Cloud & Serverless Architect/Developer

Eric Roijen
Dutch National
Currently living in Cambodia

eric@naguras.com
https://portfolio.naguras.com
https://github.com/eric-naguras

Skills

See a nice version online with pictures here:

AWS skills: Cloudfront, Cognito, API Gateway, S3, DynamoDB, EC2, Certificate Manager, Lambda, Lambda@edge, CloudWatch, SQS, Event Bus, Parameter Store, Cloud Development Kit, CloudFormation, IAM, SNS.

Recent skills: MongoDB, Javascript & ECMAScript, Node.js, Express, Quasar Framework, Tailwind CSS, Bootstrap CSS, Json Web Tokens, Let's Encrypt, Mocha Unit Test Framework, Cypress Integration Test Framework, PM2 Process Manager, jQuery, Vue.js, **Sveltekit**, Vite, Nodemailer, Stripe Payment API, Websockets, Cloudflare Workers, Cloudflare Pages, Cloudflare Key Value store, Webpack, Docker, Docker Compose, Linux, Microservices, Progressive Web Apps, Electron, Forex Trading, MQL, **Blockchain**, Crypto Currencies.

Older skills: Dot-net, c#, SQL Server, ASP .Net MVC web apps, WFC and SOAP Web services, Windows Forms, Azure.

Introduction

I have been involved in *cloud computing* since the year 2000, way before the term was coined by then Google's CEO Eric Schmidt in 2006. I'm a strong advocate of *serverless* because it moves the responsibility for security of the infrastructure and operating systems where it belongs: the hosting provider. It also eases the life of developers and makes deployment and maintenance so much safer, faster and thus cheaper. I am a member of AWS's Customer Council.

I am easy to work with, a creative but pragmatic thinker, excellent communication skills, result oriented, good negotiating skills, problem solver, analytical, marketing skills, broad knowledge of ICT market and products, ICT architect, commercial skills, Internet infrastructure & applications skills, application and implementation experience.

One of my greatest strengths is a deep understanding of a customer's business needs and translating those needs into a solid, maintainable, scalable, secure and price efficient architecture. Subsequently coding and implementing the architecture and deploying it to the desired or most suitable infrastructure.

I am a curious person. I love to keep my skills current and spend a considerable amount of (free) time learning new technologies and products in the field of web development and cloud computing.

I am a staunch believer of "Make everything as simple as possible, but not simpler".

My coding style is clear and expressive. I am a bit of a perfectionist and I'm always looking out for possible security risks. I try to reduce the amount of dependencies while coding mainly out of a security perspective. If I would need only one or two functions from a library or module, I'd rather write that function myself (or copy it) than to include the whole module in my application.

I have worked a long time as a computer journalist for major Dutch IT-publications like Computable, InfoWorld, CM Corporate and others. As a journalist I have met and talked with industry leaders like Bill Gates (Microsoft), Larry Ellison (Oracle), Billy Joy (creator of Java), Charles Wong (Computer Associates), and Eric Schmidt (Novel, Google).

Experience

Typescript,
Javascript,
docker,
nodejs,
cloudflare workers,
fly.io,
svelte,
geolocation,
sveltekit,
SQL,
Supabase,
Tailwindcss

Car Guard Mobile App

This mobile app allows motorists to give gratuities to the informal car guards that are ubiquitous in South Africa.

Car guards are individuals who provide an informal parking attendant service. They are typically dressed in high-visibility vests and assist motorists in finding parking spaces, keeping an eye on vehicles while the owner is away, and sometimes helping drivers maneuver out of tight spots. Although car guards are not officially regulated or employed by the parking facilities or retail establishments they frequent, their presence is a well-known aspect of urban life in South Africa.

This mobile application leverages modern web technologies and delivers a seamless user experience akin to native store apps. It can be installed directly onto devices and offers offline functionality, employing Progressive Web Applications (PWA) technology for caching capabilities that mitigate the dependence on an internet connection.

Upon completing a registration process within the app, car guards initiate their monitoring service by photographing vehicles enrolled in the program, identifiable through a designated sticker, which serves as verification of their oversight. They are prompted every 15 minutes with a push notification to capture successive images, ensuring continuous supervision. The guards' interface displays these photographs sequentially, arranged based on the required timing for the subsequent snapshot.

Owners of the monitored vehicles receive real-time updates via WhatsApp, allowing them to confirm and reward the service with a tip if desired.

The app is built using SvelteKit, the versatile TypeScript framework suitable for constructing full-stack applications. Full stack here alludes to the framework's capability to handle both client-side and server-side code within a unified environment. TailwindCSS, a utility-first stylesheet framework has been implemented as the styling component. The architecture is robust, with data housed in a Supabase managed Postgres database and vehicle license plate images stored in Supabase Cloud Storage. The application leans on specially crafted Stored Procedures within Supabase to manage tasks -- such as updating multiple tables within a transaction, to ensure consistent database operations.

Additionally, Supabase Triggers enhance the application's interactivity by invoking SvelteKit endpoints whenever pertinent table adjustments are identified. The application is crafted to employ the built-in geolocation features of mobile devices, highlighting address lookup against sent GPS coordinates for geospatial contextualization.

For optimal performance, the app's infrastructure is deployed on Cloudflare's workers edge computing environment to ensure proximity-related efficiency vis-à-vis user interaction. A discrete push server is provisioned for handling notification delivery using a customized Docker container, with server capabilities supported via a Node.js runtime. This container is hosted on Fly.io's expansive Edge network – notably including a strategic South African presence – to enhance the service availability for the intended demographic.

Websockets
Javascript
Docker

Scalable SaaS-Based Customer Service Chat Application

A large-scale software-as-a-service (SaaS) application for customer service chats. The application is designed to be efficient and cost-effective and support tens of thousands of users concurrently.

nodejs,
Redis
Supabase
Jsonwebtokens
Fly.io
Svelte
Sveltekit
Typescript
Tailwind Css

I was responsible for the entire development process of this application, from gathering requirements to writing the code. I understood the needs of the users, designed the architecture of the application, and implemented the code. I also tested the application to ensure that it worked correctly. I'm proud of this app because it showcases my coding skills and ability to build applications.

This is a chat system that allows website visitors to ask questions to human chat agents. Although artificial intelligence is improving, many organizations and visitors alike prefer human answers. Humans are better at understanding complex questions and providing nuanced answers. They are also better at building rapport and trust with customers.

This chat app was made with businesses and organizations in mind that want to provide top-notch customer service. It recognizes the importance of human interaction in resolving customer issues. It can be used in a variety of industries, including e-commerce, corporate, marketing and sales, education, nonprofits, and NGOs. The chat app is also highly optimized for outsourced customer service organizations. These organizations typically have a large customer base and manage chat operations on their behalf. The administrative side of the app is designed to make it easy to add new customers and their websites and domains.

Overall, this app is a comprehensive, user-friendly live chat solution that helps businesses provide superior customer engagement and service. It seamlessly blends technology with the irreplaceable human touch, ensuring that customer issues are resolved quickly and efficiently.

The app is fast, but speed was not the primary focus of the design. The main goal was to keep resource usage low so that pricing for users could be kept as low as possible. This is important because the app is used by a large number of people, and the cost of running the app can be significant. This low resource usage is achieved by using efficient technologies and hosting providers and by giving agents the tools to handle as many chats in the shortest time possible. In the context of this type of chat app, speed is not as important as it may seem. This is because website visitors are often slow typists, and agents are often handling dozens of chats at the same time. This means that there will always be some waiting time, regardless of how fast the app is.

Features

The app consists of three modules:
An Admin app for administration and agents chatting with visitors
A chat component that is embedded in a webpage
A chat server that handles the traffic between chat agents and website visitors
Chat agents are grouped by their areas of expertise or subject matter to answer a variety of questions from a diverse customer base. Messages are first placed in a group queue before being assigned to multiple agents who handle the same group. An agent can handle multiple chats at once, depending on their speed, the complexity of the questions and the visitors' response time. A complex scheduling module distributes messages from the group queue to the agent depending on the performance of previous messages.

The chat system is designed to optimize both speed and accuracy, which are essential components of any customer service interaction. It has a smart search feature that scans through a catalog of frequently asked questions (FAQs) based on the visitor's inquiry, and instantly suggests the most relevant

responses to the agents, not the visitors. This feature is being further enhanced with artificial intelligence, which will allow the system to comprehend the visitor's question and craft multiple tailored responses, ensuring that customers receive quick and accurate information every time.

The agents also have the ability to expand the list of FAQs, which will continually improve the system's knowledge base. In addition, agents have access to a repository of pre-set phrases, or "snippets," which are linked to keyboard shortcuts. This allows them to communicate more quickly and efficiently. The chat system also promotes collaborative problem-solving. If an agent faces a more complex inquiry, they can quickly consult with seasoned colleagues or supervisors. If necessary, they can transfer the chat to another user, ensuring that each customer's needs are best served by the appropriate expert.

All of these features are designed with efficiency in mind. Faster response times mean that more queries can be handled per hour, and more resolved issues mean happier customers. This reduces the number of customer service agents required, which decreases overhead costs. The system's smart features make customer service not only more streamlined but also more cost-effective for businesses.

The admin app lets you manage the chat app. You can add and assign agents, add and modify frequently asked questions, create chat subject groups, allow internet domains to access the chat app, and set shortcut snippets for agents. Managers can also set and change greetings or status messages that visitors see when they open the chat by clicking on the chat icon. These welcome messages can be automatically changed based on the time of day, if desired. The admin app also gives you the ability to set parameters for domains and websites, both for security and management purposes.

Visitors can share screenshots or other files with agents during a chat. After the chat is over, a rating page will appear where visitors can rate their experience and provide feedback on the chat. All chats will be stored in a database where they can be reviewed by customers and chat managers to improve the quality of service and rate agents on their performance. Additionally, data analysis can be conducted to identify the most frequently asked questions and other issues that visitors have difficulty with. The findings can be used to improve service, web design, and documentation.

Future Features will have audio/video calling, co-browsing (This allows the agent to browse the website with the user to guide them real-time, improving the user experience and resolving issues faster) and a better AI-assisted solution for chat agents. Additionally, CRM integrations or even a light CRM functionality will be available.

Architecture & Technologies

Here, we will examine some of the design decisions, technologies used to achieve the desired outcome, and how we achieved maximum agent efficiency.

The Chat Server is a service that handles the traffic between chat agents and website visitors. It ensures that messages are delivered to the correct recipients and that the chat experience is smooth and reliable. It is also the service that will use the most resources as it is the heart of the application. It sits between the visitor chat app and the Admin app where the agents work. The chat server, constructed as a Node.js service via WebSockets, operates within a Docker container. This setup not only bolsters its robustness and security, but critically, empowers its scalability. Whether scaling out to accommodate high demand or scaling back when less capacity is needed, the accommodation of fluctuating needs is simplified.

Further accentuating this adaptability is the use of Fly.io a globally represented cloud hosting platform specializing in fully managed Docker containers. Fly.io's strength lies in its global span, featuring servers across all continents. This

ensures that the chat server is proximate to the chat agents, yielding optimal performance. Moreover, Fly.io simplifies chat server management with features

like automatic load balancing and health checking, allowing for nimble recalibration. Easing management effort while maintaining steady, high-quality performance, Fly.io guarantees reliability and continuous availability of the chat service. The combination of Docker container utilization and partnership with Fly.io inherently promotes scalability.

Provisions of both horizontal (quantity) and vertical (quality) resources adjust automatically to the server's load, preventing over- or under-provisioning. This intelligent use of resources guarantees swift message handling, all the while optimizing cost. Simply put, the chat server exemplifies smart scaling and strategic deployment at its best. Complementing the scalability of the chat server is an efficient communication strategy facilitated through an agile, managed Redis server. By harnessing Redis' publish-subscribe (pub-sub) mechanism, rapid, real-time messaging between Docker instances is achieved. Redis, renowned for its speed and efficiency, forms the perfect backbone for managing inter-container communication. The pub-sub messaging paradigm it utilizes enables decoupled communication, making it easy to distribute updates across multiple Docker instances in a secure, organized manner. This collaborative infrastructure greatly enhances the chat server's performance, ensuring that no message delay or loss occurs even when the server scales up or down according to demand. Overall, the blend of Docker, Fly.io and Redis presents a well-rounded solution, delicately balancing scalability, performance, and efficient resource utilization.

The Admin App is designed to support tens of thousands of users, with strict data separation between different customers and privacy policies. The admin app is written in Sveltekit and uses server-side rendering for ultra-fast response times. It is hosted on Cloudflare's edge network, which has over 250 locations worldwide. This ensures that users will always be close to a Cloudflare access point, improving performance. Cloudflare also has a large set of data protection mechanisms and security features. Supabase was chosen for data storage because it is a robust, scalable, and PostgreSQL-based database that supports complex data structures. RLS (Row-Level Security) was a key factor in choosing Supabase over other SQL databases. RLS allows you to control which users have access to which rows in a database, which is useful for SaaS applications that need to restrict access to data based on user roles and organizations. RLS also allows you to make database calls from the client, which can reduce server resource usage and results in significantly lower operating costs. Supabase has a decent authentication module, but it does not have good support for SaaS applications or Role Based Access. Therefore, I created my own authorization system using RLS.

The Chat Component is a piece of code that customers can embed in their websites. It can be added by adding two lines to the header of their site: one for the JavaScript code and one for the CSS with the styling information. A separate CSS file makes it possible to completely adapt the style of the chat component to the customer's website. The JavaScript is built using Svelte, which compiles to very compact but fast JavaScript code. The javascript code is only 200 Kbytes in size. Additionally, this component is housed on the Cloudflare Edge network too.

Mobile Walking App Part 2

Typescript
Javascript
Websockets
ESP8266
C++
AWS CDK
AWS DynamoDB
AWS Lambda
Serverless

The first version was inconvenient to use. Your mind wanders off and you forget to press the lap button on the phone. So I designed a hands-free solution with the help of some electronics. I used a microcontroller (MCU) and a motion detector with a beeper. All for under $5. I wrote a program in C++ for the MCU sending out a lap time via websockets every time the motion sensor picked up someone passing by. A node js app on my PC would receive this signal, calculate some averages and different display formats and pass it through to a web page on mobile phone also via websockets . This web page is also being served by the node js app. The web page is written using pure javascript, no frameworks or libraries are used except for a websockets package. When a stop signal from the phone is received, the node js app will send the walk data via a AWS Lambda to Dynamodb for later analysis. All AWS parts are set up using AWS CDK to describe infrastructure in code.

Mobile Walking App

Typescript
Svelte
Svelte Native
Nativescript
AWS CDK
AWS Lambda
AWS Dynamodb

Walking App is a health-focused tool designed for indoor walkers with specific health conditions. The application counters the limitations of the Apple Watch, enabling accurate tracking of pace and distance. To achieve this, I employed web frameworks like Svelte, Sveltekit, Svelte Native, and Nativescript, crafting a cross-platform mobile application with an intuitive user interface.

One of the features I incorporated was haptic feedback, offering users a tactile response and enriching their interaction with the app.The app sends all data to an AWS backend of storage and for a future website that will display and analyze this data.

The backend is quickly set up with AWS CDK, a toolkit by Amazon that lets you define the infrastructure in code. So I used Typescript to define a Dynamodb database with a single table design and two lambdas, one for storing data and one for retrieving data. The CDK makes it easy to deploy updates with a single command.

Apollo GraphQL Server on Cloudflare Workers

Javascript,
Apollo Server,
Apollo Gateway,
Cloudflare Workers,
Serverless

A small proof of concept to run Apollo GraphQL servers on Cloudflare workers. Copilottravel.com, a startup that is building the next generation travel search and booking engine, asked me to make a small example on how to run apollo graphql servers on Cloudflare workers. For their travel search engine they need combined data from dozens of different sources. GraphQL was made for these kinds of applications.

Copilottravel already started their development on Google Cloud Services because CGS is node js compatible. But they plan to run everything on Cloudflare because it has much more point of presence and is faster than GCS.

The Apollo server products are built for and with nodejs and do not run by default on the Cloudflare Workers platform because the latter does not support node js. With the help of an experimental package from Apollo and some guidance from the Cloudflare Workers product manager, I was able to get a few graphql services running on Workers and establish a basic connection to Apollo Studio (the management app for Apollo services). My work gave them a handle to continue their porting efforts from Google Cloud to Cloudflare.

100% Serverless Website with Sveltekit & Cloudflare Workers

Javascript,
HTML,
CSS,
Tailwind,
Svelte,
Sveltekit,
Cloudflare Workers,
Responsive Layout,
Server-Side-Rendering,
Serverless

A simple website built in a day with Sveltekit and Tailwind running on Cloudflare's workers. You can see the result at rep-it.nl.

Normally I don't do very plain websites but I did this for my brother and I wanted to play around with Sveltekit and Cloudflare Workers. So while this is a simple website from an HTML perspective, there are also complicated technologies involved. It's a serverless app running on Cloudflare's edge network. This means it's infinitely scalable, super secure, very fast for visitors where it doesn't matter if you are in Alaska or Zimbabwe and - in the case of

Cloudflare - can be deployed with a single command. (Actually, with the latest version of Cloudflare Workers/Pages, you only need to push it to a repository like Github or Gitlab)

This was my first attempt to build something with Sveltekit, a new web framework with a radically different approach than most other web frameworks.

Svelte is a component framework — like React or Vue — but with an important difference. Traditional frameworks allow you to write declarative state-driven code, but there's a penalty: the browser must do extra work to convert those declarative structures into DOM operations, using techniques like that eat into your frame budget and tax the garbage collector. Instead, Svelte runs at build time, converting your components into highly efficient imperative code that surgically updates the DOM. As a result, you're able to write ambitious applications with excellent performance characteristics.

Sveltekit can generate different types of websites, static and server side rendered and uses adapters to tailor the result to specific deployment types like Cloudflare Workers, Node.js, Netlify, Vercel, etc. In this case I used Cloudflare Workers. For styling I tried Tailwind, something I wanted to work with for a long time. With very little effort I got a perfect score on Google's Lighthouse, a website performance toolkit, both on Desktop and Mobile.

The website was originally created with plain html/css. To rebuild it using Sveltekit was not very difficult. Each file in a routes folder automatically becomes a route so I just added a file per page. Additionally I added a special layouts file that contains the header, footer and sidebar that will be repeated on every page. No specific Svelte code or Javascript was needed for this. The header navigation bar was built as a separate component and had to be included in the layout page. This navigation bar is the only part that has some specific Svelte code for showing and hiding the mobile drawer menu. The menu also uses Svelte specific transitions.

100% Serverless Project Portfolio

Javascript,
HTML,
CSS,
Tailwind,
Svelte,
Sveltekit,
Cloudflare Workers,
Responsive Layout,
Server-Side-Rendering,
Serverless

Built with Sveltekit and Tailwind running on Cloudflare's workers. Because LinkedIn does not offer a nice way to show a project's portfolio, I built this website. I wanted to show every page as a separate link on my profile so I needed a *Server-Side-Rendered* website.

So while this is a simple website from an HTML perspective, there are also complicated technologies involved. It's a serverless app running on Cloudflare's edge network. This means it's infinitely scalable, super secure, very fast for visitors where it doesn't matter if you are in Alaska or Zimbabwe and - in the case of Cloudflare - can be deployed with a single command. (Actually, with the latest version of Cloudflare Workers/Pages, you only need to push it to a repository like Github or Gitlab) I also wanted to use Serverless hosting so I don't have to pay for a full server running 24x7. Sveltekit, together with the Cloudflare adapter makes this just a deployment option without the need of any code changes.

The Tailwind CSS utility classes made it possible to not have a css file or create classes on every page. Instead you can do a sort of real-time design; keep a browser window open next to your editor and add utility classes directly on elements, like font-bold, text-gray-500 or mt-5 for a top margin.

I like to keep two browser windows open next to my editor; one for a mobile view and one for a desktop view. This way I can immediately see if I need different classes/layouts for mobile or desktop. This way of working is very efficient and a real time saver.

This was my 2nd Sveltekit/Tailwind trial after working for a few years with Vue.js. I like Sveltekit very much because it lets you write exceptionally clear and concise code, it compiles to extremely fast, pure Javascript.

Javascript,
Node.js,
Express.js,
MongoDb,
Vue.js 3,
Quasar Framework,
AWS EC2,
AWS S3 Web Server,
JSON Web Tokens,
Docker,
Docker Compose,
NGINX,
Let'sEncrypt,
Responsive Layout

Custom CRM Application

I came across this project on one of the freelance IT-projects websites. Because, for obvious reasons, I cannot share that much on projects I did for customers, so I decided to spend a week and build part of this app to show what I can do, how it would look like and how much I can archive in a week's time.

I plan to add some more functionality whenever I have some spare time. I'm also thinking of making several versions of this app, each showing different technologies. The specifications document is at my Google Drive. I also put the source code into public repositories that you can find here: crm-frontend and here: crm-backend.

This project is a good example of how I translate business requirements in an application. After reading the specifications I still had a lot of questions but also enough to build something functional.

The Front-end contains everything you see except the variable data. The frontend is technically a static Single Page Application and can be hosted on any simple web server. Currently it's hosted in Amazon's London datacenter as an S3 website. This kind of static hosting is extremely cheap. For a business app with under a dozen users the costs would stay well under a dollar a month. You only pay for the transfer of the single webpage when it is loaded in a user's browser. So for an app like this it's probably only twice a day. This all means that it is infinitely scalable and extremely secure. Although an empty frontend would not need any security as there is no data or secrets inside the webpages. The frontend is built with Vue.js 3. and the Quasar Framework.
Then there is a backend server process that, on request, will provide the data to the frontend like lists of dealerships, venues and detailed information about a dealership or venue. It will also handle update requests for deleting or modifying data. Access to the data is limited to authenticated and authorized users only. Through the frontend you need to login providing a username and password. When a login has been approved the server will return a token with a limited expiration. With every subsequent request this token needs to be provided. Without the token or with an invalid token no data will be returned. The token contains secured information about the user who requested it like his userId and his role. By using role based authentication it's possible to have a fine grained system of who can see and alter what information. The backend is built with node js, an extremely fast javascript runtime especially suited for high transaction backend systems. This node js code is containerized using Docker. This Docker container can run on any high-end hosting facility and can be automatically scaled up or down depending on the demand. Currently it's running in Amazon's London datacenter on the smallest EC2 virtual machine. Amazon has more elegant container systems but they come at a higher price and would be overkill for this demo app. To serve data over HTTPS, I added two Docker container apps to the deployment. A nginx proxy web server together with a special nginx companion app that automatically creates and updates Let's Encrypt SSL certificates without any manual work.

The last part is a database server where all the data is persisted. Currently the datastore is a Mongodb hosted solution which is fine for small installations. Depending on the demand and use cases of an application, this database could also be hosted on Amazon.

The app is meant for desktop displays because of its high information density. But the app can be used on a mobile device. In fact the top menu bar as seen in the desktop browser is replaced on mobile by a so-called hamburger menu (the three stacked bars in the top left corner). Clicking on the menu icon will open a side drawer with an access menu to all pages in the app. Also the tables in the Desktop are replaced by cards because tables are difficult to display on small screens. Because of the high information density, other parts of the application will need much more adaptation to small screens. It is doubtful if a full mobile application would be needed or useful but the Call List and the Call Report could come in handy on mobile.

Javascript,
HTML,
CSS,
Node.js,
Express.js,
MongoDb,
Vue.js 2,
Vue.js 3,
Quasar Framework,
Mocha Unit Test Framework,
Cypress Integration Test Framework,
PM2 Process Manager,
AWS EC2,
AWS S3 Web Server,
AWS Lambda,
AWS SQS Messages,
AWS Parameter Store,
AWS CloudFront,
AWS Event Bus,
AWS Cloud Development Kit,
Nodemailer,
Stripe Payment API,
Websockets,
JSON Web Tokens,
Docker,
Docker Compose,
NGINX,
Let'sEncrypt,
HTTPS,
SSL,
Responsive Layout,
Server-Side-Rendering,
Serverless,
Linux shell scripts

Rewrite of Music Search Web Application

musicaltheatersongs.com is a web application for musicians and users in the musical and theater sector to search for musical songs based on musical-technical terms. The app is mainly used by musical students and teachers. It contains more than 11,000 songs and is used by tens of thousands of users globally. The original app was written more than 6 years ago and used a MVC framework called Derby which became obsolete. It was difficult to maintain the code or to add new functionality. I was asked to rewrite the application using a more modern toolset.

The original application had two functional parts (actually three if you count the Mongodb database as a part of the application) but was built as a single app (except for the database which is a separate service). The two functional parts are the website front-end, the part end users see and use for their searches and store their songs lists. The other part is an admin section where new songs, shows, composers and lyricists are entered and modified. It also manages user's subscriptions, both individual and institutional (universities, etc).

I split the app into three separate parts that work independently. I created a separate admin front end written in Vuejs 2 combined with the Bootstrap CSS framework and some 3rd party components like a versatile table component. The admin front-end is a Single Page Application that runs on Amazon's S3 web server combined with AWS CloudFront for world wide cached content delivery and HTTPS access. I wrote Cypress tests to test the front-end after and during modifications. The front-end is fully stateless and does not rely on previous requests. Some of the information that is not too large, is held in memory for as long as the page remains open. Login info and the JSON web token are stored in LocalStorage and will be reused if a user closes or refreshes the page.

Admin Back-end
To access and modify data from the admin front-end, I created an API service that serves data securely and stateless to both the admin front-end and a new website front-end. This API service is built using nodejs with Express.js. The routes are secured by using JSON Web Tokens for every request accessing non-public data. There is also a "development" version of the API to test new features under development. Both the production and development services are packed inside Docker containers.
Inside the containers I use the PM2 process manager to monitor the node js app and immediately restart it if it would ever crash. Both the front-end and API service are completely stateless. There is no reliance on previous requests so a quick restart of the service is not noticed by the users.

Back-end servers need a lot of sensitive information like database passwords, secret keys to external APIs like the Stripe payment API or the external email service, etc. It's always a bit problematic where to store this sensitive info, it's very important these secrets stay secret. For this I created a small module that, when the service starts, reads this info from an AWS Parameter Store. In addition to secrets I also added some other bits of key information that is likely to change during the lifetime of the application like subscription prices. Without making changes to the code and a need for redeployment, you can simply and securely change this info though the AWS Admin console and the updated info will be used by the application.

Jobs Service
The app shows some statistical information to its users about their logins and searches. This information needs to be calculated from login and search logs. Because it consumes considerable resources, I decided to implement this collecting of statistics as a separate service so it would not interfere with normal operations. This is also a node js app that runs periodically through the use of a cron plug-in. This service is built in a way that new job types can be easily added. There is a main process that imports jobs as separate javascript files and adds those to a cron queue.

Public Website
Lastly, I created a new front-end for the public website (the one customers use to search for songs). This web app is made with Vue.js 3 and the Quasar Framework. It uses Server Side Rendering (SSR) for SEO reasons. SSR needs a web server (the code for this is generated automatically by the Quasar Framework) so for this, a Docker container is used as well. The new website is built with a Mobile First approach resulting in a much better mobile usability compared to the old website. (Actually, the old website is more or less unusable on a mobile device)

Lambda
Some services like payment processing when subscribing through the website or sending emails to expiring subscribers are implemented with AWS Lambda functions. The reason for using lambdas is to offload processing time from the API server, easier maintainability of those services and to gain experience with full serverless and Lambda functions. Lambda functions communicate their status to the back-end through the AWS Eventbus. Deployment of the Lambda and eventbus functions is done with the new AWS Cloud Development Kit (CDK). This is a toolset for Infrastructure as Code whereby everything is set up and configured through Typescript/Javascript. This makes it possible to store modifications of infrastructure in a code repository and keep different versions of the infrastructure in use.

Deployment
All separate parts of the application have production and development/test versions. Modifications can be made and tested first before adding them to the production app. This brings the total of services to 6 (2 api backends, 2 jobs services and 2 websites) all created as Docker containers. I added two more Docker container apps to the deployment. A nginx proxy web server together with a special nginx companion app that automatically creates and updates Let's Encrypt SSL certificates without any manual work. To ease the management and deployment of 8 containers I'm using Docker Compose. All instructions for running the containers, their names, ports, domain names, etc are stored in a .yaml file. When a service needs updating, a new Docker container is built and automatically deployed without interrupting the unmodified, already running containers. To automate deployment, I wrote linux shell scripts that will copy files from my development machine (linux or Mac) to the EC2 instance and remotely execute a docker build command and apply changes with docker-compose. All of this is done through a secure shell (ssh). The application has many users but does not have many transactions. We can run all containers out of a single EC2 instance. If the transaction volume would increase it would be very easy to add another EC2 instance and use a load balancer in front of the multiple EC2 instances. We could also switch to AWS Elastic Container Service or AWS Fargate and run the containers from there. Since everything is already containerized no code changes would be necessary. Instead of using Docker Compose we would switch to Kubernetes.

Bitcoin Fork

Javascript,
HTML,
CSS,
Node.js,
Express.js,
MongoDb,
C++

I was asked to help with a Bitcoin clone (fork) called Bitcoin Euro (BTE). I installed a number of BTE nodes on virtual servers and set up the necessary infrastructure around it like a blockchain explorer, a mining pool and a wallet. For this I adapted a few Open Source projects to work with BTE instead of Bitcoin.

I'm not really a blockchain expert but I do have a reasonably good insight in how blockchains work. I did make some small changes to the Bitcoin source code (made with c++). The explorer, mining pool and wallet software were all built with Javascript technologies and node.js.

Javascript,
HTML,
CSS,
Node.js,
Express.js,
MongoDb,
Electron,
Pusher,
AWS EC2,
AWS S3 Web Server,
Redis,
Websockets,
Vue.js,
Bootstrap CSS,
queues,
micro-services,
JSON Web Tokens,
Docker,
PM2,
Jelastic,
Winston logging,
Loggly logging

Cloud Trade Copier & Desktop Trade Copier

The Cloud Trade Copier is a cloud based Binary Options Trading System system where Signal Providers send signals to the service using a web based interface. When a signal arrives, all the subscribers to that signal are gathered from a Redis in-memory key-value store. Per user we check which broker they use and send the signal to other microservices (one or more per broker) that will send the purchase transaction to the broker's server.

The front end website is a responsive design built using Vue.js and Bootstrap. It's used for generating signals and gives the Signal Provider feedback on this performance and his customers. The system has its own subscription system where end-users can subscribe and join the channels of Signal Providers. The front end is stateless and uses JWT (Jason Web Tokens) for authentication and authorisation. It's role-based where Signal Providers can designate other traders to send signals on their behalf. The front end is a static site hosted on AWS S3 buckets and uses AWS CloudFront as its Content Delivery Network.

The backend consists of a collection of loosely coupled micro-services using node.js as the run-time. In Options Trading, fast execution speed and low latency is the most important feature. By implementing all functions as micro-services that can run in parallel on different servers we can simply scale horizontally to keep the total throughput under the required one second, no matter how many users the system has to trade for. We just add more instances of services. The main service will receive signals via a websockets interface and add every user who subscribed to the signal to a Redis in-memory queue. The users are also stored in the same Redis database for ultra-fast retrieval. When all applicable users are added to the queue, the service will send a message via websockets to all broker services. Upon the reception of the message the broker service will fetch a user from the queue and process the signal. When done, it will get a new user from the queue until the queue is empty. It will then sleep until it gets a new message. This way multiple instances of the same broker service act as workers to process the queue as fast as possible.

The system also uses a Mongo NoSQL database for storage of users, their expiry dates and their trading results. Mongo however, is too slow for retrieving users while a signal is received. When the main service starts, all valid users are copied to the Redis in-memory store. When a new user is added or removed the API server will signal the signal service via websockets and a user is added or removed from the Redis store All parts of the system, except the front end, are hosted inside Docker containers using the node process manager PM2 and running on a Jelastic hosting platform. PM2 monitors the node processes and gives feedback about their performance and can automatically restart any crashed process it manages. Jelastic is a platform for running services and provides automatic horizontal and vertical scaling. Lastly there is a separate API server for serving data to the front end web site, all protected by Jason Web Tokens. The API server is built with node, express and MongoDb. The API server is behind a load balancer to improve resilience and enable A/B testing with different implementations of features. It also enables updating code with zero downtime.

The Desktop app is made for Windows and Mac that can trade Binary Options based on signals from a Signal Provider. The signal provider uses a simple app as pictured below to signal a trade opportunity. Because the binary options trade is a very fast moving business where a single second can mean the difference between a losing or winning trade, the interface has to be really simple so with just a single click the trader (the person generating the signal) can send out the signal.

The app is built using HTML/CSS/Javascript bundled as an Electron app for Windows, Mac and Unix. Electron is a framework for building desktop applications using JavaScript, HTML, and CSS. By embedding Chromium and

Node.js into its binary, Electron allows you to maintain one JavaScript codebase and create cross-platform apps that work on Windows, macOS, and Linux — no native development experience required.

Desktop Signal Receiver

Electron,
HTML,
CSS,
Node.js,
jQuery,
Express.js,
MongoDb,
AWS EC2,
AWS S3 Web Server,
JSON Web Tokens,
Docker,
Pusher,
Winston logging,
Loggly logging

End-users who subscribe to one or more signal providers use another Desktop app for Windows or Macintosh to trade the signals the provider sends out. Binary Option Brokers do not give API access to their trading systems. So I built an app that will open a broker's website and inject some Javascript code in the webpage to be able to control the website and execute trades. The user chooses from a number of available brokers, enters their credentials and sets some money management settings.

When a Signal Provider enters a trade in a special Desktop app created for that purpose, that signal is sent to our server, validated and forwarded to pusher.com, a messaging service. The user's Desktop Trade Copier app receives the signal and my custom code that was injected into the broker's web site, is clicking the right buttons and filling in the required fields.

This Desktop app is built with node, html and css, together with Electron. The latter is a sort of a shell program that includes the Open Source Chromium browser together with the V8 Javascript runtime. Electron creates full native desktop programs but using html/css/js. It has full access to the file system and any other native resource. It also allows you to inject javascript into existing websites. This function is heavily used in the Desktop Trade Copier. This app utilizes jQuery in many places for getting screen objects and firing events. It supports ten different Binary Options brokers. It also includes a user subscription service with automatic expiries.

Award winning bookkeeping program

C#,
Dot-net,
DevExpress UI components,
SQL Server,
Azure,
ASP.net,
XML,
SOAP,
PKI

A shrink wrapped Windows Desktop program for bookkeeping created for the Dutch market. I did the design, architecture and programming of this app. At its time it was a refreshing new approach to accounting and got thousands of users. The goal was to make it as easy as possible for non specialists to keep their books up-to-date.

The program received several awards from the Dutch computer press. It's created using C# and other dot.net technologies. For storing data it uses MS SQL Server and for the User Interface, DevExpress libraries were used. DevExpress made it very easy to adapt screens to your own preferences. Extensive reporting is an important feature of this product.

From a single codebase, four different versions were created. Two invoicing only and two bookkeeping versions both in a regular and a pro version. During installation it would ask for a serial number. When the user entered the serial number, the installer program would contact a webservice that I also created using ASP.net running on MS Azure, where the serial number would be checked against a SQL Server database. If found, the webservice returns a config file that contains the version type of the product.

This config file with an XML syntax, is signed with a Private Key on the server. The desktop program has the public key and checks the validity of the config file every time the program
starts. It is therefore impossible to change the config file because this would invalidate the PKI signature.

Education

Atheneum (Coornhert Lyceum, Haarlem)

Technical University (HTS Haarlem)

Systems Analyses (SASO IBM Netherlands)

NIAM (Cap Gemini)

System 36 (IBM Netherlands

Extra Curriculum & Publications

Member and director of the Rotary Club of Cebu South, Philippines.

Co-Author of published research reports about Graphical User Interfaces, SQL Databases, and Multimedia (published by Disk'AD).

Book dBase for Windows (Addison Wesley ISBN 9789067895729).

Speaker at IT seminars about Graphical User Interfaces, IT Standards in the Nineties and Object Oriented Programming.

Judge of several programming competitions.