Index of contents:

G.Y.A.
(GetYourAssistant)

Introduction to the system:

The GetYourAssistant (GYA) system is a general purpose architecture designed to make users able to get assistance from the organization they would like to contact. This means that also the organization workers should be able to obtain the requests and elaborate them. We also provided specific attention to the organization's supervisors. These last ones should be able to design system improvements based purely on the real-time statistics offered by the system. They will be able to also query the log database in order to obtain specific views that fit what they would like to examine. Also users will be able to see statistics about the system's general performances in order to obtain insights about it, and manage better their time, choosing for example hours of the day with less expected waiting time, to issue their requests. It's important also that all the different sections of the distributed system are provided with access management. In this way only who's allowed to take specific actions, or see specific sections, will be able to do it. Only users will be able to register, since the organization's role will be explicitly treated by the managers themselves. The project is available here.
After this small introduction we'll deepen better, how each specific section is composed and how it technically works in every of its parts.

Overview of the main components:

The system is composed of different parts, and each of them will be a different deployment in the distributed system. This architecture has been founded onto the orchestration system called Kubernetes. It gives the system many different properties, like the fault tolerance, the portability, the easier management and many others. The different main components that have been deployed are the following ones:

- User web panel: it allows the user to obtain an easy to use GUI, where he can apply all the requests for the assistance that he needs, and see the statistics of the requests' answering, using specific filters. Authentication management is provided to this component.

- Worker web panel: it facilitates the work to the organization components in charge of managing and solving the users' assistance requests, providing logging capabilities and authentication management.

- Supervisor web panel: through this portal each supervisor is able to login and visualize the statistics of the organization's workers. In this way he's able to make the company obtain improvements, simply based on the available data.

- User API server: this servant is mainly used to forward the requests from the users to the RabbitMQ queue. This will be in charge of adding also the current time to it, which will be coherent with the server which will be fetching the requests at the end of the process.

- Primary API server: this component automatically fetches the requests from the RabbitMQ server, when they’re available. Once the final worker panel will be requesting for new requests it will be in charge of providing them.

- RabbitMQ server: this node is needed for making asynchronous communication possible between the two servers in charge of dispatching the requests from the users to the organization workers.

- ElasticSearch server: this storage component will be responsible for storing the loggings and make them available for the Kibana plugin for easier access, under the shape of dashboards.

- Kibana server: this plugin to the ElasticSearch deployment will be exposing statistics as dashboards in the respective web panels of interest. In this case, it won’t need authorized access from the visualizing users.

- PostgreSQL databases: there are three databases, each one corresponding to a different web panel implemented, and they simply stored the credentials for accessing them.

Components communication:

For development purposes all the system’s deployments have been exposed in the local network, so they were accessible during the simulation at the “localhost” address. In this way it was possible for the developer to test each of them, without having the need of using multiple machines. In particular there were created different host entries under the same remote address “localhost”, and they were addresses significant with respect to what they were exposing. In this way it was also easier to remember and recognize the addresses of the different services, without having the need of finding their respective IP addresses at every deployment. The component in charge of exposing HTTP services, was the one called “ingress”. It is a component able to expose some services deployed into Kubernetes. As a choice I selected the ClusterIP as a default service for exposing each service, since it was sufficient for our purposes. The ingress was of type NGINX so it was respecting all the specifications concerning this type. All the host entries it was exposing were using an SSL certificate that was securing all the addresses needed (through the SAN specification). It was a simple certificate that obviously was catched by the browser as “not secure” because it was self-signed. To remove this problem it was sufficient to install the certificate into the device at the first start, and remove the warning. The different components were not using the ingress, which was only useful to the developer, but they were able to communicate between them, simply through the network created by the Kubernetes deployment. In this last specific environment, the nodes are able to communicate between them simply considering as a network address their deployment name. It will be automatically resolved by the Kubernetes DNS resolver. In this way the DNS eliminated the necessity of passing through the localhost ingress which would have been an additional step in the communication.

The ingress exposes the following host addresses under the remote “localhost” address:

- https://user-web.localhost → refers to the user web panel
- https://supervisor-web.localhost → refers to the supervisor web panel
- https://worker-web.localhost → refers to the worker web panel
- https://primary.localhost → refers to the primary server
- https://user-api.localhost → refers to the user api server
- https://pending-queue.localhost → refers to the RabbitMQ server
- https://elasticsearch.localhost → refers to the ElasticSearch deployment HTTP access point
- https://kibana.localhost → refers to the Kibana deployment HTTP access point

The internal addresses that each deployment exposes in order to be reached by the other, are the following:

- user-web → refers (internally) to the user web panel
- supervisor-web → refers (internally) to the supervisor web panel
- worker-web → refers (internally) to the worker web panel
- primary → refers (internally) to the primary server
- user-api → refers (internally) to the user api server
- pending-queue → refers (internally) to the RabbitMQ server
- elasticsearch-es-http → refers (internally) to the ElasticSearch deployment HTTP access point
- kibana-kb-http → refers (internally) to the Kibana deployment HTTP access point

Kubernetes deployment general characteristics:

Each node simply corresponds to a Kubernetes deployment, and it is applied simply retrieving a corresponding Docker image. Some nodes like ElasticSearch, Kibana and the PostgreSQL databases are based on default images provided by the respective producer organizations. They are then configured after the deployment, to make them match the system’s purposes. The other components instead have been deployed starting from a personalized Docker image built and pushed to DockerHub, to make Kubernetes able to fetch it during the deployment process. Each deployment is then associated with a ClusterIP service, which is a Kubernetes component able to expose an entire cluster generated by a deployment as a unique service. This is needed in order to make some functionalities accessible directly by the developer. Only the databases are exposed by another type of service which is called NodePort. In this case it was preferred because the DBs were exposing protocols different from the HTTP one. In this case this distinction was necessary and they won’t be included obviously (since no HTTP) in the ingress host entries. Another important ingredient to make them accessible is the ingress. As already mentioned, the NGINX ingress is a server which is able to expose HTTP services over a specific address. In this case the address is “localhost” so it’s the default one, and an SSL certificate has been added so the different host addresses are accessible using HTTPS. Very important is also the role of the PersistentVolumes. These are Kubernetes node types that are able to make some deployments maintain a partition of memory even between different redeployments. In this case it was necessary for a database to maintain the state of the credentials of the different users in the system. PersistentVolumes need an associated PersistentVolumeClaim which is simply the point of connection between the deployments and the PVs. Considering a last component we have to mention the Secret, which is simply storing the SSL certificate needed by the ingress to expose hosts using HTTPS. In this case the certificate is simply for developing purposes, so it’s just self-signed and it doesn’t guarantee any amount of security as it can be seen by the warnings that the browsers provide when trying to access the pages of GYA.

Let’s now deepen each component in order to have a clear understanding of the whole system, together with a small explanation of the interfaces and how the user can use them.

Components technical overview:

User web panel:

General functioning:

The user web panel is the only point of access for the customers of the organization that provides assistance. They can register inside the web portal and then after authenticating they will be ready to send assistance requests simply pressing a button. The website will automatically fetch all the data relative to them, and it will send it directly to the API in charge of forwarding it. Another important function available to the users is the one of obtaining relevant data with respect to the system’s performances in specific hours of the day. In this way they will be able to better manage their time, choosing suitable moments to issue their requests, based on average data. The statistics can be filtered thanks to the Kibana interface which won’t request any additional login, but will provide data with anonymous and automatica authentication of the user.

Coding structure:

Starting from the image from which the relative deployment is created, we have that in this case we have a personalized one (not default) so it will be fetched directly from the GYA DockerHub section. The image is derived from an Apache server enabled to PHP. However in this case further configurations were needed in order to make it able to interact with the PostgreSQL database deployed. In fact, it has been simply necessary to execute (at build time) a command which was installing the library PGSQL for PDO, that enables the PDO library to interact with PostgreSQL entry points. Once this modification was applied it was sufficient to insert into the image (for it to be built) the files relative to the website that needed to be implemented. The node is deployed with the name of “user-web” and the components connected to it make references to that name. The pages exposed by the HTTP-PHP server are mainly composed of the languages: HTML, PHP and JS. The PHP part consists in mainly three sections: the first one dedicated to the login part, so it has to manage the validation of the credentials inserted by the user, to see if they belong effectively to a user. In that case it has also to take further information about it, in order to set the session variables and make these information ready to be sent. A second section is dedicated to the registering phase, in which the data inserted by the user is confirmed or rejected based on the entries already available in the database. Once they have been confirmed, the user is registered successfully. The third section, which is also used by the other two, is the simple connection to the database, to enable communication between the servers and this last one. Javascript is required to make the portal request to the user his current position, and elaborate the request that has to be sent using JSON as a result. HTML is simply describing the design, and the components of the page. The statistics, although, are not designed inside this coding part, but embedded as a simple iframe HTML component.

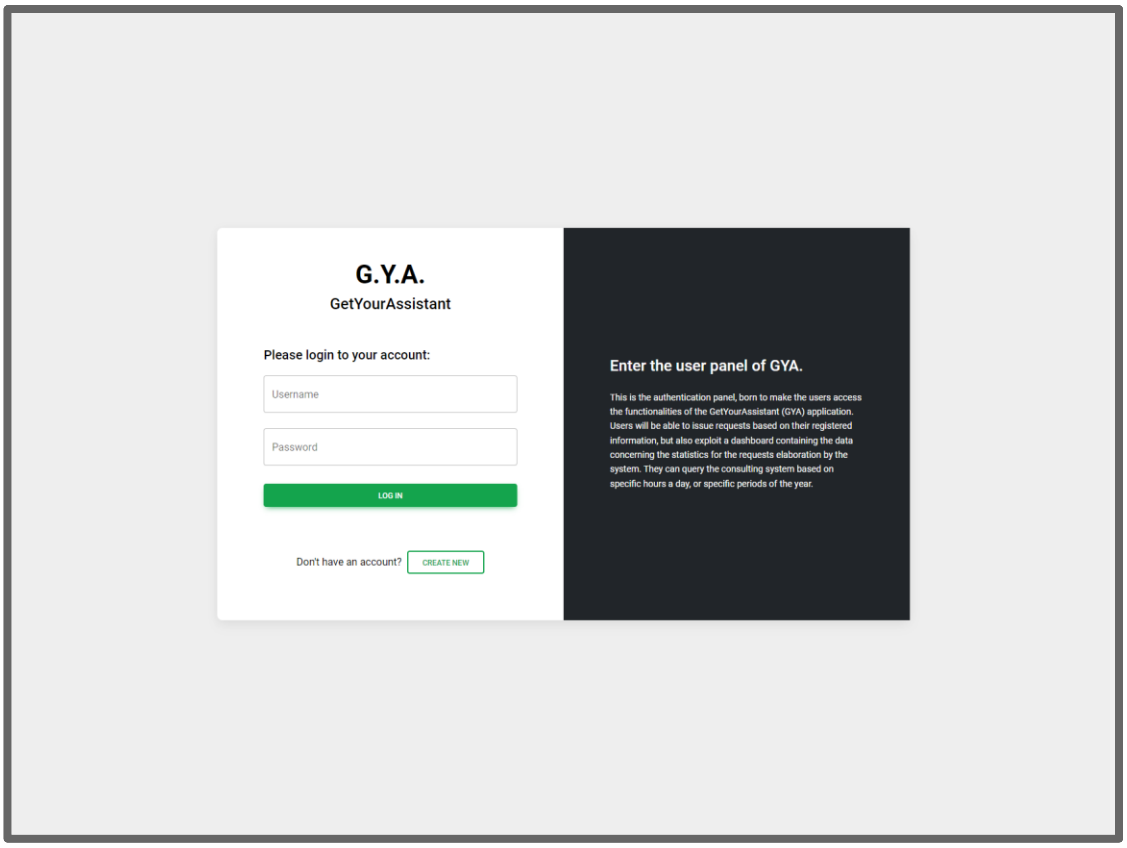
Kubernetes deployment of the component:

The deployment simply fetches the Docker image from the dedicated DockerHub. In this case no further configurations are required.

Component networking:

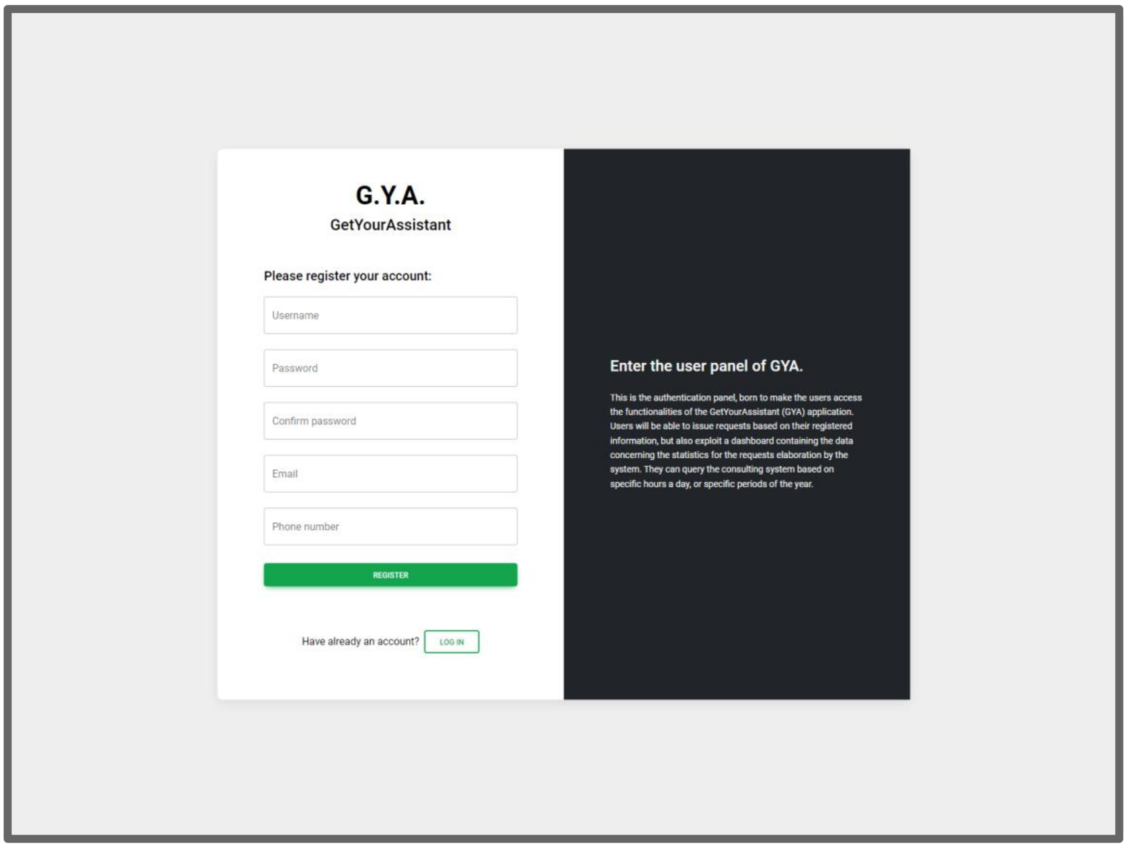
The Docker container relative to this deployment exposes the service over the port 80, the classical one adopted for the HTTP protocol. The deployment is then referenced by a ClusterIP service. This last service is then useful to be available for the ingress, which will make the developer able to see and access the website through a practical host web address. The developer will then just search the address inside its browser to obtain the wanted result, since the DNS entry of Kubernetes, in the “hosts” configuration file, will do the rest.

Graphical user interface:



Enter the user panel of GYA.

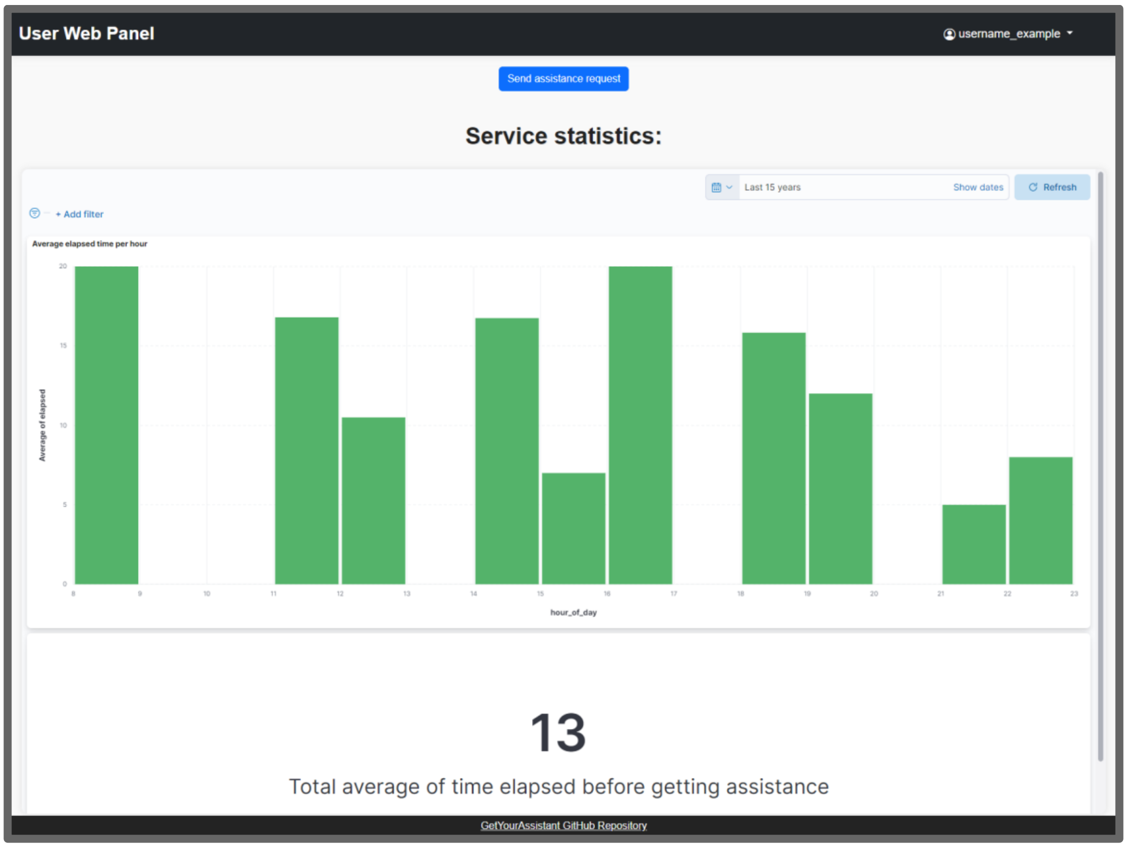
This is the authentication panel, born to make the users access the functionalities of the GetYourAssistant (GYA) application. Users will be able to issue requests based on their registered information, but also exploit a dashboard containing the data concerning the statistics for the requests elaboration by the system. They can query the consulting system based on specific hours a day, or specific periods of the year.



Enter the user panel of GYA.

This is the authentication panel, born to make the users access the functionalities of the GetYourAssistant (GYA) application. Users will be able to issue requests based on their registered information, but also exploit a dashboard containing the data concerning the statistics for the requests elaboration by the system. They can query the consulting system based on specific hours a day, or specific periods of the year.

In the previous images you can see the login and registration page of the web user panel.



In the last image instead it’s possible to see the homepage for the user web panel, with the functions mentioned in the other sections.

Worker web panel:

General functioning:

The worker web panel is the interface that allows the organization workers to interact with the overall system, and answer to the assistance requests. The worker will have to login with the credentials provided by its organization in order to access the main functionality concerning his role (answering assistance requests). For this reason no registration has been provided to this kind of web panel. Once a worker is ready to answer a customer, it simply has to press the button that fetches the next request. If requests were available at the time of pressing the button then the worker will be provided with all the data concerning the user requesting for assistance. In the case where there wasn’t any request pending, then the worker will be notified with a message explaining this particular condition. Once a request is fetched, we have many information shown about the user who issued that last one. In particular we have the following data (notice: this kind of information are easy expandable in a system of this type, however we provided only basic data for testing purposes):

- Username
- Email address

- Phone number

- Time elapsed: it is the time elapsed from when the request has been issued and when it has been taken in charge from a worker
- Location: it is the location from where the request has been issued. It is available if the customer allows the browser to obtain it. It is shown using a Google Maps iFrame that shows in a comprehensible way the geographical coordinates embedded with the request received.

Some of this data is useful to the worker purely to contact the user, while other kinds of information (like the geographical location) are also useful to collect statistics logging and allow the manager (or supervisors) to obtain insights about the system working and usage from the citizens of different countries.

Coding structure:

This component has been created using entirely the languages HTML, JavaScript and PHP. It has been created a Docker image derived from the original of PHP-Apache, to which, during the building, it adds the option of using the PDO_PGSQL library, which is an extension of the PDO library that allows to access a PostgreSQL database directly from the PHP code. The final docker image contains in the html folder the pages created for the portal that we want to expose. The portal uses PHP to manage all the authentication process, starting from the validation of the form, ending to the verification of the credentials consulting the database. In this way it is the server to perform the different validations of the form, not allowing users to tamper the validity of the data. Once the worker is able to login, the main functionality that he could use is the one of fetching a new request. This functionality has been realized entirely using JavaScript, which simply performs a POST request using AJAX, in order to obtain results without the need of refreshing the entire page, that would have been needed using a simple form submit. In this case the results are shown simply fetching the data from the JSON received and displaying it in a comprehensible way, using a tabular layout. The map instead is realized simply using an iFrame produced by Google. It is directly connected to Google Maps and it allows the requestor to indicate longitude and latitude directly inside the address. In this way the worker will be able to adopt a familiar and practical interface to see the request’s provenience.

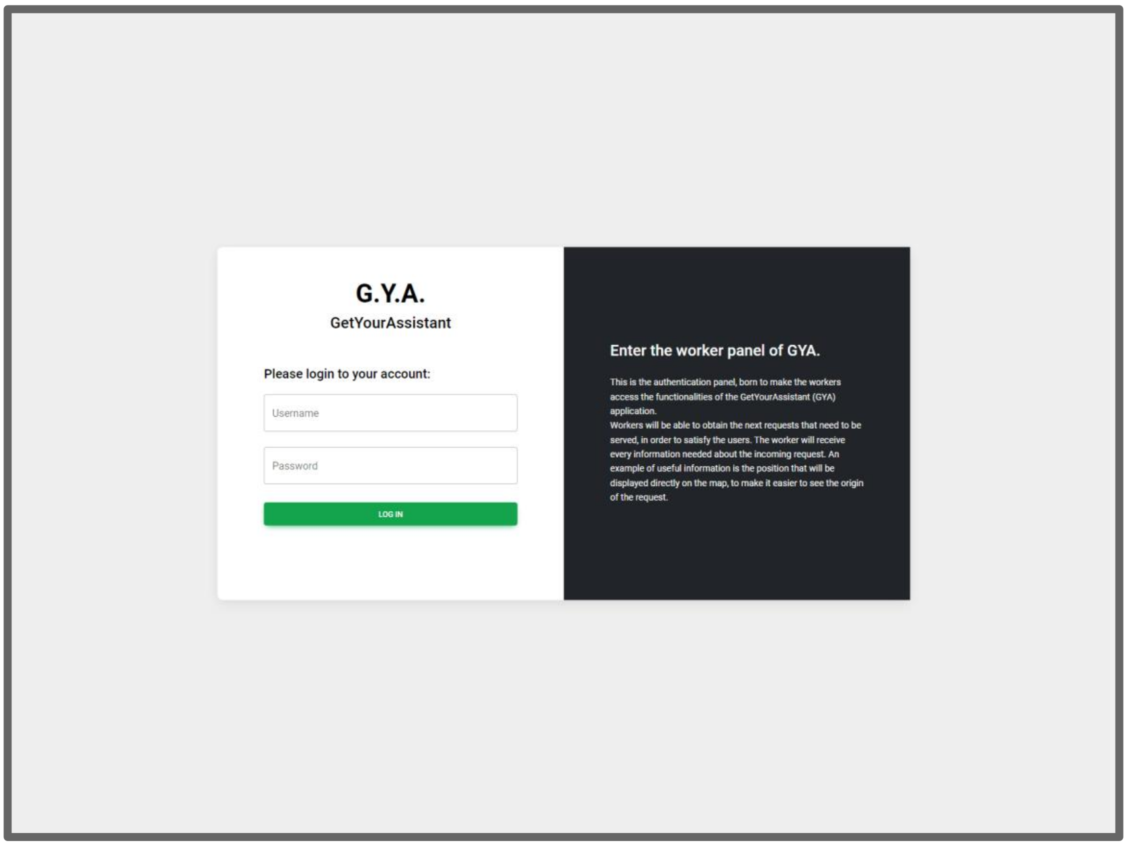
Kubernetes deployment of the component:

In this case no particular configurations were needed, since the Docker image which is fetched by the Kubernetes deployment has been created directly from us, and so configured already at deployment in the best suitable way to fit the system needs.

Component networking:

In this case we have that the Docker image dedicated to realizing the user web panel, simply exposes the service over the port 80. It is then needed to realize a ClusterIP Kubernetes component service, that references the deployment. Once it has been realized, the ingress will then reference this last one, making the developer able to test and obviously access the portal, which will be then accessible through its browser. Networking addresses have been already treated in the previous phases of this document.

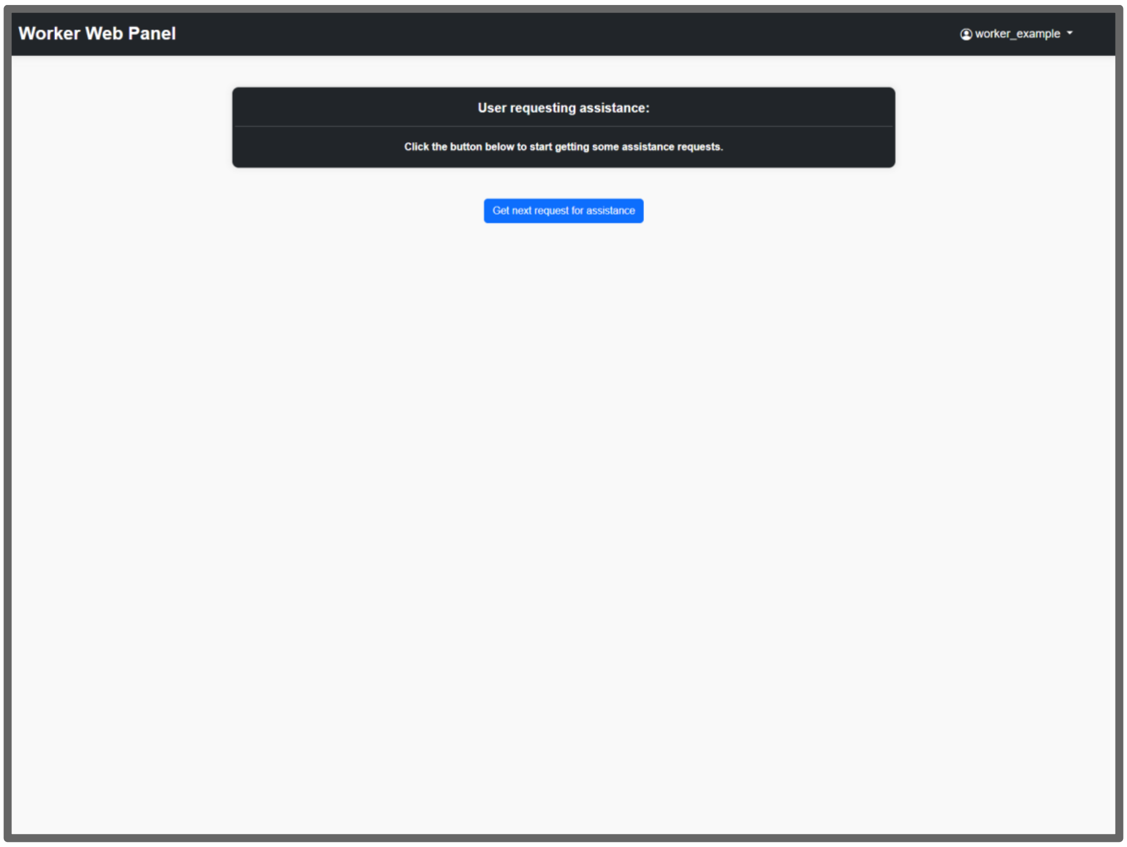
Graphical user interface:



Enter the worker panel of GYA.

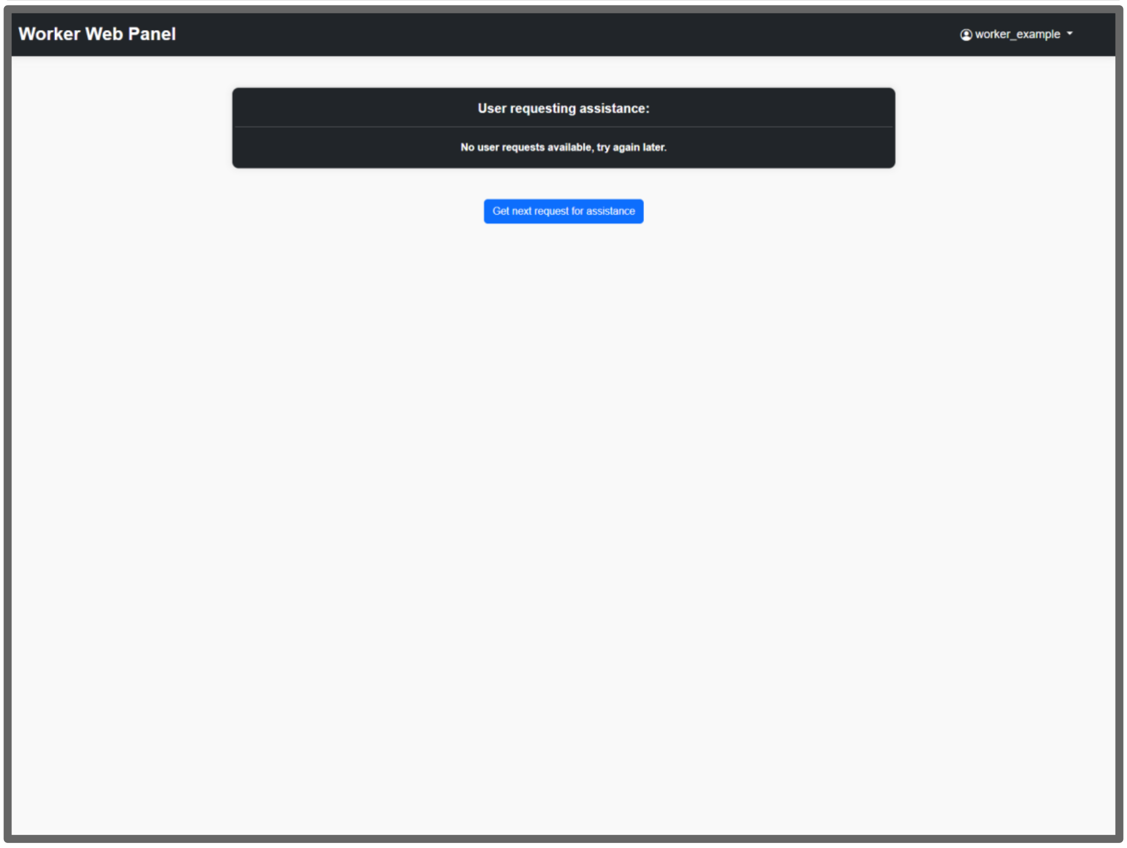
This is the authentication panel, born to make the workers access the functionalities of the GetYourAssistant (GYA) application.
Workers will be able to obtain the next requests that need to be served, in order to satisfy the users. The worker will receive every information needed about the incoming request. An example of useful information is the position that will be displayed directly on the map, to make it easier to see the origin of the request.

In the last image it’s possible to see the authentication (login) page for the worker web panel.



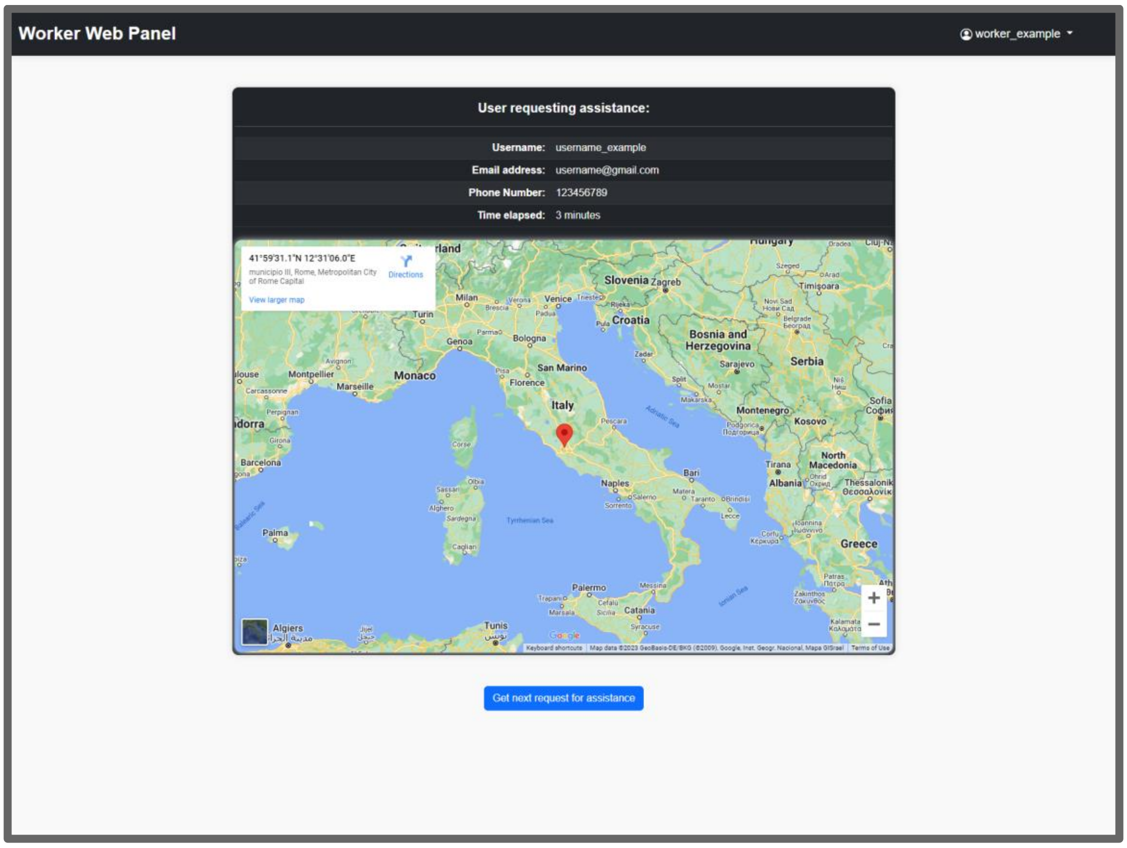
User requesting assistance:

Click the button below to start getting some assistance requests.



User requesting assistance:

No user requests available, try again later.


Worker Web Panel
worker_example
41°59'31.1"N 12°31'06.0"E
municipio III, Rome, Metropolitan City of Rome Capital
Directions
View larger map
Get next request for assistance

User requesting assistance:

Username:	username_example
Email address:	username@gmail.com
Phone Number:	123456789
Time elapsed:	3 minutes

In the last three images we see three statuses of the worker web panel. In the first one the operator still has to ask for a request. In the second image instead he tried to fetch a request, but no one was available. The third image instead shows a request fetched with the relative data included in a comprehensible way.

Supervisor web panel:

General functioning:

The point of interaction with the system, dedicated to the managers of the organization is the one called Supervisor Web Panel. This one allows the supervisors to observe and better manage the distribution and performances of their workers. Also in this case, for a supervisor to access the functionalities of the portal, it is required to login using the organization's credentials. However, once the correct credentials are provided to the system, it will be possible for him to interact with the statistics dashboards, filtering the data concerning the system performances based on different parameters. Some parameters of interest for example could be the time range, the hours of the day, the workers considered etc. Since this data is easily accessible and query could be applied, we've that the supervisor could better make smart decisions with respect to the good behavior of the organization. The information available to the supervisor consists of data like the amount of requests of assistance managed by specific workers and location of the requests in specific time ranges.

Coding structure:

This implementation has been built starting from a pre-existing Docker image that deploys an Apache web server with PHP functionalities. Once it has been configured to add the PDO_PGSQL library, it is possible to add the file containing the dedicated code and build the system into a new personalized Docker container. It will be then pushed into the GYA's DockerHub section, in order to be ready for the Kubernetes fetching. In this case the coding structure of this application component could be divided into two main parts. The first one is entirely dedicated to the interface of the index, which is purely implemented in HTML, with the statistics' dashboard simply consisting in a Kibana iFrame embedded, and so the relative interface hasn't be implemented directly by the team developer, however offering good usability and reactivity. The second part of the coding structure consists instead in the authentication phase. It allows to proceed to the system functionalities only the subjects authorized by system's credentials provided by the organization itself. It is needed since it wouldn't be normal for a user to directly register as a supervisor, as this is a kind of role managed by the direction of the company. This last part is implemented exclusively in PHP mixed with simple HTML. It is needed in order to validate the form and also communicate with the database to check the authenticity of the credentials provided by the supervisor who's trying to access.

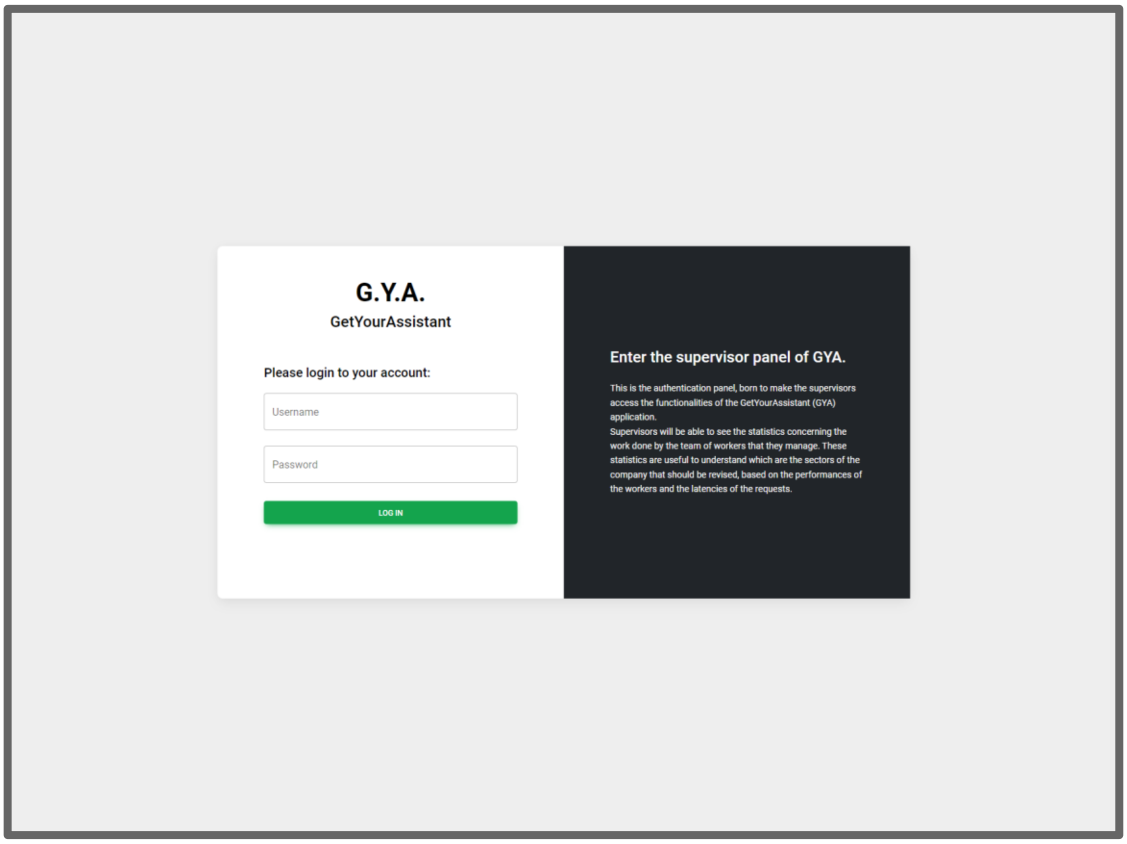
Kubernetes deployment of the component:

The supervisor web panel is simply composed by a deployment who's fetching the personalized image directly from the GYA's DockerHub. For these reasons, no further configurations were needed, and the deployment was already fitting the system requirements desired.

Component networking:

The default Docker image from Apache already exposes the HTTP service over the port 80, so it was necessary to just create an additional Kubernetes ClusterIP service. It then was referenced by the NGINX ingress in charge of allowing the developer to make the deployment accessible from a simple local Internet browser bypassing the warnings relative to the self-signed SSL certificates (for testing purposes).

Graphical user interface:



G.Y.A.

GetYourAssistant

Please login to your account:

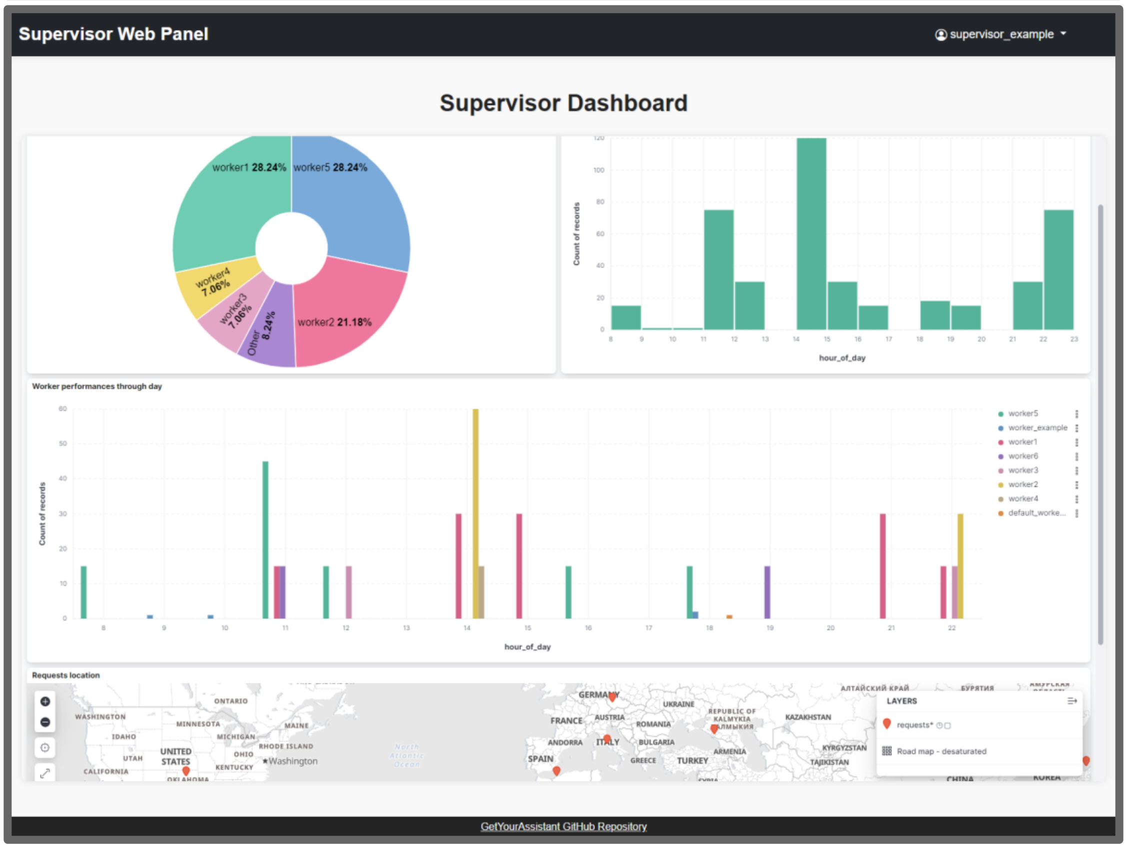
Username

Password

LOG IN

Enter the supervisor panel of GYA.

This is the authentication panel, born to make the supervisors access the functionalities of the GetYourAssistant (GYA) application.
Supervisors will be able to see the statistics concerning the work done by the team of workers that they manage. These statistics are useful to understand which are the sectors of the company that should be revised, based on the performances of the workers and the latencies of the requests.



In the previous two images we can see the following: in the first one we have the login page for a supervisor in the supervisor web panel, while instead in the other one we have the homepage for the same page. The supervisor web panel, as already said, only shows relevant statistics for the supervisor role.

User API server:

General functioning:

The user API server is a component entirely developed using Java. In particular, to deploy it, it has been used the framework Spring Boot, which allows the deployment of a RESTful service, which is fitting a system with this purpose. The general requirements for this node were the ones of having it forward the requests sent to it by the user web panel. More in detail, he handles POST requests containing requests' data. Once it receives a JSON containing the relevant data, it adds only the current time of sending. It is needed that it is the server (and not the web panel) to add the current time, because it is synchronized with the receiving server called "primary API". In this way, when the elapsed time is calculated, we have that the time is coherent with the reality. Then, when the data is elaborated, he sends it to the RabbitMQ server, which is in charge of storing requests for asynchronous communication between the two API servers.

Coding structure:

This node's Docker image has been built starting from the original OpenJDK made by Alpine. In this way it has been sufficient to inject into it the jar files built from the Java application created, and launch the "java -jar" command to execute them. The Java application has been created starting from a Spring template (composable from the dedicated website) which was enabled for the RESTful server functionalities, but also for the direct communication with a RabbitMQ queue. In this way with a brief configuration and few lines of code it has been possible to make it receive the requests, add the current time and forward them to the RabbitMQ server. To deploy the system it was necessary to build the application (in the Java editor, in this case Netbeans was used) and then build and push the Docker image to DockerHub. In this way it was possible for Kubernetes to fetch the last version of the Java application considered.

Kubernetes deployment of the component:

In this case it wasn't needed further configuration of the Docker container once the image was fetched from the DockerHub, since it was already pre configured in a suitable way.

Component networking:

The component is exposing port 8081, which is handled by a ClusterIP service. This last one is then referenced by the ingress, which forwards the requests to the relative host, directly to the port mentioned below. Obviously only the HTTP POST method is supported to fetch the page, since it is needed to provide some data about the user to the API.

Graphical user interface:

No graphical user interface is provided by this component.

Primary API server:

General functioning:

The primary API server is in charge of retrieving the requests from the RabbitMQ server, when they're available. Once the worker web panel issues a request to this API server, it must send two messages, the first one, containing only logging information, to the ElasticSearch node, the second one instead directed to the worker web portal which actually asked for this output. It is useful since, being synchronized to the other API server, it allows to have a field named "time elapsed" useful for statistics purposes. It accepts HTTP POST requests where the requestor has to indicate its worker id, in order to log also his identity inside the system's statistics.

Coding structure:

This component has been realized starting from the original Docker image OpenJDK, developed by the organization Alpine. This has been only modified in order to inject inside the dedicated folder the jar files produced by the compilation of the code relative to this server API. Another obvious step needed was the one of building these files. Once this personalized Docker image is built and pushed to DockerHub, it is ready for deployment under Kubernetes. The Java code concerning this application takes as an origin a template coming from the Spring framework website, where only the RESTful and the RabbitMQ additional libraries were selected. This allows the server API to accomplish its three main communication tasks that are:

- receive the requests' data from the RabbitMQ server
- log data to the ElasticSearch node
- send the response to the Worker web panel POST HTTP requests

All these three tasks can be easily accomplished using the libraries mentioned above. For what concern the first there will be the RabbitMQ library providing automatic access to the queue providing simply its address. When this message is fetched, it is then saved into a local queue to make it available for the worker in the correct order when he requests it. Logging of the data can be done simply issuing a POST request directed to the ElasticSearch HTTP endpoint, that will contain indexes with the necessary information. Responses instead are sent to the worker web panel thanks to the remote controller which is enabled by the REST Spring library. In this case it simply manages the POST requests, providing as a response the first element waiting in the local queue (of the messages fetched from the RabbitMQ server). Important is also the operation of completing the information present inside the requests' messages, adding the time elapsed, calculated based on the current time minus the time contained already into the requests' data.

Kubernetes deployment of the component:

Once the Kubernetes deployment fetches the personalized image pushed on DockerHub, no further configurations are needed. This is because all has been set during the building process of the image.

Component networking:

The Docker image which is then built and executed, exposes the port 8082 as an entry point. However the ClusterIP intercepts this port exposing it as a service, and then the ingress references this last one to make it accessible through the localhost address. In this way the developers are able to create test calls using the Postman tool, which is able to create all the types of HTTP requests, generating also personalized content.

Graphical user interface:

No graphical user interface is provided by this component.

RabbitMQ server:

General functioning:

The RabbitMQ server is the fundamental component that enables the communication between the two API servers. This is useful to decouple these two nodes, in order to allow asynchronous communication between them and provide better reliability in the exchange of messages. The access to the RabbitMQ configuration panel requires a login access, and the credentials are provided during the configuration procedure. Also in this case this procedure for establishing credentials is not completely secure, but it has been done for testing purposes, without deepening the security aspects that would be necessary in a real development environment.

Configuration:

This RabbitMQ server has been built from a personalized Docker image. This last one has been created based on the RabbitMQ (management version) original image. After considering this image as a solid base, the configuration files have been pushed inside the dedicated folder for this typology of files. The configuration files of this deployment have been obtained through the exporting feature of the RabbitMQ panel. In this way it has been necessary to configure the deployment only one time, and after importing the configuration inside the personalized image to obtain things like credentials and queues already set up.

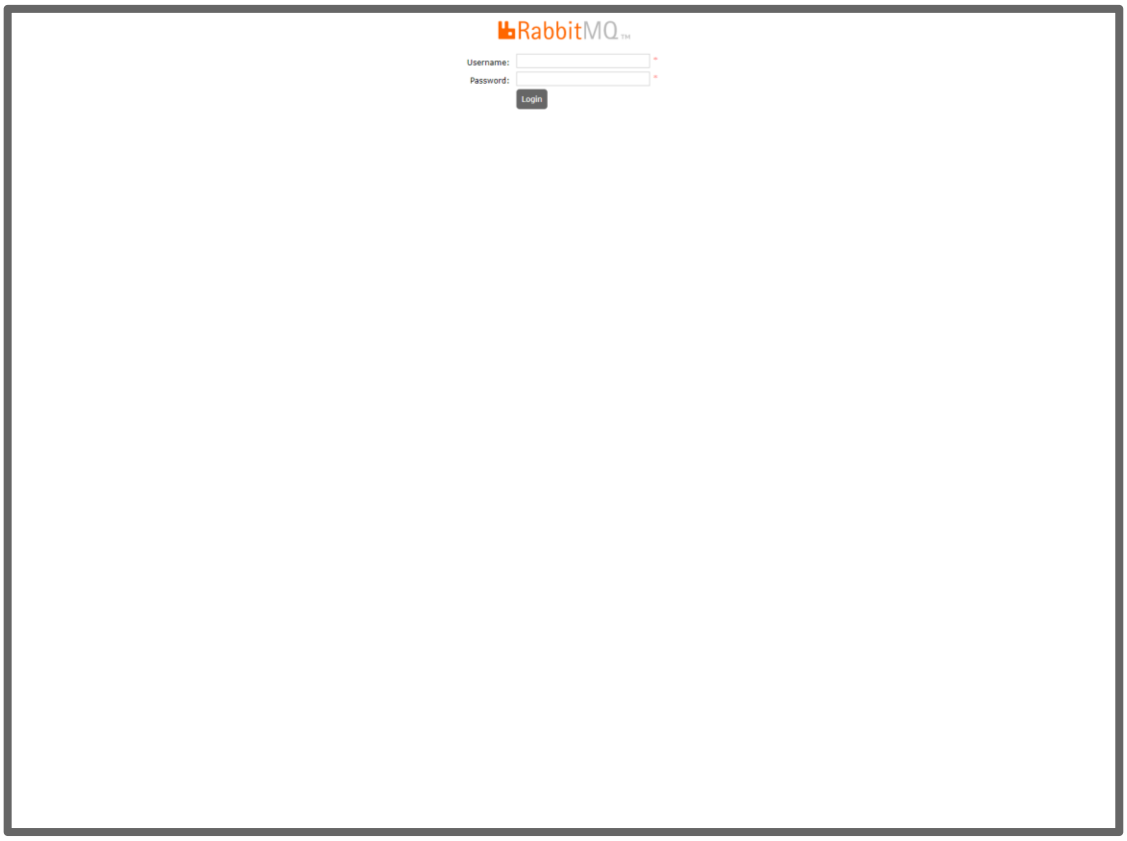
Kubernetes deployment of the component:

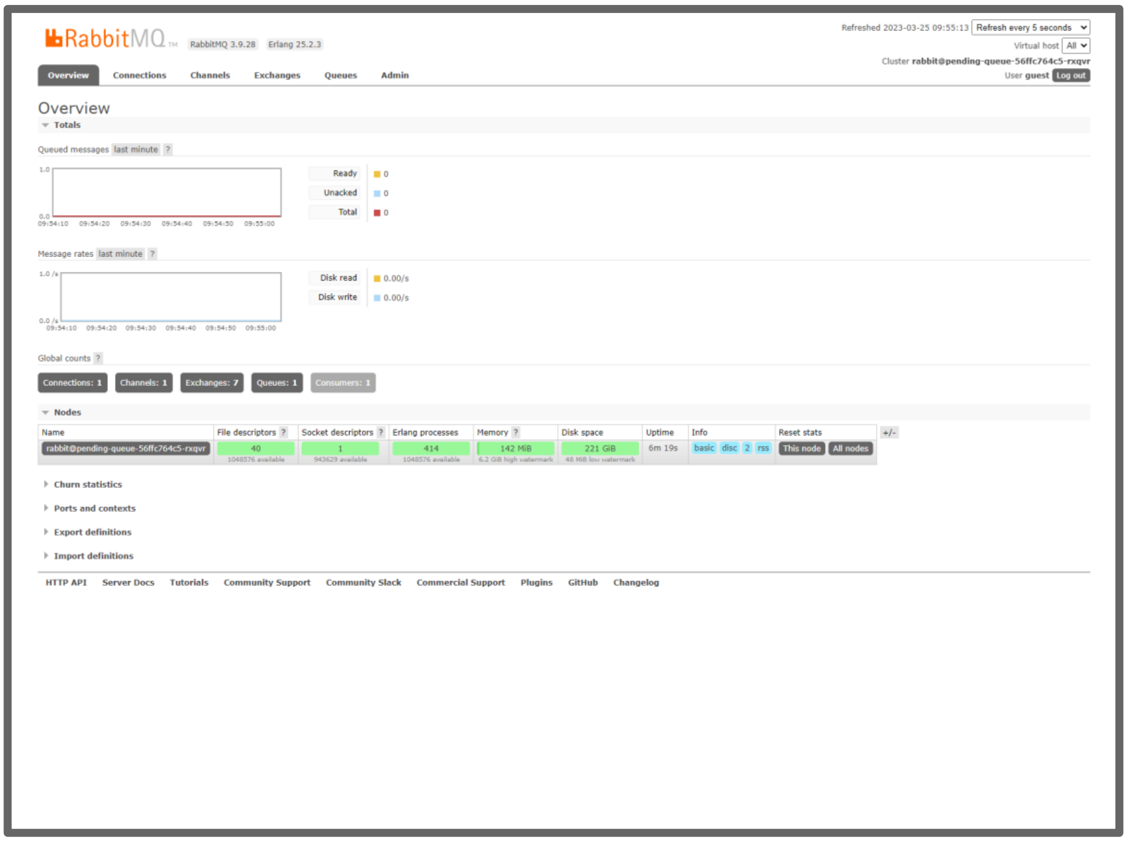
The Kubernetes deployment is pretty simple since it’s needed to fetch the image from the personalized GYA’s DockerHub. Further configurations are not needed since they have been completed in the building phase of the personalized Docker image.

Component networking:

The RabbitMQ server node exposes two different ports as entry points, one (5672) is used for the purpose of accessing the queue for communications and inserting elements, while the second one (15672), also called management port, instead is used to access the web portal exposing a graphical user interface allowing the visitors to easily modify settings.

Graphical user interface:





Refreshed 2023-03-25 09:55:13 Refresh every 5 seconds

Virtual host All

Cluster **rabbit@pending-queue-56ffc764c5-rxqvr**

User **guest**

Overview

Queued messages last minute ?

Ready	0
Unacked	0
Total	0

Message rates last minute ?

Disk read	0.00/s
Disk write	0.00/s

Global counts ?

Connections: 1 Channels: 1 Exchanges: 7 Queues: 1 Consumers: 1

Nodes

Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Info	Reset stats
rabbit@pending-queue-56ffc764c5-rxqvr	40 1048576 available	1 943629 available	414 1048576 available	142 MiB 6.2 GiB high watermark	221 GiB 48 MiB low watermark	6m 19s	basic disc 2 rss	This node All nodes

In the previous images it’s possible to see the interface for RabbitMQ's server management. It is possible to observe the current queue’s work, and modify some settings about them. All of this is easily possible thanks to an intuitive interface.

ElasticSearch server:

General functioning:

The ElasticSearch server provides an endpoint where it’s possible to store in a NoSQL approach the data concerning logging for statistics. In particular this will be adopted to make Kibana (able to interact with it) fetch the relevant data and show it using dashboards and views. The data is provided to ElasticSearch through POST requests coming from the Primary API. This is in fact the only source of information that the ES server receives, since a complete view of a request’ elaboration is possible only at the end of the process. In this case it has been provided also some sample data to that node, in order to have some statistics to be shown, directly when the application is deployed, so only for testing purposes.

Configuration:

This is a deployment that needs further configurations after deployment. It is in fact needed to perform the following steps after deployment, and they are performed into the bash script made for automatizing the deployment:

- get the API key from the security section of the ElasticSearch URL
- configure the mappings for the geographical point field named “location”
- push the sample entries in the log database

The first configuration step is required to use the obtained API key in the configuration of the Kibana deployment. It is in fact strictly required an API key, certifying that you’re the owner of the development, to proceed in the Kibana dashboard configuration. The second step of the configuration is needed in order to enable Kibana dashboard to embed a map, showing markers in correspondence to the index entries. Maps require then mappings in index fields, otherwise the Kibana system wouldn’t know which field is representable in that specific format. The last point instead is needed only for testing purposes, it in fact enables the developer to check if the statistics are working correctly thanks to some randomly generated data. Notice, the data provided is not always realistic, being random, it could happen that some requests’ location are placed in the middle of the oceans, so consider it carefully.

Kubernetes deployment of the component:

In this case we have a deployment kind provided by the ElasticSearch producers. In order to use it however it is required to apply a deployment containing all the required components for it to work correctly. Inside the deployment string, we also provided the option for the users to login with an anonymous account, in this way it is possible to access Kibana dashboards without providing credentials. As I already mentioned, a configuration is obviously necessary after the deployment, and it is applied into the bash script used to deploy the entire system, together with the application of the k8s components for this kind of deployment.

Component networking:

The ElasticSearch deployment exposes already a service, which can be referenced by the ingress directly, without the need of creating one ourselves. Many access points are created for ElasticSearch but we're interested in exposing under the localhost address only one of them, the HTTP one. It is needed to be exposed to the developers mainly for the configuration purposes, but also to basically check if the environment is correctly configured and working. The components communicating directly with ElasticSearch won't use this access point, but they will use the network created internally to Kubernetes deployments, in order to make the nodes communicate through DNS entry names associated with their deployments' name.

Graphical user interface:

No graphical user interface interface is offered by this component.

Kibana server:

General functioning:

The Kibana server will be strictly connected to the ElasticSearch deployment. It is needed to show to the user some dashboards containing the statistics needed in order to visualize general performances of the system. Users (through the User web panel) will see the average data for requests' answering throughout the day, while the supervisors instead will be able to check how the workers of the company are performing during the working days. Kibana dashboards will be produced using the Kibana interface, and then exported into iFrames to be embedded into the different portals. This operation requires two things: HTTPS connection, so an SSL certificate was provided, and a configuration procedure to enable anonymous login and create the dashboards needed to the system directly during the deployment phase.

Configuration:

The configurations needed for this type of deployment (once the one to ElasticSearch has been done) are simple. In fact it is needed only to load the configuration file into the deployment through an HTTP request. The configuration file has been generated through the exporting functionality available at the Kibana website access point.

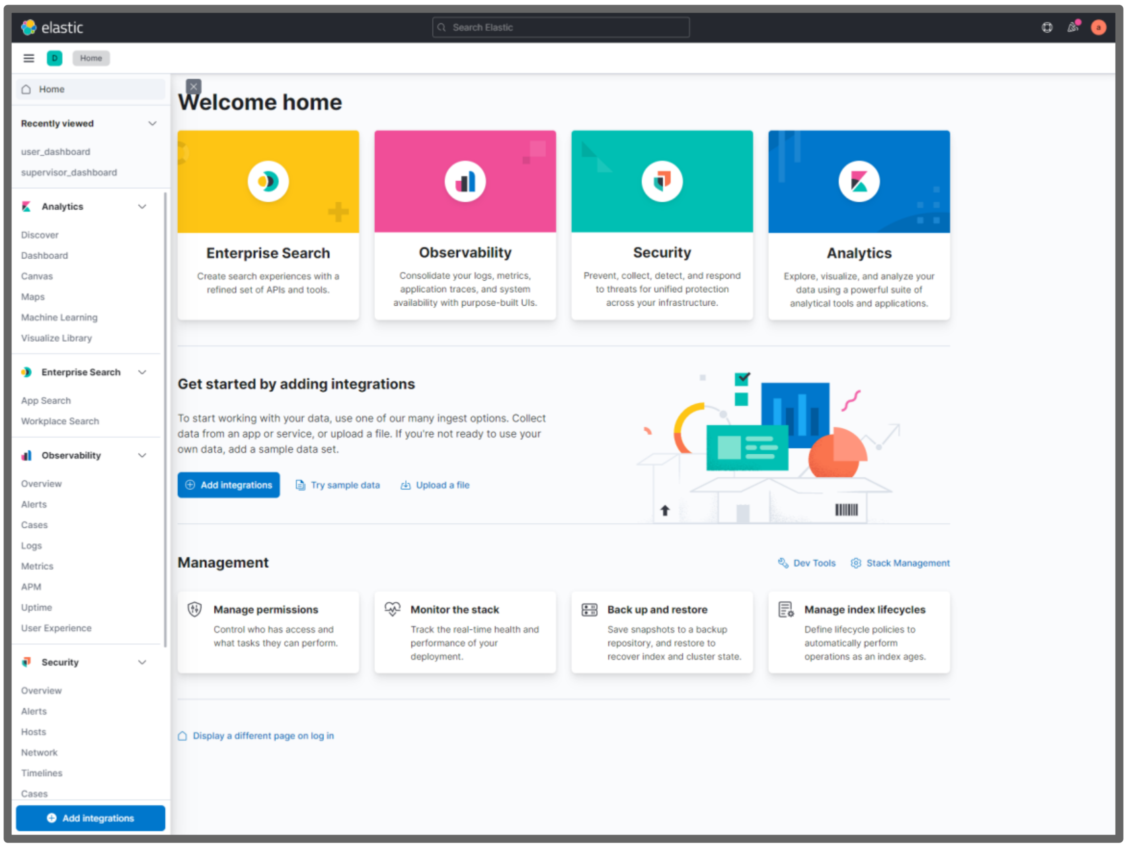
Kubernetes deployment of the component:

In this case it is required to apply an ElasticSearch particular deployment before creating the one of Kibana. It is needed in order to deploy the components necessary to deploy the needed one. Once this operation is performed, the deployment string of Kibana is pretty standard, only some basic configurations have been added, like for example the automatic login functionality, so the anonymous user, and also the option enabling the use of self-signed certificates to expose Kibana as an HTTPS entry point.

Component networking:

The Kibana deployment already exposes the different services needed to make it available for external access. The ingress can then reference the HTTP service for the Kibana deployment to expose it for developers' use. They can, at the end, edit Kibana dashboards using an easy interface and modify some settings to improve the system's statistics quality.

Graphical user interface:



Create search experiences with a refined set of APIs and tools.

Consolidate your logs, metrics, application traces, and system availability with purpose-built UIs.

Prevent, collect, detect, and respond to threats for unified protection across your infrastructure.

Explore, visualize, and analyze your data using a powerful suite of analytical tools and applications.

To start working with your data, use one of our many ingest options. Collect data from an app or service, or upload a file. If you're not ready to use your own data, add a sample data set.

Control who has access and what tasks they can perform.

Track the real-time health and performance of your deployment.

Save snapshots to a backup repository, and restore to recover index and cluster state.

Define lifecycle policies to automatically perform operations as an index ages.

In this image we see the homepage of the Kibana server, where configurations can take place thanks to an intuitive interface.

PostgreSQL database:

Configuration:

The PostgreSQL database, after the deployment, is configured using the bash command “psql” able to perform some DDL queries on the database endpoint. In this case, foreach database it has been necessary to execute two basic queries. The first one to define the users table, containing only the credentials and some general information about the profiles. The second one instead is to insert a sample credential to allow the developers to access without registering in all the three web portals.

Kubernetes deployment of the component:

This specific type of deployment needs to be connected to a PersistentVolume (through a PersistentVolumeClaim) in order to maintain the database state even after undeploying the entire system. The Docker Image from which the deployment has been applied is the basic PostgreSQL one. It has been simply necessary to add some environment variables containing data like: the initial database name, the Postgres user able to access it and its password. In this case the password has been saved in clear only for developing purposes.

Component networking:

The component exposes the container port 5432, and it is intercepted by the NodePort service that exposes this TCP application. Since we have a NodePort it is possible (from whatever address) to access the service, through the port associated with the container's one. In this case a NodePort was necessary since we couldn't pass through the Ingress, which is only able to expose HTTP services.

How to deploy the system for the first time:

The following explanation is valid for a Linux system. Actually the deployment has been tested on a Linux system virtualized over a Windows 11 distribution.

Requirements

- Docker (shell "docker" utility)
- Kubernetes (shell "kubectl" utility)
- CURL (shell "curl" utility)
- JQ (shell "jq" utility)
- PostgreSQL client (shell "pgsql" utility)
- Root privileges

First configuration and deployment

The following commands have to be executed after starting Docker and Kubernetes.

1. Clone the repository executing the following CLI command

```
git clone https://github.com/RicMash/GetYourAssistant.git
```

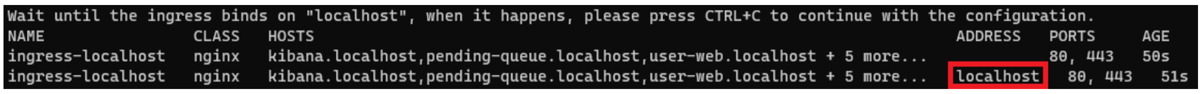
2. Move into the cloned repository directory

```
cd GetYourAssistant/
```

3. Deploy the application using the bash file provided

```
./deploy_gya.sh
```

4. Wait until the following the content of the red square in the following image is displayed

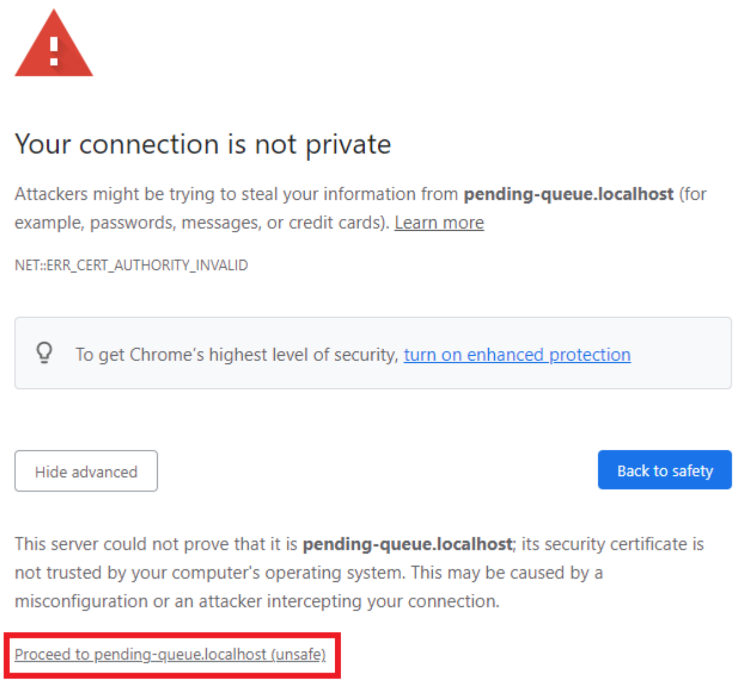


```
Wait until the ingress binds on "localhost", when it happens, please press CTRL+C to continue with the configuration.
NAME                CLASS   HOSTS                                                          ADDRESS     PORTS     AGE
ingress-localhost   nginx   kibana.localhost,pending-queue.localhost,user-web.localhost + 5 more...             80, 443   50s
ingress-localhost   nginx   kibana.localhost,pending-queue.localhost,user-web.localhost + 5 more...   localhost   80, 443   51s
```

5. Press "CTRL + C" to continue with the deployment and wait until the end
6. Add the following DNS entries to the file with path "etc/hosts"

```
127.0.0.1 kibana.localhost
127.0.0.1 primary.localhost
127.0.0.1 user-api.localhost
127.0.0.1 user-web.localhost
127.0.0.1 pending-queue.localhost
127.0.0.1 elasticsearch.localhost
127.0.0.1 supervisor-web.localhost
127.0.0.1 worker-web.localhost
```

7. Add the SSL certificates to the local store to make the domains be considered as secure. The following steps are shown using Chrome, and they have to be reproduced for each of the following domains:
    1. https://user-web.localhost
    2. https://supervisor-web.localhost
    3. https://worker-web.localhost
    4. https://primary.localhost
    5. https://user-api.localhost
    6. https://pending-queue.localhost
    7. https://elasticsearch.localhost
    8. https://kibana.localhost
8. Procedure (**to apply for each domain**):
    1. Browse the domain using the browser
    2. Download the certificate of the website (red squares in the following images indicate where you'll have to click):



Your connection is not private

Attackers might be trying to steal your information from **pending-queue.localhost** (for example, passwords, messages, or credit cards). Learn more

NET::ERR_CERT_AUTHORITY_INVALID

To get Chrome's highest level of security, turn on enhanced protection

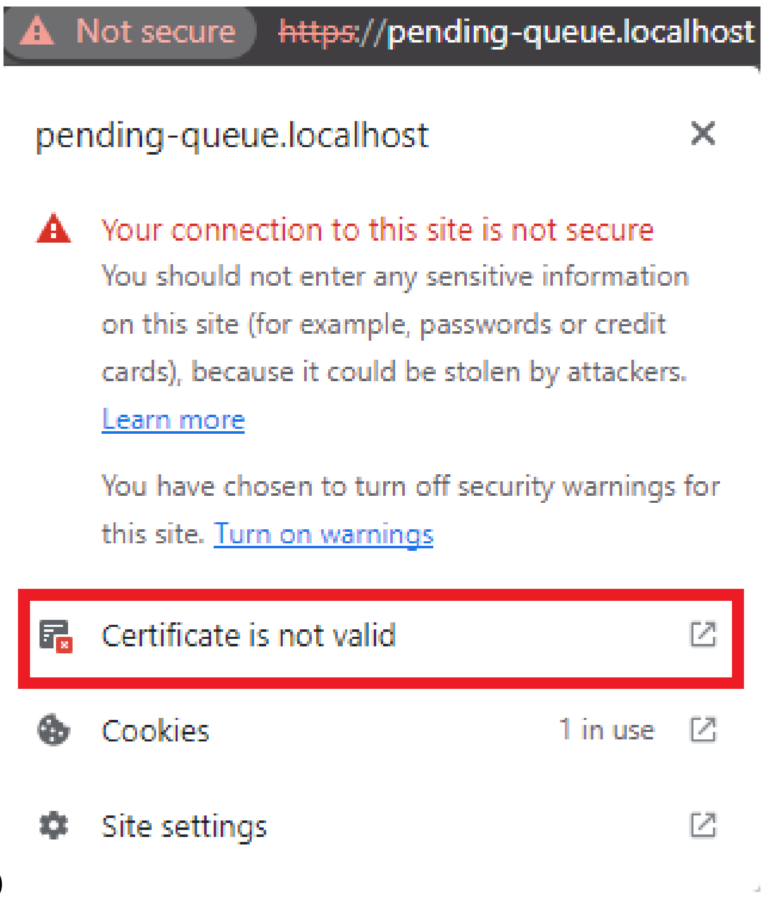
Hide advanced

Back to safety

This server could not prove that it is **pending-queue.localhost**; its security certificate is not trusted by your computer's operating system. This may be caused by a misconfiguration or an attacker intercepting your connection.

Proceed to pending-queue.localhost (unsafe)

i)



Not secure ~~https~~://pending-queue.localhost

pending-queue.localhost

Your connection to this site is not secure

You should not enter any sensitive information on this site (for example, passwords or credit cards), because it could be stolen by attackers. Learn more

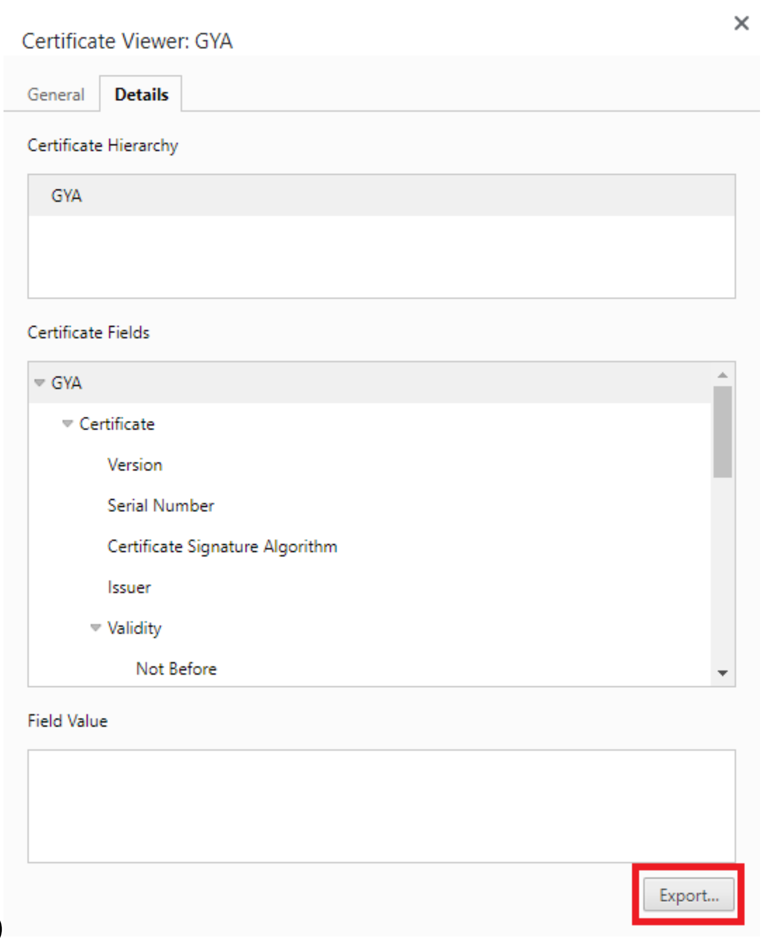
You have chosen to turn off security warnings for this site. Turn on warnings

Certificate is not valid

Cookies 1 in use

Site settings

ii)

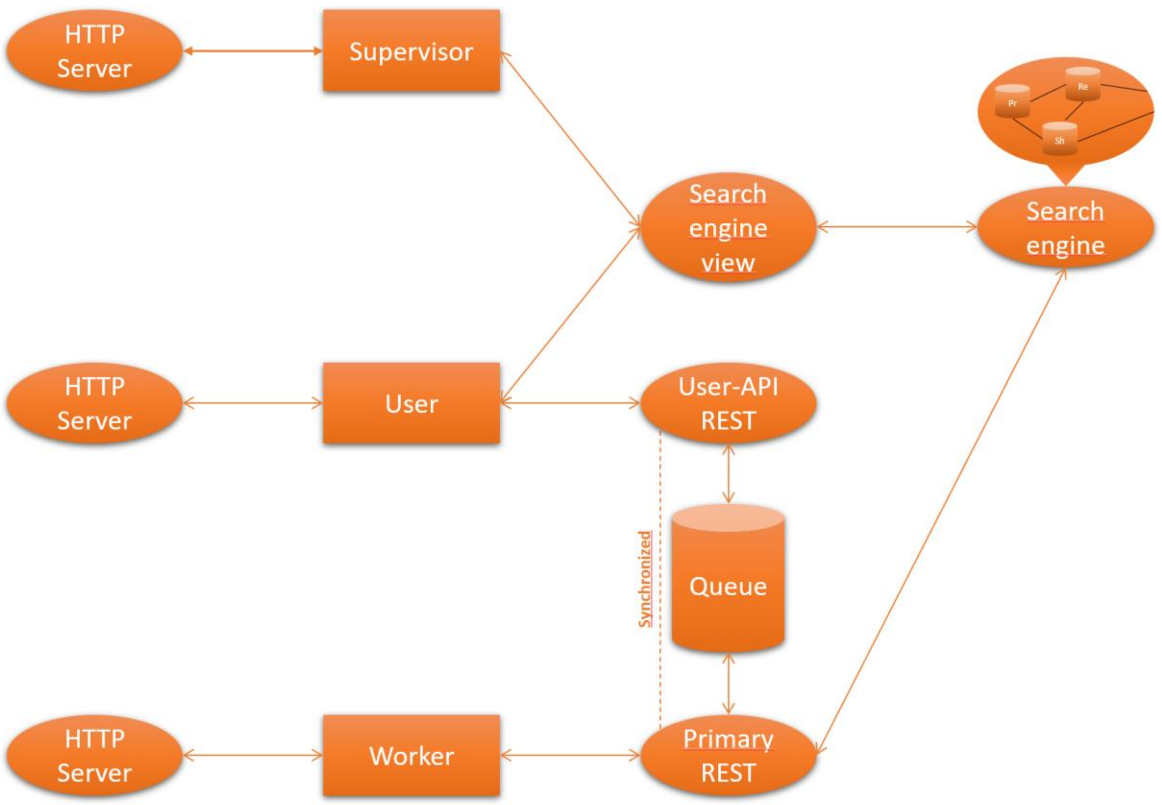


iii)

3. Install the certificate using the file downloaded

Now the setup should be ready to be used.
NOTICE: Steps 1, 2, 6 and 7 are needed only the first time the application is deployed.

Final picture of the system’s architecture:


HTTP Server
Supervisor
Pr
Re
Sh
Search engine view
Search engine
HTTP Server
User
User-API REST
Synchronized
Queue
HTTP Server
Worker
Primary REST