



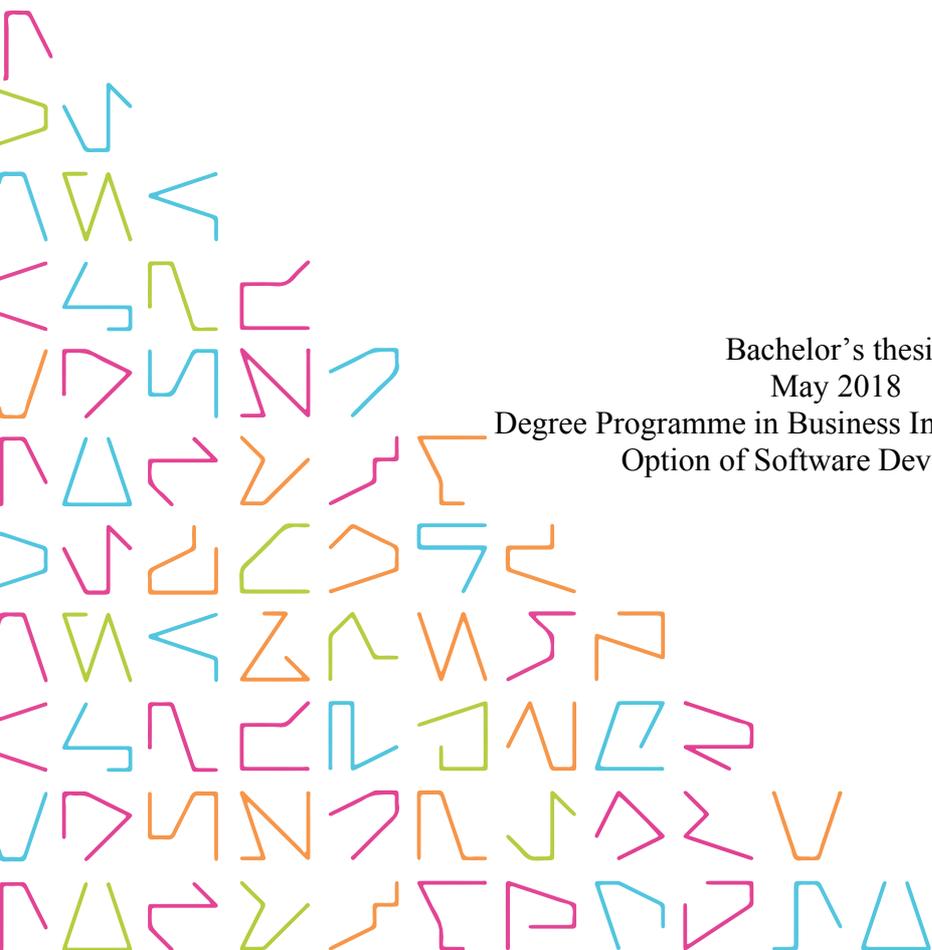
TAMPEREEN
AMMATTIKORKEAKOULU

PRODUCTIZATION OF WEB SERVICES

Atte Huhtakangas

Bachelor's thesis
May 2018

Degree Programme in Business Information Systems
Option of Software Development



TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

HUHTAKANGAS, ATTE:
Internet-palveluiden tuotteistaminen

Opinnäytetyö 34 sivua
Toukokuu 2018

Opinnäytetyön tavoitteena oli selvittää mitkä keinot auttavat olemassa olevan Internet-palvelun tuotteistamisessa. Tarkoituksena oli rakentaa Elenia Aina -järjestelmään tuotteistamistuki.

Mikropalvelut havaittiin hyväksi tavaksi viedä yksittäisiä rajapintakokonaisuuksia ylemmälle tasolle, jotta useita eri tietolähteitä voidaan käyttää. Windows-palvelimista siirtyminen Linux-palvelimiin ja Dokku-pohjaiseen konttiarkkitehtuuriin tuki useiden palveluiden pyörittämistä rinnakkain. Webpackin (JavaScriptin moduulien niputus -kirjasto) ja styled-components-kirjaston käyttöönotto helpotti useampien konfiguraatioiden hallintaa ja teemoitustukea useammalle eri tuotevariantille.

API gatewayn rakentaminen todettiin hyödylliseksi, sillä siinä voitiin tehdä käyttäjien oikeuksien hallinta ja keskittää rajapintojen kutsut yhteen paikkaan. Kirjautumispalvelun tekeminen havaittiin parhaaksi tavaksi keskittää tuotteistamisesta seuranneiden eri kirjautumispalveluiden logiikkaa.

Dokku soveltui useamman palvelun pyörittämiseen kohtalaisen hyvin. Dokkun huono tuki HTTP-proxyille kuitenkin aiheuttaa ylimääräistä konfigurointia. Järjestelmää voisi kehittää lisäämällä palveluiden tapahtumien kirjausta ja -monitorointia. Jatkuva toimitus-järjestelmän integrointi voisi edistää uusien ominaisuuksien nopeampaa julkaisua.

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Option of Software Development

HUHTAKANGAS, ATTE:
Productization of web services

Bachelor's thesis 34 pages
May 2018

The aim of this thesis was to figure out which features are important when a web service is productized. The purpose was to make the Elenia Aina extranet service a productized system.

Microservices was found to be a useful way to abstract API endpoints so that more data sources could be utilized. Moving from Windows-servers to Linux-servers and a Dokku based container architecture helped to run multiple services parallel. Utilizing Webpack and styled-components helped to bootstrap better support for configuration management and theming support for multiple different product flavors.

Building an API Gateway was found to be a good way to move user authorization to a single point and centralize the API calls to one place. Creating an authentication service was found to be the best way to centralize all of the different IAM provider logic which became apparent after the productization started.

Dokku worked relatively well for running multiple services. Dokku's bad support for HTTP proxy causes extra configuration. The Aina extranet-system could be improved by adding more logging and monitoring. A continuous delivery integration would enable faster delivery of new features.

Keywords: web service, productization, theming, microservice

Contents

1	INTRODUCTION	8
2	Productization of web services	9
3	MICROSERVICES.....	10
3.1	What is a microservice?	10
3.1.1	The purpose of a microservice	10
3.1.2	Stateless vs stateful microservice.....	11
3.2	Moving from a monolith to a microservice architecture in Aina	11
3.2.1	Moving from Windows to Linux.....	11
3.2.2	Productization	12
3.3	Multiple sources of data	12
3.4	API gateway	13
3.4.1	What is an API gateway?.....	13
3.4.2	The need for an API Gateway in Aina.....	15
3.4.3	Automatic developer documentation.....	15
3.5	Authentication service.....	16
3.5.1	Cookie authentication	16
3.5.2	Token-based authentication	16
3.5.3	Multiple authentication methods.....	17
3.5.4	Authentication service in Aina.....	17
3.6	Container infrastructure for microservices using Dokku and Docker	19
3.6.1	Dokku	19
3.6.2	Scaling microservice architecture	20
3.6.3	Docker.....	20
3.6.4	Dokku in Aina.....	21
4	THEMING SUPPORT	23
4.1	Frontend build system.....	23
4.1.1	Webpack	23
4.1.2	Replacing Aina's frontend build process.....	23
4.2	Custom styles for product flavors.....	24
4.2.1	Styled-components	24
4.2.2	Styles as an integral part of productization.....	25
4.3	Tailor-making pieces of the system	25
5	REFACTORING.....	27
5.1	What is refactoring?.....	27
5.2	Refactoring strategies.....	27

5.3 Refactoring frontends.....	28
5.3.1 Mobile device support	28
5.3.2 Refactoring Aina's frontend.....	29
5.4 Testing.....	29
6 CONCLUSIONS.....	31
REFERENCES.....	33

Abbreviations and terms

Aina	Elenia's Extranet service that is the system that was productized and is used as an example in this thesis. Elenia is an electric power distribution company operating in Finland.
API	Application Programming Interface, a way to use the functionality or data of another service.
Apollo Engine	A GraphQL gateway.
Backend	The server that does all of the state handlings and serves those to the frontend.
Brownfield project	A project that has existing code base.
Browserify	An open source tool to build JavaScript bundles from modules.
Contentful	Content management API.
Cypress	A JavaScript End to End testing framework.
Docker	An operating system level virtualization platform.
Dockerfile	A file format defined by Docker. Used to define the container structure of a Docker container.
Dokku	A self-hosted PaaS, used to deploy new applications and new versions of code.
DOM	Data Object Model. The programming interface of HTML.
E2E	End to End testing. UI testing application complete application flows from start to finish.
Frontend	A visual UI that is seen by the user.
GraphQL	A query language.
GraphiQL	An in-browser IDE for exploring GraphQL.
Greenfield project	A project that is new and has no legacy burden.
Heroku	A PaaS providing containers with plugins to run applications on.
HTTP	Hypertext Transfer Protocol.
IAM	Identity and Access Management. A service that controls and verifies user access. Also known as IDM (Identity Management)
JSON	JavaScript Object Notation.

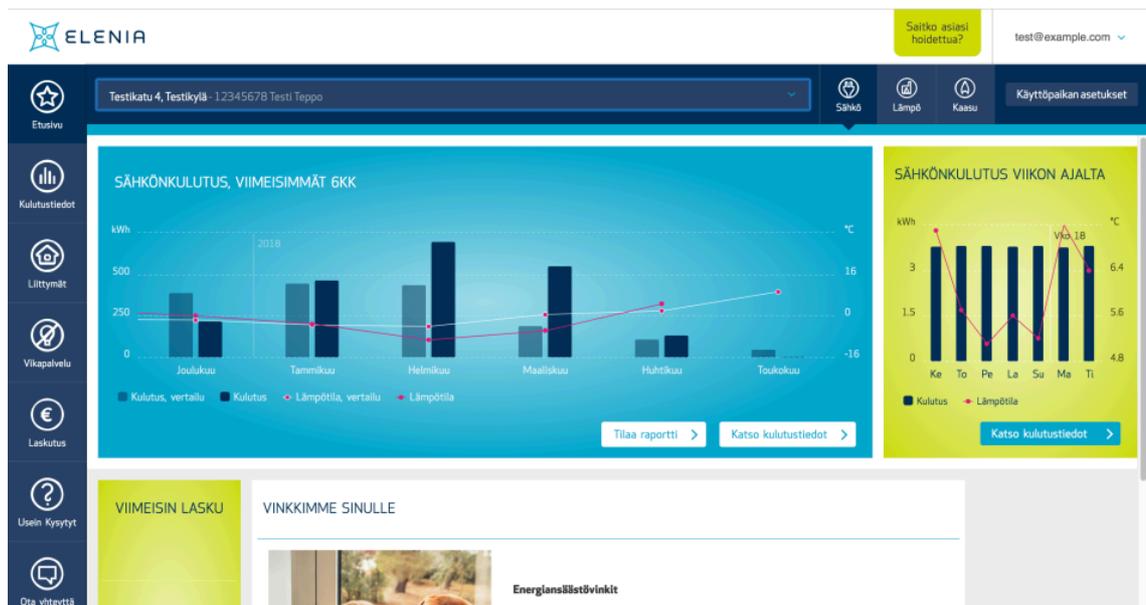
JVM	Java Virtual Machine.
Logspout	Log router for Docker containers. Enables to route Docker containers' logs to a set location.
Marshal	The process of transforming data to a format suitable for storage or transmission.
Microservice	A service that is generally made to do one thing only, e.g. handle all invoicing related data handling.
npm	A package manager for JavaScript.
PaaS	Platform as a service.
Papertrail	Cloud-hosted log management service.
PostgreSQL	An open source database. Also known as Postgres.
Procfile	A file format defined by Heroku for their services. Used to define which commands are run in a Heroku application.
Productization	Making or crafting a service into a product.
Redis	An in-memory data structure store.
Scala	A functional programming language, operating on top of JVM.
SOAP	Simple Object Access Protocol. XML-based messaging protocol.
Ubuntu	A Linux distribution.
WordPress	A Content Management System.

1 INTRODUCTION

This thesis was made as a part of the productization effort of Elenia Palvelut Oy's Aina extranet service. The aim of this thesis is to figure out the best practices on how to productize an existing service. Also finding out how to alter the current functioning service so that it can be sold as a product to other companies. The productization of Aina was started in 2017 and is still ongoing. The effort is being done incrementally and support for new services is implemented constantly.

In this thesis the productization of the *Elenia Aina* -service is used as an example (picture 1). Elenia Aina is an extranet service where Elenia's customers can track their energy consumption, browse invoices, see power outage related info and contact Elenia through various forms. The purpose of this thesis is to make the Aina-service a productized service.

By reading this thesis the reader gets an insight of what it means to productize a web service and know how some of the applied solutions affected the Aina-service. The reader will get to know what solutions worked well and what are the possible future improvements that could be carried out in this or similar cases.



PICTURE 1. The home page of Aina Extranet service.

2 Productization of web services

The concept of productization means making a service into a product. Productization of software is taking a custom-built software and building it to be a more general use product. When applied to a custom-built service, the needs in a service are generally even more specific to the original user. Making the service into a product can require setting new servers for the new customers, making new integrations and building support for custom themes. “Software productization means abstracting the common features of multiple implementations to form a base product also called as plain-vanilla product.” (Madhuda Oak, 2015).

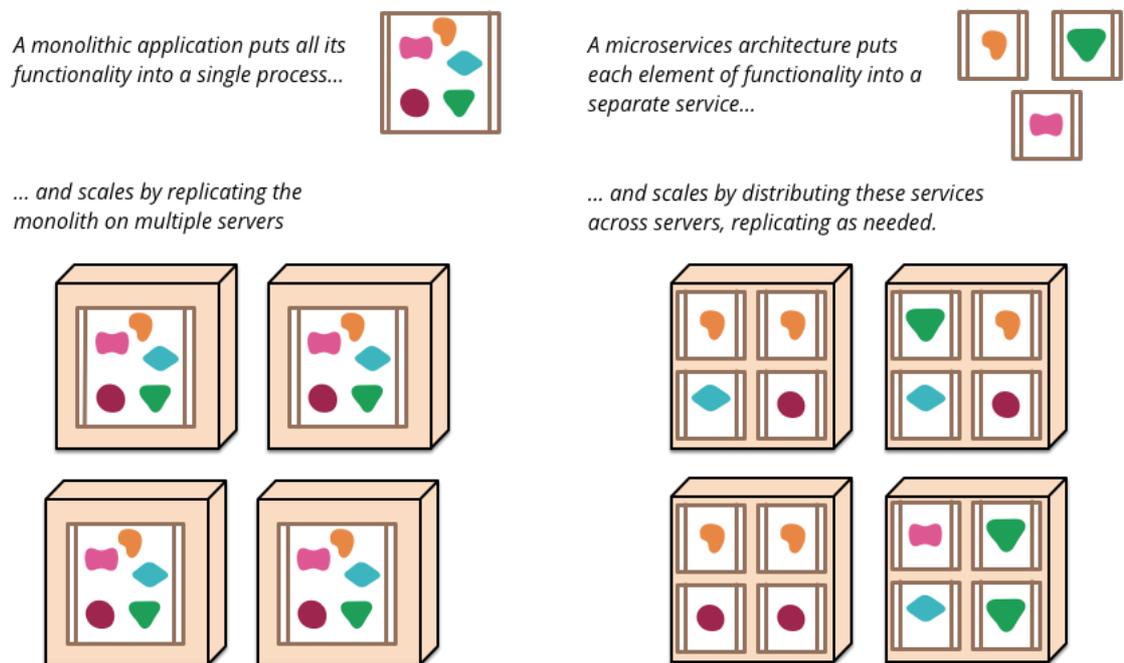
Productization can also include some parts of white labeling. “A white label product is manufactured by one company and packaged and sold by other companies under various brand names.” (Investopedia, 2008). Even in a white label product, there might be a need to customize the product to take in the other companies’ data sources. So effectively productizing can be white labeling a product.

3 MICROSERVICES

3.1 What is a microservice?

3.1.1 The purpose of a microservice

Most of the time at first the backend is built as a large system that spans multiple different responsibility areas. It might, for example in the case of Aina, handle user authentication, email report generation, and electricity consumption data handling. Once the backend gets big and there's a lot of users, simply scaling the backend with more RAM isn't feasible. The issue could be disk IO or more CPU time for example. Martin Fowler and James Lewis (2014) have described that sometimes the stress can only affect a certain area of the system meaning only scaling that part would improve the system's performance (picture 2). In this case can be beneficial to the performance to separate that part of the system as a microservice.



PICTURE 2. Scaling services and monolithic applications. (Martin Fowler & James Lewis 2014)

A microservice is another backend that generally handles just one thing. They are meant to be vertically scalable, meaning if there's a need for more performance it's possible to

increase the amount of that specific service and the container management system should handle the routing. Microservice is a good architectural solution when there's a need to abstract away multiple different data sources to provide the same form of data from one API.

3.1.2 Stateless vs stateful microservice

Microservices are sometimes *stateless*, meaning they do not store stateful data. If a microservice has a state, it could have a session stored for the user and it would have to store that data in a persistent storage of some sort, usually a database. When a microservice is stateless, it's easily scalable since the external data source can be used by multiple instances of the same service (Adam Michael Wood, 2017).

Statelessness is also a good idea for microservices in the way that it's not a good idea to duplicate code and build user access control to each microservice separately. Instead, it can be beneficial to implement it on another service that is proxying the requests to the microservices. It's much easier to maintain when the responsible code isn't duplicated so it leads to fewer maintenance fees and the possibility of forgetting to replicate bug fixes in multiple places is not an issue.

3.2 Moving from a monolith to a microservice architecture in Aina

3.2.1 Moving from Windows to Linux

Before the productization was started, Aina had a single monolith backend. There was a single backend service that was built with Scala (a programming language) running on top of JVM on a Windows Server. It used in-memory solution to store session ids to match the cookies to users, making it stateful.

To modernize this stack the plan was to move to Linux servers and use Dokku to handle the deployments and containerization of the current backend and frontend. Ansible playbooks were written to bootstrap the new servers. This was implemented and currently, the stack is now Ubuntu as the Operating System and Dokku as the container management system.

The new microservices now are partly used behind the old backend, which checks the user's authorization before passing the request onwards to the microservices. The newer microservices are gradually being moved to be used through an *API gateway*. The API gateway is also stateless microservice and instead of storing session ids in memory, we opted to use token-based authorization. This was done to support mobile clients better in the future.

The microservices now can be scaled vertically if there's a need. They do not persist any data in memory. Instead, they use PostgreSQL as the database and Redis as the cache. Each service has their own pair of PostgreSQL and Redis, if they need one, in their own separate Docker containers.

3.2.2 Productization

After moving to Dokku, the need to productize the service was realized and this change that was done before made the effort easier. Now the need would grow to make multiple servers running almost identical services. This is much simpler now as the need is to run the playbook and configure some environment variables, instead of manually running commands and configuring services through the Windows graphical user interface.

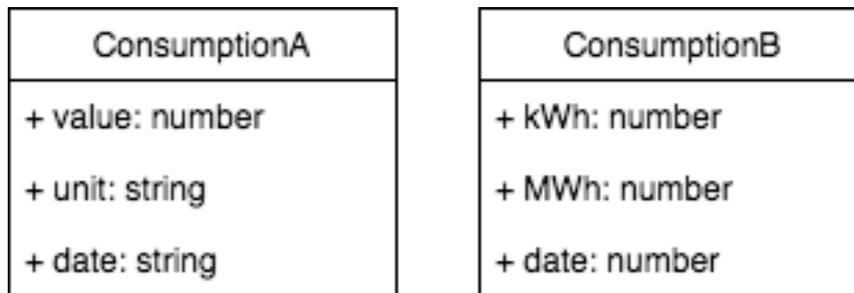
Since the new products will have different data sources, authentication methods and different versions of their systems, a need for gathering data from multiple systems realized. This was the initial need for microservices, as it would make the monolith even bigger than before if all of that logic would be in the same service.

3.3 Multiple sources of data

In a productized system there's going to be different data sources for one thing, like multiple providers for customers invoices. It's not good to build handling of different data sources in a frontend. Possibly multiple frontends would then have to handle all of the data processing separately, so it's a good idea to place it in some backend, or even a microservice.

The data from source A might have a customer's name in a field *Name* and source B might have it in fields *first_name* and *last_name* and source C in *f_n* and *sn*. Or the

consumption data might have a different form (picture 3). It's then crucial that we provide it in one form to our frontends.



PICTURE 3. Potential differences between consumption data from different sources

Multiple data sources can be handled in the business logic layer of a microservice. The same validation logic and routes can be utilized allowing code sharing. Then in the business logic layer, the target can be determined and passed to the different service. The data is then formatted to be similar for all of the different responses.

In Aina, we have the need to combine different kinds of consumption data to a similar format. The consumption data is for electricity, gas and district heating. Some of the formats are in *kWh*, some in *MWh* and some gas data is in cubic meters (m^3). The resulting data is combined from multiple different consumption data services, which have differing endpoints.

3.4 API gateway

3.4.1 What is an API gateway?

Generally, an API gateway takes a request in, let's say in the form of *https://example.com/api/customers* and from there the logic what happens next has to be defined. The immediate part after */api/* can be directly mapped to some other service and the rest of the query can be forwarded to the microservice responsible for that. Then the response, containing either success or failure, is streamed back to the API caller. It's crucial to not alter the HTTP status codes or headers if there's no specific reason for it.

In a microservice environment, the need for a single source for APIs is usually wanted. If a system doesn't have one, the user of an API such as a frontend web application will

need to care which service or domain it fetches the data from. Even though it would seem like there's only one service, the API would be split up into several domains.

Also, without an API gateway, each microservice will have to implement possible user authorizations and rate limits. It's much simpler to build microservices that are stateless and don't have to handle irrelevant things when they can be implemented once on a higher level on the API gateway.

API gateway is a service that acts mostly as a bridge between the different microservices and the user applications. It handles the API calls as the single external source of data, checking the user's claims from passed token or cookie and decides if the user can access the requested resource (figure 1).

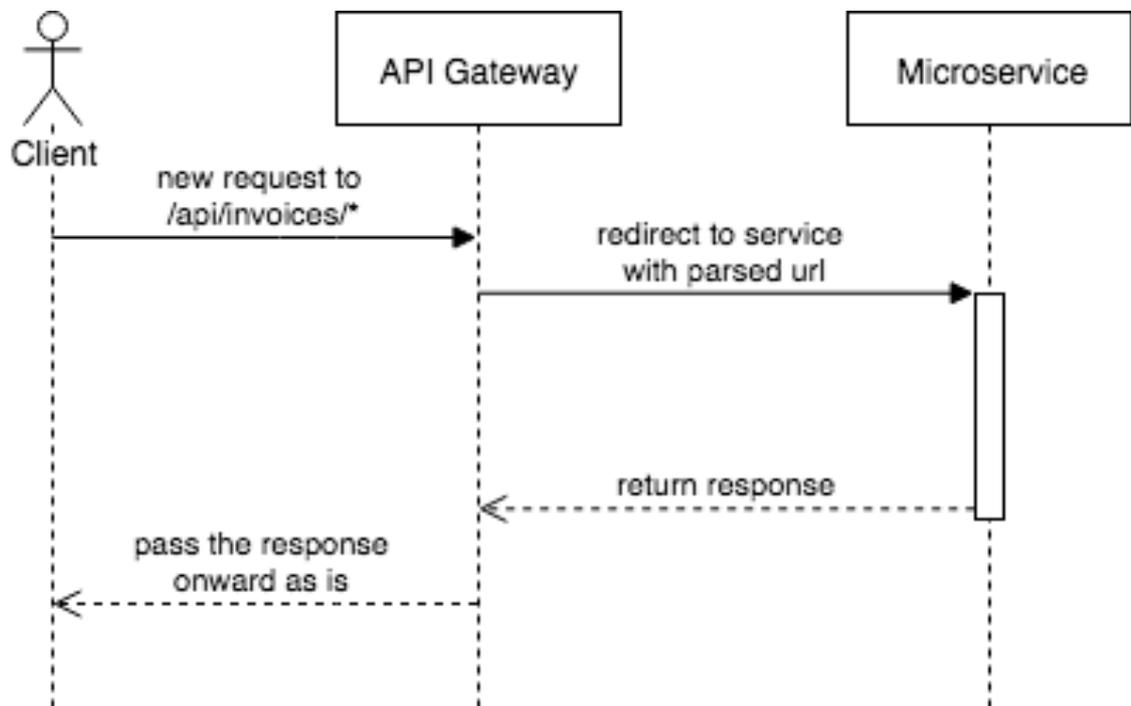


FIGURE 1. API middleware proxying the request.

Most of the time the API gateways are meant to pass on the request to the relevant microservice untransformed but it's also possible to modify the requests and responses. It's also possible to implement specific access control to APIs and rate limits if needs arise. Logging, analytics, and monitoring are also great additions to an API gateway.

3.4.2 The need for an API Gateway in Aina

Once some of the microservices were built and some were integrated into the current system we realized the biggest time sink is our current backend and decided to do something about it. The issue was none of the developers were really fluent with Scala anymore and having to marshal and unmarshal data between the requests was taking too much time.

We decided to make a new backend with Node.js, an API gateway to handle the traffic between the microservices and the frontend. After consulting our coworkers, we investigated the feasibility to use an existing product to handle the API gateway part. We found out about Kong, but it didn't meet the one requirement we had. Kong wasn't able to validate JWTs (JSON web tokens) using a custom function, which was a must for our service (Kong: JWT plugin).

The previous authentication in the old system was made with cookies. A cookie had information which customer IDs belonged to the user. The backend then had the responsibility to check if an incoming query had a customer ID and validate it against the cookie. This was done by checking if the user's customer IDs included the customer ID. Because making this kind of functionality seemed impossible with Kong, we decided to implement our own API gateway, aiming to make it as light and automated wrapper as possible.

3.4.3 Automatic developer documentation

We also wanted a central place for documentation for the new microservices. The original backend has Swagger documentation and we wanted to have similar documentation for the new services as well. After considering other options like using GraphQL (a query language) since it has GraphiQL (IDE for exploring GraphQL endpoints), we decided to build a custom feature that collects the swagger JSON from the microservices and shows that in our API Gateway.

The decision to not use GraphQL was done because Apollo Engine cannot be self-hosted, and the servers are located in the US. Apollo Engine would've handled stitching the

schema from our microservices. GraphQL could have then been used instead of Swagger to browse the APIs the microservices provide.

3.5 Authentication service

Authentication service is a service that handles the user's login related flows. The service ensures the user's rights to use the product, usually with username and password. The user is then redirected to the service. Authentication service might not be needed for services that only need one source of user information. In a productized system where the different products might have different user management systems which need to be integrated into the system, it might be useful to have.

In a productized system's authentication service, it's mandatory to make those account's data look as similar as possible. The account information contains all of the needed info to identify a user and make sure the user can access all of their information on the service.

The user information is then passed on to either the API gateway or directly to the client, depending on which authentication method is to be used, cookies or tokens. In both cases, the authentication service makes a signed JWT with a private key that is passed onwards.

3.5.1 Cookie authentication

In the case of cookies, the user is redirected to the API gateway and the gateway forms a session from the given JWT after it's verified with a matching public key of the authentication server's key pair. After the session is stored in the database, the user is redirected to the frontend with a cookie matching the domain and the client then can make requests to the API gateway that are authorized. The cookie is sent on each request to the API gateway.

3.5.2 Token-based authentication

If tokens are used, the signed JWT is passed directly to the frontend that then stores it somewhere. A web page stores it in browser's local storage. The token must be explicitly passed in requests, either in a query or in HTTP headers.

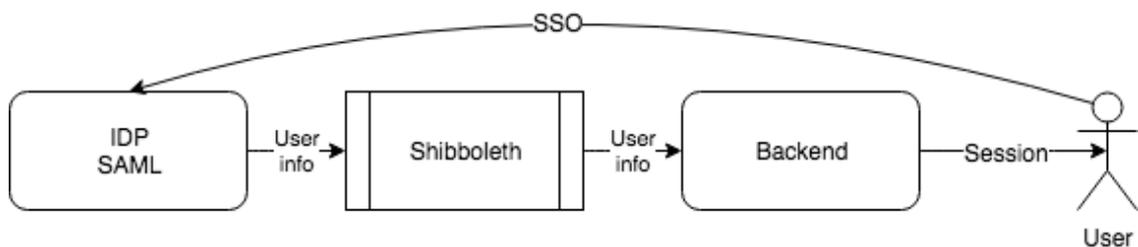
3.5.3 Multiple authentication methods

One way of building an authentication service that accepts multiple different authentication methods is to use Passport.js-library. It's an authentication middleware for Node.js (Passport.js documentation). Passport.js has a concept of authentication *strategies*, which are different authentication methods such as OAuth, SAML, OpenID and *local*. Local means implementing the Passport.js local strategy by storing and encrypting the user's authentication details in a database locally, instead of in an existing service. More strategies can be implemented but the support for different methods is exhaustive.

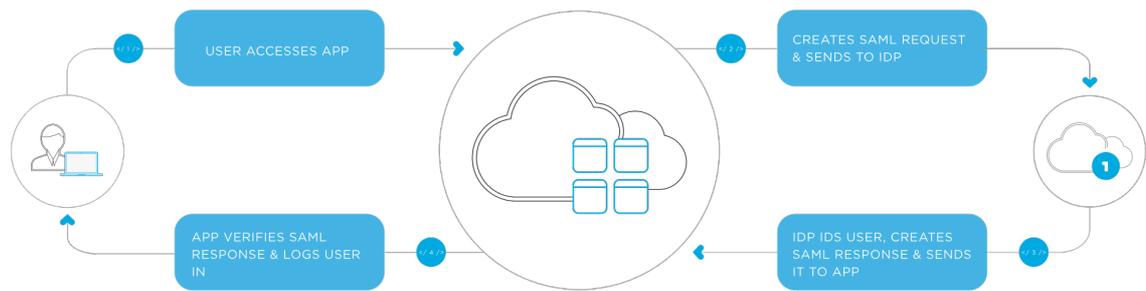
By using Passport.js, it is possible to implement custom logic that uses a different authentication method for each product variant. The API and the user objects the authentication service provides has to be as similar as possible between those product variants to reduce the need to build custom logic for handling different user types in the consuming services.

3.5.4 Authentication service in Aina

In Aina before the productization there was only one way to log in, and only one IAM (Identity and Access Management) provider (picture 4). The IAM acted as an IDP (Identity Provider) and our Shibboleth as the SP (Service Provider). As can be seen in picture 5, the communication between these services is handled with the SAML2 protocol (OneLogin Developers).



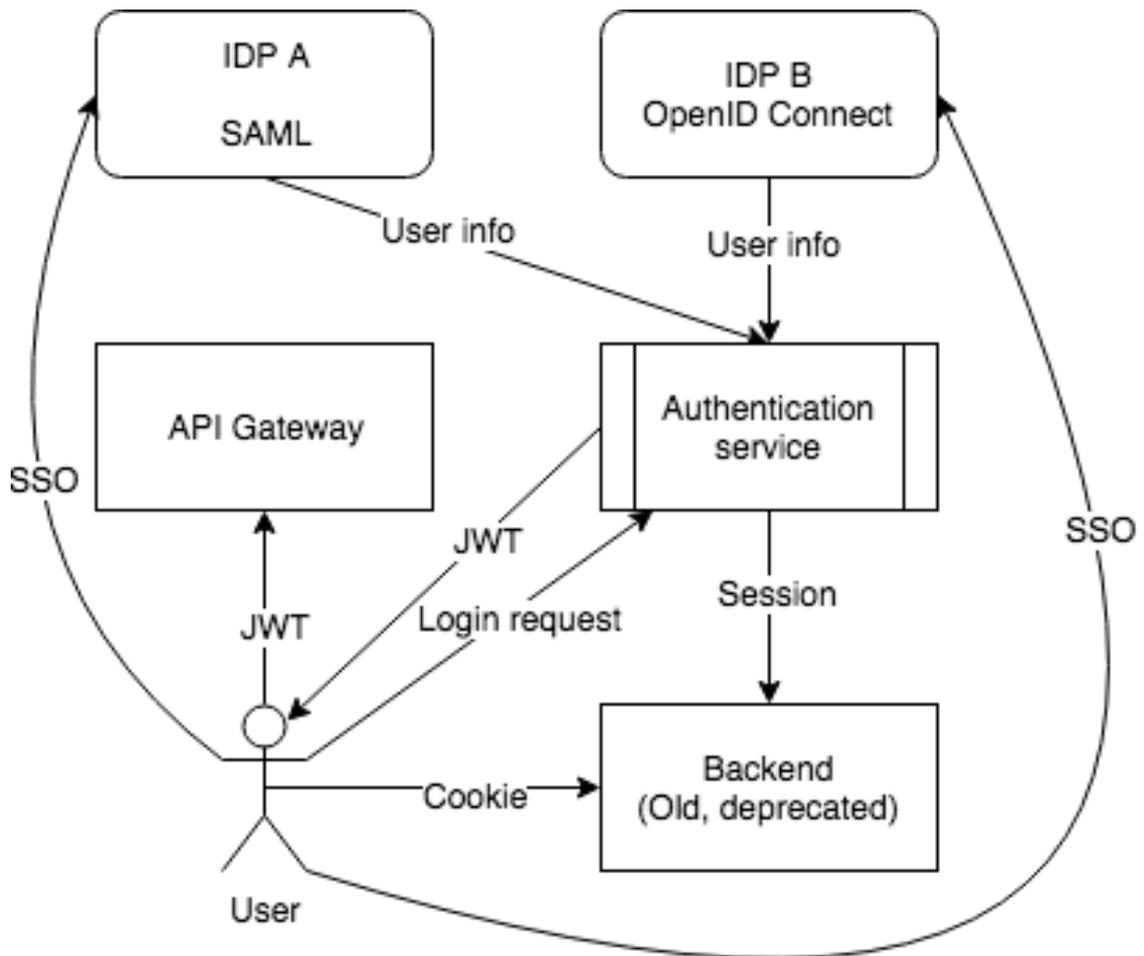
PICTURE 4. Authentication flow with only one IDP using SAML.



PICTURE 5. SAML SSO Flow (OneLogin Developers)

After the need to productize we decided to use Passport.js as the new base for the Authentication service. It was the only solution that we found that had some kind of way to use multiple different authentication methods programmatically. We built the service to support the current SAML-based login and evaluated if it works. After successfully testing we then moved to implement the next known authentication method, OpenID Connect. This is still work in progress, but the initial tests are showing successful results.

The Authentication service handles the session creation and giving the user the token (picture 6). The most important thing we want is that we can return a similar user from the Authentication service. We need to display the user's basic info and check the users claim to some customer ids and we want that data to be in the same format. The service also ensures these both have the same data for the authorization claim.



PICTURE 6. Authentication flow with Authentication service.

3.6 Container infrastructure for microservices using Dokku and Docker

3.6.1 Dokku

Dokku is a service that provides the ability to deploy self-hosted Heroku-like services. It utilizes Heroku's Procfiles to determine what to run inside the deployed application. Procfile is a file format defined by Heroku for their services. With Procfiles it's possible to define which commands should be run in an application.

Dockerfiles can also be used instead of Procfiles to specify in more depth what the running environment of an application should be like. It's often needed to do so in environments that require using system level proxy.

Heroku is a PaaS that aims to be a really simple platform to run web services on. Heroku provides a system that lets the users run their containerized applications inside their managed runtime environment. Heroku provides easy scaling through, several add-ons for the user's applications and app metrics. (Heroku product page)

Dokku is a self-hosted PaaS, marketed as self-hosted Heroku. The developer experience is intended to be simple through simple deploys after setting up an app, namely just having to push to a git remote. (Dokku product page)

Dokku provides the ability to configure multiple different instances with different environment variables, making the same service behave in a different way. We use this in Aina to host the different products different services to hook up to different data sources and use different hostnames.

3.6.2 Scaling microservice architecture

Scaling a service is making the service be able to serve more requests or users. If a service is running on too few containers, the service might reach its maximum capacity of handling requests. This can be avoided by scaling the service vertically.

Using a container system, such as Dokku, means it is possible to scale the service, by vertically setting up more process containers. With Dokku it's possible to define the number of processes per service that should be running. Dokku operates as a load balancer when there are multiple instances of a process in a container.

3.6.3 Docker

Docker is a platform enabling the use of containers. Docker uses images that are executables and containers which are runtime instances of images. The containers are run on Linux using the host machines kernel. (Docker documentation)

Docker is configured using Dockerfiles. Dokku can be run without them but in several of Aina's microservices, we need a custom one. Some of the APIs we utilize require the date calls to be in Finnish time. Most of them require a private npm package that provides

common utilities for express. We need to pass an environment variable of an access token with which we can then download the package during our builds.

3.6.4 Dokku in Aina

The decision to use Dokku in Aina was made since it deemed good enough in another project for Elenia. The change from the previous system, which mostly didn't utilize containers is drastic. Previously the backend was running as a Windows Service and the frontend was being served by Apache. The deployment process was manual and involved lots of steps the developer needed to carry out (figure 2).

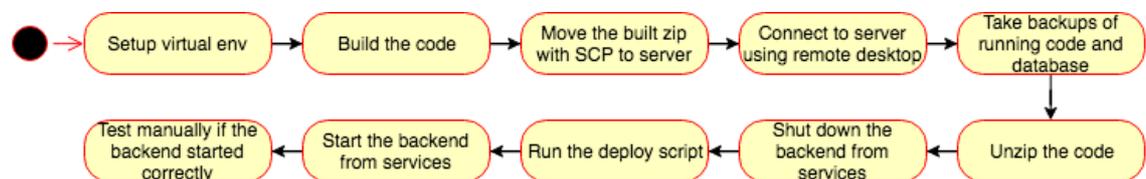


FIGURE 2. Deployment process for old Windows servers.

Now the backend and frontend have their own Docker containers. The amount of services that have been split away from the backend is also significant and Dokku has helped to run the connections between them. The current deployment process involves fewer steps for the developer and the deploy times take less time from the developer (figure 3).

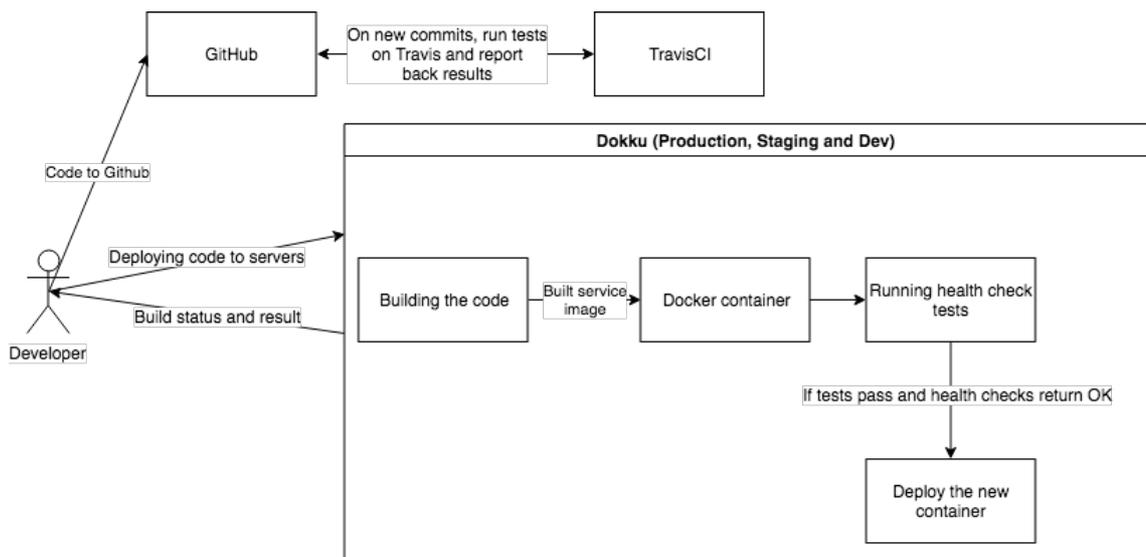


FIGURE 3. Current Deployment process on Dokku environment

Using Dokku in an enterprise environment as a Heroku replacement has worked mostly fine. Dokku doesn't work as straightforward with HTTP proxies as it does without. (Jose Diaz-Gonzalez, 2016) Elenia servers use HTTP proxy server for outgoing connections. The proxy is set up for security reasons. A proxy server is a layer between the Internet and the servers. They relay the incoming and outgoing requests. In case of a security breach, an attacker could only gain access to the proxy server, instead of the application server. (Derek Malkowski, 2016)

Because Dokku doesn't handle proxy servers too well., this has resulted in lots of extra work to define the HTTP proxy as environment variables in multiple places. To define an HTTP proxy in Dokku's build phase, it's required to define the variable in Dokku's docker-options build phase, which is tedious to maintain. The frontend builds need the environment variables during build time, so they have to be defined in Dokku's *docker-options*, taken in as *ARG* (Docker argument) in Dockerfiles and set to the environment in the Dockerfile.

We have a plan in Aina to build a utility to help with this in the future or investigate the possibility to define the HTTP proxy in some missing place. This way we wouldn't need custom Dockerfiles for every service we set up.

4 THEMING SUPPORT

4.1 Frontend build system

4.1.1 Webpack

Webpack is a JavaScript module bundler. “[Webpack] builds a *dependency graph* that includes every module your application needs, then packages all of those modules into one or more *bundles*.” (Webpack documentation) With Webpack it’s possible to create multiple build configurations, for example with varying environment variables and browser targets. In the world of frontend build systems, Webpack is one of the most common ones. As it’s relatively new, released in 2012, it has improved over the years. Webpack is currently at version 4.

4.1.2 Replacing Aina’s frontend build process

In Aina, we used to have Grunt as the build system with manual scripts to handle assembling the development and production builds. Browserify was used to bundle the JavaScript sources to a single module. We replaced Grunt and Browserify with Webpack as it can handle more than one configuration more gracefully without the user needing to build all of that via code. Rather the user can build and extend configuration files to create a system. Grunt was replaced mostly as it’s an outdated platform, most of the development has halted. Webpack also provides the ability to include CSS, fonts, images imported in the JavaScript files.

We now build different product variants based on the environment variables that are read during the build time. We have a base config which we extend in the product flavors, using custom entry points for each JavaScript bundle. This allows us to only include certain parts of the application for a single product.

4.2 Custom styles for product flavors

4.2.1 Styled-components

Styled-components is a JavaScript library that provides CSS in JavaScript (Varayut Lerdkanlayanawat, 2017). With styled-components it's possible to make *styled components* with React as the library's name suggests (picture 7).

Less is a language extension for CSS. With Less, it's possible to define variables like in modern CSS, but they are essentially limited to values like colors and numbers. The option that can be done with Less is to modify class names of a DOM-node and assign specific styles that are predefined to the node.

Because styled-components is defined in JavaScript's template literals, using functions to use certain styling definitions whenever a condition is filled is possible (Styled-Components documentation). This allows for greater flexibility and less manual work of modifying and managing the resulting class names.

```
import styled, { css } from 'styled-components';

const Container = styled.div`
  background: #bada55;
  border: 1px solid gold;
  {props => props.product === 'myproduct' && css`
    background: red;
    float: left;
  `}
`;

// Resulting CSS
.Container {
  background: #BADA55;
  border: 1px solid gold;
}
```

PICTURE 7. Defining a component using styled-components.

4.2.2 Styles as an integral part of productization

Styling is an important piece of making products look different from one another. Aina used to have styles made with Less but due to styled-components providing much more robust theming support a change was made to it. Less only supports variables that can be used. These variables can be used separate stylistic choices, such as colors and backgrounds.

Instead of continuing to use Less like Aina used before, we changed to use styled-components incrementally where we had to make changes. This allows us to use JavaScript in better ways to modify how some of the products look. We can dictate that for a certain product variant we want a gradient background and for rest of the products we don't.

4.3 Tailor-making pieces of the system

In a productized system the need to make specific features for certain customers is often needed. In this case, it's crucial to make this in a way that it doesn't end up complicating the code so much that it'll become unmaintainable at some point.

One solution in the frontend is to define a product configuration file which has feature flags and other per-product configurations. "Feature Toggles (often also referred to as Feature Flags) are a powerful technique, allowing teams to modify system behavior without changing code." (Pete Hodgson, 2017). Using configuration files for changing the app behavior like with feature flags is an effective way to build product specific features (picture 8). With these, it's possible to specify which features like pages in a web application a product variant can display.

```

// The product configuration
const productConfig = {
  productA: {
    counter: true,
    /* ... */
  },
  productB: {
    counter: false,
    /* ... */
  }
};

// Current product from environment variables, here using example string
const currentProduct = 'productA';

// A function to determine if product should see feature
const showFeatureForProduct = (product) => (feature, { defaultValue = false }) =>
  productConfig[product][feature] || defaultValue;

const showFeature = showFeatureForProduct(currentProduct);

const App = () => (
  <div>
    Welcome to app!
    {/* Show the feature only if counter is enabled for current product*/}
    {showFeature('counter') && <Counter />}
  </div>
);

```

PICTURE 8. Product configuration usage to conditionally render an element in React.

Often web services have some kind of content that is only relevant to one product, so maintaining these separately would be an increasing burden on the development team. It's much more cost-efficient to build or integrate some kind of CMS system where the people responsible for the product can maintain the content that is visible. This way the developers aren't required to change the contents of some text or other content. Some solutions like WordPress and Contentful exist for this and should be considered as they are cheaper than making one's own service. Sometimes the complexity might not allow using such readymade systems, so a custom interface and database is needed.

5 REFACTORING

5.1 What is refactoring?

Refactoring is making changes to the underlying code of the program, to make it easier to understand and less time consuming to add more features, without changing what it does. (Martin Fowler, *Refactoring: Improving the Design of Existing Code*, 1999)

” A code smell is a surface indication that usually corresponds to a deeper problem in the system. “(Martin Fowler, *CodeSmell*, 2006) Code smells are usually a sign of code that can be inspected whether it should be refactored. Refactoring can be improving the quality of the code by removing code smells, such as repetition, to make the codebase more maintainable and make it easier to add more functionality. Refactoring eventually has to be done at some point, if it’s not done the codebase will become unmaintainable and adding new features and fixing bugs will start to take increasingly more time.

In a *brownfield* project, there can be an existing code base that has some legacy burden, so some decisions were made without taking the new aspects or requirements into consideration when building those. Over the years features could’ve been just added to “hack points” because those are the easiest. This is where having an architect working with the team day to day – keeping the big picture on track (Michael C. Feathers, 2004, 215).

The opposite of a brownfield project is a *greenfield* project which is a project that has no existing code and as such, no decisions have yet been made. If the requirements change, a brownfield project most likely will need some refactoring of the code to possibly higher abstraction levels.

5.2 Refactoring strategies

It is essential to refactor one’s code all the time. If a codebase is not refactored, adding new features will end up making the code extremely unmaintainable. As can be seen in figure 4, this will increase costs in the long term (Zsolt Herpai, 2010).

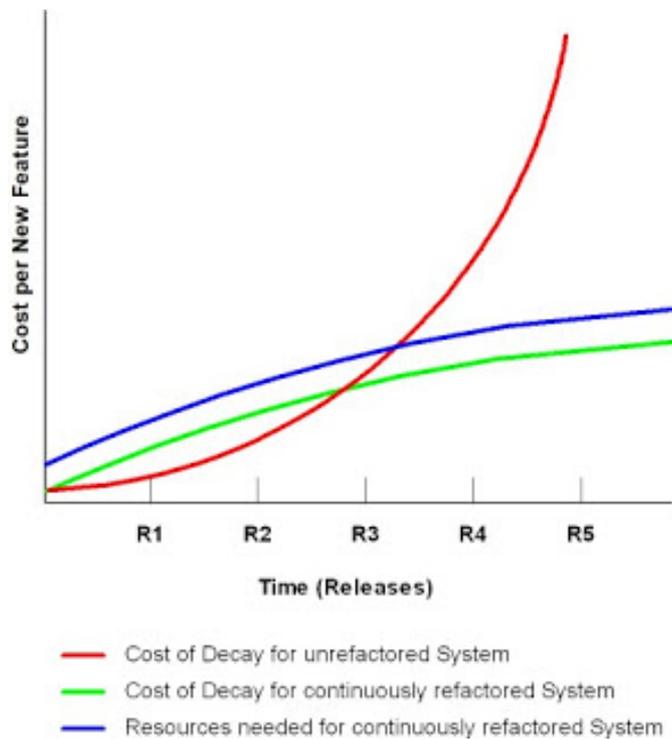


FIGURE 4. Cost per feature with or without refactoring. (Zsolt Herpai, 2010)

A good point to refactor is when a feature is added. After that, the implementation should be refactored to make more sense so that future additions are less time consuming to integrate.

5.3 Refactoring frontends

5.3.1 Mobile device support

If a web app was once designed for desktop use only and then later it's required to be used with mobile devices as well, it's going to require refactoring. In the case of React, the app will need styling improvements but also some functional changes. If an app uses tooltips, those will need to be changed to be shown on click, instead of on hover, for mobile users. Mobile devices don't support hover events (Karal Max, 2015). Instead, this functionality must always be replaced with touch events or always placing the hover-content to the screen.

5.3.2 Refactoring Aina's frontend

In Aina's frontend, we must refactor the page to support mobile layouts as well. Currently, the Aina Mobile is implemented with web technologies as well, so there's a possibility to use the React components from there as well. Still, there are lots of graphs which currently heavily rely on hover events to show extra data about the graph. This needs to be implemented some other way for touch devices. The most likely solution is to provide the tooltip information below the graph itself when a data point is touched, similarly as it would display a tooltip when hovered.

5.4 Testing

Whilst building a system which does have to use different sources of data and still hoping to maintain the same API for the backend it is crucial to have tests. Manually checking the code for bugs won't catch nearly all of the potential bugs or issues. An effective testing strategy is needed and a way to implement it. Automatic testing also can give feedback in minutes, instead of weeks. (Ham Vocke, 2018)

As can be seen in figure 5, automatic testing has a cost of setting up but it starts to pay off when the technical debt starts to accumulate (Albert Albala, 2014). It can be really hard to try to get rid of technical debt by refactoring if there are no automated tests because existing functionality might break. A thorough manual testing might not cover all of the cases testing would catch and vice versa.

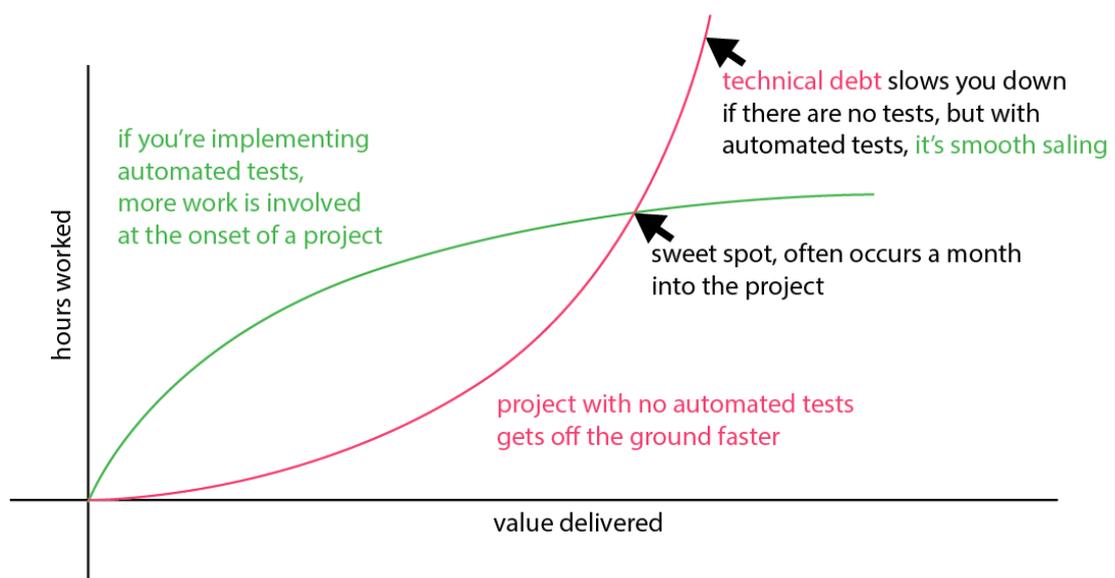


FIGURE 5. Effect of automated testing on technical debt. (Albert Albala, 2014)

If there are no tests before splitting the system to microservices or to use multiple data sources, this is a crucial moment to implement them. It's important to ensure the data always looks as same as possible.

We made a plan to write tests to the microservices to ensure similar data output from the different data sources. The consumption data is the most critical aspect in this sense.

6 CONCLUSIONS

The productization of Aina is still ongoing. We're making progress on the systems that operate behind the scenes and we've made the bootstrapping of new microservices easier for us. We still need access to some APIs which would enable us to move forward with the more business critical aspects.

Moving the deployment process from a manual process to a more modern process has helped speed up the iteration cycle. There are some initial setup costs, but they are worth it in the long run. This has helped us test new features faster than before.

Webpack has allowed us to make configurations to create multiple builds based on environment variables for our frontend application. Some theming has been replaced with styled-components and it has proved to be useful in the cases where we need more customization per product flavor.

Authentication service has been integrated into the system. It's fully operational and is awaiting the other IAM integrations. API gateway is deployed to staging and is slowly being taken to use.

Dokku has proved to be quite nice in containerization. It doesn't provide the same level of easiness of use than services like Heroku, but it still does its job. Dokku doesn't work too well with HTTP proxies so it has caused extra work to configure the proxy settings in multiple places. Overall it is still possible to recommend trying out Dokku as the infrastructure unless there is an HTTP proxy that must be used for outside connections.

Logging is an area of improvement for the service. Currently, the logs are stored in Docker's logs somewhere on the filesystem. Those logs have a set maximum length and they are overwritten. A service like Logspout that would send the logs to Papertrail would store the logs so that they could be read later. Papertrail also provides the ability to search and set alerts based on the log events.

We started writing more tests along with the creation of new microservices. The new tests are now run on TravisCI whenever new code is pushed there. We can only run unit- and

integration-tests, as integration tests require the applications to be on the same network as the production. Since the access to the servers is limited with IP address restrictions we also cannot set up a full CI (Continuous Integration) to the staging server. Travis IP address block would need to be whitelisted as well. This would improve the speed of deployments even more as the current development version would be on the staging server.

There's a need for more tests still. Some of the APIs are using SOAP which is more time consuming to test against. SOAP endpoints often have edge cases that are hard to notice. We didn't initially make tests for some of the first endpoints that were split away from the current backend so writing tests for those is a step we must do at some point. Also having some E2E-tests (End to End) written with Cypress (A JavaScript E2E testing framework) is something that would help with keeping the business-critical services operating between new versions. We cannot make these runs on a CI currently, as the staging server's IP needs to be to the hostname in the computer's hosts file. Travis doesn't support doing this at this time. These could be run on the developers' computers but that would take away from their development time. There's no fully mocked APIs either so running a mocked environment isn't a possibility either.

REFERENCES

Adam Michael Wood. 6.6.2017. Managing stateless microservices and stateful data stores. Read 21.4.2018.

<https://www.oreilly.com/ideas/managing-stateless-microservices-and-stateful-data-stores>

Albert Albala. 26.2.2014. Eight tips to remember on your path to automated testing. Blog post. Read 17.4.2018.

<http://blog.dcycle.com/blog/52/eight-tips-remember-your-path-automated-testing/>

Derek Malkowski. N.d. Proxy servers tapped as gatekeepers to protect security and privacy. Blog post. Read 15.4.2018.

<http://thirdcertainty.com/guest-essays/proxy-servers-tapped-as-gatekeepers-to-protect-security-and-privacy/>

Docker Documentation. N.d. Docker. Read 21.4.2018.

<https://docs.docker.com/get-started>

Dokku. N.d. Product page. Read 22.4.2018. <http://dokku.viewdocs.io/dokku/>

Elenia. N.d. Aina Extranet. Extranet service. Read 17.4.2018. <https://asiakas.elenia.fi>

Feathers, M. 2004, Working Effectively with Legacy Code. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference

Fowler, M. 1999, Refactoring: Improving the Design of Existing Code. Reading (MA), Boston: Addison-Wesley Professional

Ham Vocke. 26.2.2018. The Practical Test Pyramid. Article. Read 15.4.2018.

<https://martinfowler.com/articles/practical-test-pyramid.html>

Heroku. N.d. Product page. Read 22.4.2018. <https://www.heroku.com/>

Jose Diaz-Gonzalez. Installing and running Dokku behind a http proxy. GitHub Issue Response. Read 17.4.2018.

<https://github.com/dokku/dokku/issues/2467#issuecomment-258715674>

Karal Max. 19.8.2015. How to Deal with Hover on Touch Screen Devices. Blog post. Read 17.4.2018.

<https://knackforge.com/blog/karalmax/how-deal-hover-touch-screen-devices>

Kong. N.d. Kong JWT plugin. Read 18.4.2018. <https://getkong.org/plugins/jwt/>

Madhura Oak. N.d. Software Productization vs Customization. Blog post. Read 17.4.2018.

<https://madhuraokblog.wordpress.com/2015/11/07/software-productization-vs-customization/>

Martin Fowler. 25.3.2014. Microservices. Article. Read 15.4.2018.

<https://martinfowler.com/articles/microservices.html>

Martin Fowler. N.d. Microservices Resource Guide. Read 15.4.2018.
<https://martinfowler.com/microservices/>

Martin Fowler. 9.2.2006. CodeSmell. Wiki page. Read 17.4.2018.
<https://martinfowler.com/bliki/CodeSmell.html>

OneLogin. N.d. Overview of SAML. Read 21.4.2018.
<https://developers.onelogin.com/saml>

Passport.js contributors. N.d. Passport.js Documentation. Read 21.4.2018.
<http://www.passportjs.org/docs/>

Pete Hodgson. 9.10.2017. Feature Toggles (aka Feature Flags). Read 21.4.2018.
<https://martinfowler.com/articles/feature-toggles.html>

Productize. N.d. Investopedia. Read 21.4.2018.
<https://www.investopedia.com/terms/p/productize.asp>

Styled-Components contributors. N.d. Styled-Components Documentation. Read 17.4.2018.
<https://www.styled-components.com/docs>

Varayut Lerdkanlayanawat. 27.6.2017. Styled-Components in Action. Blog post. Read 17.4.2018.
<https://hackernoon.com/styled-components-in-action-723852f2a93d>

Webpack contributors. N.d. Webpack Documentation. Read 17.4.2018.
<https://webpack.js.org/concepts/>

White Label Product. N.d. Investopedia. Read 8.4.2018.
<https://www.investopedia.com/terms/w/white-label-product.asp>

Zsolt Herpai. 6.4.2010. Why invest in TDD? Refactoring. Blog post. Read 17.4.2018.
<https://codejargon.blogspot.fi/2010/05/why-invest-in-tdd-refactoring.html>