Expertise
and insight
for the future

Tri Hoang

JAMStack Continuous Integration and Continuous Deployment with CircleCI and Netlify

Metropolia University of Applied Sciences

Bachelor of Engineering

Name of the Degree Programme

Bachelor's Thesis

1 May 2020

Author	Tri Hoang
Title	JAMStack continuous integration and continuous deployment with CircleCI and Netlify
Number of Pages Date	31 pages + 12 appendices 1 May 2020
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Kari Salo, Principal Lecturer

The purpose of this study is to introduce Continuous Deployment and Continuous Integration for JAMStack web application. These new development solutions are able to enhance reliability, keeping the development flowing fast and smoothly. The paper first deep dives into JAMStack then discuss the benefit of having a Continuous Deployment and Continuous Integration pipeline.

In conclusion, this paper emphasizes the benefit of Continuous Deployment and Continuous Integration, how to leverage CircleCI and Netlify to its full potential and putting it into actual software development.

Keywords	Continuous Deployment, Continuous Integration, Testing, CI/CD pipeline, JAMStack

Contents

List of Abbreviations

AWS	Amazon Web Services
CI	Continuous Integration
CD	Continuous Deployment
CMS	Content management system
DDOS	Distributed Denial of Service
DNS	Domain name system
IaaS	Infrastructure as Cod
JS	Javascript
REST	Representational State Transfer
SaaS	Software as a Service
PaaS	Platform as a Service
SSR	Server-side rendering

Glossary

YAML

A human-readable data-serialization language commonly used for configuration files.

JAMStack

JAMstack, a cloud-native web development architecture based on client-side JavaScript, reusable APIs, and markup.

ME(X)NStack

MongoDB(M) as Database, Express.js(E) as the server framework, a Javascript front-end framework such as React, Angular, Vue … as the client framework(X), and Node.js(N) as the server environment.

LAMPStack

Most popular web stack, LAMP stack is made of four core technologies: Linux (L), Apache(A), MySQL(M) and PHP, Pearl or Python (P).

1 Introduction

Nowadays in the advanced technological era, almost every single business is using at least some form of technology to be able to run their business efficiently. The need for better software applications to support business logic has led to a growing number of software developers all around the world. Moreover, globalization has made all businesses around the world to change their business models in great speed to adapt to the market.

Big demands come with big challenges; software developers not only have to deliver software applications fast to meet the demands but also have to ensure and maintain the quality of the software applications. This raises the need for methods to help developers meet the requirements for quality, delivery time, and service cost. Continuous Integration, Continuous Deployment, and JAMStack are important solutions to this problem.

Continuous Integration, Continuous Deployment and JAMStack connect developers with their products and services, enable ease on maintaining content for the client through Content Management System (CMS). These technologies aim to automate product packaging, quality assurance, and publishing processes, making development work more transparent, improve the quality and, above all, more working easier for everyone.

Software developers are trying to find ways to apply these technologies by asking basic questions: Where to start? What services and architect should be used? This thesis attempts to answer the above questions by evaluating the concept and market. Another aim of this study is to provide an exemplary implementation of Continuous Integration, Continuous Deployment and JAMStack in the real-world scenario.

2 JAMStack compare to Traditional stack

2.1 Traditional stack

Websites traditionally have been powered by monolithic architectures, with the user interface (UI) provided by front-end and content served by the back-end. These two parts are tightly coupled together, every change made on one of which required software developers to maintain the other.

Two of the most famous architectures are LAMP stack and ME(X)N Stack.

2.1.1 LAMP stack

Considered to be the oldest and most popular web stack, LAMP stack is made of four core technologies: Linux (L), Apache(A), MySQL(M) and PHP, Pearl or Python (P). This raises the difficulty for beginner developers and takes a long time to master all the required technologies to be able to develop a good website out of LAMP stack:

For instance, Apache Web Server acts as a director that helps route HTTP requests from the front-end to their corresponding back-end functions and controllers. The developer must learn how to configure Apache to route correctly and efficiently, in case of heavy traffic, bad Apache configuration could lead to slow experience for the end-user and at the worst unresponsive website.

Not only that, using MySQL means that there should be a server providing database service, which leads to high operation costs and harder to maintain. Database structure in MySQL is crucial and needs to be done right from the start. Poor structure usually required schemas changed, which required migration data from old schemas to newer schemas. This procedure costs a lot of time and money and some cases lead to data loss.

2.1.2 ME(X)N stack

Just like LAMP stack, the components of ME(X)N include MongoDB(M) as the Database layer, Express.js(E) as the server-side application framework, a Javascript front-end framework such as React, Angular, Vue … as the client-side application framework(X), and Node.js(N) as the server-side environment. The best part is that all of its components are opensource and all have a Javascript base.

ME(X)N has many advantages over LAMP stack, for example, MongoDB is a very unique database because it is unlike traditional relational databases that usually require SQL to interface with the underlying data. MongoDB uses a Javascript like a set of calls to pull data and stores data in JSON vs tables. This can be ideal especially if the client's website is very document-heavy such as blogs.

Javascript typically can only run in the browser. Node.js was developed as a Javascript based runtime environment that allows developers to run code outside of the browser. This allowed developers the ability to code from the back to front-end all in Javascript.

Express.js plays a similar role in Node.js that Flask does for Python. Express helps manage routes and incoming HTTP requests. This makes it easier to develop web applications and makes it much easier to develop between the back and front-end of your project.

Still ME(X)N stack retains some characteristics from LAMP stack. It still has a front-end and back-end connect to a database service. Also deploying ME(X)N stack still requires a web server framework such as Apache or Nginx. In the end, even though ME(X)N stack is more advanced than LAMP stack, it still has not solved the traditional front and back end architecture. That is why JAMStack was invented for.

2.2 JAMStack

JAMStack was invented to solve a lot of problems, some of the most challenging are:

- Providing a way for the client, who has no knowledge of coding, to be able to change content on the website, without having assistance from software developers.

- Monolithic apps are rarely conducive to superior site performance. They need to generate and deliver HTML every time a new visitor arrives on the site. This significantly slows down page loading time. [1, 1]
- Having a backend and database service results in more security issues.

- The traditional stack comprises front-end and back-end, which means higher cost to operate since there are 2 services to maintain. Also, this architect required skillful developer in operation (DevOps)

Fortunately, JAMStack can solve all the problems through its architecture. The JAMStack is not about specific technologies. It is a new way of building websites and apps that delivers better performance, higher security, lower cost of scaling, and a better developer experience [2]. [3, 17]

At its core, JAMStack pre-renders all the HTML content, after that, it is sent to the visitor on the site. This approach provides unmatched performance because the visitor browser does not have to run all the Javascript (JS) code and render the HTML. Moreover, with the integration of a headless content management system such as Contentful or Netlify-CMS. JAMStack drops the need for a back-end system and relies on the REST-API for the content of the website. The benefit of a headless content management system is huge: there is no need to maintain because the service is handled by the provider and this provides an interface for the client to interact and change the content without the assistance of developers.

Also hosting JAMStack is a big advantage compare to other stacks. Since JAMStack pre-renders all HTML content, hosting servers only need to serve the HTML and do not need to run any complicated code nor compute complex calculations, this simplified the hosting server technology. With JAMStack, security threats will be reduced to a minimum as the website will rely on static webpages and get content from CDNs. This limits the plausible methods hostile actors can attack your site.

3 Characteristics of Continuous Integration and Continuous Deployment

One of the most important problems that professional developers face is how to convert a good idea, to bring a business case into real life as fast as possible. The answer to that is to focus on build, deploy, test and release procedure.

3.1 Old software deployment pattern

In many software projects, releasing is a manually intensive process. The environments that host the software are often crafted individually, usually by operation or IS team. Third-party software that the application relies on is installed. The software artifacts of the application itself are copied to the production host environments. Configuration information is copied or created through the admin consoles of web servers, applications servers, or other third-party components of the system. Reference data is copied, and finally, the application is started, piece by piece if it is a distributed or service-oriented application. [3, 17]

This old model proved to be prone to errors. If one of many steps is executed wrongly, the whole application will not run properly. Moreover, when an error occurred, it may not be clear which is the error and which step is wrong causing development to take a big hit not only in quality but also delivering speed.

3.2 The new deployment pipeline

To prevent this problem, an approach focusing on automating the software development lifecycle is a must. And the continuous integration and continuous deployment pipeline pattern – CI/CD for short – is the answer.
CI/CD is an automated implementation of an application's build, deploy, test, and release process. Every organization will have differences in the implementation of their deployment pipelines, depending on their value stream for releasing software, but the principles that govern them do not vary [3,18]. An example of a deployment pipeline is given in (see Figure 1).

Figure 1: Continuous Integration and Continuous Deployment pipeline

Every change that is made to an application's configuration, source code, environment, or data, triggers the creation of a new instance of the pipeline. One of the first steps in the pipeline is to create binaries and installers. The rest of the pipeline runs a series of tests on the binaries to prove that they can be released. Each test that the release candidate passes gives us more confidence that this particular combination of binary code, configuration information, environment, and data will work. If the release candidate passes all the tests, it can be released. [3,18]

The deployment pipeline has its foundations in the process of *continuous integration* and is, in essence, the principle of continuous integration taken to its logical conclusion.

3.3 Continuous Integration

In the old fashion way, to implement quality checks on software development. All planning phases, building application and testing its functionalities must be done by the software developers (see Figure 2). These processes are repeated every single time if there is any small change to the application itself. Setting up and performing these tasks manually make a big impact on time efficiency and quality control.

The first step to delivering consistent and high-quality software is Continuous Integration (CI). CI is all about ensuring your software is in a deployable state at all times. That is, the code compiles and the quality of the code can be assumed to be of reasonably good quality [4, 8]

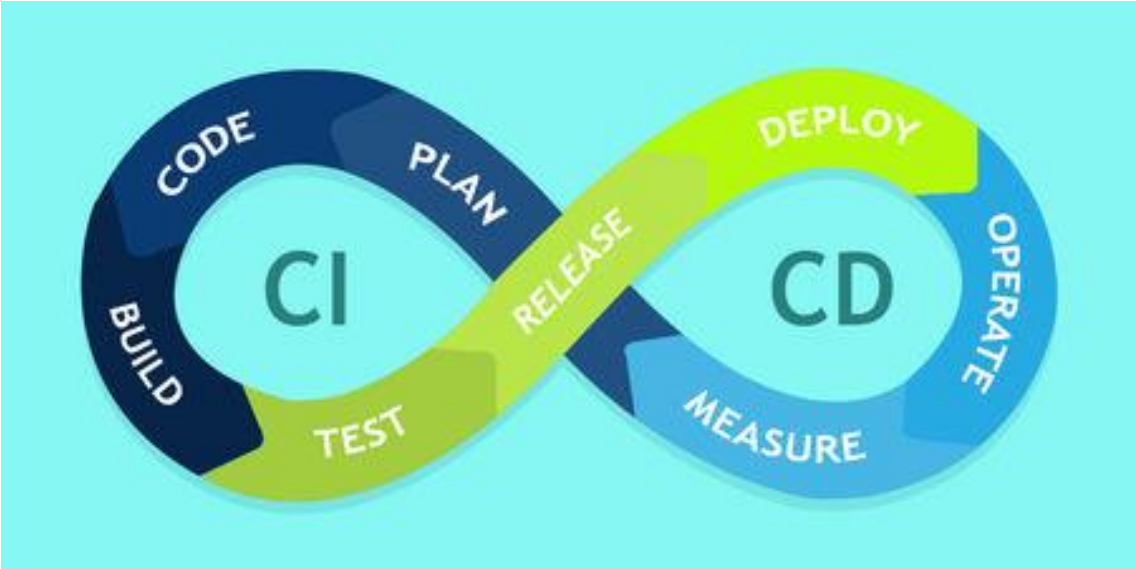

Figure 2: Continuous Integration and Continuous Deployment

Developing software requires planning for change, continuously observing the results, and incrementally course-correcting based on the results (see Figure 1). This is how CI operates. CI is the embodiment of tactics that gives us, as software developers, the ability to make changes in our code, knowing that if we break software, we'll receive immediate feedback. This immediate feedback gives us time to course-correct and adjusts to change more rapidly [5, 10].

3.4 Continuous Deployment

At the end of any software development cycle, software developers have to put their applications into production by deploying. Usually, after testing, the software application will be manually deployed by software developers. This process not only takes a lot of time but also prone to error due to human mistakes. Automate the deployment phase of a software application is an essential key to ensure delivery speed and eliminate human error out of the equation.

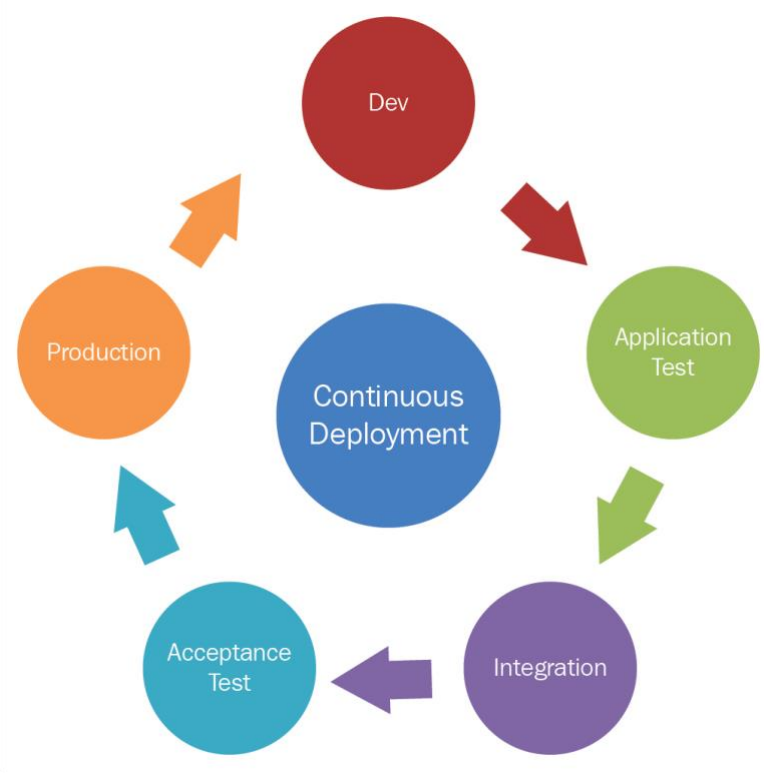

Figure 3. Continuous Deployment characteristics.

The final stage of automating software development process is Continuous Deployment. When practicing Continuous Deployment, every check into the source control is deployed to a production (like) environment on a successful build. The rationale behind this is that software developers are going to deploy the software to production sooner or later. The developers do this, the better the chance bugs will be detected and able to be fixed faster [4, 10].

4 Implementing Continuous Integration and Continuous Deployment

Developers can choose between three models of implementations: On-premises, Platform as a Service (PaaS), and Infrastructure as a Service (IaaS).

In a traditional On-premises data center, the IT team has to build and manage everything. Whether the team is building proprietary solutions from scratch or purchasing commercial software products, they have to install and manage one-to-many servers, develop and install the software, ensure that the proper levels of security

are applied, apply patches routinely (operating system, firmware, application, database, and so on), and much more. With so many manual tasks in different fields make this model prone to error and cause high operating cost.

The other options are two cloud service models: Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) (see figure 4). Each cloud service model provides a level of abstraction that reduces the efforts required by the service consumer to build and deploy systems. Each cloud service model provides levels of abstraction and automation for these tasks, thus providing more agility to the cloud service consumers so they can focus more time on their business problems and less time on managing infrastructure.

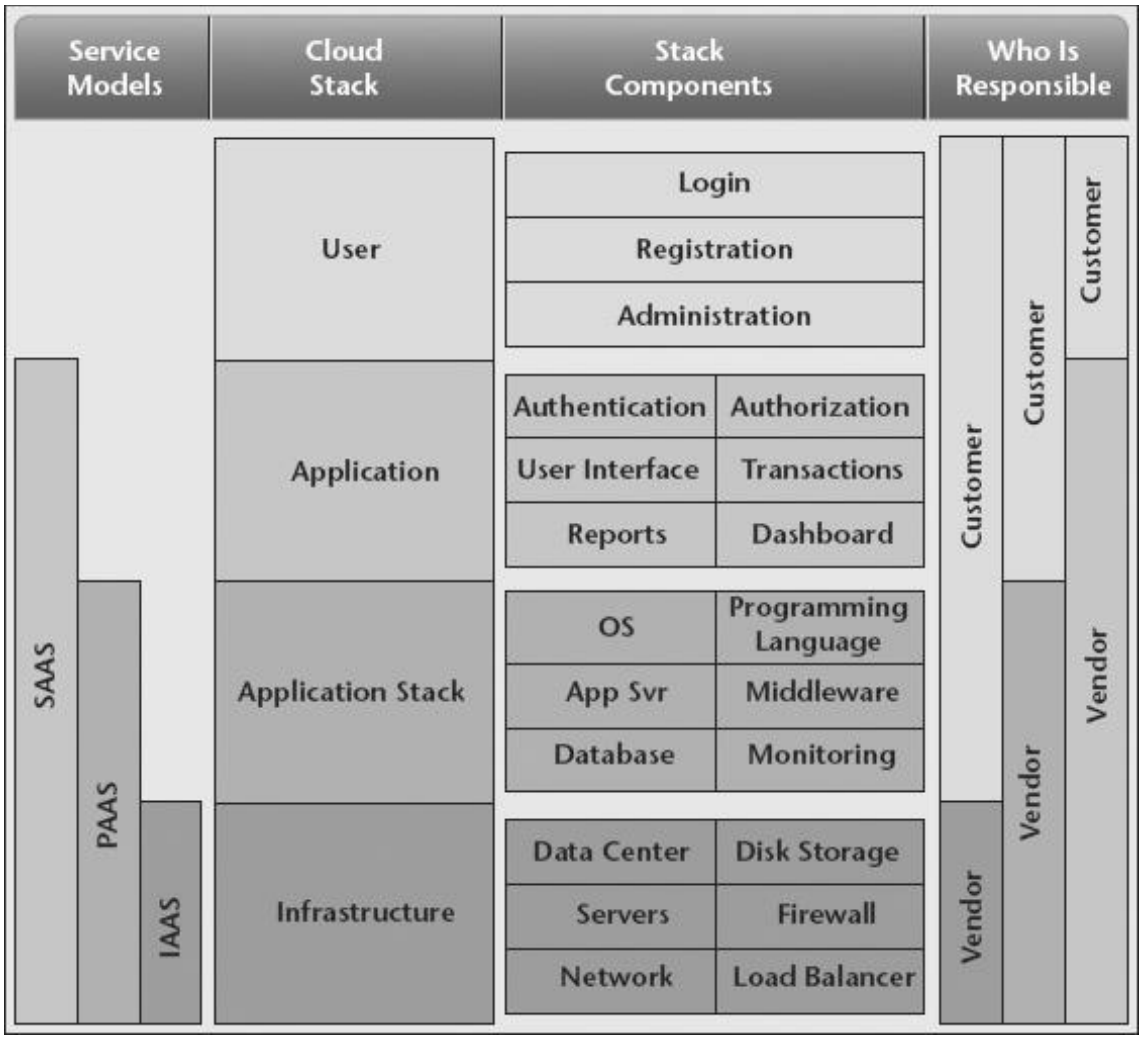

Figure 4. Continuous Deployment characteristics.

4.1 Infrastructure as a Service

With IaaS, many of the tasks related to managing and maintaining a physical data center and physical infrastructure (servers, disk storage, networking, and so forth) are abstracted and available as a collection of services that can be accessed and automated from code- and/or web-based management consoles. There is no physical infrastructure to manage anymore. Gone are the long procurement cycles where developers would order physical hardware from vendors that would ship the hardware to the buyer who then had to unpackage, assemble, and install the hardware, which consumed space within a data center [6, 45]. This reduces high TCO costs and simplified the requirement for server/dev-ops developer. There are many IaaS vendors but the most popular one is AWS with its Code Build and Code Deploy solutions for CI/CD.

However, developers still have to design, code entire applications and administrators still need to install, manage, and patch third-party solutions. This required a Cloud developer to manage the infrastructure and also introduce complexity in optimizing the infrastructure to work best with the business model.

4.2 Platform as a Service

The next level up on the stack is PaaS. What IaaS is to infrastructure, PaaS is to the applications. PaaS sits on top of IaaS and abstracts much of the standard application stack–level functions and provides those functions as a service. PaaS eliminates the need for a Cloud Developer to manage the infrastructure. Also, since the entire infrastructure has been built and optimized, there is a very low risk of error compared to building and managing our infrastructure. Right now, the market of PaaS for CI/CD is huge. There are many players with many products, and since PaaS is usually heavily optimized for a specific purpose when it comes to JAMStack, CircleCI and Netlify surface as heavy contenders for our CI/CD PaaS.

4.2.1 CircleCI

For integrating Continuous Integration into the software development pipeline, developers either have to develop their solution or use prebuilt solutions. When it comes to the Continuous Integration Service market, there are many Continuous Integration providers, but it is split into two categories: hosted and cloud base.

Hosted solutions such as Jenkins required a big infrastructure set up by developers in order to run properly. Not only that, Jenkins also required developers to build their plugin if there is no prebuilt solution, also maintaining dependencies and plugins raise a big challenge in scalability if the software development pipeline becomes too big. Fortunately, cloud-based Continuous Integration services fix all the issues. Unlike hosted solutions, cloud-based Continuous Integration runs on the cloud, removing the needs of infrastructure. Companies and developers do not have to allocate resources to build and maintain the server because it is done by the service provider.
Right now, on the market, there are hundreds of cloud-based Continuous Integration Providers such as Travis, Bamboo, Drone, etc. But stands out is CircleCI which leads its category with 4,65% of the market [7] (see figure 5).

CONTINUOUS INTEGRATION MARKET SHARE TABLE

Ranking	Technology	Domains	Market Share
1	Jenkins	15,926	65.88%
2	JetBrains TeamCity	3,396	14.05%
3	CircleCI	1,124	4.65%
4	Hudson	820	3.39%
5	Travis CI	752	3.11%
6	Atlassian Bamboo	698	2.89%
7	CruiseControl	583	2.41%
8	GoCD	312	1.29%
9	CloudBees	177	0.73%
10	Codeship	111	0.46%

Figure 5. Market share of Continuous Integration service. (Datanyze.com 2019)

CircleCI provides enterprise-class support and services, with the flexibility of a startup. We work where you work: Linux, macOS, Android, and Windows - SaaS or behind your firewall [8].

CircleCI runs nearly one million jobs per day in support of 30,000 organizations. Organizations choose CircleCI because jobs run fast and build can be optimized for speed. CircleCI can be configured to run very complex pipelines efficiently with

sophisticated caching, docker layer caching, resource classes for running on faster machines, and performance pricing [8].

```
version: 2
jobs:
  build:
    docker:
      - image: circleci/<language>:<version TAG>
    steps:
      - checkout
      - run: <command>
  test:
    docker:
      - image: circleci/<language>:<version TAG>
    steps:
      - checkout
      - run: <command>
workflows:
  version: 2
  build_and_test:
    jobs:
      - build
      - test
```

Figure 6. CircleCI/config.yml sample

Developers using CircleCI can SSH into any job to debug build issues, setting up parallelism in .CircleCI/config.yml file (see Figure 6) to run jobs faster, and configure caching with two simple keys to reuse data from previous jobs in your workflow. As an operator or administrator of CircleCI installed on your servers, CircleCI provides monitoring and insights into your builds and uses Nomad Cluster for scheduling.

4.2.2 Netlify

For integrating Continuous Deployment to software development pipeline, developers have to handle these tasks:

- Hosting service
- DNS service
- Server for the website

- Security protection

Every single task mentioned above required enormous development effort from developers. Traditionally, this was done manually which proved to be inefficient and expensive. Nowadays, there are a lot of service providers for each of these single tasks. But Netlify turns out to be the strongest since it offers an all-in-one solution from hosting, DDOS protection, DNS services, building web application.

Netlify is an all-in-one platform for automating modern web projects. Replace your hosting infrastructure, continuous integration, and deployment pipeline with a single workflow. Integrate dynamic functionality like serverless functions, user authentication, and form handling as your projects grow. [9]

By using Netlify, software developers no longer need to build and maintain a deployment service. Everything is handled by Netlify. Every time software developers publish an application, Netlify will automatically build and deploy the web application with only the configuration file written by developers.

With the netlify.toml file (see Figure 7), Netlify support the Infrastructure as Code model. Developers can enjoy the benefit of maintaining the infrastructure through a single file of code and easy to share the configuration for different projects.

Figure 7 netlify.toml sample.

4.3 The benefit of using both Netlify and CircleCI

Netlify and CircleCI are very different services at its core:

Netlify beside providing continuously building your project, it also provides Content Delivery Network (CDN) feature for your website. Netlify can host your website and re-route the traffic to edge location, ensure the End User to have the best connectivity and speed while accessing your website hosted on Netlify. Unfortunately, Netlify does not provide container service for you to run your owned code, which, you cannot run tests, auditing your code before building or integrate with other services at build time.

However, CircleCI provides just that. CircleCI provides containers services for you to run any code related to your project. You can run tests, audit security issues, report any found bug back to the developer via email or store report in other storage like Amazon S3 bucket. Being a container service means CircleCI can build your website after it passed all tests and checks set up by developers and deploy it to Netlify. Since the website is already built-in CircleCI, Netlify is not required to build the website again but deploy the already built package by CircleCI immediately. This is a significant cost reduction because both CircleCI and Netlify are Lambda services. You only paid for the computing time your website required, that means you only paid for the time CircleCI is building and testing your website, when deploying to Netlify, since it is already built, Netlify will not do any computing task but deploy it right away and because Netlify is not building anything, there is zero cost generate at Netlify. This workflow is applied to the project mention in this thesis and could be described in the diagram (see figure 7).

Furthermore, since CircleCI is a container service. We can spin up Linux container and bash/ssh code right in the container. Developers can leverage this huge advantage by writing customized scripts to add and customize CircleCI even more. For example, in *5.2.3 CircleCI Configuration,* all four scripts are written to communicate with Github using Github's API to send an easy preview of deployment directly in Github without going to CircleCI.

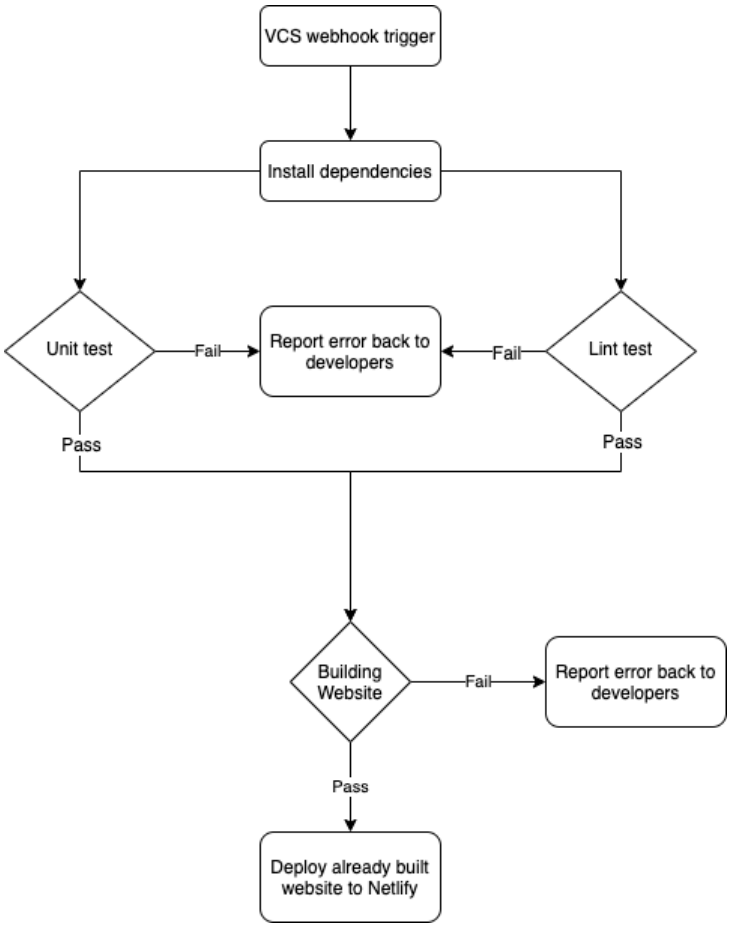

Figure 7 Workflow of CircleCI and Netlify CI/CD.

5 Implementation and Evaluation CI/CD with CircleCI and Netlify

5.1 Project overview

As brought up in the introduction, this project is an implementation of Continuous Integrations and Continuous Development with JAMStack using Netlify and CircleCI. The goal of this project is to evaluate all theories discussed above. This project is a Command Line Interface that can generate a JAMStack project with these features:

Get started immediately: You **don't** need to install or configure tools.
They are preconfigured so that you can focus on the code. However, all configuration files are exposed so you can customize them whenever you need them.

Continuous Integrations and Continuous Development pipeline support out-of-the-box: You **just** need to sign up for services, get access tokens and connect your services and the rest is handled by create-gatsby-web
Integrated with scripts to provide real **develop-staging-production** experience.

Content Management Service support: Prebuilt webhooks support for CMS, you can configure your CMS to call to CircleCI webhooks. CircleCI is already preconfigured to run CI/CD job if being called by this webhook.

5.2 Implementation method

5.2.1 Prerequisite

Before the project work is started, there are requirements to fulfill before being able to develop the JAMStack boilerplate. Requirement and chosen tools are as follow:

- An Integrated Development Environment (IDE)
- NodeJS v10 an up installed on the system
- Yarn package manager
- Github personal access token for CircleCI reporting deployment to Pull Request and Github Deployment, note down the token as GITHUB_DEPLOYMENTS_TOKEN=<the-token>.
- Netlify **development** personal access token for CircleCI to deploy preview and staging (this one belongs to Dev team Netlify's account). Note down the token as NETLIFY_ACCESS_TOKEN=<the-token>. This is your staging website.
- Netlify **production** personal access token for CircleCI to deploy production (this one belongs to Client Netlify's account - if you owned the project then development access token is enough). Note down the token

as NETLIFY_CLIENT_ACCESS_TOKEN=<the-token>. This is your production website.

Installing NodeJS on macOS or Linux

Download NodeJS from https://nodejs.org/en/download/

Install NodeJS to the system. To make sure NodeJS is installed, run the below command (should get current node version, v10.15.0 at the time of writing)

```
node -v
node version 10.15.0
```

Installing Yarn on macOS or Linux

Follow the instruction on https://yarnpkg.com/en/docs/install#mac-stable

To make sure Yarn is installed, run the below command (should get current yarn version, 1.19.0 at the time of writing)

```
yarn -v
1.19.0
```

Sign Up Github Account

Follow the instruction on https://github.com/join

Sign Up Netlify Account

Follow the instruction on https://app.netlify.com/signup

Sign Up CircleCI Account

Follow the instruction on https://CircleCI.com/signup/

5.2.2 Gatsby configuration

Gatsby is a static website generation tool for React developers. In essence, this tool lets you build React components and captures their rendered output to use as the static site content. However, Gatsby takes static site generation to the next level. In particular, it provides mechanisms for sourcing your website data and transforming it into GraphQL that's more easily consumed by React components. Gatsby can handle anything from a single page brochure site to a site that spans hundreds of pages. [10, 52]

In order for Gatsby to work correctly, it relies on the configuration file name as gatsby-config.js (Appendix 2). This file contains all the plugins needed for your Gatsby project. Since our aim to provide the best performance website out-of-the-box. We will specifically pre-installed some of Gatsby-plugins:

Gatsby-plugin-canonical-URLs: When multiple pages have similar content, search engines consider them duplicate versions of the same page. For example, desktop and mobile versions of a product page are often considered duplicates. Search engines select one of the pages as the *canonical*, or primary, version and crawl that one more. Valid canonical links let you tell search engines which version of a page to crawl and display to users in search results [11]. This plugin will auto add canonical URLs to HTML pages generated by Gatsby [12].

Gatsby-plugin-offline: Enable service-worker for caching, help web load faster and better under poor network connection. Also, enable Progressive Web App capability [13].

5.2.3 CircleCI configuration

config.yml (see Appendix 1) is configured to run in the latest Ubuntu container with browser preinstalled. The Configuration contains eight types of jobs:

- install-dependencies-test-lint: this job is to install all dependencies needed to build the project. Here, we use a feature from CircleCI called caching. Caching is to persisting data between two different workflows. When using caching, we checksum the lock file, if the lock file is changed, CircleCI will download it from the beginning. But if the lock file does not change, CircleCI will just download the persisted cache from storage without having to download and install from the internet, saving a huge amount of time as well as resources.

- checking-performance: this job is configured to run lighthouse ci to check for website performance, accessibility, SEO, best practices as well as PWA capability. Passing threshold can be configured in *lighthouserc.json* (see Appendix 3)

- gatsby-build-preview-deploy: this job is configured to only run on branches beside Develop and Master, it will run *gatsby-deploy-start.sh* (see Appendix 5) then use Gatsby to build the website. After finishing the build, it will run *gatsby-deploy-end.sh* (see Appendix 4). Both scripts are for reporting deploying status back to Github, this will save developers a lot of time, not having to go to Netlify to see the preview build. In *gatsby-deploy-end.sh* (see Appendix 4), the script is written to use netlify-CLI to deploy the recently built website (Netlify preview built). This helps developers can always checking for run-time error before releasing to production.

- storybook-build-preview-deploy: this job is configured to only run on branches beside Develop and Master, it will run *storybook-deploy-start.sh* (see Appendix 6) then use Storybook to build the website. It will run *storybook-deploy-end.sh* (see Appendix 4). Both scripts are for reporting deploying status back to GitHub. After finishing building, the built will be saved das CircleCI artifact and can be view directly by accessing the path to the artifact.

- cms-gatsby-build-staging-deploy: this job is configured to rebuilding the demo-website to fetching updated content in the CMS from Develop branch. CMS webhook will be configured to automatically trigger this job if there are any changes to the content. Storybook will not be updated because it is not affected by the updated content.

- gatsby-storybook-build-staging-deploy: this job is configured to build both the website and storybook documentation using Gatsby and Storybook from Develop branch. After that, CircleCI will use Netlify CLI to deploy to demo link with tag –prod (production).

- cms-gatsby-build-release-deploy: this job is configured to rebuilding the production-website to fetching updated content in the CMS from Master branch. CMS webhook will be configured to automatically trigger this job if there are any changes to the content. Storybook will not be updated because it is not affected by the updated content.

- gatsby-build-release-deploy: this job is configured to use Gatsby to build the website from Master branch and release the built to production.

5.2.4 Netlify Configuration

Since everything is built-in CircleCI, Netlify does not need any *Netlify.toml configuration* for building the website. Being a very optimize CDN and hosting service. Netlify handles all traffic routing, caching in edge network and best of all cost nothing since we do not run any building task in Netlify.

5.2.5 Setting up the CLI for CI/CD

1. **Clone this Repo**

```
npx create-gatsby-web <your-project-name>
```

2. **Install all packages**

   Using either yarn/npm install

```
cd <your-project-name>
yarn
```

   or

```
cd <your-project-name>
npm install
```

3. **Setting up Continuous Integration and Development**

   - Upload project to Github's repo, if you want to use CircleCI for free, Github's repo must be **public**.

   - Install LighthouseCI via this LINK, let LighthouseCI access your project repo, note down the TOKEN provided on the authorization confirmation page as `LHCI_GITHUB_APP_TOKEN=<the-TOKEN>`.

   - Run `yarn build/ npm run build`, you will get Gatsby built **public** folder in the root directory.

- Run `yarn build-storybook/ npm run build-storybook`, you will get Storybook built **build-storybook** folder in the root directory.

- Login to **Dev team** Netlify, upload the **public** folder via the image below **(DO NOT USE new site from GIT)**, after upload you should get new project deployment in Netlify, click on it and go to site settings, note down the API ID as `NETLIFY_SITE_ID_STAGING=<the-API-ID >`.

- Still in **Dev team** Netlify, upload the **build-storybook** folder via the image below **(DO NOT USE new site from GIT)**, after upload you should get storybook deployment in Netlify, click on it and go to site settings, note down the API ID as `NETLIFY_SITE_ID_STORYBOOK=<the-API-ID >`.

- Log in to **Client** Netlify, upload the **public** folder via the image below **(DO NOT USE new site from GIT)**, after upload you should get new project deployment in Netlify, click on it and go to site settings, note down the API ID as `NETLIFY_SITE_ID_RELEASE=<the-API-ID >`.

- Login to CircleCI, click on add project -> set up project -> start building -> add manually

- Go back to the main dashboard -> jobs -> click on the setting of your project -> environment variable -> add variable. Then add all the 7 ENV that we just got.

Now the project is ready for integrating with CI/CD

4. **Setting up Content Management Service Webhook**

- CircleCI personal access token for CMS to call CircleCI webhooks. Note down the token as CIRCLE-TOKEN=<the-token>.

- CMS account, recommended Contentful
- Connection keys from CMS, if you use Contentful, click on settings -> API keys -> Content delivery/preview tokens -> Add API key. Note down as:

```
CONTENTFUL_SPACE_ID=<the-Space-ID>
CONTENTFUL_ACCESS_TOKEN=<the-Content-Delivery-API-access token>
```

1. **Setup Contentful**

- Create a **.env** in project root directory, put this in the .env file:

```
CONTENTFUL_SPACE_ID=<the-Space-ID>
CONTENTFUL_ACCESS_TOKEN=<the-Content-Delivery-API-access token>
```

- Run `yarn add gatsby-source-contentful / npm install --save gatsby-source-contentful`
- Navigate to **gatsby-config**, uncomment the block of code:

```
/***** REMOVE COMMENT TO ENABLE CONTENTFUL CMS
{
  resolve: `gatsby-source-contentful`,
           options: {
                    spaceId: process.env.CONTENTFUL_SPACE_ID,
                    accessToken: process.env.CONTENTFUL_ACCESS_TOKEN,
           },
  },
******/
```

- Login to CircleCI, main dashboard -> jobs -> click on the setting of your project -> environment variable -> add variable. Then add your CONTENTFUL_SPACE_ID and CONTENTFUL_ACCESS_TOKEN.

2. **Setup webhook**

Webhook for staging website

- Login to Contentful, settings -> webhooks -> add webhooks
- Details -> name `Trigger CircleCI Build Develop Branch`
- Details -> URL -> POST -
  `> https://CircleCI.com/api/v2/project/github/<org-name-or-your-account-name>/<repo-name>/pipeline`
- Triggers -> Select specific triggering events -> tick all Publish + Unpublish
- Headers -> add custom header -> Name: `Circle-Token` -> Value: `<the-token-value>`
- Content type -> application/json
- Payload -> Customize the webhook payload:

```
{
  "branch": "develop",
  "parameters": {
    "trigger": false,
    "cms-develop": true
  }
}
```

- Click save, voila! you got the **staging** webhook setup. Every time there is a change in contentful, the webhook will trigger CircleCI to run the pipeline and deploy the new content.

Webhook for production website

- Login to Contentful, settings -> webhooks -> add webhooks
- Details -> name `Trigger CircleCI Build Master Branch`
- Details -> URL -> POST -
  `> https://CircleCI.com/api/v2/project/github/<org-name-or-your-account-name>/<repo-name>/pipeline`
- Triggers -> Select specific triggering events -> tick all Publish + Unpublish
- Headers -> add custom header -> Name: `Circle-Token` -> Value: `<the-token-value>`
- Content type -> application/json
- Payload -> Customize the webhook payload:

```
{
  "branch": "master",
  "parameters": {
    "trigger": false,
    "cms-master": true
  }
}
```

After setting this up, every time you push, CircleCI will check for lint + testing error, if passed you can click details in CircleCI check and see the Web URL deployed on Netlify.

Note that for master branch, CircleCI will build the App and release to Netlify with tag --prod for Production deployment.

5. **Open the source code and start editing!**

```
yarn start
```

or

```
npm run start
```

Your site is now running at http://localhost:8000!

Open the React-Gatsby-Typescript-CircleCI-Netlify-Boilerplate directory in your code editor of choice and edit src/pages/index.js. Save your changes and the browser will update in real-time!

5.2.6 Project structure

A quick look at the top-level files and directories in this boilerplate.

```
.
├── .CircleCI
│   └── config.yml
├── .storybook
│   ├── addons.js
│   ├── config.js
│   └── webpack.config.js
├── config
│   └── testing
│       ├── setupTests.js
│       └── __ mocks __
│           ├── file-mock.js
│           ├── gatsby.js
│           └── styleMock.js
├── node_modules
├── src
│   ├── images
│   ├── pages
│   │   ├── index.jsx/tsx
│   │   └── your_page
│   │       └── index.jsx/tsx
│   └── components
│       ├── seo
│       │   └── index.jsx/tsx
│       └── < your component >
│           ├── index.jsx/tsx
│           ├── __ stories __
│           │   └── your_component.stories.jsx/tsx
│           └── __ tests __
│               └── your_component.test.jsx/tsx
├── tasks
│   └── deployment
│       ├── gatsby-deploy-end.sh
│       ├── gatsby-deploy-start.sh
│       ├── storybook-deploy-end.sh
│       └── storybook-deploy-start.sh
├── .eslintignore
├── .eslintrc.js
├── .gitignore
├── .huskyrc.json
├── .lintstagedrc.json
```

```
├── .lighthouserc.json
├── .prettierignore
├── .prettierrc
├── gatsby-browser.js
├── gatsby-config.js
├── gatsby-node.js
├── gatsby-ssr.js
├── jest-preprocess.js
├── jest.config.js
├── LICENSE
├── loadershim.js
├── README.md
├── package.json
├── tsconfig.json -- only available on typescript template
└── yarn.lock/package-lock.json
```

1. **./CircleCI**: This directory contains CircleCI configuration file. Note that there are 4 type of jobs: preview-staging-release-webhook

2. **.storybook/**: This directory contains all the configuration files for Storybook.

3. **./config/testing**: This directory contains all the MOCK configuration files for Jest testing.

4. **./node_modules**: This directory contains all of the modules of code that your project depends on (npm packages) are automatically installed.

5. **./src**: This directory will contain all of the code related to what you will see on the front-end of your sites (what you see in the browser) such as your site header or a page template. src is a convention for “source code”.

6. **.eslintignore**: This file tells eslint which files it should not track.

7. **.eslintrc.js**: Eslint configuration file.

8. **.gitignore**: This file tells git which files it should not track / not maintain a version history for.

9. **.huskyrc.json**: Husky configuration file. Already set up with a pre git commits hooks.

10. **.lintstagedrc.json**: Lint-staged configuration file. Already set up to auto lint and format code before commit.

11. **.lighthouserc.json**: Lighthouse configuration file. You can adjust passing parameters here. Already configured with optimum parameters.

12. **.prettierignore**: This file tells prettier which files it should not track.

13. **.prettierrc**: This is a configuration file for Prettier. Prettier is a tool to help keep the formatting of your code consistent.

14. **gatsby-browser.js**: This file is where Gatsby expects to find any usage of the Gatsby browser APIs (if any). These allow customization/extension of default Gatsby settings affecting the browser.

15. **gatsby-config.js**: This is the main configuration file for a Gatsby site. This is where you can specify information about your site (metadata) like the site title and description, which Gatsby plugins you'd like to include, etc. (Check out the config docs for more detail). SEO component already preconfigures, only production deployment will get index by Google bots all preview and staging will have **noindex** meta tag.

16. **gatsby-node.js**: This file is where Gatsby expects to find any usage of the Gatsby Node APIs (if any). These allow customization/extension of default Gatsby settings affecting pieces of the site build process.

17. **gatsby-ssr.js**: This file is where Gatsby expects to find any usage of the Gatsby server-side rendering APIs (if any). These allow customization of default Gatsby settings affecting server-side rendering. Preconfigured to convert stylesheet inline to link, preventing too long head which prevents Facebook, Twitter ... scraping data.

18. **jest-preprocess.js**: This file contains babel options to build gatsby project for Jest testing.

19. **jest.config.js**: This file contains all of Jest configurations.

20. **LICENSE**: This boilerplate is licensed under the MIT license.

21. **loadershim.js**: This file contains loader setting for Jest.

22. **package.json**: A manifest file for Node.js projects, which includes things like metadata (the project's name, author, etc). This manifest is how npm knows which packages to install for your project.

23. **README.md**: A text file containing useful reference information about your project.

24. **tsconfig.json**: This file contains all of typescript configurations for type checking.

25. **yarn.lock/package-lock.json** (See package.json below, first). This is an automatically generated file based on the exact versions of your npm dependencies that were installed for your project. **(You won't change this file directly).**

5.3 Evaluation

To evaluate the outcome of the implementation. We decided to migrate a JAMStack E-commerce website with the following specification:

- Currently running on Gatsby and is a JAMStack website
- Manually deploy on AWS
- Content served by Contentful
- Project code was hosted on Github without any CI/CD

5.3.1 Performance

These scores were measured by Google's Page Insight and Lighthouse audit by just simply migrate the current website into using the *create-gatsby-web* (with Netlify and CircleCI).
After migrating, the website is running 75% faster than the original version, certified to be a Progressive Web App and got a perfect 100% score on SEO.(see figure 8, 9)

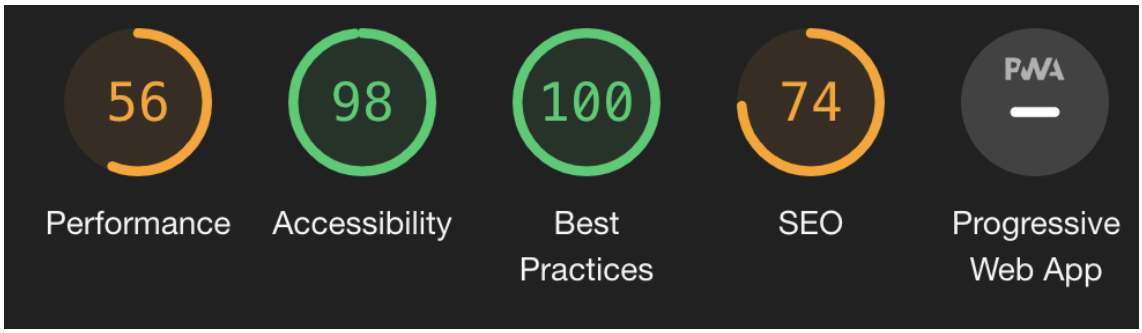

Figure 8 Original Lighthouse score.

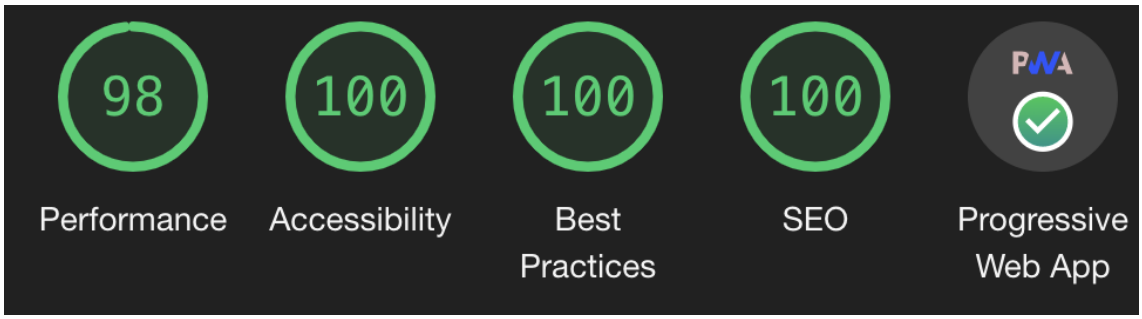

Figure 9 Lighthouse score after migrating to *create-gatsby-web*

5.3.2 Development workflow

Before migrating, the website had some downtime due to errors that got past manual inspection by developers (see figure 10).

After migrating, we observe a 0% failing rate (see figure 11)

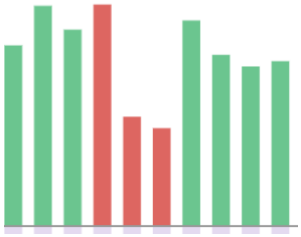

Figure 10. Website uptime before migrating.

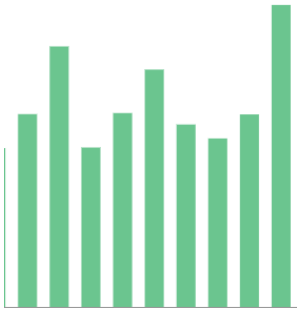

Figure 11. Website uptime after migrating.

Also, reporting from developers involving in the project are very optimistic. Pull request is being reviewed at a much faster pace and more informative than the original project, thanks to the CircleCI customized scripts. They also reported feeling less stressed because CI/CD preventing any bug that causes the website to shut down to pass through.

11/11 developers said to be very satisfied with the CI/CD implementation and would use it again in future projects.

6 Conclusion

This thesis advocates the benefits of software development with Continuous Integration Continuous Deployment and JAMStack by discussing the history of software development and how the future of business market leads to the trend of Continuous Integration and Continuous Deployment. These two methods with their many advantages over traditional software development enhance the delivery speed, reduce cost and maintenance upkeep delegation. Web development will shift from a monolithic architecture to JAMStack, which will vigorously continue to develop with the help of Continuous Integration service and Continuous Deployment service.

The paper also introduces CircleCI as Continuous Integration service and Netlify as Continuous Deployment service. These are the most popular service in its category. Moreover, these services combined under Infrastructure as Code architecture to create the JAMStack boilerplate, which proves the concept of CI/CD to be very beneficial, low cost and very efficient.

The combination of Continuous Integration, Continuous Development and JAMStack will be the future of web development, bringing more value to businesses relying on technology. The possibility of the internet is endless and these technologies offer unlimited scalability in the future market. Already, more and more new services support this pipeline model creating an ecosystem around CI/CD and JAMStack. These new services have higher abstraction levels and functionality to meet the upcoming requirements of the industry.

References

1 Biilmann Mathias. Modern Web Development on the JAMStack. O'Reilly Media, Inc; 2019

2 JAMStack Documentation. JAMStack organization; Available from: https://jamstack.org [cited December 15, 2019]

3 Humble Jez, Farley David. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Video Enhanced Edition; O'Reilly Media, Inc; 2019

4 Rossel Sander. Continuous Integration, Delivery, and Deployment. Packt Publishing; 2017

5 Duvali Paul M, Glover Andrew, Matyas Steve. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley Professional; 2007

6 Kavis Michael. Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS). Wiley; 2014

7 Continuous Integration market share report 2019; Datanyze.com; Available from: https://datanyze.com/market-share/ci [cited December 15, 2019]

8 CircleCI.com; Available from: https://CircleCI.com/docs/2.0/about-CircleCI/#section=welcome [cited December 15, 2019]

9 Netlify Documentation. Netlify.com; Available from: https://docs.netlify.com/#get-started [cited December 15, 2019]

10 Boduc Adam. React 16 Toolings; Packt Publishing; 2017

11 Google Web Dev best practices. Web.dev; Available from https://web.dev/canonical [cited May 8, 2020)

12 Gatsby Documentation. Gatsby.js.org; Available from https://www.gatsbyjs.org/packages/gatsby-plugin-canonical-urls/ [cited May 8, 2020)

13 Gatsby Documentation. Gatsby.js.org; Available from https://www.gatsbyjs.org/packages/gatsby-plugin-offline/ [cited May 8, 2020)

1 config.yml

```
version: 2.1
orbs:
  jq: circleci/jq@2.0.1
parameters:
  # This parameter is used to trigger the main workflow
  trigger:
    type: boolean
    default: true
  cms-develop:
    type: boolean
    default: false
  cms-master:
    type: boolean
    default: false

executors:
  node-executor:
    docker:
      - image: circleci/node:lts-browsers
    working_directory: ~/repo

jobs:
  checking-performance:
    executor: node-executor
    steps:
      - checkout
      - jq/install
      - attach_workspace:
          at: ~/repo
      - run:
          name: Install Moreutils
          command: sudo apt-get install moreutils
      - run:
          name: Detecting all pages
          command: ./tasks/lighthouse/detect-pages.sh
      - run:
          name: Lighthouse Auditing
          command: yarn lhci:run
      - run:
          name: Lighthouse Report To Github
          command: yarn lhci:upload

  install-dependencies-test-lint:
    executor: node-executor
    steps:
      - checkout
      - attach_workspace:
          at: ~/repo
      - restore_cache:
          keys:
            - yarn-cache-11-{{ checksum "yarn.lock" }}
      - run:
          name: Install Dependencies
          command: yarn install && npm rebuild
      - run:
          name: Check lint error
          command: yarn lint
      - run:
          name: Testing
          command: yarn ci-test
      - save_cache:
          key: yarn-cache-11-{{ checksum "yarn.lock" }}
```

```
          paths:
            - ./node_modules
      - persist_to_workspace:
          root: .
          paths:
            - ./node_modules
  gatsby-build-preview-deploy:
    executor: node-executor
    steps:
      - checkout
      - jq/install
      - attach_workspace:
          at: ~/repo
      - run:
          name: Create GitHub Gatsby Deployment
          command: ./tasks/deployment/gatsby-deploy-start.sh >
gatsby_deployment
      - restore_cache:
          keys:
            - gatsby-public-cache-{{ .Branch }}
      - run:
          name: Gatsby Build
          command: GATSBY_ACTIVE_ENV=staging GATSBY_CPU_COUNT=2 yarn build
      - save_cache:
          key: gatsby-public-cache-{{ .Branch }}
          paths:
            - ./public
      - run:
          name: Add GitHub Gatsby Deployment success status
          command: ./tasks/deployment/gatsby-deploy-end.sh success
          when: on_success
      - run:
          name: Add GitHub Gatsby Deployment error status
          command: ./tasks/deployment/gatsby-deploy-end.sh error
          when: on_fail
      - persist_to_workspace:
          root: .
          paths:
            - ./public
  storybook-build-preview-deploy:
    executor: node-executor
    steps:
      - checkout
      - jq/install
      - attach_workspace:
          at: ~/repo
      - run:
          name: Create GitHub Storybook Deployment
          command: ./tasks/deployment/storybook-deploy-start.sh >
storybook_deployment
      - restore_cache:
          keys:
            - v9-storybook-public-cache-{{ .Branch }}
      - run:
          name: Storybook Build
          command: yarn build-storybook
      - store_artifacts:
          path: build-storybook
      - save_cache:
          key: v9-storybook-public-cache-{{ .Branch }}
          paths:
            - ./build-storybook
      - run:
          name: Add GitHub Storybook Deployment success status
          command: ./tasks/deployment/storybook-deploy-end.sh success
          when: on_success
      - run:
```

```
          name: Add GitHub Storybook Deployment error status
          command: ./tasks/deployment/storybook-deploy-end.sh error
          when: on_fail
  cms-gatsby-build-staging-deploy:
    executor: node-executor
    steps:
      - checkout
      - jq/install
      - restore_cache:
          keys:
            - yarn-cache-{{ checksum "yarn.lock" }}
      - run:
          name: Install Dependencies
          command: yarn install && npm rebuild
      - run:
          name: Check lint error
          command: yarn lint
      - run:
          name: Testing
          command: yarn ci-test
      - save_cache:
          key: yarn-cache-{{ checksum "yarn.lock" }}
          paths:
            - ./node_modules
      - run:
          name: Create GitHub Gatsby Deployment
          command: ./tasks/deployment/gatsby-deploy-start.sh >
gatsby_deployment
      - restore_cache:
          keys:
            - gatsby-public-cache-{{ .Branch }}
      - run:
          name: Gatsby Build
          command: GATSBY_ACTIVE_ENV=staging GATSBY_CPU_COUNT=2 yarn build
      - run:
          name: Add GitHub Gatsby Deployment success status
          command: ./tasks/deployment/gatsby-deploy-end.sh success
          when: on_success
      - run:
          name: Add GitHub Gatsby Deployment error status
          command: ./tasks/deployment/gatsby-deploy-end.sh error
          when: on_fail
      - save_cache:
          key: gatsby-public-cache-{{ .Branch }}
          paths:
            - ./public
      - run:
          name: Netlify Deploy Gatsby
          command: ./node_modules/.bin/netlify deploy --site
$NETLIFY_SITE_ID_STAGING --auth $NETLIFY_ACCESS_TOKEN --prod --dir=public
  gatsby-storybook-build-staging-deploy:
    executor: node-executor
    steps:
      - checkout
      - jq/install
      - attach_workspace:
          at: ~/repo
      - run:
          name: Create GitHub Gatsby Deployment
          command: ./tasks/deployment/gatsby-deploy-start.sh >
gatsby_deployment
      - restore_cache:
          keys:
            - gatsby-public-cache-{{ .Branch }}
      - run:
          name: Gatsby Build
          command: GATSBY_ACTIVE_ENV=staging GATSBY_CPU_COUNT=2 yarn build
```

```
      - run:
          name: Add GitHub Gatsby Deployment success status
          command: ./tasks/deployment/gatsby-deploy-end.sh success
          when: on_success
      - run:
          name: Add GitHub Gatsby Deployment error status
          command: ./tasks/deployment/gatsby-deploy-end.sh error
          when: on_fail
      - save_cache:
          key: gatsby-public-cache-{{ .Branch }}
          paths:
            - ./public
      - run:
          name: Netlify Deploy Gatsby
          command: ./node_modules/.bin/netlify deploy --site
$NETLIFY_SITE_ID_STAGING --auth $NETLIFY_ACCESS_TOKEN --prod --dir=public
      - restore_cache:
          keys:
            - v9-storybook-public-cache-{{ .Branch }}
      - run:
          name: Storybook Build
          command: yarn build-storybook
      - save_cache:
          key: v9-storybook-public-cache-{{ .Branch }}
          paths:
            - ./build-storybook
      - run:
          name: Netlify Deploy Storybook
          command: ./node_modules/.bin/netlify deploy --site
$NETLIFY_SITE_ID_STORYBOOK --auth $NETLIFY_ACCESS_TOKEN --prod --dir=build-
storybook

  cms-gatsby-build-release-deploy:
    executor: node-executor
    steps:
      - checkout
      - jq/install
      - restore_cache:
          keys:
            - yarn-cache-{{ checksum "yarn.lock" }}
      - run:
          name: Install Dependencies
          command: yarn install && npm rebuild
      - run:
          name: Check lint error
          command: yarn lint
      - run:
          name: Testing
          command: yarn ci-test
      - save_cache:
          key: yarn-cache-{{ checksum "yarn.lock" }}
          paths:
            - ./node_modules
      - restore_cache:
          keys:
            - gatsby-public-cache-{{ .Branch }}
      - run:
          name: Gatsby Build
          command: GATSBY_ACTIVE_ENV=production GATSBY_CPU_COUNT=2 yarn build
      - save_cache:
          key: gatsby-public-cache-{{ .Branch }}
          paths:
            - ./public
      - run:
          name: Netlify Deploy Gatsby
```

```
          command: ./node_modules/.bin/netlify deploy --site
$NETLIFY_SITE_ID_RELEASE --auth $NETLIFY_CLIENT_ACCESS_TOKEN --prod --
dir=public

  gatsby-build-release-deploy:
    executor: node-executor
    steps:
      - checkout
      - jq/install
      - attach_workspace:
          at: ~/repo
      - restore_cache:
          keys:
            - gatsby-public-cache-{{ .Branch }}
      - run:
          name: Gatsby Build
          command: GATSBY_ACTIVE_ENV=production GATSBY_CPU_COUNT=2 yarn build
      - save_cache:
          key: gatsby-public-cache-{{ .Branch }}
          paths:
            - ./public
      - run:
          name: Netlify Deploy Gatsby
          command: ./node_modules/.bin/netlify deploy --site
$NETLIFY_SITE_ID_RELEASE --auth $NETLIFY_CLIENT_ACCESS_TOKEN --prod --
dir=public

workflows:
  version: 2
  build-deploy:
    when: << pipeline.parameters.trigger >>
    jobs:
      - install-dependencies-test-lint
      - gatsby-build-preview-deploy:
          requires:
            - install-dependencies-test-lint
          filters:
            branches:
              ignore:
                - develop
                - master
      - storybook-build-preview-deploy:
          requires:
            - install-dependencies-test-lint
          filters:
            branches:
              ignore:
                - develop
                - master
      - gatsby-storybook-build-staging-deploy:
          requires:
            - install-dependencies-test-lint
          filters:
            branches:
              only:
                - develop
      - gatsby-build-release-deploy:
          requires:
            - install-dependencies-test-lint
          filters:
            branches:
              only:
                - master
      - checking-performance:
          requires:
            - gatsby-build-preview-deploy
          filters:
```

```
          branches:
            ignore:
              - develop
              - master

cms-webhook-develop:
  when: << pipeline.parameters.cms-develop >>
  jobs:
    - cms-gatsby-build-staging-deploy
cms-webhook-master:
  when: << pipeline.parameters.cms-master >>
  jobs:
    - cms-gatsby-build-release-deploy
```

2 gatsby-config.js

```
module.exports = {
  siteMetadata: {
    title: `jamstack-javascript-boilerplate`,
    description: `jamstack-javascript-boilerplate`,
    author: `@tripheo0412`,
    type: process.env.GATSBY_ACTIVE_ENV || 'staging',
    siteUrl: `https://www.your-app-domain.netlify.app/`,
    hostname: `your-app-domain.netlify.app`,
  },
  plugins: [
    `gatsby-plugin-react-helmet`,
    {
      resolve: `gatsby-source-contentful`,
      options: {
        spaceId: process.env.CONTENTFUL_SPACE_ID,
        accessToken: process.env.CONTENTFUL_ACCESS_TOKEN,
      },
    },
    {
      resolve: `gatsby-plugin-canonical-urls`,
      options: {
        siteUrl: `https://www.your-app-domain.netlify.app/`,
      },
    },
    {
      resolve: `gatsby-source-filesystem`,
      options: {
        name: `images`,
        path: `${__dirname}/src/images`,
      },
    },
    {
      resolve: 'gatsby-plugin-i18n',
      options: {
        langKeyDefault: 'en',
        useLangKeyLayout: false,
        prefixDefault: false,
      },
    },
    {
      resolve: `gatsby-plugin-manifest`,
      options: {
        name: `jamstack-javascript-boilerplate`,
        short_name: `starter`,
        start_url: `/`,
        background_color: `#663399`,
        theme_color: `#663399`,
        display: `minimal-ui`,
        icon: `src/images/gatsby-icon.png
      },
    },
    `gatsby-plugin-sitemap`,
    `gatsby-plugin-offline`,
    `gatsby-transformer-sharp`,
    `gatsby-plugin-sharp`,
  ],
}
```

3 lighthouserc.json

```
{
  "ci": {
    "collect": {
      "url": ["http://localhost/"],
      "numberOfRuns": 2,
      "staticDistDir": "./public"
    },
    "assert": {
      "preset": "lighthouse:recommended",
      "assertions": {
        "first-contentful-paint": [
          "warn",
          {
            "maxNumericValue": 2500,
            "aggregationMethod": "optimistic"
          }
        ],
        "interactive": [
          "warn",
          {
            "maxNumericValue": 5000,
            "aggregationMethod": "optimistic"
          }
        ],
        "uses-long-cache-ttl": "off",
        "uses-http2": "off",
        "canonical": "off",
        "is-crawlable": "off",
        "link-name": "off",
        "meta-description": "off",
        "dom-size": ["error", { "minScore": 0.98 }],
        "uses-rel-preconnect": "off",
        "unused-css-rules": "off",
        "offscreen-images": ["warn", { "minScore": 0 }]
      }
    },
    "upload": {
      "target": "temporary-public-storage"
    }
  }
}
```

4 gatsby-deploy-end.sh

```
#!/bin/sh

set -eu

token=${GITHUB_DEPLOYMENTS_TOKEN:?"Missing GITHUB_TOKEN environment variable"}

if ! gatsby_deployment_id=$(cat gatsby_deployment); then
  echo "Deployment ID was not found" 1>&2
  exit 3
fi

if [ "$1" = "error" ]; then
  curl -s \
    -X POST \
    -H "Authorization: bearer ${token}" \
    -d "{\"state\": \"error\", \"environment\": \"storybook\"" \
    -H "Content-Type: application/json" \

"https://api.github.com/repos/${CIRCLE_PROJECT_USERNAME}/${CIRCLE_PROJECT_REPO
NAME}/deployments/${gatsby_deployment_id}/statuses"
  exit 1
fi

if ! netlify_deployment_url=$(./node_modules/.bin/netlify deploy --json --site
$NETLIFY_SITE_ID_STAGING --auth $NETLIFY_ACCESS_TOKEN --dir=public | jq
'.deploy_url'); then
  echo "Netlify preview deployment failed"
  exit 1
fi

echo ${netlify_deployment_url}

if ! gatsby_deployment=$(curl -s \
                  -X POST \
                  -H "Authorization: bearer ${token}" \
                  -d "{\"state\": \"success\", \"description\": \"deployed on
Netlify\", \"environment\": \"gatsby\", \"environment_url\":
${netlify_deployment_url}, \"target_url\": ${netlify_deployment_url},
\"log_url\": ${netlify_deployment_url}}" \
                  -H "Content-Type: application/json" \

"https://api.github.com/repos/${CIRCLE_PROJECT_USERNAME}/${CIRCLE_PROJECT_REPO
NAME}/deployments/${gatsby_deployment_id}/statuses"); then
  echo "POSTing deployment status failed, exiting (not failing build)" 1>&2
  exit 1
fi
```

5 gatsby-deploy-start.sh

```
#!/bin/sh

set -eu

token=${GITHUB_DEPLOYMENTS_TOKEN:?"Missing GITHUB_TOKEN environment variable"}

if ! gatsby_deployment=$(curl -s \
                  -X POST \
                  -H "Authorization: bearer ${token}" \
                  -d "{ \"ref\": \"${CIRCLE_SHA1}\", \"environment\":
\"gatsby\", \"description\": \"Gatsby\", \"transient_environment\": true,
\"auto_merge\": false, \"required_contexts\": []}" \
                  -H "Content-Type: application/json" \

"https://api.github.com/repos/${CIRCLE_PROJECT_USERNAME}/${CIRCLE_PROJECT_REPO
NAME}/deployments"); then
  echo "POSTing deployment status failed, exiting (not failing build)" 1>&2
  exit 1
fi

if ! gatsby_deployment_id=$(echo "${gatsby_deployment}" | python -c 'import
sys, json; print json.load(sys.stdin)["id"]'); then
  echo "Could not extract deployment ID from API response" 1>&2
  exit 3
fi

echo ${gatsby_deployment_id} > gatsby_deployment
```

6 storybook-deploy-start.sh

```
#!/bin/sh

set -eu

token=${GITHUB_DEPLOYMENTS_TOKEN:?"Missing GITHUB_TOKEN environment variable"}

if ! storybook_deployment=$(curl -s \
                  -X POST \
                  -H "Authorization: bearer ${token}" \
                  -d "{ \"ref\": \"${CIRCLE_SHA1}\", \"environment\":
\"storybook\", \"description\": \"Storybook\", \"transient_environment\":
true, \"auto_merge\": false, \"required_contexts\": []}" \
                  -H "Content-Type: application/json" \

"https://api.github.com/repos/${CIRCLE_PROJECT_USERNAME}/${CIRCLE_PROJECT_REPO
NAME}/deployments"); then
  echo "POSTing deployment status failed, exiting (not failing build)" 1>&2
  exit 1
fi

if ! storybook_deployment_id=$(echo "${storybook_deployment}" | python -c
'import sys, json; print json.load(sys.stdin)["id"]'); then
  echo "Could not extract deployment ID from API response" 1>&2
  exit 3
fi

echo ${storybook_deployment_id} > storybook_deployment
```

7 storybook-deploy-end.sh

```
#!/bin/sh

set -eu

token=${GITHUB_DEPLOYMENTS_TOKEN:?"Missing GITHUB_TOKEN environment variable"}

if ! storybook_deployment_id=$(cat storybook_deployment); then
  echo "Deployment ID was not found" 1>&2
  exit 3
fi


if [ "$1" = "error" ]; then
  curl -s \
    -X POST \
    -H "Authorization: bearer ${token}" \
    -d "{\"state\": \"error\", \"environment\": \"storybook\"" \
    -H "Content-Type: application/json" \

"https://api.github.com/repos/${CIRCLE_PROJECT_USERNAME}/${CIRCLE_PROJECT_REPO
NAME}/deployments/${storybook_deployment_id}/statuses"
  exit 1
fi

if ! repository=$(curl -s \
                  -X GET \
                  -H "Authorization: bearer ${token}" \
                  -d "{}" \
                  -H "Content-Type: application/json" \

"https://api.github.com/repos/${CIRCLE_PROJECT_USERNAME}/${CIRCLE_PROJECT_REPO
NAME}"); then
  echo "Could not fetch repository data" 1>&2
  exit 1
fi

if ! repository_id=$(echo "${repository}" | python -c 'import sys, json; print
json.load(sys.stdin)["id"]'); then
  echo "Could not extract repository ID from API response" 1>&2
  exit 3
fi

path_to_repo=$(echo "$CIRCLE_WORKING_DIRECTORY" | sed -e "s:~:$HOME:g")
url="https://${CIRCLE_BUILD_NUM}-${repository_id}-gh.circle-
artifacts.com/0/build-storybook/index.html"

if ! storybook_deployment=$(curl -s \
                  -X POST \
                  -H "Authorization: bearer ${token}" \
                  -d "{\"state\": \"success\",\"description\": \"deployed on
CircleCI\",  \"environment\": \"storybook\", \"environment_url\": \"${url}\",
\"target_url\": \"${url}\", \"log_url\": \"${url}\"}" \
                  -H "Content-Type: application/json" \

"https://api.github.com/repos/${CIRCLE_PROJECT_USERNAME}/${CIRCLE_PROJECT_REPO
NAME}/deployments/${storybook_deployment_id}/statuses"); then
  echo "POSTing deployment status failed, exiting (not failing build)" 1>&2
  exit 1
fi
```