19294349

COMP7033

Design Docs

GUI DESIGN

A white sheet of paper with purple writing

Description automatically generated

A rough initial mock up is seen above. First column from the left is a selection of services, the second column is a selection of functions using the selected service. The third and fourth columns would be populated by content from the requests made to and from the selected service. Calls to edit or create new records could use a pop-up window for data entry or use the fourth column if it’s free.

A prototype was built with some minimal functionality using the tkinter module within python3:

A screenshot of a login box

Description automatically generated

This is the first window the user sees. A prompt to login that is connected to the cloud to perform verification and authorisation for the user account.

A screenshot of a login box

Description automatically generated

Clicking login here makes a call to the admin-auth service with a GET[/login, params=(username=testuser, password=testuser)]. The admin-auth service checks with the mongo database that these sent details match stored details, upon seeing that they do the service creates a session token and sends it back to the application for all further authentication requests this session. If these credentials did not match stored data, the server would return JSON containing the string: “"username or password incorrect".

On successful response from the service the user sees:

A screenshot of a phone

Description automatically generated

“Managing Adminstration Staff” is the default state of the GUI. The first column of buttons is the highest tier of menu selection. The second column containing “Show All Admins”, and “Create New Admin” is the next layer of menu. When the user selects “Show All Admins”, the admin-staff service is contacted with a GET[/, params=(token=token)]. This API takes the call along with a session token, contacts the admin-auth service to validate the token with a GET[/, params=(token=token)] and when successful searches the database for all admin records and returns them in JSON format.

A screenshot of a computer

Description automatically generated

PROTOTYPE GUI!

This shows the returned list admin records. The “Create New Admin” button produces a form which the user can fill in to add a new user to the system if the logged in user has sufficient permissions:

A screenshot of a phone

Description automatically generated

Clicking on the “register new admin” button sends a POST[/, params=(username=username, password=password, admin\_name = name, admin\_permission=permission\_level, token=token)] to the admin-staff service. This checks with the admin-auth service that the token is valid with a GET[/, params=(token=token)], and checks that the user with the token has permission to create new admin records. When these are true it then adds a new record containing these details to the admin staff database and to the admin login database with the admin-auth service with a GET{/register/newuser, params=(username=username, password=password)].

This is the extent of implemented functionality in this current prototype.

Ideal GUI would have more button for more menu selection, better visuals for the returned data and allowing infix editing and updating of this data. Implement validation for the input fields such as username only containing a limited set of chars, or passwords having to fulfil certain criteria such as length and character variance. A focus on accessibility would be a priority. Internationalization and locale management would also be needed if the software was to be used in other countries.

A diagram of a cloud

Description automatically generatedThis is the general theoretical design of the app with:

A white rectangular sign with black text

Description automatically generated being a general representation for accessing the internet. All of the microservices are accessed on their own IP addresses following this schema:

A sign with black text

Description automatically generated = A diagram of a user account manager

Description automatically generated

Every service represented like:

A white sign with black text

Description automatically generated

Is handled as:

A diagram of a system

Description automatically generated

Quality Requirements

In the cloud each micro-service is accessed at the external ip address of its load balancer. Each micro-service is being continually monitored to: check for crashes, in which case it attempts to reboot the docker container for the service; check for a need for horizontal scaling, where if the load to the service is high it can spin up more Kubernetes pods with the service docker running inside, and if the load is reducing it can drain a pod to reduce CPU and memory burden on the server while never going below one pod to maintain uptime. Functionality for vertical scaling, i.e. adding more hardware resources per node is available in the google cloud service but not used in this current testing version.

The databases are stored in the cloud using mongo atlas and have replicas, allowing for more access to read operations for services while still enforcing atomic principles of changes to the database across its replica nodes. This allows many micro-service instances to have read access to data (which will be the most common request due to the application requiring get requests so that a user can then select a record to edit) which helps meet scalability and reliability requirements.

From the previous Coursework: on performance -

“In the context of ClaClo we will be serving data and responses in a similar fashion to website browsing so users will expect similar results. This means the goal would be to have response times of less than 500ms for small requests, and 3 seconds for larger requests.”

Small requests would be simple tasks such as validating the user or requesting all admins, as this requires only reading an entire collection from the database, no editing or searching. Larger requests would be editing student course assignments, as this would need to use multiple micro-services to accomplish.

The database being stored on separate servers with separate infrastructure will lead to worse performance due to latency but allows simpler implementation and would allow for greater reliability and scaling as other services could be used to host the micro-services if needed and they could still access the same database infrastructure independently.

From the previous Coursework: on scalability –

“The service needs to be able to scale horizontally to serve the required users. If we were limited to a single application on one server handling all requests, it would buckle under high load. This can be achieved through using a load balancer to distribute the tasks between multiple instances of the same service, coupled with a load manager to spin up new services when needed or spin down when not. The ClaClo system would be UK based and looking to have 20 Universities join up within the first year. If we assume ~25,000 users per university on average, 20,000 being students, the rest being admin staff or teachers, this gives us the goal of being able to cope with ~500,000 users within the first year. The service would be in constant use throughout the academic year, it would have large spikes of usage as the year begins in September when students need to enrol and courses need to be set up. Another spike in periods of examination with work being submitted and results/feedback being returned. So we would need to build into the design the capability to serve all these users in small periods of time at certain points of the year.”

As previously mentioned the Kubernetes cluster will auto scale horizontally to increase the number of pods running each microservice as demand increases.

A screenshot of a phone

Description automatically generated

10 requests a second across 2 active pods gave peak usage of 7.5% when averaged, requested by Kubernetes on this google cluster. It is hard to see regular usage of this admin service leading to saturation of more than 3 pods capacity on this cluster. Perhaps overnight batch processing of 100 requests a second could be the biggest use of resources for a university but still would be within scope.

From previous Coursework: Reliability -

“Reliability is a metric of how much the service can provide it’s task throughout a given period. Technical issues, bugs or power outages can always occur and disrupt service uptime but obviously are ideally minimised. For the ClaClo system the aim would be for 24 hour 7 days a week uptime with no more than 3 hours of contiguous downtime during peak periods and no more than 24 hours of contiguous downtime during slower periods. Over a whole year it should have no more than 100 hours of downtime. This requires good ops teams with solid domain knowledge as well and potentially strong technical support from ISPs and server hosts to help rapidly fix any issues.”

Reliability can be helped by having a secondary cluster which is monitoring the primary. When an issue is detected, the secondary cluster can use Kubernetes to spin up pods with the micro-services which have failed to maintain operation. This can also be used to prevent maintenance on the primary cluster from limiting uptime for the service. The database has replicas which should help to mitigate the risks of full downtime. The load balancer and monitoring on the primary cluster itself can also help achieve reliability due to killing crashing or failing services and starting fresh instances to replace them. As the micro-services are stateless themselves, it doesn’t matter which instance of one serves a user’s requests, the tokens to authenticate and the parameters sent are the only thing needed to fulfil API requests. The micro-services also use async calls to the database so that if anything else can be done while the database response is awaited, it can. If an error occurs or validation checks are not valid, the micro-service returns text describing the issue such as “Invalid Token” if the token being used has timed out or isn’t in use, "Insufficient Privileges" if the admin user does not has the right permission level to perform the requested task, or "failed to update teacher in API" if the service isn’t able to update a teacher record through the API.

Implementation

Along with the prototype GUI, only the admin-auth micro-service, the admin-staff micro-service, and a teacher-stub microservice have been implemented. Each micro-service provides separate functions in a stateless fashion through RESTful API implemented with python based FastAPI. They are hosted with the google cloud using their Kubernetes implementation allowing for auto-scaling both horizontally and vertically. This also provides the load balancer for controlling ingress and monitoring for restarting faulty containers and can send alerts to system admins when issues occur. The system uses a mongo atlas hosted MongoDB to store its persistent data. NoSQL provides a good platform for scaling up with large amounts of data readily and atlas provides replication of data nodes allowing for many micro-services to read data at once without slowing each other down.

Collaboration

Unfortunately, the group didn’t seem to want to collaborate at all this time. It was rough trying to get anything the first-time round, but this was a no go. This led to me not having any other part of a system to use or test with, so I made a teacher-stub service just to do some very light testing with.