Netlife Technical Strategy

Introduction

This is a guideline describing the technical strategy of Netlife Design AS moving forward from 2019 into the future. It aims at suggesting a stable and standardized path for all new projects in order to structure the way we work and improve our efficiency, our flexibility, our quality, our ability to maintain and the expertise we offer.

Stakeholders

In any strategy there are considerations which limits the freedom and deciding power we can allow ourselves. There are areas which tech control fully and holly where we can, and should, own the domain and all the decisions within it. There are other areas where we share the

domain with other stakeholders, and we need to share decisions. And lastly there are domains where we clearly do not own the domain, even if we work within it, and can only influence or suggest certain strategic goals. We need to be considerate to the people we work with, and respect that there are several sides to any coin.

Outliers and Exceptions

A strategy can never be a perfect fit for every need be it existing solutions, relationships or simply timing. Constraints and requirements can overrule a strategy where there is good cause. For this reason the rules and guidelines provided in this document needs to be general enough to support common use cases, open enough to inspire creativity and exploration and concrete enough to give any value. We aim at aligning to the strategy in most things, but need to be flexible enough to choose other patch at times.

Evolution

Nothing lasts. Similar to code and documentation a strategy begins to deteriorate as soon as its written. Everything will eventually change. Therefore there will never be a *final* version of this strategy, there will be only a current one waiting to be improved in order to fit the larger and current picture of the world as it might be.

I encourage everyone to opinionate, improve and suggest changes to this document. It's not supposed to be owned by one single entity, it is by nature a shared commodity in which we all align to and, as far as possible, agree on.

Time is on our side

Rome was not built in a day. A strategy cannot be expected to be fruticious immediately. We must iteratively and slowly transition wherever we can to ripe the fruits of tomorrow. But we should aim at getting closer to that utopian future every single day. And that should be the fuel of our motivation. And we cannot do it alone. We need the entire organisation to believe what we believe, our partners to align with our guidelines and hope to convince new and existing customers to embrace the brand new world.

The dawn of opportunity

As of September 2019 there exists movements in the community which align almost perfectly with the internal direction we are heading. Simultaneously our room is filled with an elephant, or several elephants I would say, of challenges. And everything not related to elephant is indeed irrelephant. I firmly believe that aligning with the general directions and decisions proposed in this strategy would alleviate some of that pain quite efficiently, but in a considerate and respectful manner to those afflicted by them.

The short version of the strategy will be outlined here, while each will deservedly receive a more detailed analysis and explanation in the following sections.

Headless is the future

As all technologies gaining popularity, headless CMS isn't brand new. As autonomy, decoupling, RESTful, microservices, DomainDrivenDesign and abandonment of monolithic architecture has

been growing for quite a while, decoupling web applications from the backend follows the same pattern. There are several reasons as to why the emergence has been delayed in our field, but the maturing of SPA applications, Node/npm, GitHub, SEO requirements, SaaS cloud models and enriched tooling certainly are major beneficiary factors.

Headless options for new and existing CMS platforms are ***game changing***. There are technical improvements which certainly can improve development experiences and delivery quality, these are *nothing* compared to what headless offers to the table. And the list of platforms that are taking the headless approach is constantly growing (headlesscms.org).

The JAMstack

One cannot for certain confirm that the term JAMstack originated from Mathias Biilmann at Netlify, but one can trace most of its push back to Netlify at least.

What the JAMstack (wtf?) propose is a high focus on the front end builds, eluding server-side functions as much as possible. The three major elements are:

J - Javascript

Javascript on the client-side handling communication requests and responses. Although not limited to, front-end SPA frameworks like React fits this model well.

A - APIs

Server-side processes, including database handling, are accessed by JS-driven HTTP(s) calls. JAMstack is by default API and technology agnostic, communicating with all external sources over HTTP, caring about the data and structure only.

M - Markup

This term relate again to an agnostic attitude towards creating or rendering sites users can actually see. This declaratively suggests using build tools and encourage the use of static site generator techniques.

Completing the deploy circle

Following the JAMstack path would as such enable the use of CDNs (or ADNs) hosting of static applications fully SEO optimized with high-availability and performance. Git-centric continuous deployment strategies through pushing changes. And it reduces security concerns greatly with removing the attack vectors of traditionally coupled systems (it does not remove it completely mind you). The proposed architecture would also fit in nicely with serverless BaaS function paradigms.

Pro's and con's

By every shift in paradigms there are benefits and disadvantages to be found. It is important to understand and know about what these are so that we may face the consequences well prepared and in agreement. Hence the following sections outline how different areas are affected in a more direct manner both positive and negative effect. Because there is no such thing as one size fits all in technology. But the benefits ***are*** greater than the drawbacks in a heuristic perspective, which is why the chapters are positively named.

Strategy alignment

The overall Netlife strategy aims at moving towards delivering more product and service oriented solutions. The consequences of this could be to deliver web-applications with a whole range of different backends. Today most of our projects are content driven, often powered by a CMS. The strategy suggests a shift in that forcus.

What service/product systems have in common is that they are highly interactive, are feature driven and where the datamodel and structure is often inherited from, or dependent on, external or existing systems. These schenarios typically have a separated frontend and backend. We have failed in delivering such projects in the past. A great way to prepare would be to become experts at these kinds of structures and architectures in areas with less risk. The proposed technical strategy hence aligns quite nicely with how we want to deliver services down the road in accordance with the goals put forth in the Netlife strategy.

Customer benefits

There are few disadvantages on the customer side apart from the fact that there is an increased setup cost for full stack applications which includes both a frontend and a backend (CMS). In a JAMstack architecture most customers without technical inhouse resources would find making big changes to templates or the presentation of content more difficult without expert help. Content and text by itself should not be a problem, but the ability to add a new page template and present that on their webstite is not done purely from the CMS, which is a disadvantage for semi advanced users.

Benefits worth mentioning would be licensing and price. Most of the ecosystem (depending on platform of choice) offer very reasonable pricing models for quite extensive tooling and tech setup. It includes none of the drawbacks some architectures and stacks have on SEO. It enables a customer to choose, and replace, specific expertise on the frontend and backend side. Arguably the cost of an eventual migration will be significantly lower and easier to manage (or contain rather).

Due to separation of concern, it is safe to assume that once in place the JAMstack architecture will be cheaper and more reliable solution. However, existing or already implemented CMS platforms could challenge, and even prevent, the optimal architecture for a given set of customers. There are legacy systems out there (we have even created many), and we need to transition slowly, if at all, depending on the customer.

Partner benefits

A clear separation between the frontend and backend should make it easier to identify responsibility. Estimation and status for separate deliveries should also easier to understand and report on, as the exchange currency is data through defined endpoints. Is should improve collaboration and mandate in a more concise way than previously.

The most important beneficial effect however would be ownership and simplicity. No longer should uncertainty of handover and re-implementation of functional prototype be an issue.

The most important drawback would be that one side of a messy partnership needs to let go of a part of the solution, in other words release control of a revenue stream. If the parties are working in collaboration on the frontend, one side needs to let the other handle all of it or none of it.

Internal benefits

Let's discuss some drawbacks first and foremost. Not everyone is as adept to this way of developing applications. A starting hurdle is to be expected. There is also a risk that some will resist the suggested architecture based the restrictions they put forward, difference in opinions as to whether this is the right approach or simply because it does not sit well with their development style. There will be a defining change in how we use a CMS, and how we would work in a scenario where we deliver both the backend and frontend.

We would be required to work the same way, through the same architecture and the same methodology regardless of which parts of the stack we take responsibility for delivering. If no common ground can be found, even given the exceptions mentioned, we might even lose personnel in this process. Even if I have hopes this would not occur.

There are some consequences regarding CMS handling which will be covered in a separate section.

In my honest opinion the benefits far surpass the mentioned drawbacks. So let's get to them.

Clearly defined delivery expectations

Either we as tech, deliver the entire frontend of a solution, HTML/CSS prototypes or nothing at all. Gone are the days of developing applications which in the end is torn apart, cut to pieces and reimplemented into any given CMS templating engine.

It will also be simpler to define and explain which CMS we could develop and configure internally, which we would always use a partner for the backend and which we won't support at all.

Flexibility

As we decide on a shared and standardized way of developing applications on behalf of our customers we will all be working on the same set of frontend technologies. The overhead of a steep learning curve to understand and master CMS specific templating engines, in order to create value to customers would be greatly reduced. We should be highly effective and flexible in moving from one solution to the next in our roles as frontend developers.

Specialized experts

When working with a defined subset of technologies and a set architecture we would specialize in them to a greater extent. It would also be natural to believe that fractions would form in terms of pure frontend, fullstack and CMS/backend developers. Some would want to be involved in several of these groups, some in perhaps only one of them. I would argue that this in the end is a good thing. Becoming even more specialized; opens doors where we can start creating various contents in the name of sharing and contributing to the ecosystem. It might even prove to give us a unique standing in the technical community or if nothing else aquire high confidence and experience in what we are doing, becoming masters in the art so to speak.

Talent, training, onBoarding and recruitment

With a shared base of technical knowledge we would also benefit greatly in terms of finding new talent in certain technologies. The cost and time of training and onboarding would be reduced, and new people would be able to create value earlier than before since the landscape one needs to master is smaller and more defined. The actual recruitment process would also be more direct and concise as we know exactly what kind of talent we are looking for, or at least give transparent information of the technologies we are working with.

Lifecycles, quality, hosting and maintainance

The amount of **stuff** we need to maintain and know decreases significantly because the sum of variations decrease over time. We can also streamline the tools and functions required to monitor, surveil and ensure high availability. It enables ***"one"*** standard way of deployment and a standard way to handle persistance.

Another benefit would be on the side of testing. Even if we can allow a multiture of tools to perform testing, the separation of front/back-end would standarize the way we test communication and build integrations.

Adaptability and evolution

Even if the strategy propose a clear path moving forward, there should be plenty of room to explore and innovate. The proposed strategy focuses mainly on a direction and does not eliminate new paths or future expansion of technology in the future. The industry evolve and changes daily, and we need to keep up to harvest the benefits of future improvements. As our strategy must evolve as must our technical solutions. But even so we have a natural order and direction to follow.

Wrapup

I strongly believe that betting on a future of headless CMS, JAMstack architecture and high performing applications would long term put Netlife firmly on the map as technologists. Even if vendors, platforms and technologies go out of favor, we would be better able to adapt to an unknown future.

And in the end we enable a more fair world where we have to prove ourselves worthy to keep the trust of our customers. There is nothing in our technical strategy which does not align closely with the overlying Netlife strategy.

CMS

The domain of CMS does not really belong to technology. What I mean by that is that the choice of CMS platform should not be based on technical preference. All CMS platforms have strengths and weaknesses, some of which we can decide are too great of a weakness or so far from our strategy that they are ruled out, but in the end the choice should be made based on the requirement of the administering users of our customer and the features we need to support.

Refusing to work with a certain CMS platform if the architecture aligns with our strategy, is simply put a folly on our part. We should not, and cannot dictate this choice, but we should provide honest and well meaning advice. There is just too much opportunity at stake, in a demanding market, to turn our back based on technical preference.

That is not to say we are to say yes to deliver anything on any platform, but we should be able to provide our customers excellent frontend web applications whenever we can, or explicitly have chosen not to.

Design and CSS

This is a domain we share with others. As technologists we are merely translators of design, and we need to establish how collaboration with others to achieve the best result should work. We want others to contribute with their expertise as efficiently as possible. We aim at being an extension of their work and complement others strengths.

Hence the technologies, frameworks and tools we utilize needs to consider how we collaborate efficiently with others. The same goes for choices of implementation. We can suggest improvements and tools, and if successful we will sufficiently train and help others to be as efficient as possible when conducting their job just as we would expect them to view our challenges and efforts.

Resources of interest on the internetz

- General
- Platforms
- JAMstack Sites Showcase
- Static Site Generators
- CMS
- API
    - Authentication
    - Comments
    - Forms
    - E-commerce
    - Search
    - Database
- Serverless
- Videos
- Tutorials / Articles

- Podcasts

General

- JAMstack
- JAMstack resources - Videos and articles about JAMstack.
- the New Dynamic - Frikking awesome library of examples, tools and services.

Platforms

- GitHub Pages - Website hosting from your GitHub repo.
- Netlify - All-in-one platform for automating modern web projects.
- ZEIT Now - All-in-one serverless platform for modern web apps with config-free tools and workflows.

JAMstack Sites Showcase

- React - Built on Gatsby.
- Squoosh.app - Hosted on Netlify, demonstrates advanced features from a modern Web Application.
- Hopper - Built on Gatsby and hosted on Netlify.
- VSCode Power User Course - PWA built on Gatsby and hosted on Netlify.
- CloudyCam - PWA built on Next.js and hosted on Zeit Now v2 Serverless platform.

Static Site Generators

- Gatsby - Blazing-fast static site generator for React.
- Next.js - Lightweight framework for static and server-rendered applications.
- Metalsmith - An extremely simple, pluggable static site generator.
- eleventy - A simpler static site generator transforming various template files into HTML.

For a more complete list see StaticGen.

CMS

- Contentful - Content infrastructure for digital teams.
- NetlifyCMS - open source Git-based CMS.
- Sanity - Headless CMS and Content API.
- Headless Craft CMS - Element API and CraftQL alternatives introduction.
- Headless Enonic XP - Graphql powered CMS API with Enonic.
- Headlesss Hybrid Episerver CMS - headless decoupled CMS backend for frontend web development in Episerver.

API

Authentication

- Auth0 - Single sign on and token based authentication.
- Netlify Identity - Brings a full suite of authentication functionality

Comments

- Disqus - Global comment system that improves discussion on websites and connects conversations across the web.
- Facebook Comments - The comments plugin lets people comment on content on your site using their Facebook account.
- Utterances - A lightweight comments widget built on GitHub issues. Use GitHub issues for blog comments, wiki pages and more.

Forms

- Netlify Forms - Built-in form handling on building time by parsing HTML files directly at deploy time.

E-commerce

- Flatmarket - Flatmarket is a free, open source e-commerce platform for static websites.
- GoCommerce - A headless e-commerce for JAMstack sites.
- Snipcart - A powerful shopping cart platform for developers.
- Moltin - eCommerce API for developers.
- Trolley - A shopping cart designed for the JAMstack.

Search

- Algolia - The most reliable platform for building search into your business.
- Lunr - Search made simple (on frontend).
- CloudSh - Powerful search with a few lines of JavaScript.

Database

- GraphQL - Query language for APIs and a runtime for fulfilling those queries with your existing data.
- MongoDB - Document-based distributed database for the cloud.
- Elastic - The ELK stack for infinite searching at scale.

Serverless

- Netlify Functions - Netlify lets you deploy Lambda functions without an AWS account, and with function management handled directly within Netlify.
- Amazon Lambda - Lets you run code without provisioning or managing servers.
- Microsoft Azure
    - Azure Functions - Serverless compute service that enables you to run code on-demand without having to explicitly provision or manage infrastructure.
    - Azure Logic Apps - Simplifies building automated scalable workflows that integrate apps and data across cloud services and on-premises systems.

- Google Cloud
    - App Engine - Serverless application that completely abstracts away infrastructure so you focus only on code.
    - Cloud Functions - Serverless environment to build and connect cloud services.
    - Cloud Datastore - Highly-scalable NoSQL database with automatic sharding and replication.
    - Cloud Storage - Geo-redundant object storage for high QPS needs.
    - Cloud Pub/Sub - Geo-redundant real-time messaging for all message sizes and velocities.
    - Apigee - Enterprise API management for multi-cloud environments.
    - Endpoints - API management apps built on Google Cloud.
    - Cloud Dataflow - Serverless stream and batch data processing service.
    - BigQuery - Serverless data warehousing services that help you to deploy advanced cloud data warehousing solutions for your enterprise.
    - Cloud ML Engine - Serverless machine learning services that automatically scales built on custom Google hardware (Tensor Processing Units).
- Serverless - Toolkit for deploying and operating serverless architectures.
- Cloudinary - Serverless media (images/videos) management platform. Provides SDKs in every popular language and media widgets for JAMstack to make it easy to manage media, CDN, storage, transformations, and more.

For a more complete list see Awesome Serverless.

Videos

- The New Front-end Stack. JavaScript, APIs and Markup - Matt Biilmann.
- Rise of the JAMstack - Mathias Biillman.
- Git-based or API-driven CMS - Chris Macrae.
- JAMstack Tutorial - Full site using Netlify & Hugo - freeCodeCamp.org.
- Gatsby JS Crash Course - Traversy Media.
- How We Got Here and The Future of the Web - Kyle Mathews.

Tutorials / Articles

- Ghost on the JAMstack
- Getting Started with Gatsby and Cockpit—Part 1 of 2
- Creating Static E-commerce site with GatsbyJs
- For Static Sites, There's No Excuse Not to Use a CDN
- E-commerce front-end for Vue.js, Nuxt.js and Snipcart
- Building Paul The Octopus
- JAMstack and Netlify: Do We really need another buzzword?
- The JAMstack Startup Landscape
- How I built my blog using Gatsby and Netlify
- Developer's Guide to Headless E-Commerce
- Handling Static Forms, Auth & Serverless Functions with Gatsby on Netlify

- JAMstack for Clients: Benefits, Static Site CMS, & Limitations
- Exploring Netlify CMS, a React & Git-Based Content Management System
- JAMstack PWA—Let's Build a Polling App. with Gatsby.js, Firebase, and Styled-components Pt. 1
- Dynamic Static Sites with Netlify and iOS Shortcuts; Use Netlify Functions, a Gulp build process and iOS Shortcuts to publish dynamic content to your static site
- Gatsby for Apps
- Turning the Static Dynamic
- Going JAMstack with Netlify and Nuxt
- Getting Started With Gridsome
- The Complete Beginner's Guide to Deploying Your First Static Website to IPFS
- A Broad Discussion on JAMstack & E-Commerce (Podcast & Transcript)

Podcasts

- JAMstack Radio

Shoutout to Vilson Vieira (automata)
