Technologies

We used the following technologies:

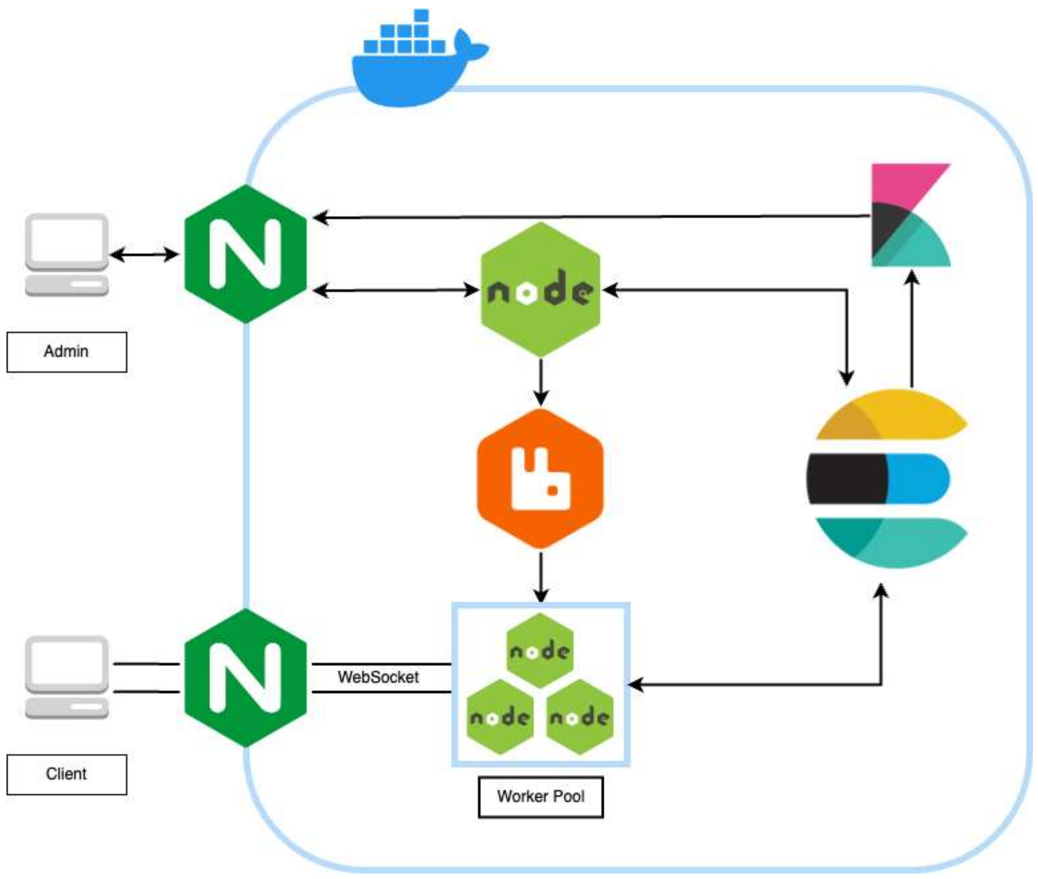
- **Docker:** to create, connect and deploy the different microservices needed for the correct working of the system. Moreover, starting from a base image, it allowed us to build personalized ones that are specific for our needs. The use of docker also simplifies cloud deployment thanks to the provided integration with AWS.
- **Elasticsearch:** as NoSQL Database. We choose it because it allows for a native interaction with another service (Kibana, see later). Being NoSQL it allows us to be flexible in how we store data without the constraints of an ER Schema.
- **Kibana:** in combination with elasticsearch, it allows us to easily create dashboards that are then shown to the admin. They provide significant information about the players and more in general about the state of the game, using data stored in the database.
- **RabbitMQ:** as a Message Broker, implemented with a Publish-Subscribe model.
- **Nginx:** we used it in several different ways depending on the type of services it needs to provide. In particular as a Web Server to provide static content to clients, as Load Balancer to evenly distribute requests among the worker and as a Reverse Proxy to handle the access to Kibana's dashboards and admin's APIs.
- **NodeJS:** to implement the different types of servers. In particular we used the express framework to create REST APIs and WebSockets.

We also need to briefly mention Amazon Web Services that allowed us to easily deploy on cloud the whole system. In particular we used:

- **Elastic Compute Cloud - EC2:** to create a load balancer that has a static address. In this way we are able to have a fixed entrypoint where to contact the system.
- **Elastic Container Services - ECS:** to automatically deploy the services as containers in the cloud.

Software Architecture

In the image we can see how we composed the different services together to build a more complex architecture.



We can divide the system into 3 distinct “logical subsets”: Base Services, Users Side and Admin Side.

Base Services: they are the backbone of the architecture; this subset includes all those services that are necessary for the system to function: elasticsearch, rabbitmq and nginx.

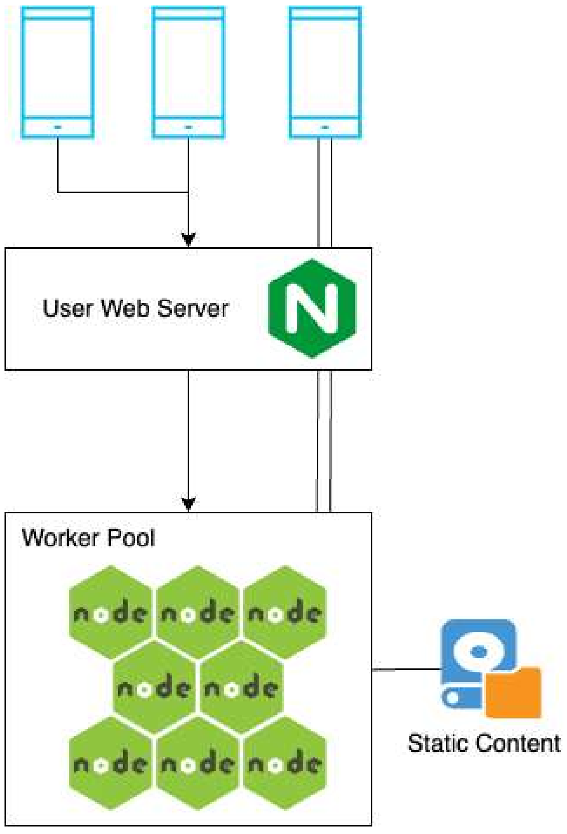
[NOTE: we will be more specific of the different instances of nginx in the discussion of the next part of the system]

Indeed without these microservices we would not be able to contact the system and the internal components would not be able to communicate within each other.

User Side: contains all those services used by users to play the game. It is composed of:

- Nginx: this instance of nginx is used as a Load Balancer for the requests. In particular, it distributes the client's request to one of the workers in the worker pool.
- Worker Pool: is a set of (almost) identical instances of a Worker Image. The role of each worker is the same and consists in being an endpoint for a websocket. Moreover it is subscribed to the message broker queues where it receives instructions from the admin such as: send a new question to all connected users or send the final score (once the game has ended). Each worker also interacts with the database to write and read information gathered during the game.

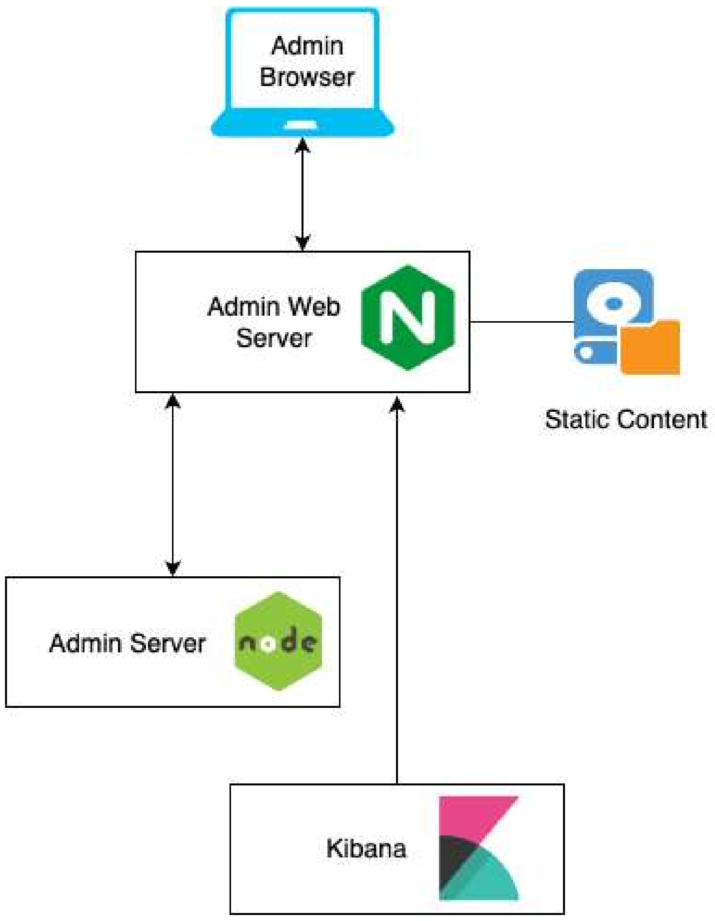
[NOTE: the instances of the worker pool are not identical due to how we deploy them. Indeed, our architecture is deployed using Docker on a single machine so we cannot use the same port for two different containers. Because of this, we are forced to specify as environment variables a different port for each worker, making them slightly different.]



(the arrows means that the client is doing a new request instead the "channel" is an established websocket connection)

Admin Side: contains all those services used by the admin to interact with the system. It is composed of:

- Nginx: the instance of nginx used for the Admin Side works as a Web Server providing static content and a SSL Connection to the user. Moreover it also works as Reverse Proxy which allows us to offer an SSL Connection to Kibana and to the APIs.
- NodeJS Server - Admin Server: that provides the REST APIs needed to start/end a game and send a new question. It interacts with the message broker as a publisher to notify all the workers in the system. Moreover it also interacts with elasticsearch to clean the database of old “game instances”.
- Kibana: provides to the Admin Web Interface two dashboards. The first shows the list of players with their id, nickname and score ordered by score. The second one shows the number of answers for each option with regard to the last question.


Admin Browser
Admin Web Server
Static Content
Admin Server
Kibana