# 

# 

# 

PROJECT REPORT

CCBDA - Spring 2022

# 

# 

# 

Aleix Boné Ribó

Alba Cañete Garrucho

Riccardo Cecco

Marc Manzanares Díez

Oriol Martínez Acón

**Professor:** Angel Toribio-González

# Index

[**Index**](#_f8u78pg135y1) **2**

[**Introduction**](#_fr19d63xcpgq) **3**

[**Development methodology**](#_60fiejtzzzqi) **4**

[**Application functionalities**](#_hxz9c6a5v3u) **4**

[Login](#_v8xmjhein4ze) 4

[Register](#_q7s89r28t9z) 5

[Dashboard](#_igu9ous4qulc) 6

[Request](#_sdmm4o6rl3vt) 6

[History](#_lybd9ivu585e) 7

[**ML Model**](#_156rwaeiuzfe) **7**

[**Infrastructure/Devops**](#_w1mi8sfzl7h3) **10**

[Dockerizing the application](#_u0wrr1tfd9c6) 10

[CI/CD Pipeline](#_bf0wwlu8fxya) 10

[AWS Cloudformation](#_tnwa6oi95wok) 11

[Agent image and docker-compose](#_8armgtyn5845) 11

[Configuration](#_ss7ub1qeamu3) 11

[AWS Architecture](#_857zu2559h9p) 15

[AWS ECR](#_dquafmivk4u) 16

[AWS ECS](#_k4q4llkcoctv) 17

[Cluster](#_w793biall9gh) 17

[Task definition](#_w5wrhrmsplax) 18

[Load balancer](#_barj29k4d17p) 19

[Service](#_x4d9c1fr4ocx) 20

[AWS CloudWatch](#_vsjynxvz0ct8) 24

[**Future work**](#_au63zlp7tj28) **25**

[**Conclusions**](#_crrkwk5hhz4o) **25**

# Introduction

In this report we present our final project of the CCBDA subject. Our idea is to scrap data from <https://www.transfermarkt.com/>, which is a web where you can access information on the football players’ value in the market.

The idea is to take different types of data. First of all, we have considered only the 5 most important European’s championships from 2010 to 2022, which are:

1. Serie A
2. La liga
3. Ligue 1
4. Premier League
5. Bundesliga

For each championship, we have taken into account this data:

* Name of the Team
* Average age of the players
* Value of the team
* Number of players.

Example:

{"squad\_name": "FC Barcelona", "avg\_age": 25.7, "squad\_value": 679000000, "year": 2021, "number\_players": 30, "championship": "primera-division"}

We have also scrapped each player inside a team by collecting this information:

* Name of the player
* Age
* Role
* Value of the player
* Name of the team
* Year
* Games played
* Minutes played
* Goals
* Assists.

Example:

{"name\_player": "Lionel Messi", "age": 33, "role": "Right Winger", "value\_player": 80000000, "squad\_name": "FC Barcelona", "year": 2020, "games\_played": 47, "goals": 38, "assists": 14, "minute\_played": 4192, "championship": "primera-division"}

If the player is a goalkeeper, the stats are different. Instead of taking goals and assists, we take:

* Goals conceded
* Clean sheets

Example

{"name\_player": "Marc-Andre ter Stegen", "age": 26, "role": "Goalkeeper", "value\_player": 90000000, "squad\_name": "FC Barcelona", "year": 2018, "games\_played": 49, "goals\_conceded": 43, "clean\_sheets": 23, "minute\_played": 4410, "championship": "primera-division"}e

At the end of the process, we have collected 43295 different data that we have had to preprocess in order to clean it before using it.

Our project is composed of a frontend and a backend, a machine learning (ML) model, and an infrastructure/devops part.

As for the frontend and the backend, our idea was to create different plots by aggregating the data that we have collected, and to create a ML model which would allow us to predict the value of a player in the future.

For the frontend part, we have used a Bootstrap template where an user can sign up/in and create personalized plots.

We offer the possibility to choose the team or the player to show different statistics such as the value in the future or numbers of goals in the last 12 years, for example.

As for Infrastructure/devops, we divided it in two parts. The first part is what we could call the Internal services, and it consists on the deployment of Team City in an AWS environment and its proper configuration that allows us to have a CI/CD pipeline.

The second part is the infrastructure where the application runs in production, for which we used AWS ECS, the containers service of AWS, and autoscaling groups. Cloudwatch is used to monitor the resources the application is using and to decide whether to scale the infrastructure.

# 

# Development methodology

The methodology used to develop the application has been the Agile methodology. Initially, we held several meetings to develop the idea together, from the choice of data to the goal and structure and, given the limited time we had, we divided the work according to our prior knowledge.

We divided our project into smaller tasks, and step by step, each of us would state what task we were working on, what the current status of the work was, and the problems we encountered along the way.

All the process was documented using Trello, where we created a Dashboard for every team: Frontend, Backend and Infrastructure, used to divide and assign tasks.

For each task, one person was in charge of developing the idea, testing it, and in case of any problems, asking someone else in the group for help so that they could work in pairs without getting stuck.

In addition to the weekly meeting done with the professor, we had several meetings with the whole team to understand the status of the project development, so as to keep track of all the various deadlines we had set.

In general, we managed the work and time very well considering the effort required by other subjects as well.

# Application functionalities

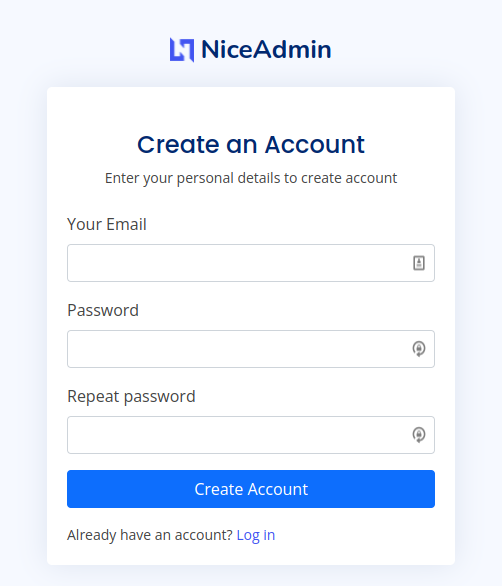
## Login

We propose a classic login interface. Every user has to login using the email and the password. Passwords are encrypted inside our db.

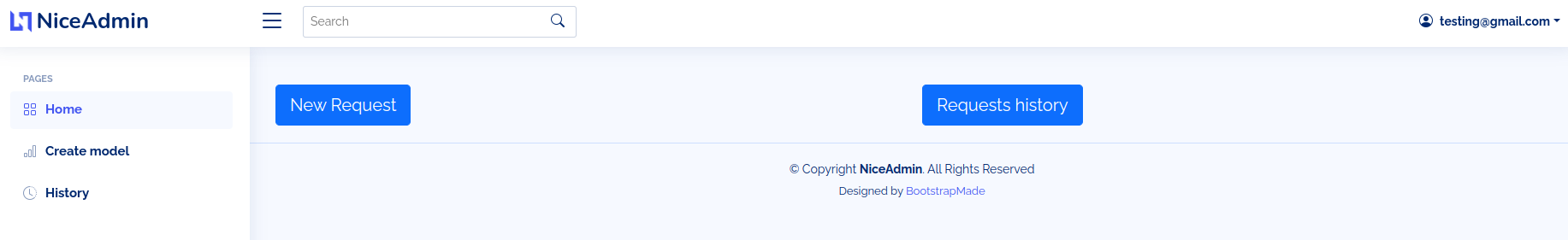
# 

## Register

We propose a classic sign up interface. Every user has to sign up using the email and the password. Passwords are encrypted inside our db.

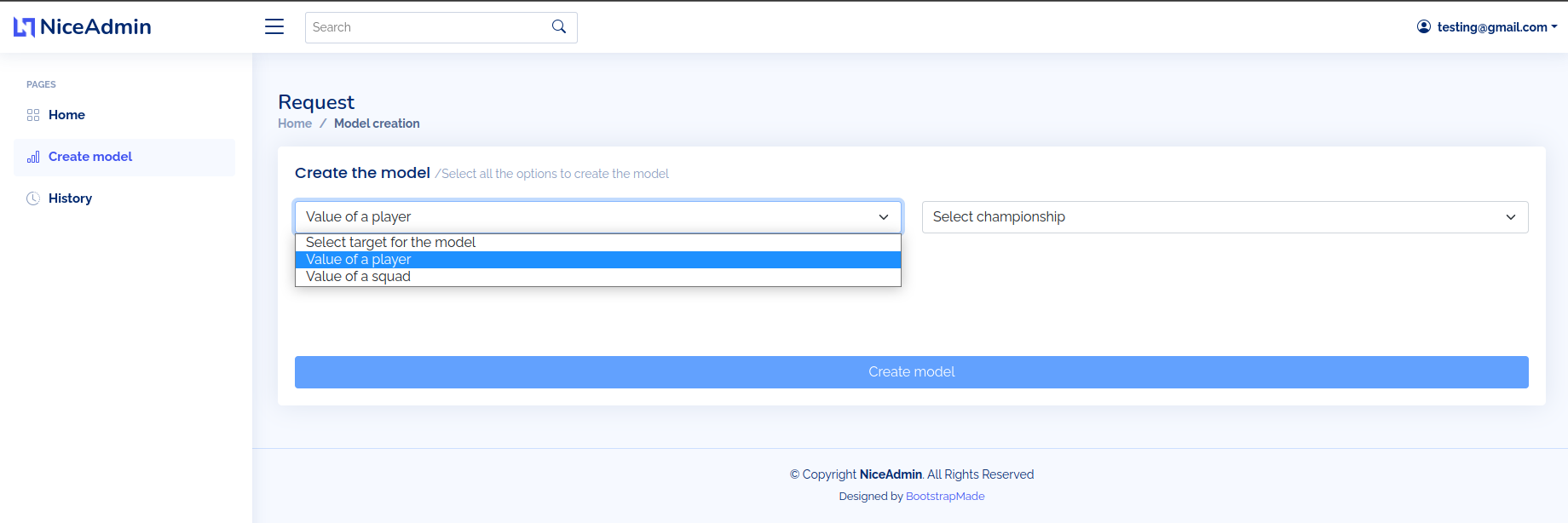


## Dashboard

The image below is showing us our dashboard. We have a sidebar on the left, from which we can access the Create model and the history functionalities.

## Request

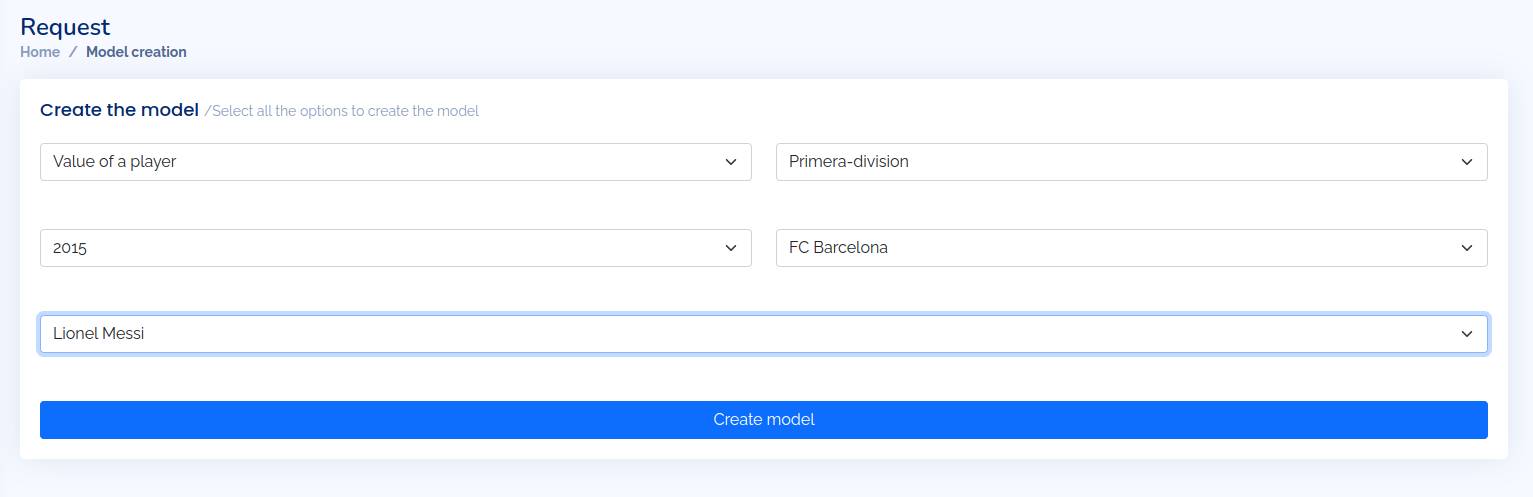
Creating a query proposes us two different searches: player value and team value.



Step by step, the GUI will ask us this information to find the player we are looking for.

Starting from the top, we are asked what league he plays in, what year, what teams, and finally his name.

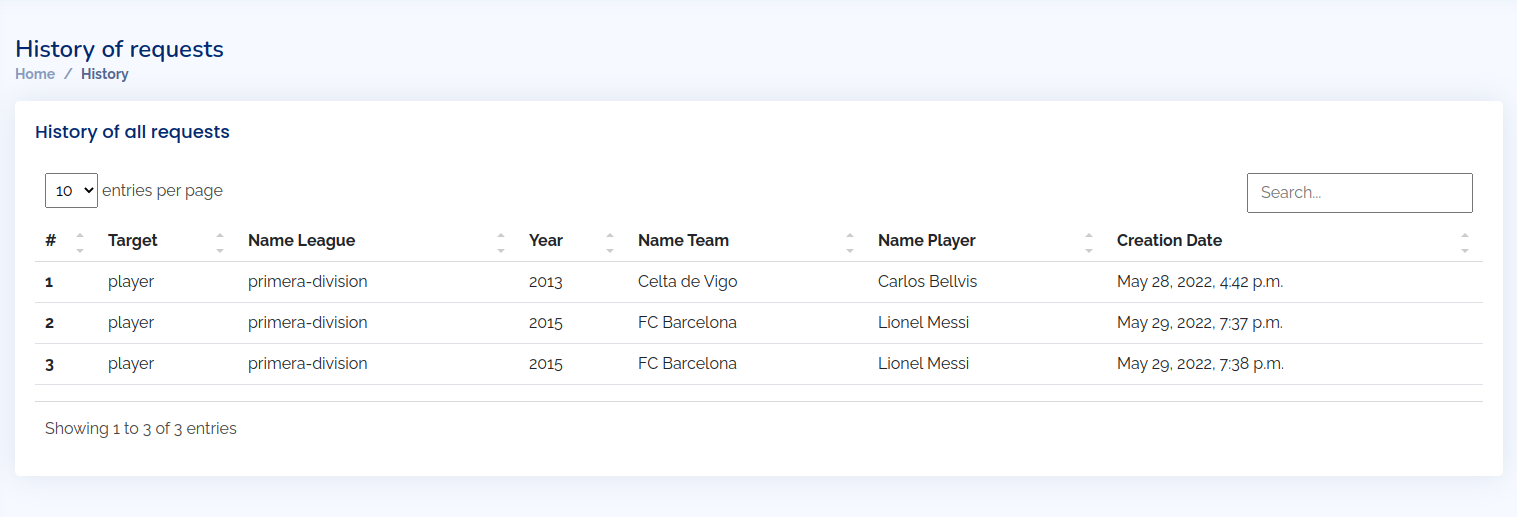
The different drop-down menus will populate based on the choice made in the previous drop-down menu.



After clicking on create model, the GUI will show us a graph with the value of the player from 2010 to 2022, as we can see in the image below.

## 

## History

For every user, we save the history of searches that this user has created, so that through their history page they can more easily access an old search.

# ML Model

Once we had all the relevant data, we wanted to create some models upon it in order to generate predictions on future player statistics. We started by using AWS sagemaker AutoPilot feature to just run various models on the data without any kind of preprocessing.

This was great to get a quick idea of how the data behaved and what could we do with it. After running 250 different models through AWS sagemaker autopilot, the best model we obainted was using XGBoost with a R2 of 0.65, when using the value of a player as the target variable. This meant that our model could explain around 65% of the variability of the data. In the image below, we can see an overview of the XGBoost model pipeline that aws used for our dataset, we can see that it performs a column transformation for the different kinds of variables we had, for the player names, since they were a lot, it did a CountVectorizer (Converting text into columns that model can understand), for categories it performed OneHotEncoding (again, a method that converts each category value into a separate boolean column) and for the numeric variables it imputed the missing values and scaled them with a standard scaler. Finally, it runs a GridSearch on the XGBoost regressor with different parameters in order to find the optimal ones.

# 

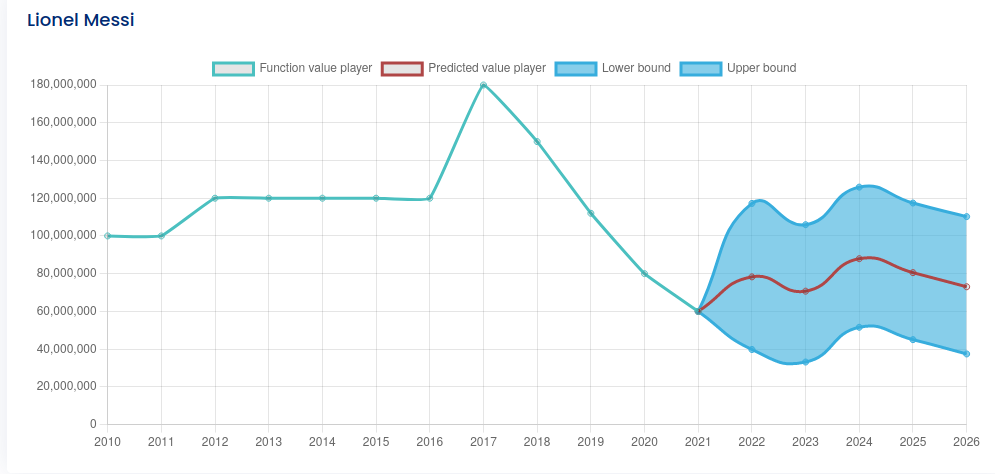
However, due to the nature of Autopilot, it was working with the data without taking into consideration that it was a time series dataset, the models we can obtain from autopilot are not great for forecasting data into the future. With our model, we can predict the theoretical value of a player if he had different statistics, but it cannot give a prediction of how his goals or value will evolve over time.

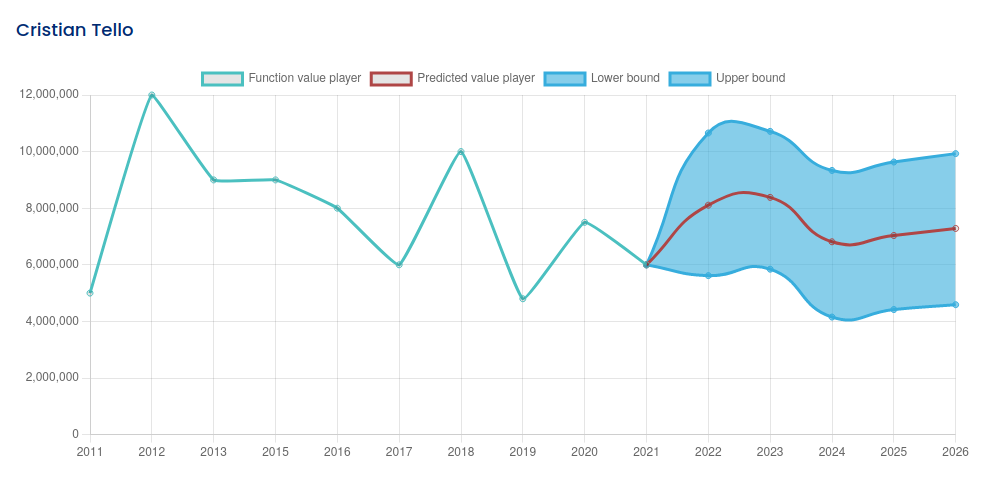
Sagemaker canvas, offers some tools to analyse and build time series models on a cloud editor[[1]](#footnote-1), and there are some prebuilt models that can be used for time series predictions such as churn analysis. We found that [AWS Forecast algorithms](https://docs.aws.amazon.com/forecast/latest/dg/aws-forecast-choosing-recipes.html), there are a couple of proprietary AWS algorithms for forecasting based on neural networks that would work, but they need hundreds of historical values, and we only have 10. Due to the limited time of the project and our budget, we could not really scrap data from more sources and obtain more data to build a robust time series based prediction for each player, with much more data collected, such a solution would be feasible.

Our main problem was that we did not have enough data to obtain proper results with a model. For each player we only had at most 10 data points, and we only had goals, assists, minutes and games played and goals they received if they were goalkeepers. With this little information, there is no magical solution that can give adequate results.

As such, we decided that we would have a general global model trained with all the data and compliment it with a smaller model that could be created on the fly inside our ec2 instance for each player that our users requested. We can then predict the number of goals, assists, minutes played… the next following season and feed this data to our model to give an estimate of a player's value.

To predict the statistics, we used [Prophet](https://facebook.github.io/prophet/), a forecasting procedure created by Facebook designed with the purpose of predicting trends in the stock market. This is suited for large historical data with lots of points, so the results will not be great either. When a user requests predictions on a player, we take the player data, build a small prophet module on the variables goals, games\_played… and predict their trend in the next 5 years. With prophet, we, this model takes into account the data of all 3 variables to predict their global trend. Once we have this forecast, we can send it to our actual model, which will give the player value estimates. We can then show the predictions to the user along with a confidence interval, as shown below:





The actual results vary greatly from player to player, the model only works somewhat properly for Forwards/Wingers, due to the fact that we have primarily goal and assists as metrics. With more columns such as success rate on tackles, passes, fouls… we would expect a much more accurate and reliable predictions. As it is now, the model is simply a proof of concept of what could be implemented with adequate training data.

# Infrastructure/Devops

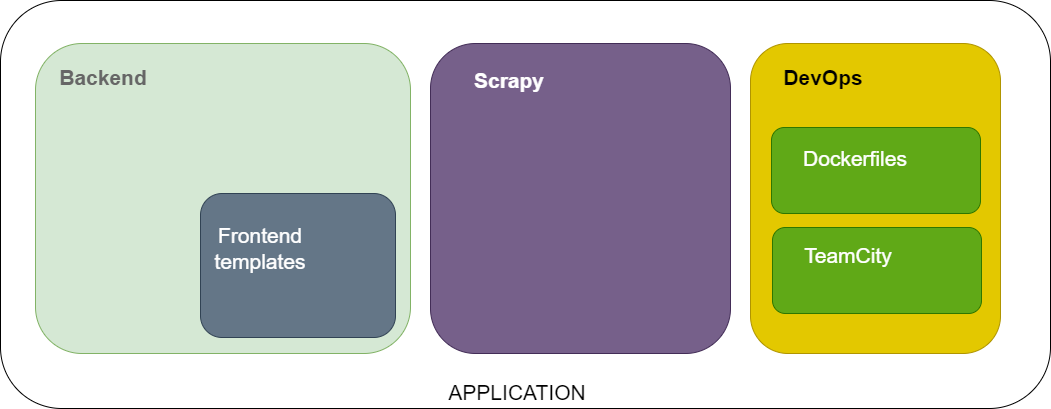
Because we are studying in a subject called “**Cloud Computing** and Big Data Analytics”, we thought that it would be interesting to go deeper into this field and try different things, so that we can use what we learned through the course and even increase our knowledge..

Therefore, we made an automated process where we dockerize the application, which is uploaded to the cloud through a CI/CD pipeline using a tool called TeamCity, and then this docker image is deployed into an entire background process of clusters, instances, and many other services.

## Dockerizing the application

As we have learned in the course, Docker is a software tool that allows us to make packages of applications, services, programming language packages... basically, whatever we want to, and store it inside a virtual container that can be executed anywhere and even used inside another docker container.

First of all, we will explain the structure of our application so that it is understood how the docker image will be done.



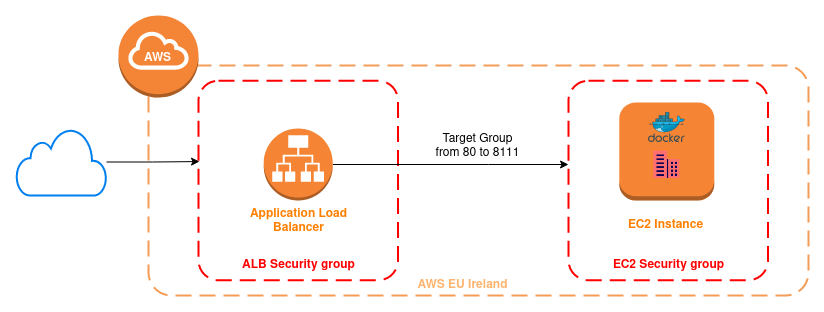
Because we have the templates inside the backend, we only made a single image of the “backend” directory. This docker image is created from this Dockerfile ([https://github.com/RiccardoCecco/Cloud-Computing-and-Big-Data/blob/master/backend/Dockerfile](https://github.com/albacg5/CCBDA-Project/blob/master/backend/Dockerfile)) which basically starts from a python template, configures some environment variables, copies and runs some configuration files and finally exposes the port where the application will run, 8000.

## CI/CD Pipeline

TeamCity (TC) is a build management and continuous integration server from JetBrains. We use it to automate the creation of the Docker images of the application.

### AWS Cloudformation

TC is deployed in AWS, in an EC2 instance, and can be accessed via a Load Balancer CNAME. The AWS Cloudformation full template can be found in [https://github.com RiccardoCecco/Cloud-Computing-and-Big-Data/blob/master/devops/cloudformation/CI-CD.yaml](https://github.com/RiccardoCecco/Cloud-Computing-and-Big-Data/blob/master/devops/teamcity/docker-compose.yaml)

****

### Agent image and docker-compose

We have dockerized the server, the agent and the postgreSQL database in order to add another layer of failure resistance. As for the server and the database, the official images provided by Jetbrains and postgres, respectively, already have all the packages that we need.

The Docker image used for the TeamCity agent does not have Docker installed by default, so we need to create our own image based on it. The dockerfile can be found here: <https://github.com/RiccardoCecco/Cloud-Computing-and-Big-Data/blob/master/devops/teamcity/Dockerfile>. As you can see, we install Docker above the Jetbrains image and configure the permissions.

Once all the images are ready, we specify all three services on a docker-compose YAML file (<https://github.com/RiccardoCecco/Cloud-Computing-and-Big-Data/blob/master/devops/teamcity/docker-compose.yaml>), so that deploying them is much easier. For each service, we specify:

* The image for each service.
* Container names.
* The ports that need to be exposed, which are 8111 for the server and 5432 for the DB.
* The volumes we want to keep for each. The agent mounts its properties file, where the URL of the server, needed to start, is written.
* The environment variables.

### Configuration

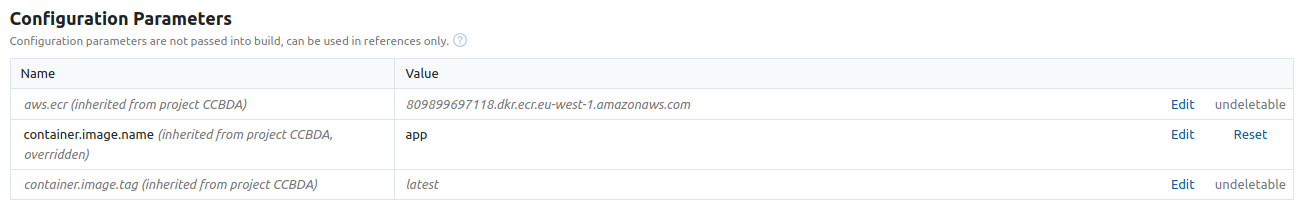
TeamCity lets you define different projects, where each project can have subprojects, Build configuration templates, and Build Configurations. For a company, a project can be defined for each application they develop, where they have a subproject for every environment (development, staging, production). A Build configuration template allows you to eliminate repetitive Build Configurations, and a Build configuration encompasses the build steps of a pipeline. Normally, a Build configuration is defined for every microservice of the application.

In our case, we define a project, called *CCBDA*, that will have one Build configuration, called *App*, with two build steps.

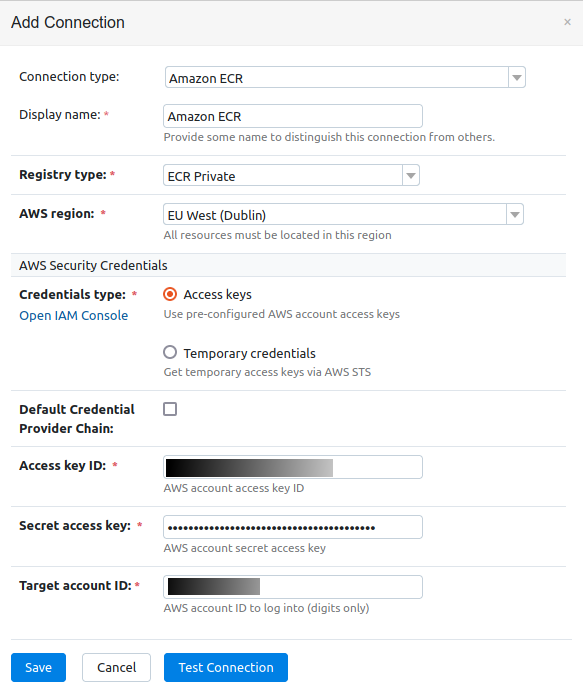
**Project: CCBDA**

Inside of a TC project, we define

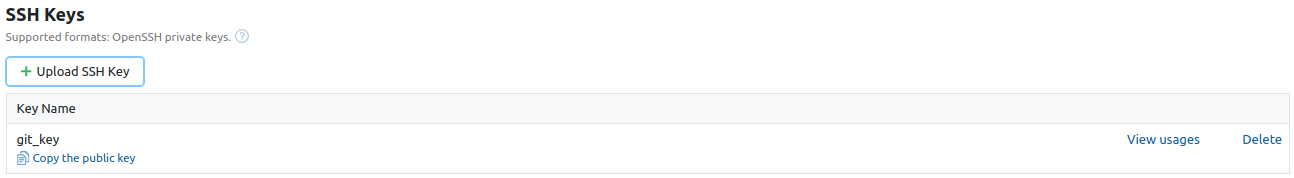
1. *Build configurations*. In this case, we have one, called *App.*
2. *Parameters*. TC lets you define parameters that are useful to not hardcore values and that are inherited by Subprojects and Build configurations. Thus, parameters can be created at Project level, and those that are global to a project can be defined at its level, and those more specific in the Subprojects or Build configurations. In our case, we define three parameters
   1. *aws.ecr.* URL of the AWS ECR Registry. It is common to all the microservices, so it is specified in the Project.
   2. *container.image.name.* Name of the Docker image. It is specific to each microservice, so it is left blank and specified in each Build Configuration.
   3. *container.image.tag*. Name of the Docker image tag. It is specific to each environment. Since we are only using one environment, we do not have Subprojects, so we define it in the Project level.



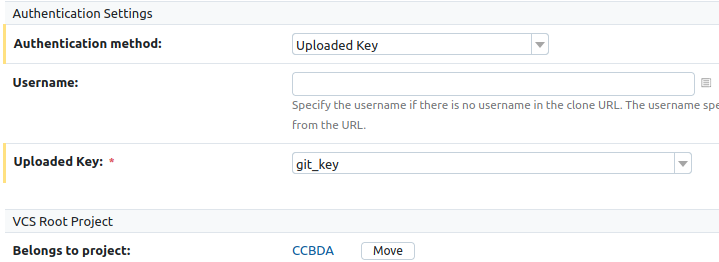
1. *Connections.* We define a connection to the AWS ECR Registry. In order for it to work, we create an IAM User that has only access to AWS ECR and specify the credentials in TC.



1. *SSH Keys*. Since the code is in a private repository, we need to create a new SSH key (*ssh-keygen -t ed25519 -C* [*test@example.com*](mailto:test@example.com)) that TeamCity will use to authenticate, and add it to Github and TC.



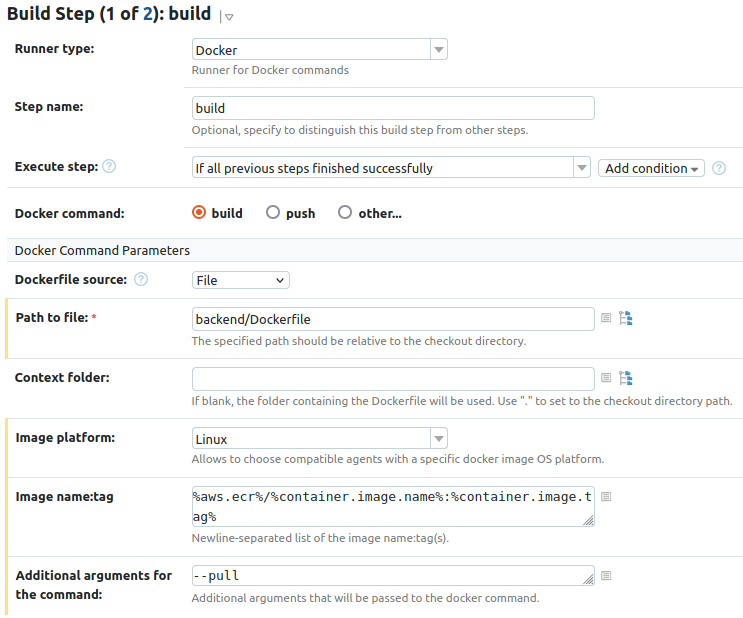
1. *VCS Roots.* We create a VCS Root for our Github repository project, using its SSH URL:<git@github.com:RiccardoCecco/Cloud-Computing-and-Big-Data/CCBDA-Project.git>.



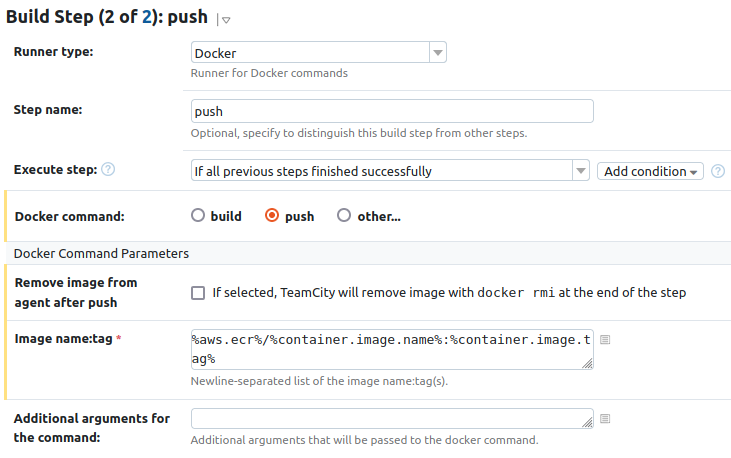
**Build configurations**

A TC Build configuration lets you define the steps of the pipeline of a microservice. We define

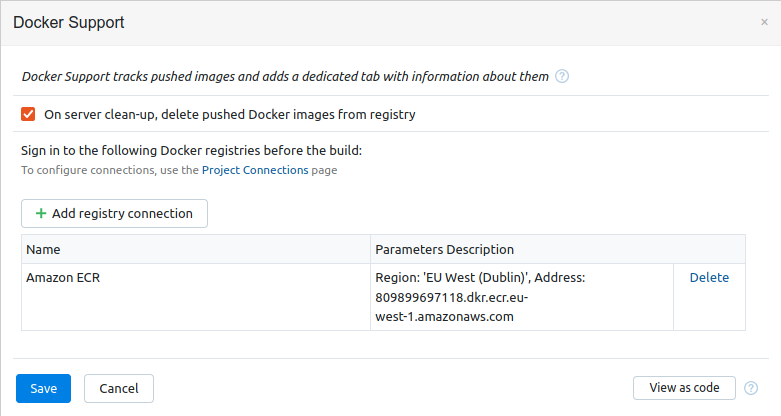
1. *Build steps.* When creating a build configuration, it is important to configure the sequence of build steps to be executed. In our case, we have two:
   1. *build.* This step takes the Dockerfile in the *backend/* folder inside the VCS Root and builds a Docker image with name *aws.ecr/container.image.name:container.image.tag*.

**

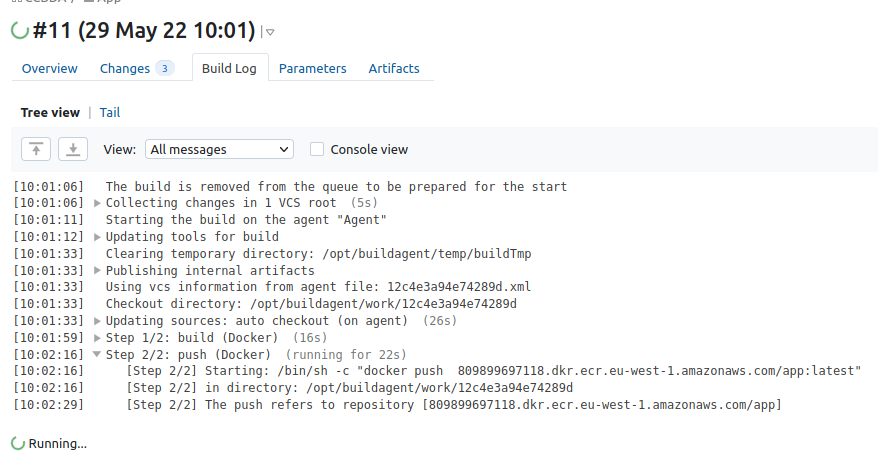
* 1. *push.* This step takes the Docker image that was created in the previous step and pushes it to the AWS ECR registry.

**

1. *Build features.* A build feature is a piece of functionality that can be added to a build configuration to affect running builds or reporting build results. In our case, we create a Build feature of type Docker support, that lets us choose the AWS ECR Registry created in the Project and that will let the *docker push*  step of the build successfully authenticate to the registry.

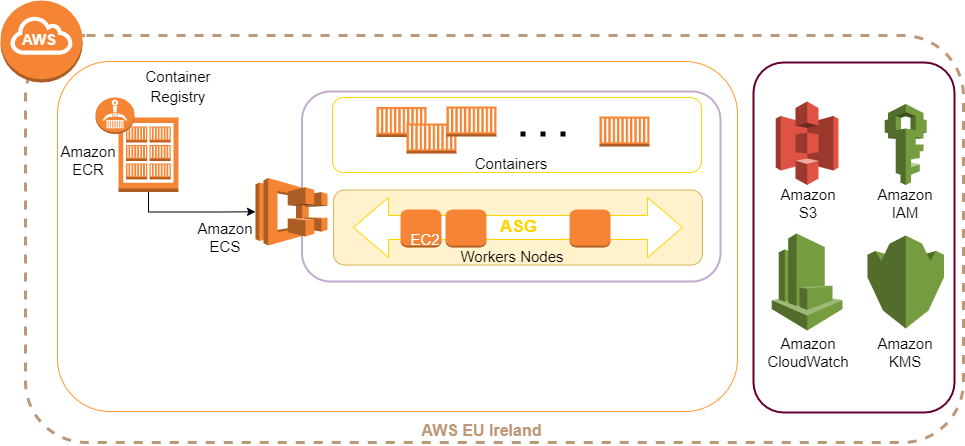


The following screenshot shows the log that can be followed when a configuration is being run.



## AWS Architecture

We already defined a first version of the AWS Architecture we will be implementing. We did some changes, and here is the final version of the architecture we are using on AWS.



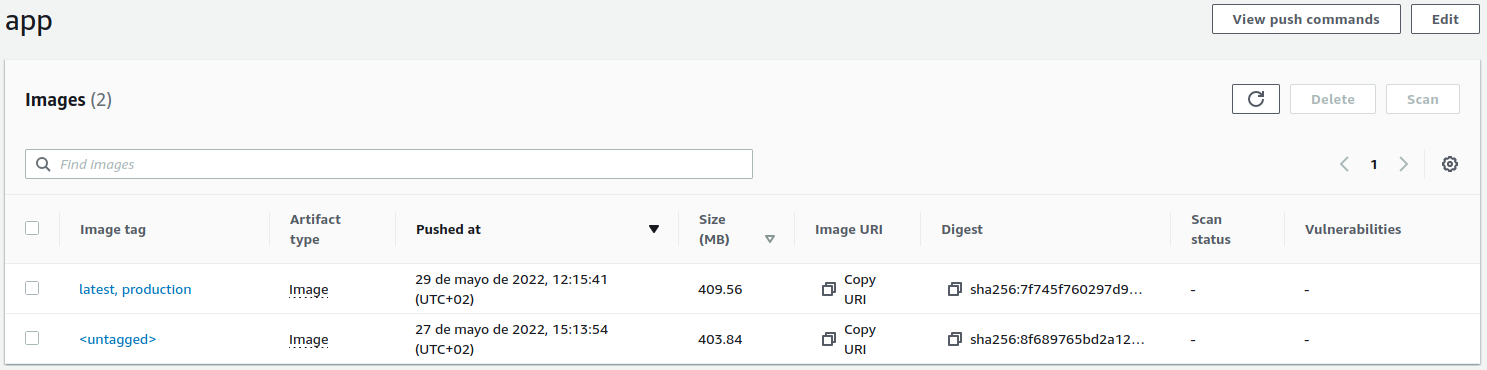
The idea is to have an Elastic Container Registry where we will push the docker images of our application. Then, using task definition in the Elastic Container Service, we will have a system of multiple EC2 instances running the docker image. These instances will be controlled by an Auto Scale Group and monitored by the CloudWatch service.

Also, we made a decision on removing DynamoDB service, because locally we have been working with a SQL database, and DynamoDB is a no relational database. The other option was to use RDS service, but we discarded that option due to the cost of the service. RDS is probably one of the most expensive in AWS. So in the end, we decided to simplify things using a sqlite database inside our application.

## AWS ECR

The AWS Elastic Container Registry is a service that lets us store Docker images in a private registry. In order to make it secure, we have created an IAM User that has only access to the ECR, and that we use to push images from Team City.

There is a repository for each image we create. The following screenshot shows the *app* repository



Every time an image with a tag is pushed, the latest version appears in the graphical interface. For every image, there can be different tags, but only the last version of each tag is kept. The image can be used by copying the URI provided.

## AWS ECS

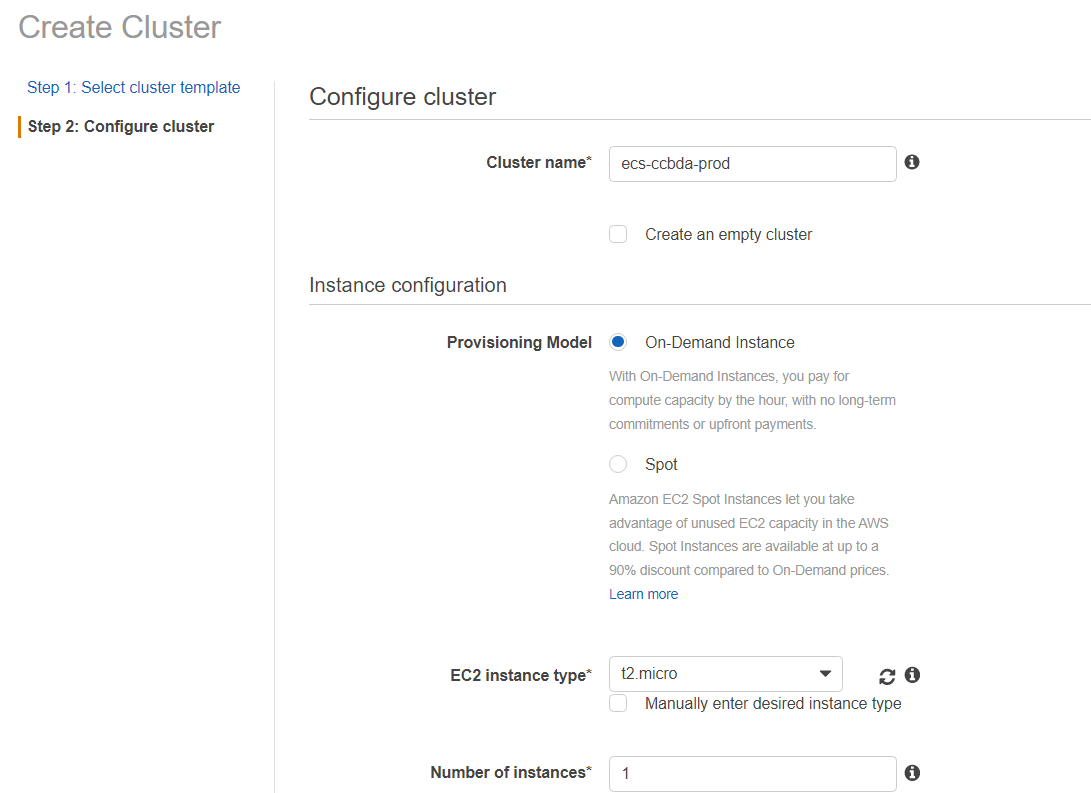
The Elastic Container Service allows us to have a cluster with multiple services running different tasks. A common scenario for a web application is to have a service for the backend, another service for the frontend, and some others for metabase, workers, etc.

To simplify things, we will have only one service, the backend service, which will have the backend code with the frontend templates inside.

In the next few sections we will explain the process to have an ECS cluster running and deploying new versions of the web application.

### **Cluster**

The first thing to do is to create the ECS cluster that will contain the backend service. You can create three different types of clusters: Networking cluster, EC2 Linux + Networking, and EC2 Windows + Networking. For our project, we have chosen the **EC2 Linux** type. Then, you have to specify the instance configuration as it shows the following image.



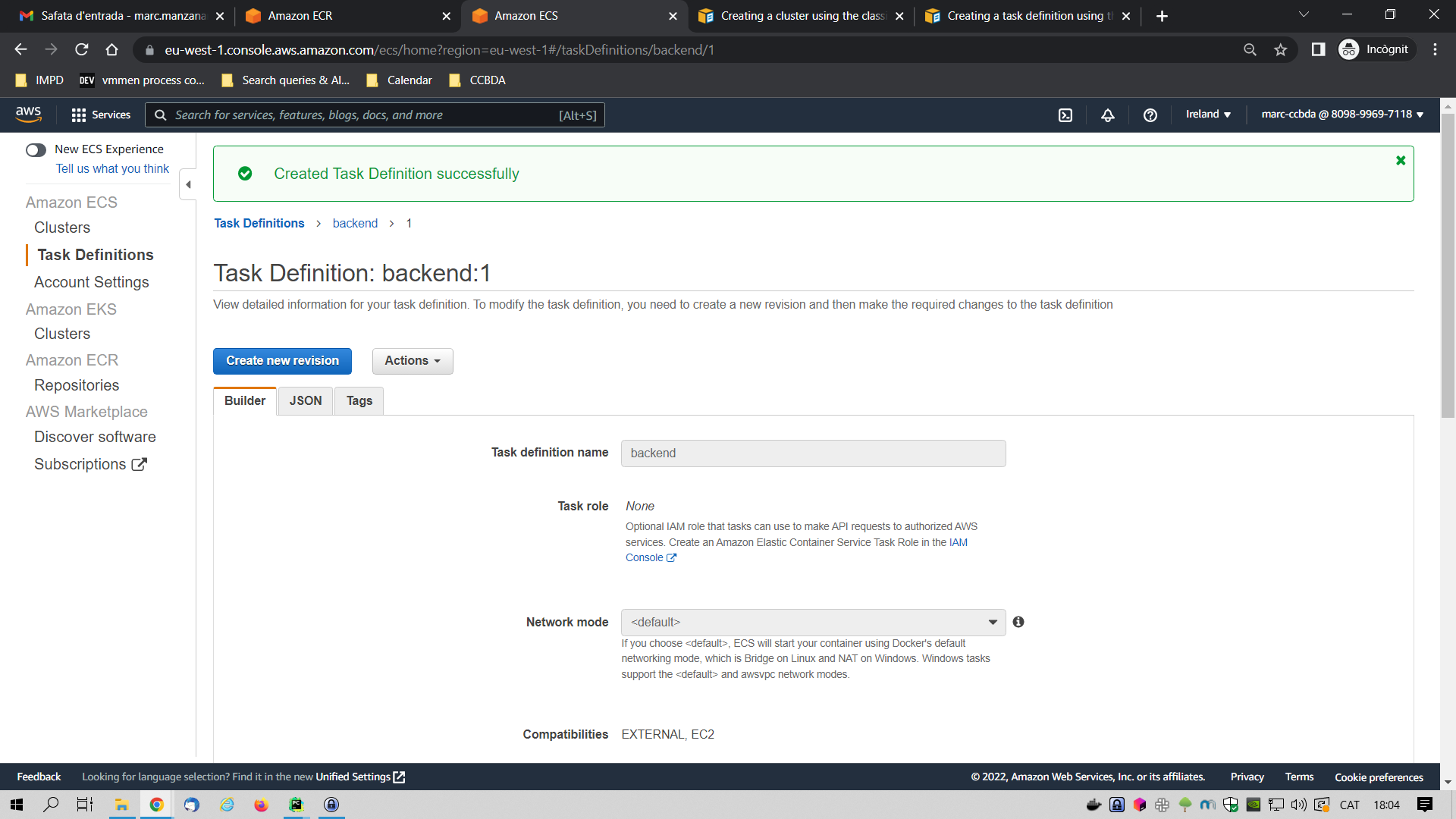
Once you finish with that part, the cluster will be created and an empty instance will be automatically deployed. So, the next step is to create a service and link it with the ECR repository. But before doing that, we still need to create two more things: a **task definition**, and a **load balancer**.

### **Task definition**

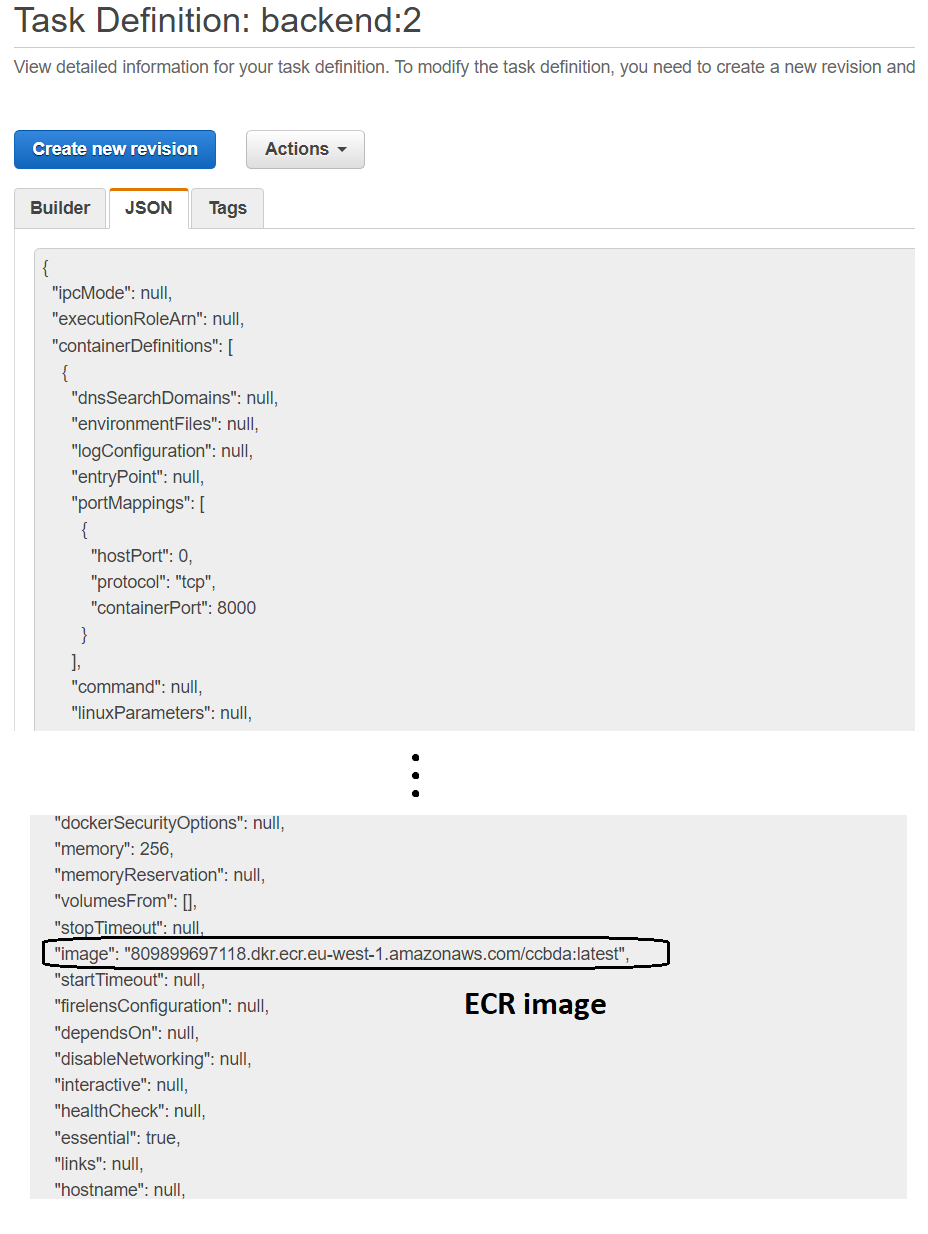
Task definitions specify the container information for our application, such as the number of containers we need, which resources they will use, and what ports they will use.

Task definitions are very useful because they have a thing called: **task revision**, which will be increased every time you do a change on the task definition without deleting the previous ones, so you can always rollback to a previous version on your ECS cluster service.

The next image shows the creation of a task definition, and as you can see, it says “Task Definition: backend:1”. This “backend:1” refers to the task revision.



How can we link the ECS with ECR repository images? In a task definition, you will be creating a container with some parameters. One of these parameters is the image. In the next screenshot you can see that we have linked the task definition “backend” with the ECR repository’s URI, which contains the docker images of our application.

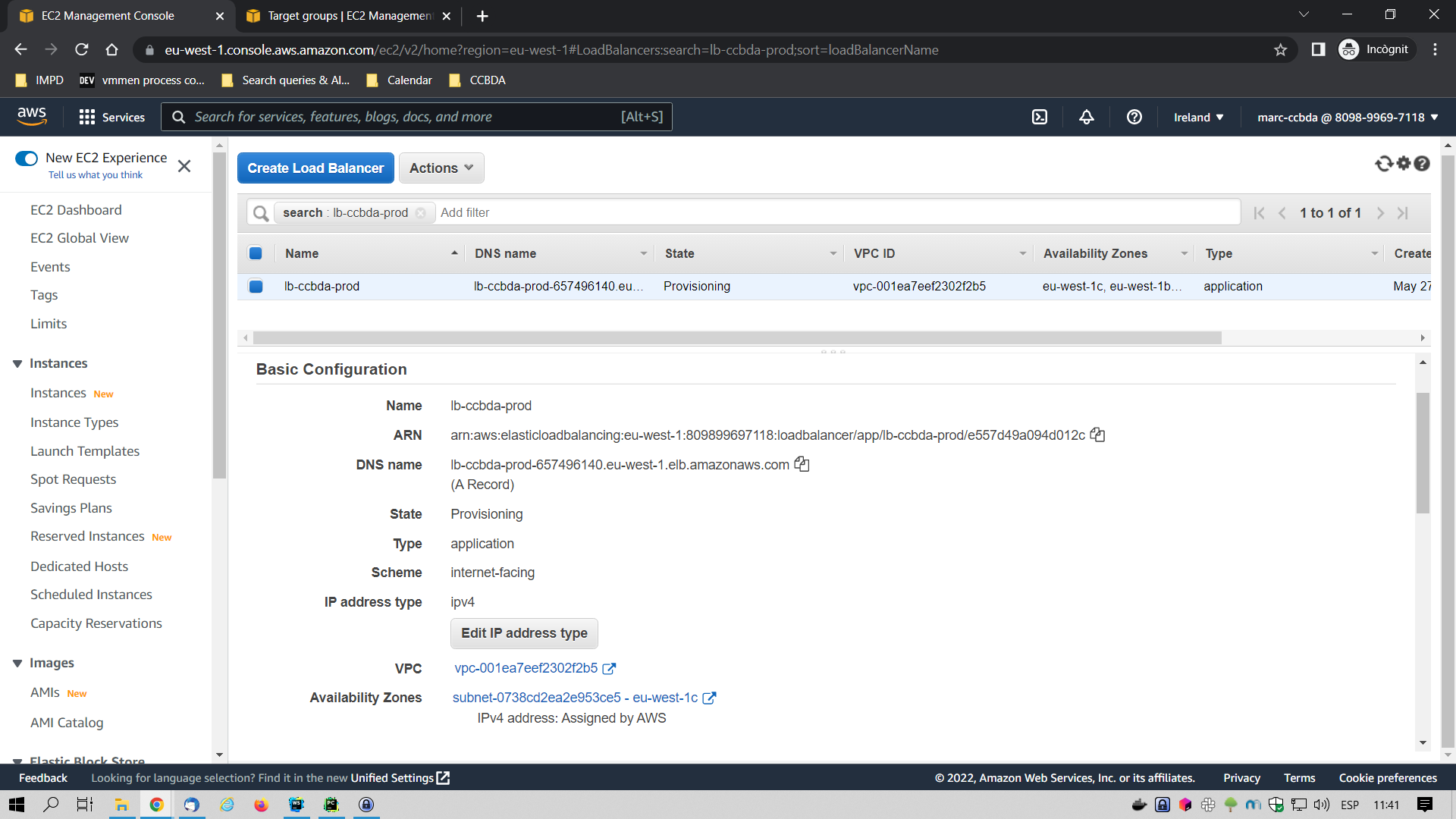


Next, we will create a load balancer for the cluster.

### **Load balancer**

The load balancer will be the entry point of our application and the one who will redirect requests to an instance or another depending on the policy we are using. We all have already seen and worked with load balancers in the lab sessions, so we are not going to go into too much detail.

The only thing we want to comment on is that we created a load balancer to have an Auto Scaling Group for our ECS cluster.



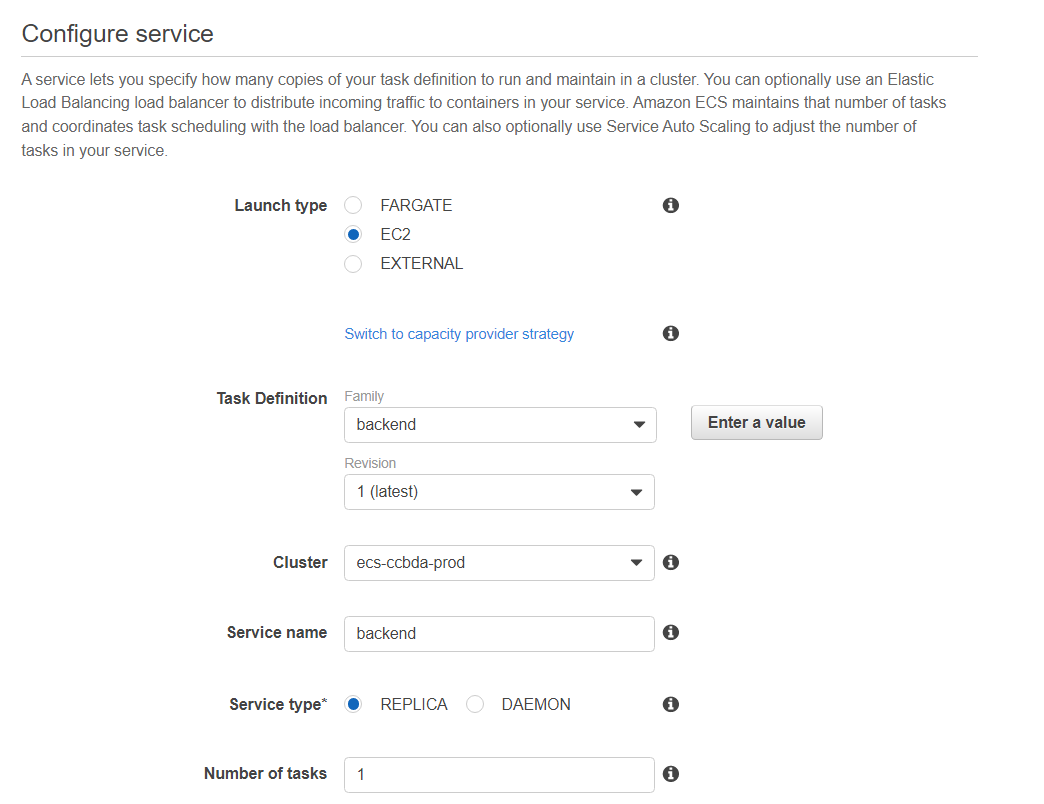
Now we are able to finally create our backend service inside the cluster.

### **Service**

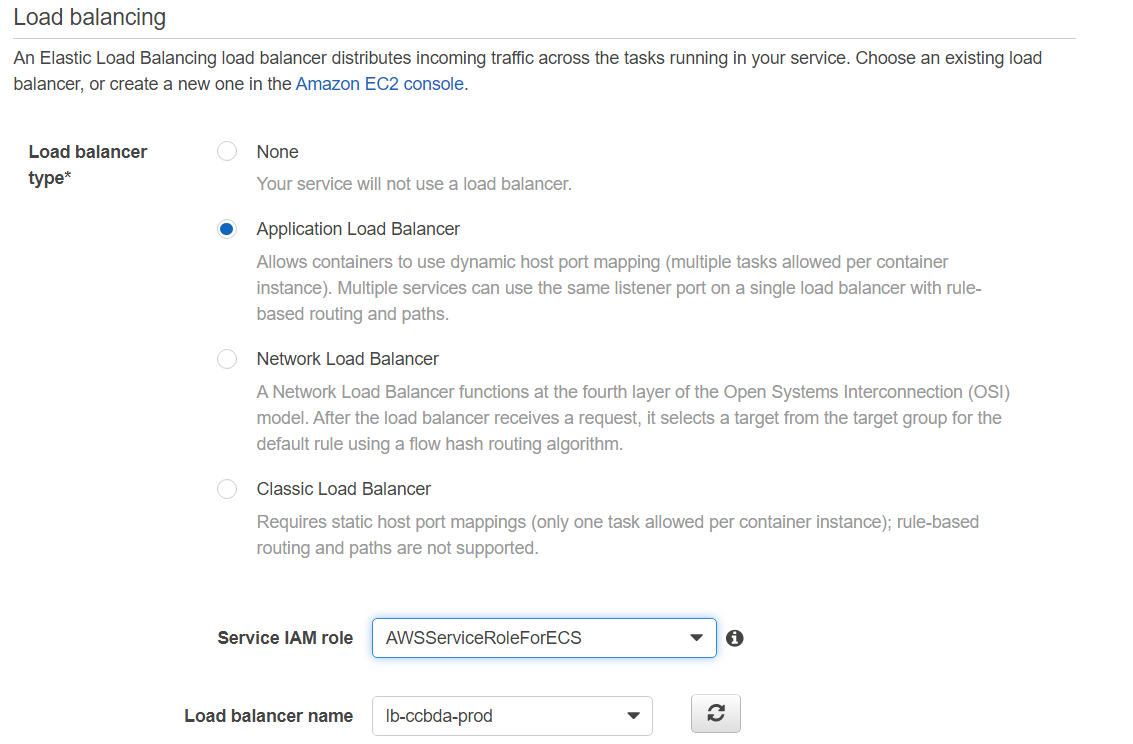
As we said before, we are going to create a single service for the backend image that already has all what we need to run the web application.

The creation of the service follows three different steps. The first step is service configuration:

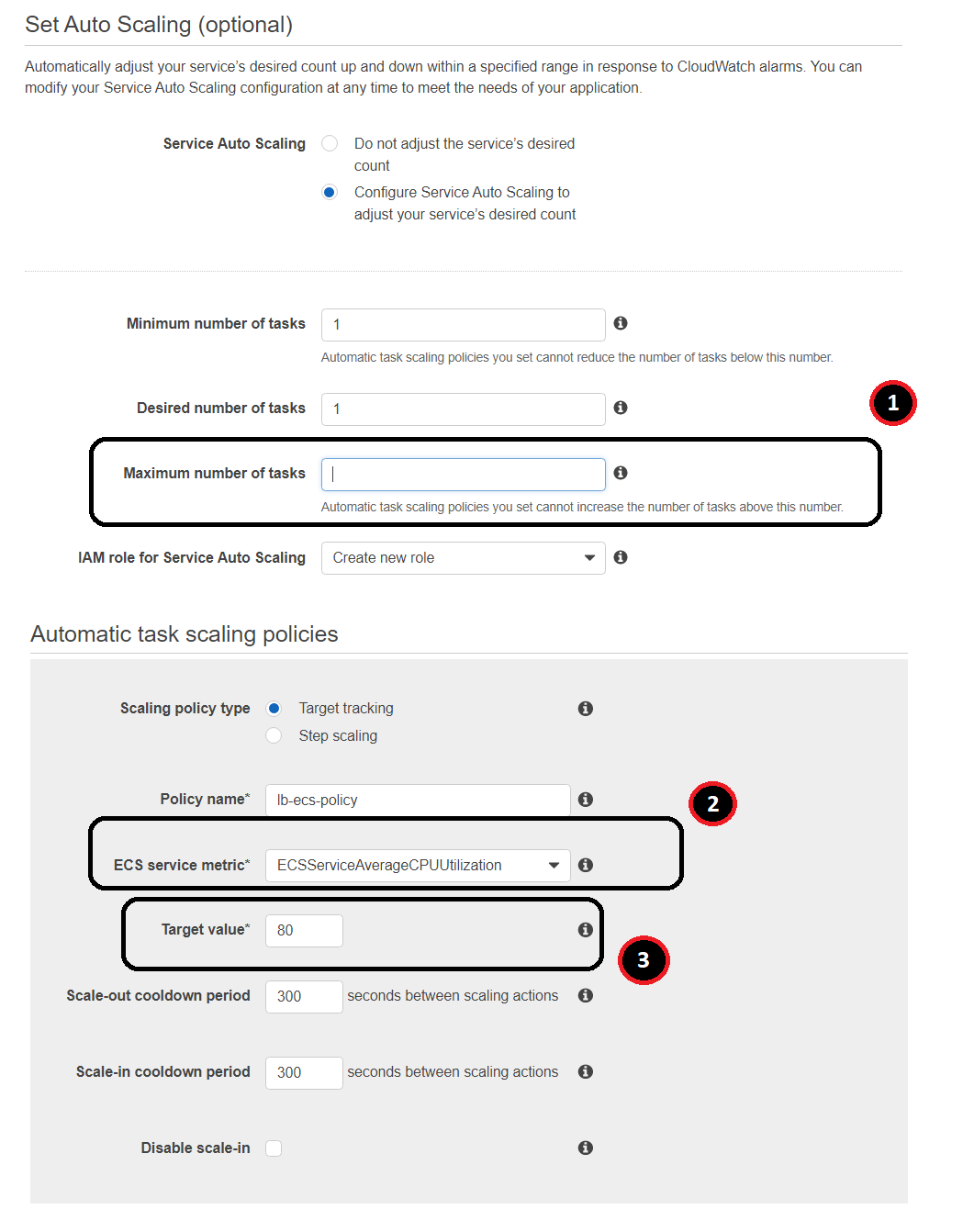
1. The first thing is to specify the service type. We will select EC2.
2. Next we will add the task definition we defined before, and also we can specify which revision we want to use.
3. Select the cluster in which we want to use this service.
4. Specify the number of tasks, with just one task it’s fine.



The second step is to add the load balancer we just created before:

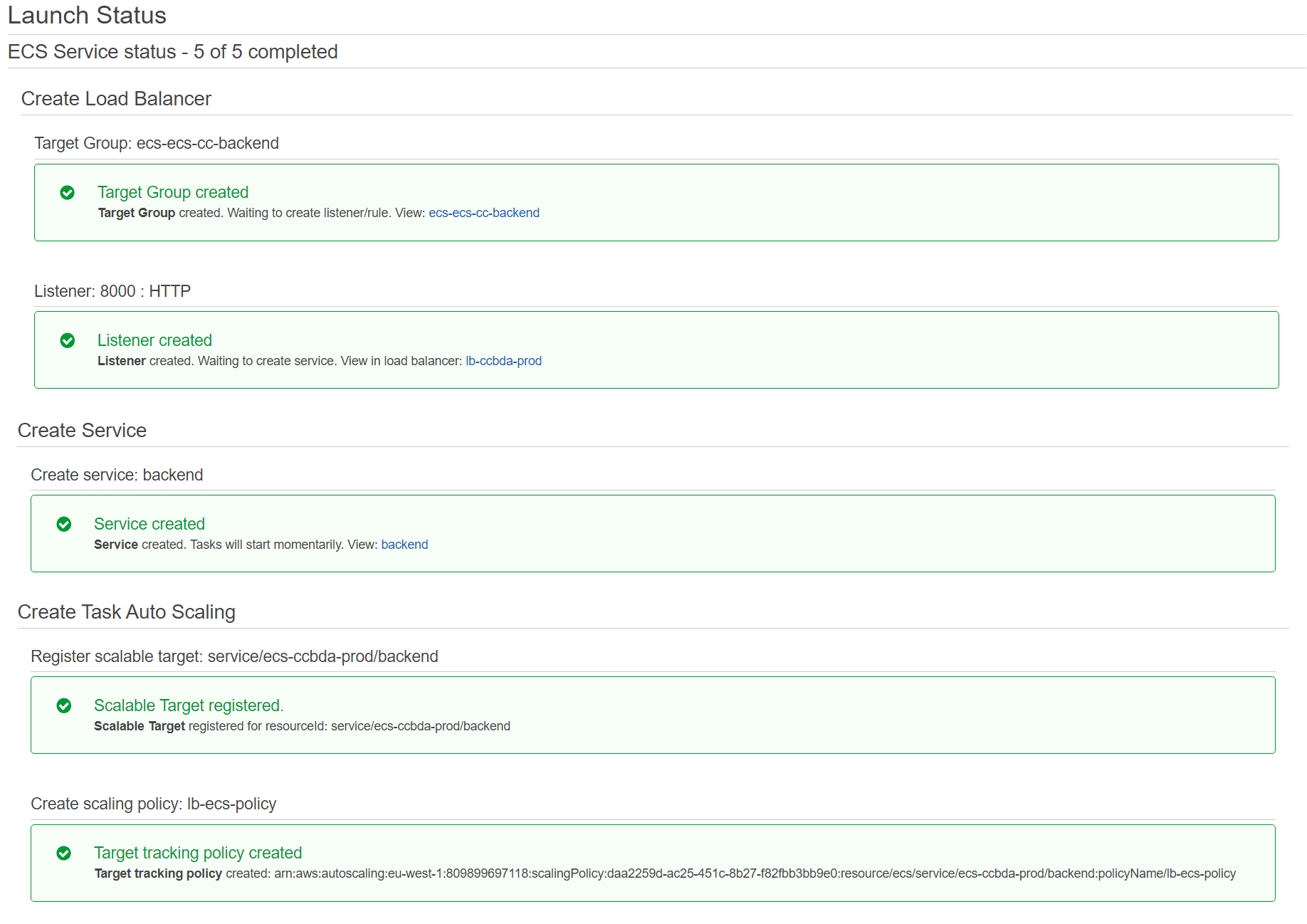


And the last step is to configure the Auto Scaling Group. In the next screenshot we have marked the most relevant fields to configure the ASG as we want. In our case, we have set up a maximum of 2 tasks that will be scaling according to the amount of CPU usage.

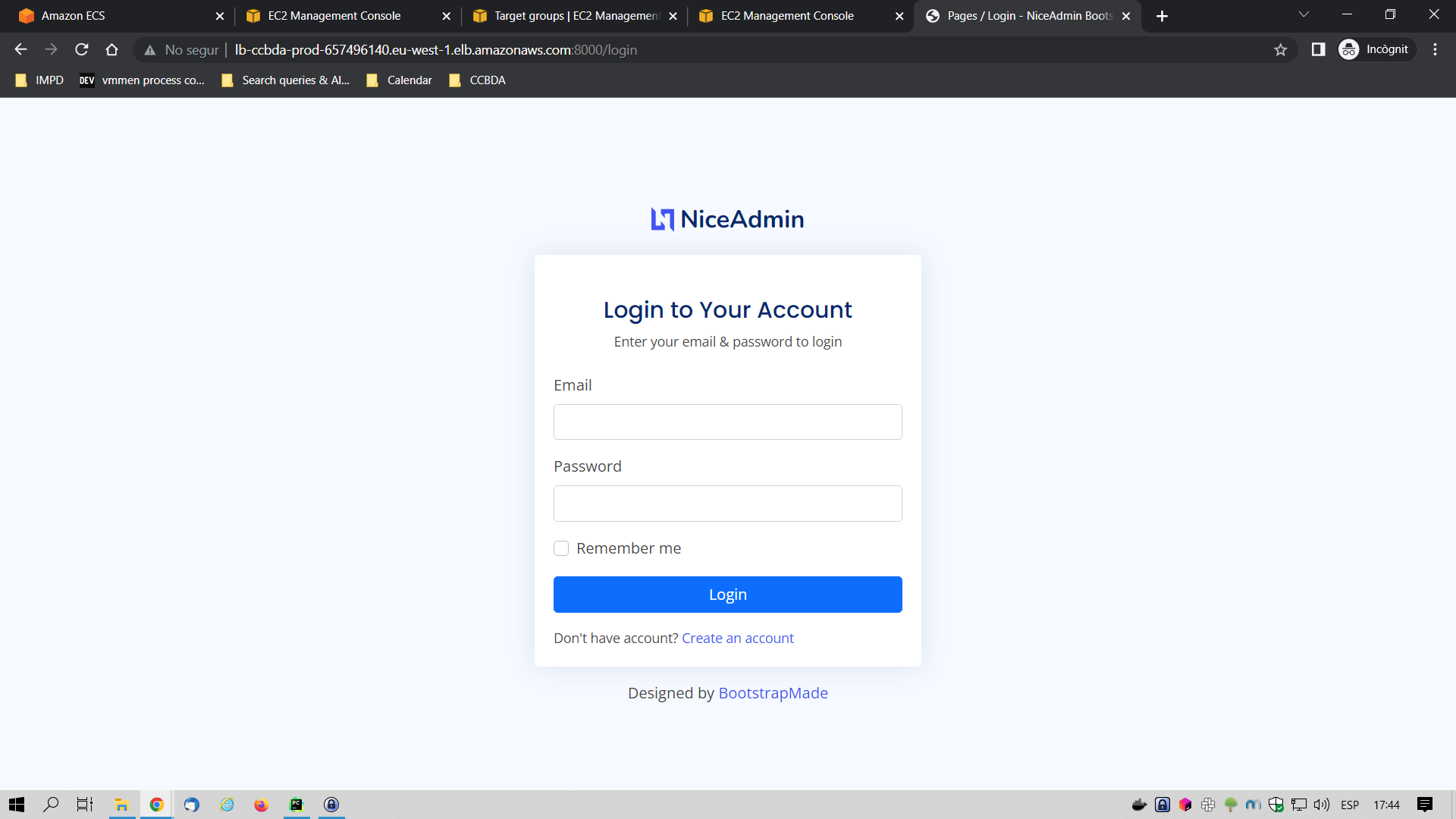


1. Limit number of scaling instances.
2. Define the metric variable which will be monitored to increase or decrease the number of instances (CPU, memory, request per target).
3. Set a value to that metric variable.

After that, we can finish the creation of the service and if we have done it right, we will see the same as in this screenshot.



Finally, we can access our web application through the load balancer DNS name.

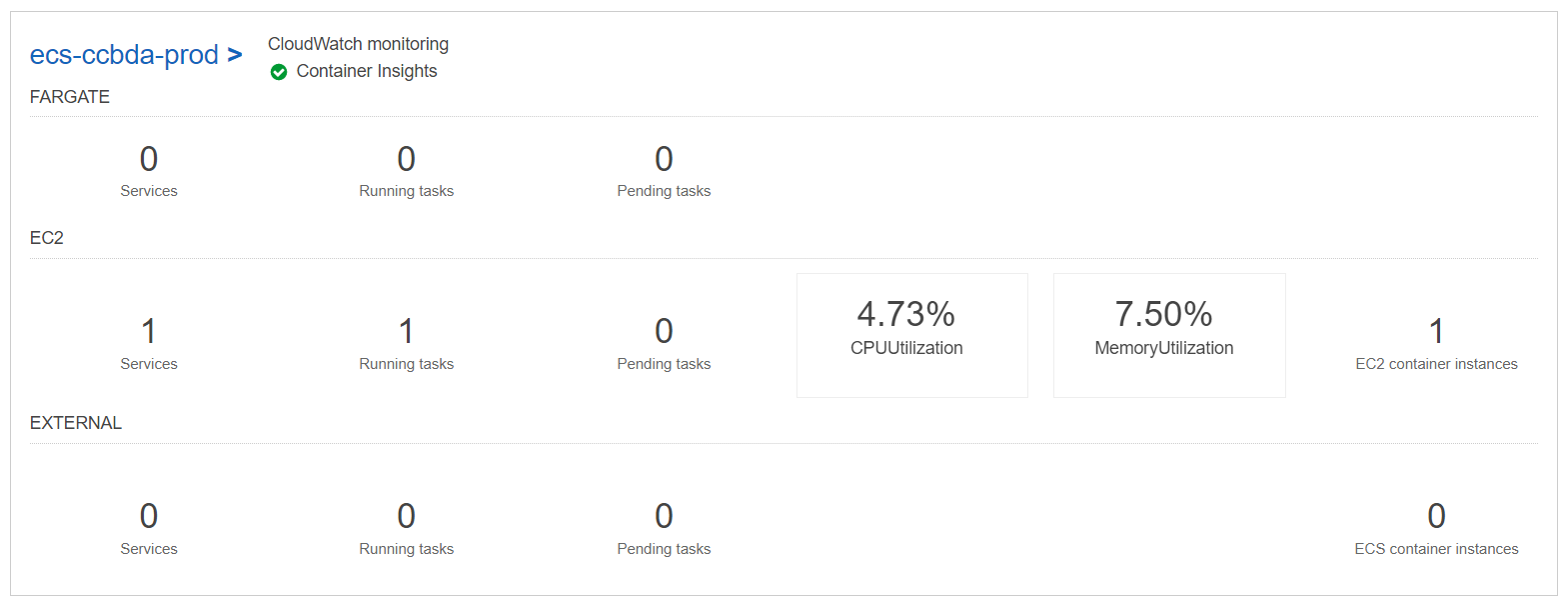


And that’s all, we have an automated process that is deploying images of our application automatically to AWS with an Auto Scaling Group and monitoring.

## AWS CloudWatch

AWS ECS provides a CloudWatch monitoring view for every cluster you have created. This way you can see and control the number of instances that are running, tasks, the amount of CPU usage, and so on.

Here is a little example of our cluster:



# Future work

In this report, we present the first version we have implemented of our application, but we would like to highlight some improvements that could be done in the future.

* Give the user the ability to create more personalized searches by, for example, choosing a range of years to show.
* Obtain more data and statistics on the players such as fouls, tackles, shot accuracy…
* Allow a user to create more personalized requests taking into account all the data (goals, assists, minutes played, games played and age of the team).
* Improve our ML model to predict the player value in the future.
* Show all the player's statistics year by year, such as goals scored or assists made.
* Implement the frontend using a framework such as Angular.
* Automate the deployment of the ECS infrastructure with CloudFormation.
* Create an AWS pipeline with ECR-to-ECS deployment.
* To have a staging environment.
* Add SSL/TLS encryption.
* Dockerize the database with PostgreSQL.

# Conclusions

Unlike the weekly workshops where we had to follow a path, in this project we were asked to invent the project based on the knowledge we had gained during the past few months. It was quite a challenge considering the time we had to work on it, the fact that we would have to work with an extended team and come up with something that was not obvious.

Our different cultural backgrounds gave us a way to teach and learn from other group members, even things that were not directly part of this course.

Overall, we can say that it was a good experience that taught us many things: from the ability to work in a group to the technicalities learned. We could experiment with some tools that we have seen in the project labs of the subject, and some others. Even though the project is at the starting point we can say that this application may be presented as a minimal viable product on the market.

1. <https://docs.aws.amazon.com/sagemaker/latest/dg/canvas-make-time-series-forecast.html> [↑](#footnote-ref-1)