Ad-supported Digital Signage System

Short project description

A complex of web applications and hardware devices (terminals) that help businesses to generate passive income from their digital screens. AdStash turns businesses' TVs or digital screens into advertising platforms that can show different kinds of content (images, videos, and RSS feeds) mixed with ads.
Our team was automating routine business processes, resolving technical issues as customer support, stabilized the application and prepared it for a high load, and improved the code base by covering it with tests.
Tech stack: Ruby on Rails, Postgre SQL and Redis DBs, Delayed job, Que and Sidekiq job schedulers, AWS (EC2, S3, Route53, Elasticache, ELB, OpsWorks, RDS, AutoScaling), Terraform, Raspberry Pi 3, Python, C

Challenge

The client runs an advertising company that provides hardware devices and analytics software to businesses to generate passive, recurring revenue. Businesses can connect the device to TVs and digital screens, schedule and run different kinds of media campaigns to earn extra income with minimal efforts. In addition, they can manage their ads content (ex. images, videos, and RSS feeds), see metrics and reports to keep track of their earnings.

Analytics dashboard for clients

When we started working together, the client already had a software solution with a huge amount of legacy code, complex business logic, but a good technical base. The biggest challenge we faced was the variety of technologies, lack of automation and tests, and documentation that didn't cover all aspects of the project.

In general, the project consisted of 2 web platforms and thousands of terminals. The project's immediate needs were to maintain their mature web platform, complete daily routine tasks, in order to

keep the platform running smoothly, as well as fix bugs, resolve customer issues, and provide technical support. An additional challenge that appeared in the middle of our cooperation was that the platform wasn't scalable enough to prepare for big traffic growth.

Approach

At first, we integrated Agile practices on the project to go with an adjustable development workflow, such as standups, planning, and retrospective meetings. After conducting fundamental steps, we completed the setup process with Slack for communication and used Trello as a Kanban board for task management.

Next, our team reviewed the existing code to learn more about the project and its functional requirements. During this process, we noted that the controller actions were well-documented, and the project used the latest dependencies and tech stack. However, the code wasn't covered with tests which was a major issue as it slowed down the development speed and made the app vulnerable. It was the reason why we focused on improving test coverage.

On the stage of negotiation, we noticed that a lot of work assigned to our team could have been automated. We discussed it with the client and started to automate some routine tasks to free up the developer's time for more important platform updates and codebase improvement. Overall, we were working on 4 main project parts during the first 10 months:

- Customer support
- Automation of routine tasks
- App stabilization and high-load preparation
- Codebase improvement

Customer support

One of our main duties was to resolve the problems customers had with hardware and applications. Throughout this step, the team was investigating different sides of the project and got to know the code better.

Every day, we were getting more requests than we could handle. The reason to believe was that every case was unique and handled manually, which required a lot of time and resources. Partly because there wasn't any knowledge base for similar cases.

We started using Notion for knowledge management and advised our client to follow the same process. It certainly slowed down our work, however, within 7 months, we created a knowledge base that contained answers on almost any customer question. After that, we set up a bridge communication channel with the customer support department and transmitted our explicit knowledge to them which resulted in a non-technical support team being able to resolve most of the problems on their own.

Installation Support Q&A

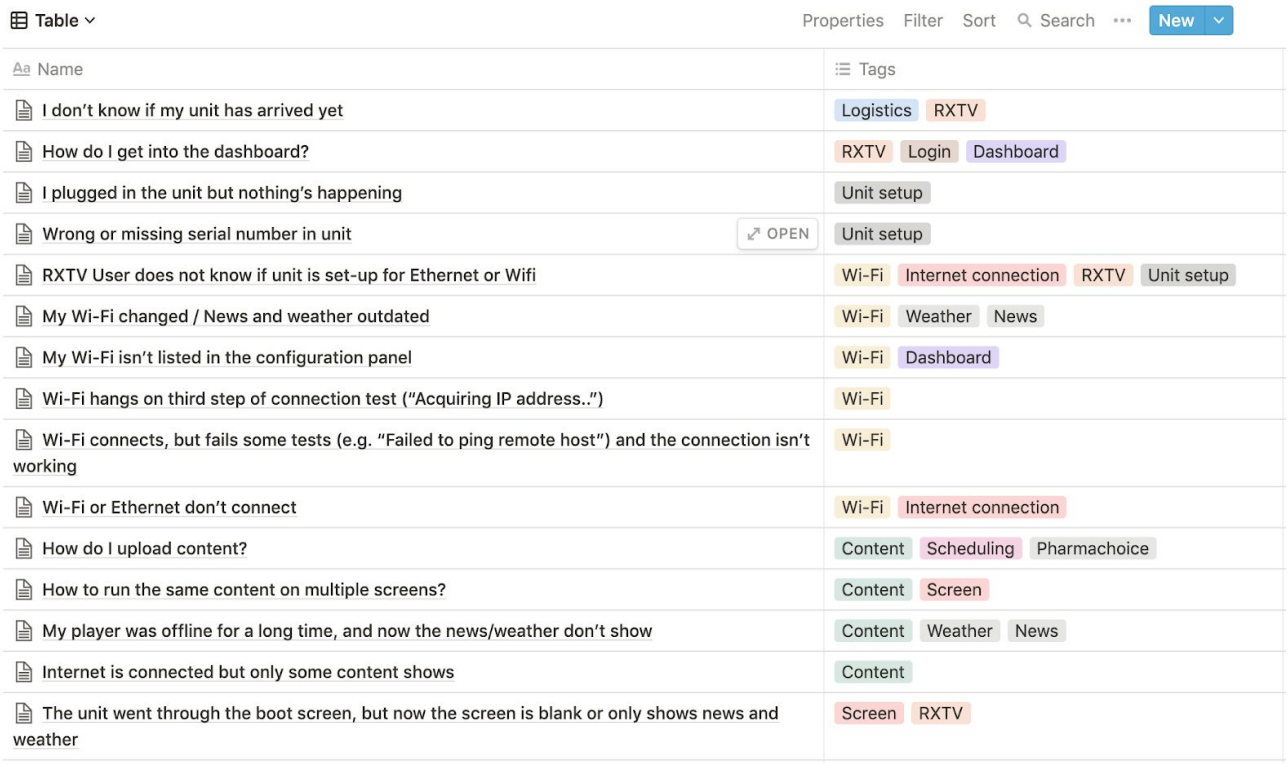


Table — Properties Filter Sort Search New

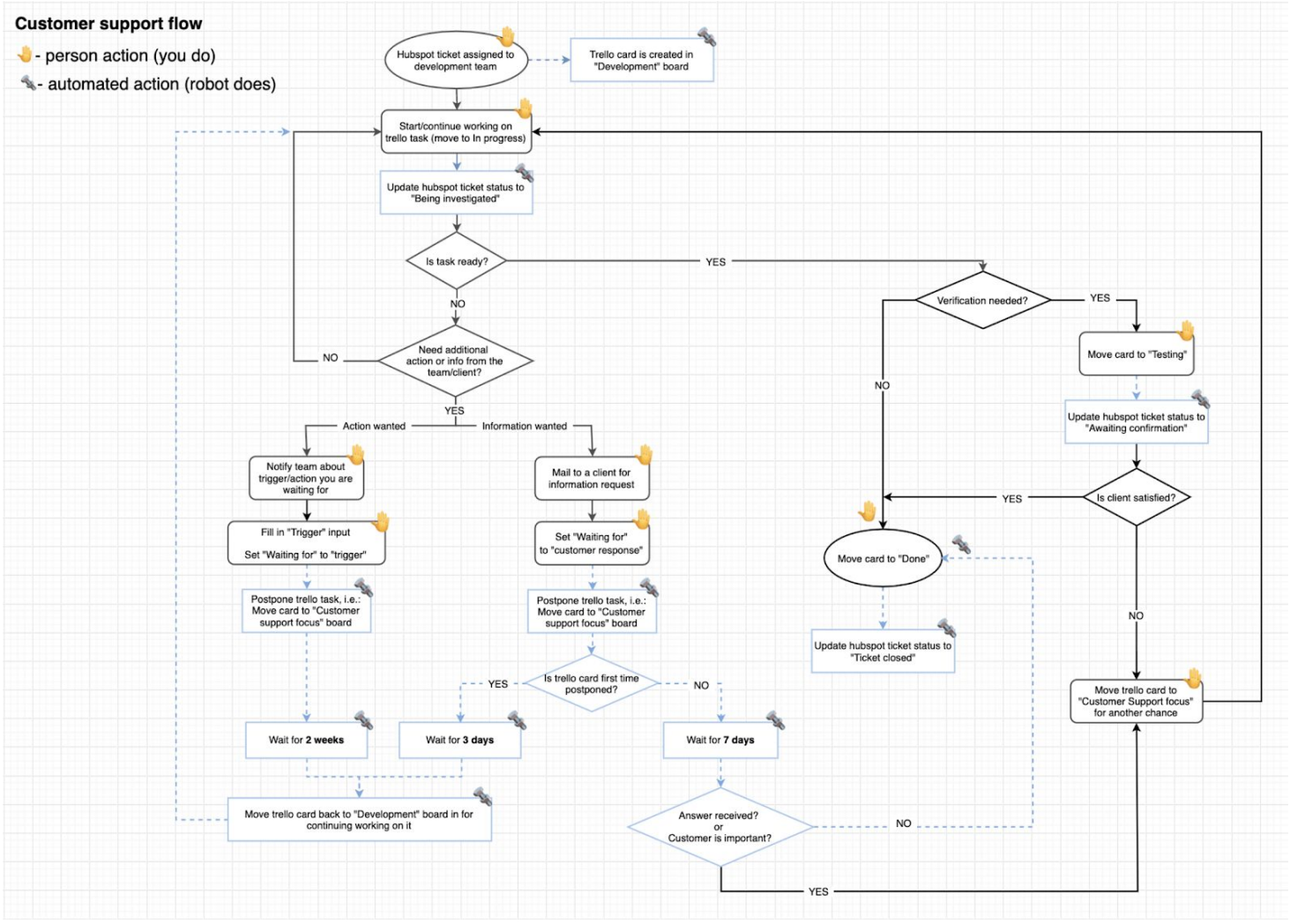
Name	Tags
I don't know if my unit has arrived yet	Logistics, RXTV
How do I get into the dashboard?	RXTV, Login, Dashboard
I plugged in the unit but nothing's happening	Unit setup
Wrong or missing serial number in unit (OPEN)	Unit setup
RXTV User does not know if unit is set-up for Ethernet or Wifi	Wi-Fi, Internet connection, RXTV, Unit setup
My Wi-Fi changed / News and weather outdated	Wi-Fi, Weather, News
My Wi-Fi isn't listed in the configuration panel	Wi-Fi, Dashboard
Wi-Fi hangs on third step of connection test ("Acquiring IP address..")	Wi-Fi
Wi-Fi connects, but fails some tests (e.g. "Failed to ping remote host") and the connection isn't working	Wi-Fi
Wi-Fi or Ethernet don't connect	Wi-Fi, Internet connection
How do I upload content?	Content, Scheduling, Pharmachoice
How to run the same content on multiple screens?	Content, Screen
My player was offline for a long time, and now the news/weather don't show	Content, Weather, News
Internet is connected but only some content shows	Content
The unit went through the boot screen, but now the screen is blank or only shows news and weather	Screen, RXTV

The implemented solution has left the datarockets engineers only serious technical requests to resolve, and secured enough resources to work on other important tasks. Next, we improved the process of managing incoming requests by integrating Hubspot and Trello.

Custom Integration between *Trello and Hubspot*

Using two task management tools, Trello and Hubspot was a real challenge as it caused the loss of focus to developers. They had to sync the tasks between both applications manually since there wasn't any integration provided by Hubspot. Besides, it was impossible to eliminate Hubspot due to the customers' feedback being collected there.

After further consideration, we decided to implement a custom integration. We created a customer support flow and automated 50% of tasks as illustrated in the flow diagram below:



Accordingly, the developers could receive tasks regarding Customer Support immediately to their board and manage them in a transparent way. Furthermore, it increased their happiness and work enthusiasm towards this kind of task.

Automation of routine tasks

Onboarding automation

There were procedures that must have been followed when the project started working with new clients. Most of them were small and similar, but the team had to repeat them from 10 to 100 times for each new client. It took a huge amount of time and effort to complete the on-boarding procedure. After investigating the process, the datarockets team analyzed the routine and then gradually started automating them.

One of the steps was configuring the scan distance to detect devices near the terminal. It took several minutes and required 2-3 Chrome's tabs to get necessary information about the environment manually, then calculate reasonable distance using a special formula, put it into a form, and save the configuration. We decided to implement a feature, which allowed us to do all these processes with one click. The automation allowed customers to perform 10 times faster at this stage, which significantly saved time and created a good impression on the client.

[redacted] 01:00

@Ula used the scan distance button today while setting up [redacted] account. Because they didn't set-up the filter themselves it automatically set it to 60 ft. This saved me about 5-10 minutes per units which made onboarding a new client much quicker and less stressful. They were impressed with how quickly we were able to onboard and are now discussing a bigger contract with us #teamwin

Reduced app downtime

One of the biggest issues the project experienced was the app downtime. The reason for this was that important services (e.g. Nginx) and workers did not start automatically after server reboot due to the maintenance on AWS. These services were responsible for processing data analytics, synchronizing the backend with terminals, starting scheduled jobs to send daily or weekly reports, etc. Therefore, every minute of downtime of these services was a big loss of income and data for the client's business.

To start all the required services, we had to connect to the server and do everything manually. To solve this problem, we configured the autostart of the services with *systemd* units. After this enchantment, services were able to start automatically after servers reboot. It resulted in decreasing maintenance costs and increasing income for the business.

Monitoring tool integration

Our team was responsible for a routine that was not too difficult, but time-consuming. It consisted of multiple small tasks including:

- Ensuring the client dashboard worked properly.
- Checking the amount of unprocessed data to be sure that workers are in a healthy state.
- Ensuring the daily tasks were completed.
- Diagnosing the unhealthy terminals and fixing them.
- Checking that we had enough free space in the DB server, etc.

The best solution was to set up some monitoring system to only receive notifications when something bad occurred. To do this, we chose Datadog due to its simple integration with our AWS infrastructure, a wide range of features that covered all our needs, and low cost or even free for the size of our servers. We managed to set up observers for our services to be sure that they were healthy and worked properly. The monitors were configured to get servers' metrics like CPU load, memory load, free disk space, etc.

Integrating this monitoring tool has put an ease to the process, reduced stress and pressure from having too many daily duties.

App stabilization and high-load preparation

After the client signed a big prospect, we were awaiting a significant increase of new terminals and audience. It was great for the business as it might have brought a big income to the client, but we discovered that the platform wasn't ready for such a rapid traffic increase.

Each component of the application (workers, request handler, cache storage, etc.) was on the same physical server, like keeping all the eggs in one bucket. The situation did not allow us to scale servers horizontally, be flexible enough to avoid redundancy of unnecessary parts to save money, and could have caused data inconsistency in the application because of several cache storage instances.

To solve this problem, we decided to implement a variety of improvements in the infrastructure:

- Switched from old AWS Classic Load Balancer to Application Load Balancer
- Moved cache stores from local Redis to AWS Elasticache

- Moved Sidekiq service to separate server
- Refactored OpsWorks deploy scripts to independently deploy workers and app
- Configured OpsWorks to horizontally auto-scale app servers on load increase
- Created custom AMIs for faster instance start on Auto-scaling
- Used Terraform to keep all infrastructure as code

These changes allowed the application to get 5 times higher load and saved the client's budget by avoiding redundant scaling. Furthermore, this adjustment has stabilized the app enough and practically eliminated the downtime, which was a big headache for the dev team.

Codebase improvement

As mentioned above, we were dealing with a large codebase lacking tests and common code-style, meaning:

1. We couldn't be sure if we wouldn't accidentally break something important when implementing a new feature or making the codebase scalable.
2. It required more effort and time to understand the codebase and business logic.
3. To reach an acceptable level of test coverage for faster and better development, we had to spend more than 6 months of work.

To overcome this challenge, we applied our simple but powerful rule: "Make it better than it was". It undoubtedly helped us increase the quality of the codebase on a daily basis. The philosophy of the rule included breaking big refactor tasks into milestones, making small but important improvements every time we worked with code. There was a couple of conditions that we checked before accepting the code changes:

- A newly implemented feature must have been covered with tests.
- If fixing was needed, refactored the place where we were working on and made sure to follow the style guide.
- If you resolved something that was unclear, put the investigation result into the documentation.

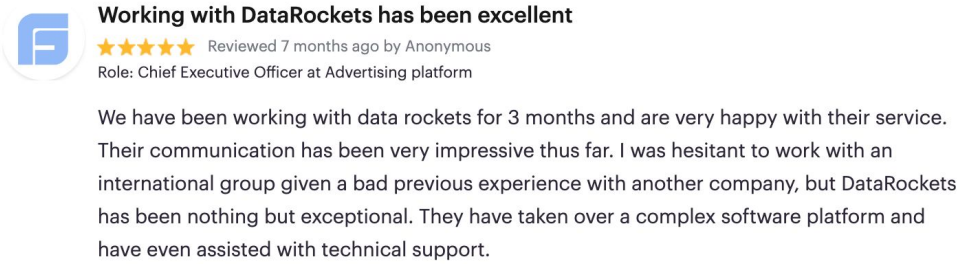
Consequently, this strict but necessary workflow allowed us to increase the test coverage from 0% to 30%, which was a big improvement and motivation for the team to continue working on it.

Result

Despite the fact that we worked with this kind of project for the first time, our team was able to:

- Automate routine business processes to save time & money for our client.
- Resolve a great number of technical issues working as a customer support team, which resulted in a better customer experience.
- Stabilize the applications by reducing the platform's downtime overall.
- Make the whole ecosystem of applications scalable so the client was able to grow their customer base significantly.
- Improve the code quality and test coverage which resulted in faster development of new features.

datarockets' team proved their excellent skills of transparent communication and problem solving that impressed the client in a different kind of experience working with international teams. After only 3 months of collaboration, the client happily shared nice feedback about us on GoodFirms:



Our team continues working on this project and has huge plans on making the project codebase better, automating more routine tasks, and integrating with new ad-exchanges that will grow the client's business revenue.

Technology

Ruby on Rails, PostgreSQL and Redis DBs, Delayed job, Que and Sidekiq job schedulers, AWS (EC2, S3, Route53, Elasticache, ELB, OpsWorks, RDS, AutoScaling), Terraform, Raspberry Pi 3, Python, C