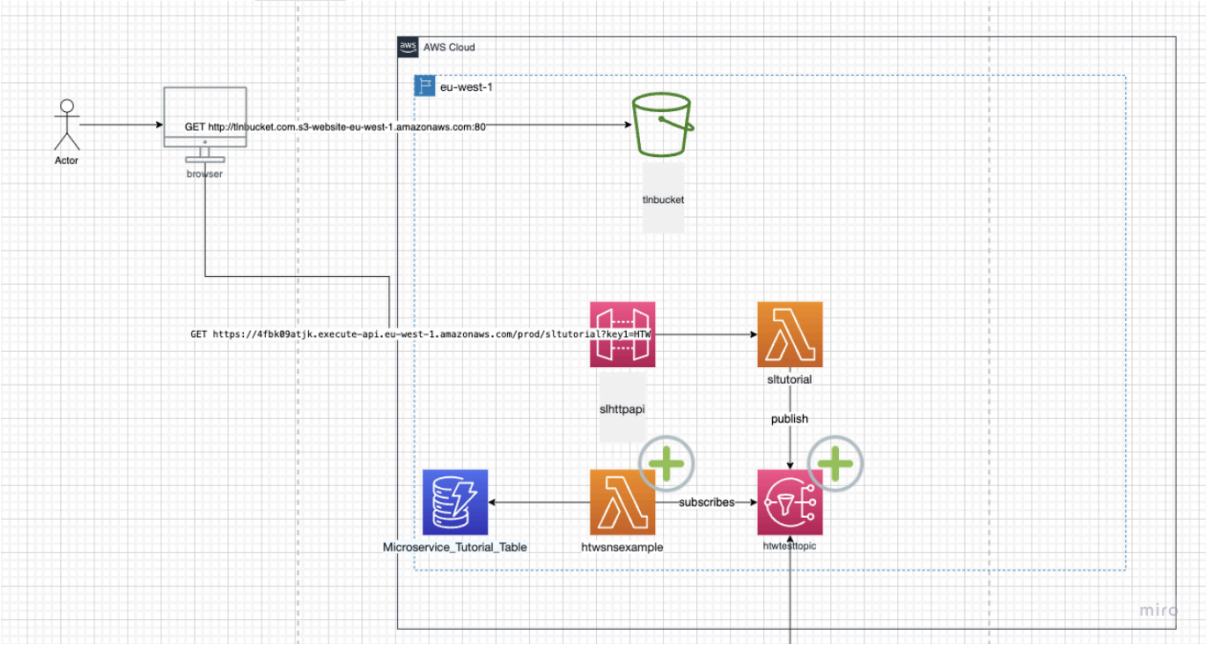
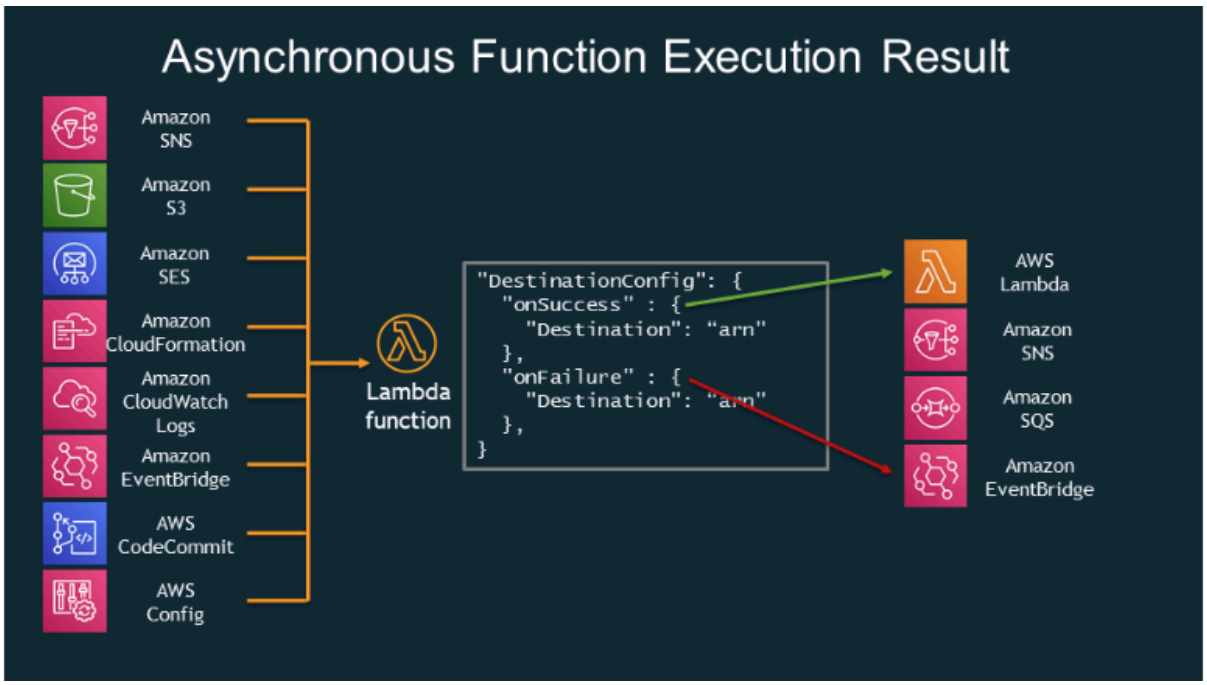


Fig. your current infrastructure

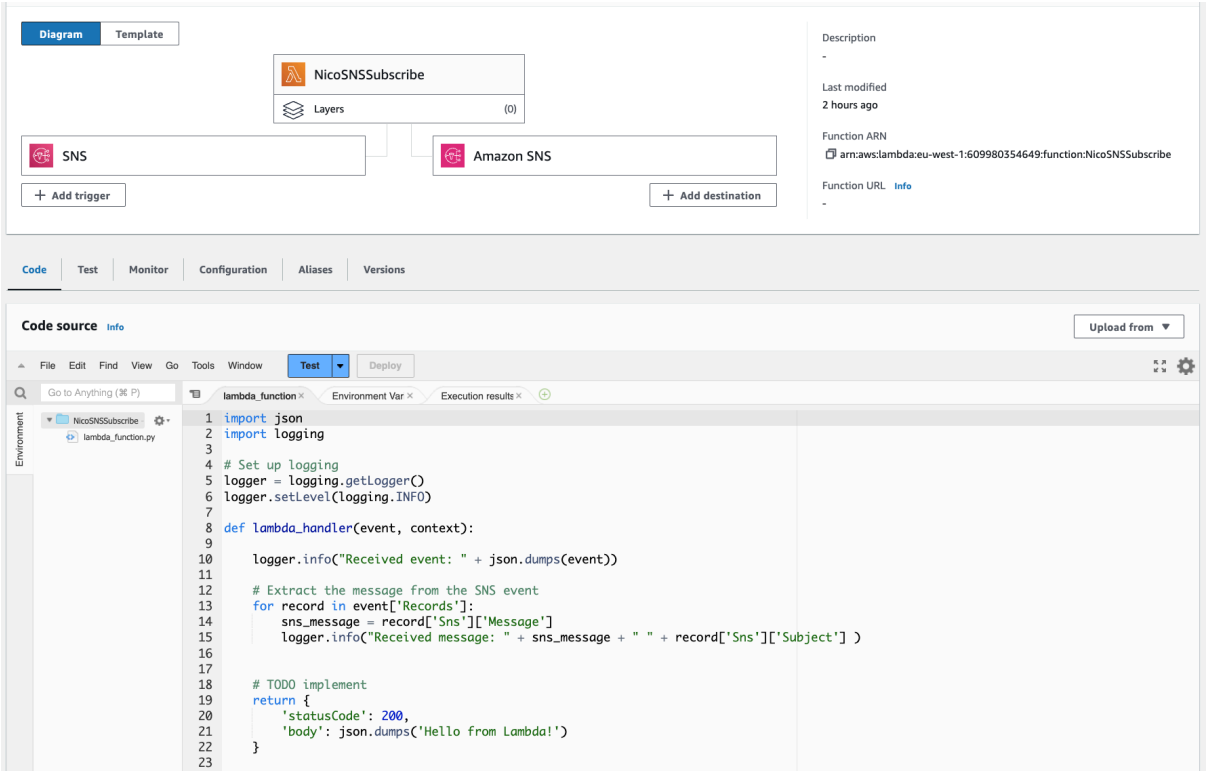
Asynchronous Calls

A lambda function can be called asynchronously. Only if the function is called asynchronously, a destination can be triggered. Thus destinations should be used only if you know exactly from where your function is triggered (see figure below).



[23] Async calls

The following function has SNS as a trigger and as a destination. If you use the TEST or a function url, nothing will be published to the sns topic. Only if the function is called throw the SNS trigger, something will be published in the SNS topic.



[23] https://aws.amazon.com/de/blogs/compute/introducing-aws-lambda-destinations/

More Details

https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html

Queuing

Instead of sending a message directly to a lambda, you can also poll for a message. Therefore the message is sent to SQS (simple queueing service) and a lambda explicit polls for it. The advantage is, that even if the service is down, the message is available to the consumer afterwards. There are different queue types which could be found here: Amazon SQS queue types

The following picture shows a good example of how SNS and SQS can be used together. A SNS topic is sending a message directly in a SQS queue. Here a service (realized on a server in EC2) is consuming the message (poll) when the service (or better server) is up.

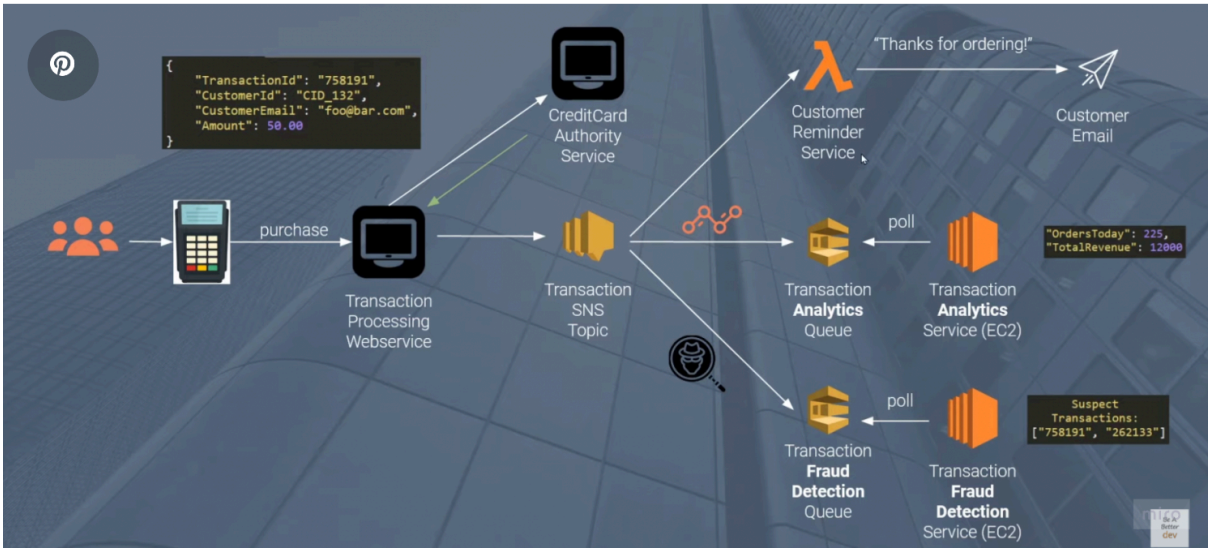


Fig: SQS and SNS[24]

On-Prem vs. Cloud

Cloud Computing

The National Institute of Standards and Technology (NIST) defines cloud computing as "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

This cloud model is composed of **five essential characteristics**, **three service models**, and **four deployment models**."[25]

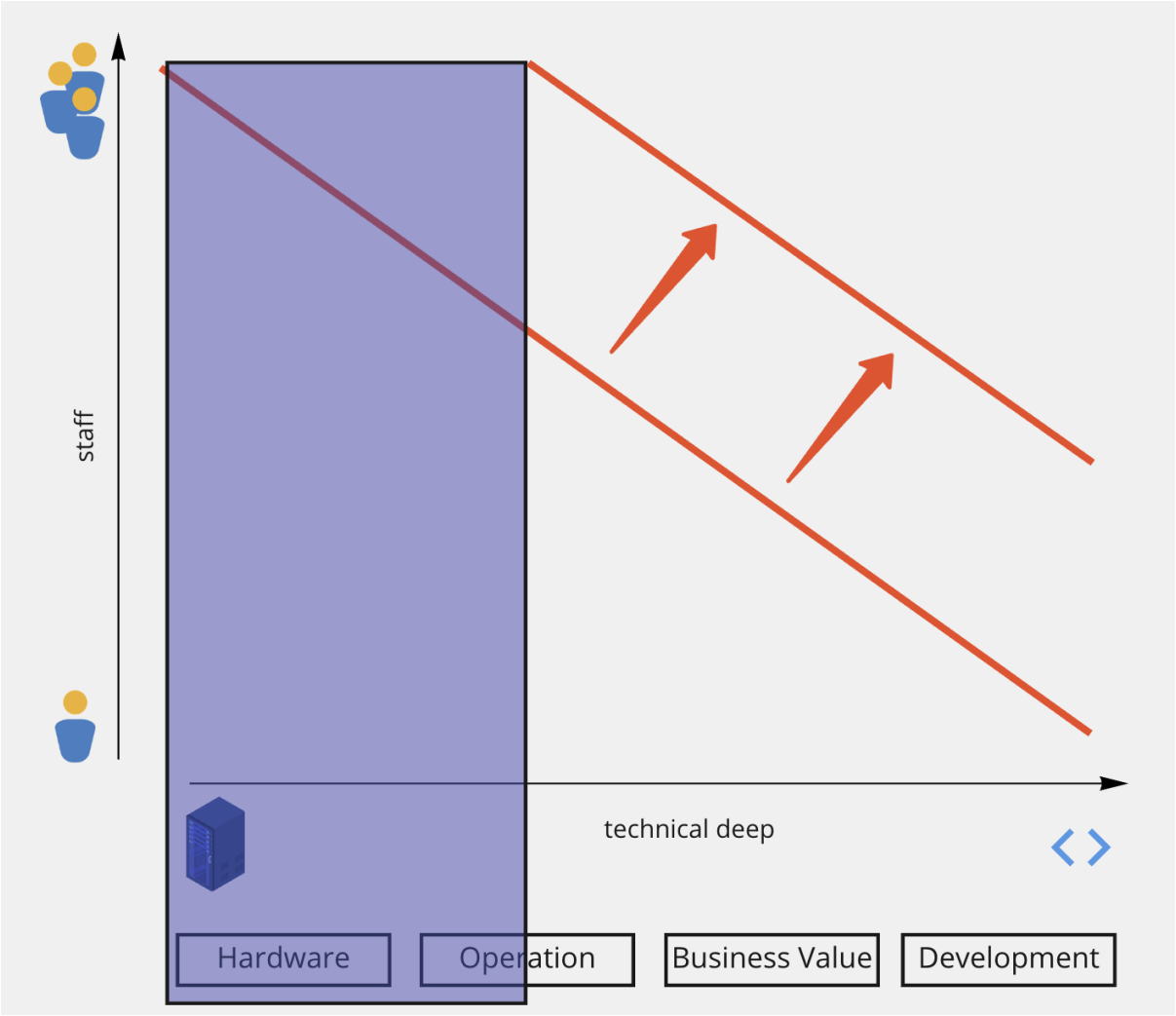
[24] https://www.beabetterdev.com/2021/08/08/aws-sns-vs-sqs-whats-the-difference/
[25] https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

That’s are a lot of things, but let’s concentrate on configurable computing resources. This is part of one of the service models: Infrastructure as a Service (IaaS).

Infrastructure as a Service means that you have the whole infrastructure as a service. No bare metal for instance. The definitions also states, that everything should be “provisioned and released with minimal management effort”.

By doing that we archive one really important thing: we can quickly set up an infrastructure and we don’t need a large knowledgebase.

The following figure shows us how we can concentrate on business value and development when using infrastructure (blue rectangle) as a service.



Cloud Computing Compliance Controls Catalogue (C5)

C5 is a set of regulations and guidelines developed by the German Federal Office for Information Security (BSI).[26] The C5 framework is designed to provide an audit standard for cloud service providers, ensuring a high level of security and compliance.

Here are some key points of the C5 regulations:

1. Basic Requirements: Cloud providers need to demonstrate that they have a minimum baseline of security measures in place. This includes aspects like physical security, network security, system security, and data protection.

2. Transparency: The C5 framework puts a strong emphasis on transparency. Cloud providers must offer detailed insights into their security and data protection measures, allowing customers to make informed decisions.

3. Redundancy: Service providers need to ensure that their services are reliable, which means having backup and redundancy measures in place.

4. Data Protection: Providers are expected to align their data protection measures with the European General Data Protection Regulation (GDPR).

5. Incident Reporting: In case of security incidents, cloud providers are required to have processes in place to report these incidents both internally and to their customers.

6. Continuous Improvement: Continuous monitoring and improvement of security measures are essential. This includes regular security assessments, penetration tests, and vulnerability management.

7. Auditability: The measures taken by cloud providers must be auditable. This means that third parties should be able to verify and validate the security measures implemented.

ISO 27001

ISO27001 provides a systematic approach to managing sensitive company information so that it remains secure. It encompasses people, processes, and IT systems. Organizations that conform to this standard can achieve certification to showcase their commitment to information security.

1. Risk Assessment: The Company regularly checks its systems to spot any weak points. They identify potential threats and figure out the possible consequences.

2. Policies and Procedures: The Company has set clear security rules and steps that all employees are aware of. This includes guidelines like password rules, access permissions, and what steps to take during a mishap.

---

26 https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/CloudComputing/ComplianceControlsCatalogue-Cloud_Computing-C5.pdf?__blob=publicationFile&v=3

3. Employee Training: The Company ensures its staff gets frequent training on online safety. This keeps everyone up-to-date about new threats and ways to dodge them.

4. Physical Security: The Company has amped up on-site security. They use tools like fingerprint scans for key areas and cameras to monitor the premises.

5. Incident Response Plan: If an issue arises, they have a ready-to-go plan. This helps them respond swiftly, reduce damage, and learn from the event.

6. Continuous Monitoring: The Company maintains a dedicated team to watch over its systems around the clock. This lets them detect and deal with any odd activity right away.

7. Regular Audits: Each year, an external firm verifies how well The Company adheres to ISO27001 guidelines.

Beyond the MVP - production as a challenge

We heard about some frameworks up to this point, but why is that important? Therefore first let's have a look at how a product can be successful.

Factors of success

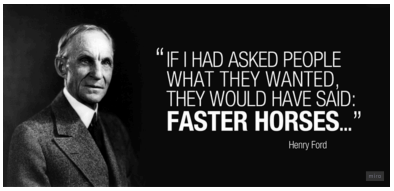
There are some things that make a product successful. That are for example:

- fast innovations and minimum time to market
  We need to be really fast in innovations to be one of the firsts who bring an idea into the market. Think about the first IPhone for example.
- maximum customer satisfaction
  We need to meet the customers' needs. When the customer is using our product, he or she should be happy. Imagine a good restaurant. If everything, i.e. the meal, the service, even the restrooms are perfect, you'll come back.
- maximum stability
  Even if the product is pretty cool, it needs to work stable, e.g. an app should crash.
- maximum individuality
  The product shall be differ from the product of our competitor.

This leads to three interesting statements:

1 be fast

2 have ideas

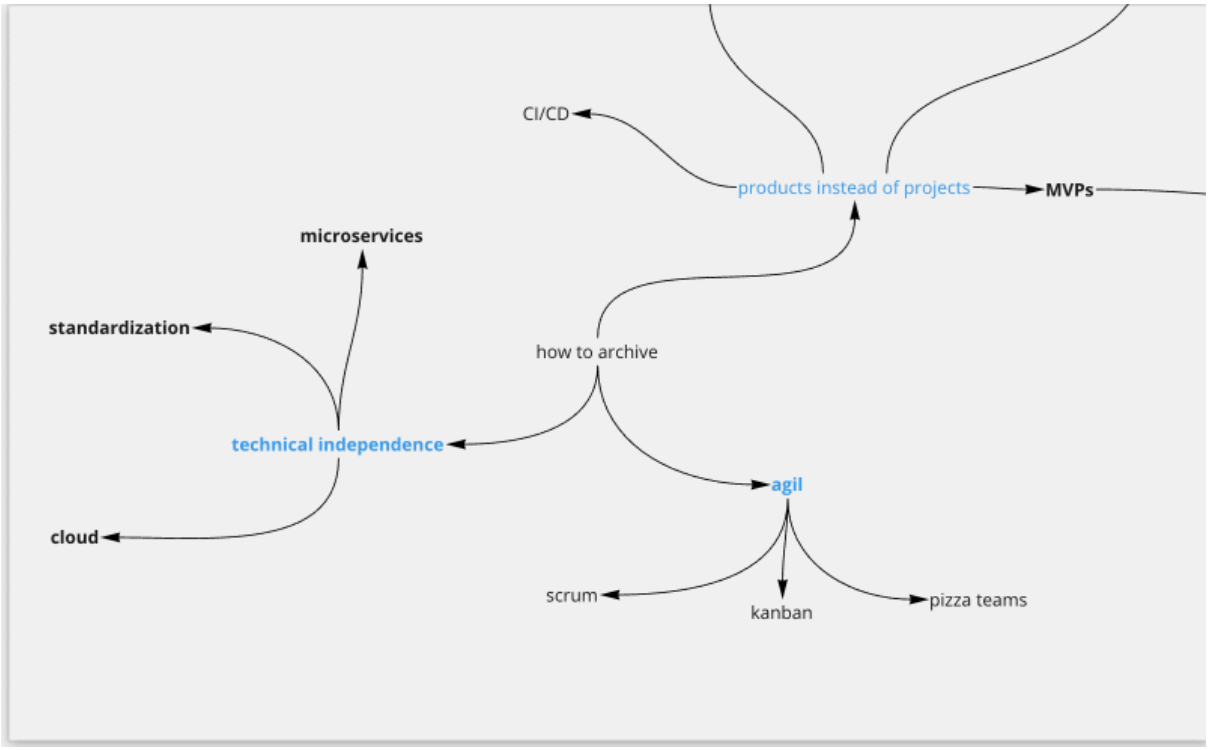


3 put the customer first

On the other hand, we are facing some restrictions. That are for example:

- lack of resources
  To be fast and to develop everything stable and brilliant at the same time, we need human power. Think about the magic triangle with cost - quality - time. When we are focusing on quality and time and don't care about costs, we still need to be able to pay somebody for working four our product. But these days there is a lack of human resources.
- don't know how the user is interacting
  We are guessing how users are interacting but we don't know for sure. We don't know which feature is relevant and if e.g. a user is using our product for the purpose and in the way it's designed for.
- market pressure
  We are not the only ones in the market. Other competitors are also fast. At the same time we have a pressure on costs.
- governance and data security
  Even if we are really fast and get everything done perfectly well, we are facing company governance and security topics, e.g. the GDPR compliance[27]

Ingredients for success



[27] https://gdpr.eu/

Projects

Possible Projects

- Cross Region VPC
- Auto Backup and Reliability of Servers
- Image (Screenshot) to Text converter for source code
- SQS / SNS Examples incl. Direct calls via API and Step function and time measurement