Technical Risk Assessment

Detailed Code Report & Recommendations

Report assembled for

Astral Yeti Energy

Analysis conducted by
Serene Yew

On the date of
January 25, 2024

PIXELTREE

Table of Contents

Executive Summary

During the course of this audit, we have reviewed all the codebases, collected infrastructure details in one place, and distilled the project setup instructions so that the codebases are easier to set up on new developer machines.

There are no glaring items that indicate that the app is not ready to be submitted to the App Store at this time. However, during the application process, unanticipated items are expected to arise. We recommend starting this process as soon as possible so that we can get feedback to work from.

While the app does work as is, and the code is in decent shape, there are a number of items that put Inspectagram at risk of data breach, as well as some that will significantly slow down turnaround time of development, and increase the risk of unknown bugs appearing.

We have determined a list of high priority security items that should be addressed as soon as possible, and come up with a proposed go-forward plan.

See Recommendations.

Codebases

api-gateway

api-gatewayhttps://github.com/astralyetienergy/api-gateway

This repository holds the code that hosts the API. It appears to host some endpoints that interact with the MySQL database.

report-gateway

https://github.com/astralyetienergy/report-gateway

This repository holds an application that renders the reports and generates the PDF files.

mobile-app-v3

https://github.com/astralyetienergy

This repository contains the React Native code for the mobile app.

realm-config

https://github.com/astralyetienergy/realm-config

This appears to be some kind of repository that is being used by a continuous integration process that contains the configuration for the realm database.

mobile-app

https://github.com/astralyetienergy/mobile-app

This appears to be an old version of the mobile app that is no longer in use.

image-cdn

https://github.com/astralyetienergy/image-cdn

This repository contains some code to host images on a CDN, although we were unable to determine if it's currently being used.

realm-backend

https://github.com/astralyetienergy/realm-backend

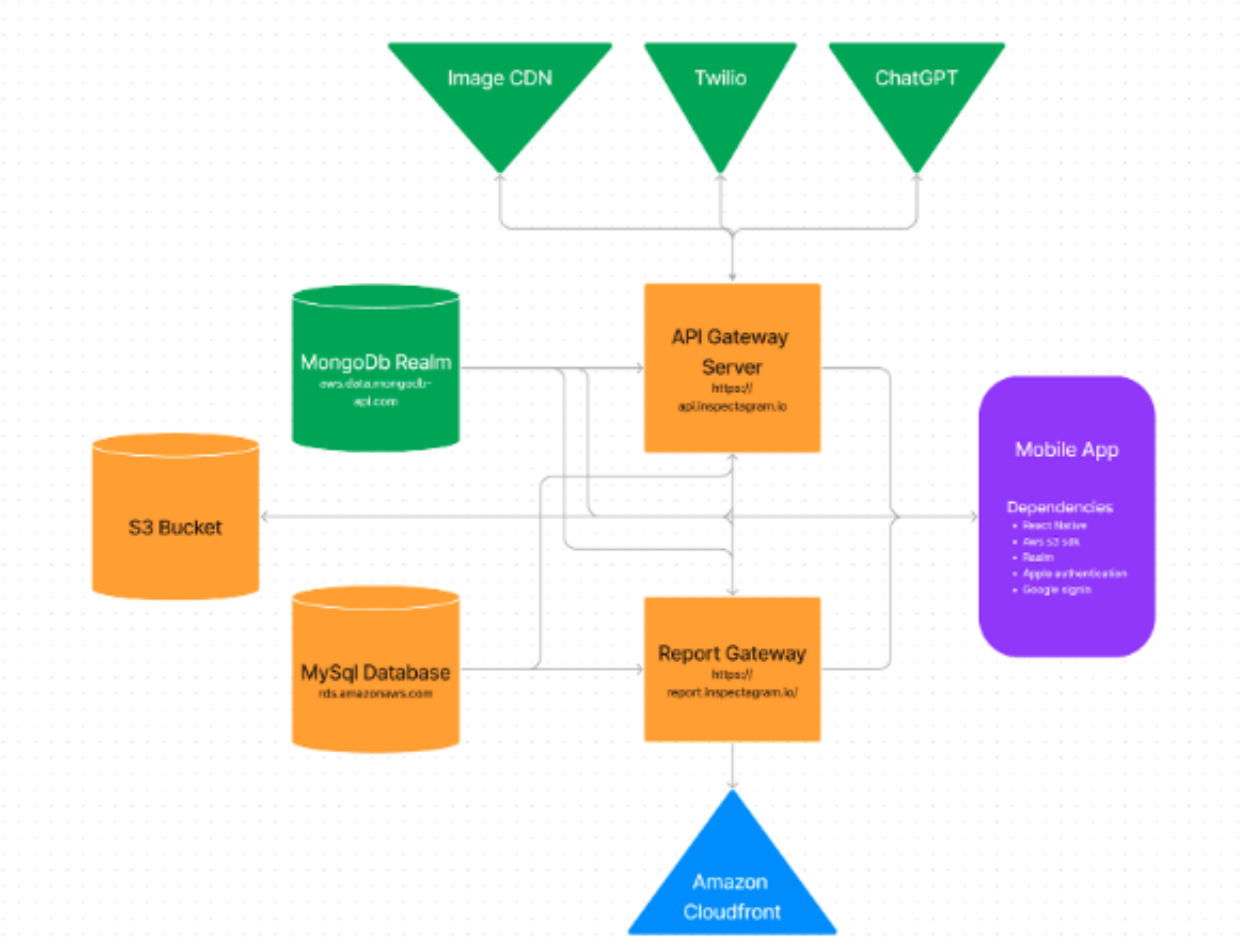
This repository appears to be an older version of the mobile app that is no longer in use.

Deployment

Currently, the deployment process is manual. The developers upload the code to the production server and restarts the web server. The mobile server is built and then distributed through Testflight.

See more in Risks.

Architecture



The architecture is mostly reasonable for the features implemented. The MongoDB instance is functional, but comes with challenges for testing. Because of the way that the MongoDB instance was built, they had to create a separate MySQL database to hold data that doesn't need to be regularly synced with the mobile app.

The report gateway is on really large EC2 boxes to be able to support report generation. It's unclear at this time if this could be optimized with a different implementation.

The S3 bucket stores generated PDFs. This is in line with industry best practices.

Risks

Credentials at Risk

Currently, there are quite a number of secure credentials in the code repository. While the repositories are private, this is considered unsafe. Repositories are breached fairly often, usually without the owner's knowledge. These credentials need to be reissued and moved to more conventionally secure locations, like environment variables. (See report-gateway, mobile-app-v3, api-gateway)

Also found were database connection credentials (api-gateway) and S3 bucket credentials (report-gateway).

Certificates in repository

As above, credentials should never be in the code repository. These are the certificates that are used to authenticate your application with Apple. If these credentials are compromised, a third-party would be able to impersonate you and replace your app. (Reference mobile-app-v3)

Plaintext Password Storage

> Note that this only applies to the email/password logins, not the ones that use Apple Sign-in.

Typically, passwords are saved as a one-way hash. It is done this way such that if your database is breached, the attacker is unable to use the information found to log in to your system, or others where the passwords may also be used. On your codebases, the default hashing behavior has been overridden so that the passwords are saved and compared in plaintext. (See report-gateway, api-gateway).

Use of Production Database from all Environments

There is no separation of the development environment from staging (testing) or production environments. This can be quite risky as it puts your live production data at risk of being deleted, modified, or even just out of sync with the code. Typically, there are separate environments for development (local environments that developers use for active coding), staging (a production-like environment where we can test pre-production code), and production (what the end users see).

Risks continued

PHP/Symfony Versions

Both the api-gateway and report-gateway are on PHP version 7.4, which was End-of-Life last November. They need to be upgraded to 8.2 to receive the new security patches. Symfony would need to be upgraded from 6.2 to 6.4 in order to support PHP 8.2.

Certificates in repository

As above, credentials should never be in the code repository. These are the certificates that are used to authenticate your application with Apple. If these credentials are compromised, a third-party would be able to impersonate you and replace your app. (Reference mobile-app-v3)

No Production Deployment Management

There appears to be no managed deployment process for pushing code to the production server. This means that it is not possible to track deployments, rollback in the event of a failure, or only swap to the new version on a successful deployment.

No Automated Tests

There are no automated test suites on any of the repositories, which makes it challenging for the development team to ensure that the product will continue to function as expected when changes and additions are made. Without automated test suites, it will be challenging to perform library updates with confidence that nothing will change or break.

Large Number of EOL Dependencies

While there are a number of out of date dependencies, there is a quick fix to update most of the high priority ones. However, without an automated test suite, it's impossible to ensure that the application will continue to function exactly the same way after the upgrades.

God Mode

God Mode was easily accessible and the pin to access the God mode login screen is stored and hardcoded inside of the mobile app code base. (See mobile-app-v3). This is quite easily breachable. iOS apps can be reverse engineered, and it is unsafe for credentials to be part of the build.

Recommendations | High Priority

Create smoke test suites

This is really the highest priority item to ensure that the team is able to iterate quickly on changes and detect breakage before code goes into production. Without a test suite, every change or addition risks breaking functionality for users without anyone knowing until an end user reports it.

There is a huge range of how much we can do here. The most basic implementation would involve creating test suite structures for the api-gateway, mobile-app-v3, and report-gateway, then writing smoke tests to ensure that each project continues to compile.

Cost: $10,000+
Timeline: 2+ weeks

All the following estimates are provided with the assunmption that we first implement at least smoke test suits.

Create a production deployment process

We propose creating docker builds that can be managed with Cloud66 so that we can track deployments, the changes made between deployments, and easily rollback failed deployments.

Cost: $6,000
Timeline: 2 weeks

Fix battery drain problem on iPhone 12

There seems to be an issue with battery drain on the iPhone 12. We would need to run diagnostics and monitoring to be able to determine the cause of the problem.

Cost and timeline are unknown until we determine the cause of the issue.

Separate development and staging environments from production

Develop separate or mock instances so that code can be tested independently of the production databases. This will allow developers to work without risking production data and for QA testers to test new functionality first without users being impacted.

We believe that we can create a scaled down staging environment that will be similar to the production environment with a lower cost. However, there is still a lot of discovery to be done here.

Cost: $10,000 - $20,000
Timeline: 4 weeks

Update libraries with critical updates

Security is of utmost importance in any codebase. A single data breach could destroy a company. Libraries need to be kept up to date in order to ensure that security patches are applied promptly.

Cost of performing the upgrade: $1,000
Cost of regression testing the application (assuming we have a separate testing server): $8,000
Timeline: 3 weeks

Upgrade PHP/Symfony

Similar to the above, frameworks need to be kept up to date. End-of-life framework versions mean that security patches are not even provided anymore, which puts your application at risk.

Here, we would upgrade both PHP and Symfony for the api-gateway and report-gateway code bases.

Cost of performing the upgrade: $2,000
Cost of regression testing the application (assuming we have a separate testing server): $8,000
Timeline: 3 weeks

Recommendations | Medium Priority

Moving credentials out of the repository

The credentials in the repository that may have already been compromised need to be rotated and moved to secure locations, like environment variables.

Cost: $2,000
Timeline: 1 week

Issue new certificates for the app store

The certificate in the repository needs to be rotated and replaced with fresh certificates that are stored in a secure location.

Cost: $2,000
Timeline: 1 week

Move god mode out of the mobile app

Since mobile apps can be reverse engineered, it's dangerous to have information in the code that can be used to access any administrative accounts. This would require building a separate web application, and then removing the god mode access.

Cost and timeline unknown until this feature is better defined.

Hash passwords

Replace the plaintext passwords in the databases with properly hashed passwords. This was likely done this way to support authentication across the services, so there will probably be some challenges in migrating the content.

Cost and timeline unknown until more testing is done on a staging server. Making this change on the production server can accidentally break access.

Recommendations | Nice to Have

Codebase says Copyright Mobility Quotient Solutions in a few places

We need to check if this is true or if it's something we can change, or if it's something we need to credit.

Cost: < $500
Timeline: < 1 day

Add new documentation to Github repository wikis

While performing the audit, we documented clean project setup processes to add to the documentation. Currently, the Github organization does not support Wikis. Ideally, we would turn this on and add it there, extracting the content currently in the readme and structure it into something more usable long term. We would also remove documentation that does not work.

Cost: < $500
Timeline: < 1 day

Move images out of MongoDB

About 400mb of images are currently being stored in MongoDB. This is not conventional but seems to be working right now. If there are more images to be added, we would recommend moving them to something like S3. However, it seems to be working well enough for now that the cost to change this implementation would not be justifiable at this time.

Cost: $5,000+
Timeline: 3 weeks

This concludes the technical risk assessment for

Astral Yeti Energy

This assessment was conducted by Pixeltree Inc.

PIXELTREE

Contact info
www.pixeltree.ca
serene@pixeltree.ca