Digitalization & Sustainability Review

Vol 1, No 1 (2021)

Getting Started Modern Web Development with Next.js: An Indispensable React Framework

Venkata Koteswara Rao Ballamudi, *Sr. Software Engineer, HTC Global Services, USA*
Karu Lal, *Integration Engineer, Ohio National Financial Services, USA*
Harshith Desamsetti, *Software Engineer, MedAllies, Inc., Fishkill, NY 12524, USA*
Sreekanth Dekkati, *Assistant Vice President (System Administrator), MUFG Bank, Arizona, USA*

Abstract
Developers spend much time and effort mixing many technologies to produce an entire web application. Frameworks like Next.js help. Next.js neatly organizes packages and configuration files. Its full-stack web application framework lets developers create front-end and back-end code in one place, making it unique. It simplifies the developer's life and speeds up product shipping. However, full-stack frameworks like next.js must compile the entire code base for every production build because we write it all in one location. There was room for improvement. In this article, we will explain how we may increase the efficiency of a production build next.js app utilizing strategies and coding patterns we learned when constructing a badminton data analytics-based web app.

Keywords: Next.js, Client-Side Rendering, Server-Side Rendering, Page Pre-Rendering, Lazy Loading, React Framework, JavaScript

INTRODUCTION

A popular React framework for server-side rendering web apps is NextJS. NextJS has three main advantages: An improved user experience. Extraordinary performance. React developers use NextJS to simplify their work. It also adds several critical features to React projects we must build ourselves. Many examples exist, including routing (Thaduri et al., 2016; Reddy et al., 2020; Desamsetti, 2016a; Lal et al., 2018; Chen et al., 2019). NextJS enhances our React app. React.js, a responsive system that "changes the principles of the game" in web development, and divides our application into small squares that can contain the rationale, construction, and styles for that part only. This heavy element made larger application bases more solid and had fewer "complex components" that broke. Web application development has changed drastically since Next.js' rise (Conolly, 2018). This technology lets programmers construct reliable JavaScript web apps without worrying about the back end. It uses the most prevalent crossover application method with client-side and server-side pages. Although the technique is simple, engineers need help to speed up their apps. Building sports-based web apps requires processing and fetching many game videos and creating data while loading the page, making performance and improvement difficult. An application's speed depends on how

long it takes to provide code, styles, and data to the client in the first entire loop. Application execution corrupts when the server sends extra resources (like photos) during the whole circle. Fortunately, developers can follow methods to speed up Next.js apps.

STATEMENT OF THE PROBLEM

getStaticProps was introduced in Next.js 9, allowing engineers to provide information during construction. In contrast to getInitialProps, this capability is almost always executed on the server side. Since the code inside this capability will not be remembered for the JavaScript pack transferred off the client, it is safe to write server-side code directly in getStaticProps. This information is readily available when we browse the page, so we do not have to rely on other API endpoints to bring the info anticipated to deliver the models on the /store page.

- **Format of the image:** The component supports the web image format (Alam, 2019), considered the next generation's picture format. Images saved in this format are typically 25–35 percent larger than their corresponding JPEG counterparts, depending on the quality index. When we compare the sizes of the photographs that were retrieved, we can see that there is a significant difference: the image of the red vehicle model used to be 1.3 megabytes when we were using the older approach, but we were able to reduce the size by -98.75 percent so that it is now only 16.6 kilobytes by utilizing the component. Image optimization is the process of reducing the overall size of a photo file, and the word "image optimization" refers to this process. Since images are one of the significant factors contributing to our app's overall performance issues, reducing the size of image files can be of assistance. The process consists of two stages: 1) Scale the photo to a more manageable size, and 2) save it in the appropriate format (a jpeg file is best for photos, while a PNG file is best for graphics).

- **Lazy Loading:** The previous implementation requested all model photographs. The component gets the image when the viewport intersects the picture's bounding box. We reduced image loading time from 3000ms to 270ms without changing the photos.

- **Dynamic Routes:** Customers can tap everything in the store to see the model more often. Previous executions used question boundaries and getInitialProps to supply the page and model information (Krill, 2017). The client can see the model after the API demand begins in getInitialProps settles, as on the/store page. Next, document framework-based unique courses were in js 9. With getStaticPaths and getStaticProps, this element lets us pre-render model pages based on their ID. We can statically construct pages for each model using a way boundary instead of one model page using getInitialProps and inquiry borders to choose which information to get and which model to deliver (Dekkati et al., 2019). Enveloping each model card by the network in part, lets us prefetch each/model/[id] page as it appears in the viewport, allowing a moment route when a client taps on it.

- **File-based routing:** React routing could be better. It has many moving parts and takes time, code, and effort to do correctly. Next.js' file-based routing technique determines our app's

routes. The required file structure organizes the software and lets developers see its navigation structure (Dekkati & Thaduri, 2017). The code-free routing method is intuitive and a good idea. Each Next.js project has a page folder with all page-level graphics.

- **Dynamic Imports:** 4 Next.js 9 supports TypeScript out of the box, unlike prior versions, which required extensive configuration! Adding a tsconfig.json file at the root of existing projects or using 'px create-next-app —ts' on freshly produced projects lets us use TypeScript instead of configuring it ourselves.

- **Client-side rendering:** Client-side delivery (CSR) uses JavaScript to provide pages directly in the program. Instead of the server, the client handles reasoning, information delivery, templating, and directing. Easy clientside delivery is hard to find and save for portability. It can show pure server-delivering by doing little work, preserving a tight JavaScript budget, and imparting value in as few RTTs as possible. HTTP/2 Server speeds up primary content and information delivery, enabling the parser. PRPL designs should be evaluated to ensure memorable introductions and pathways (Krill, 2018). Client-side rendering has a fundamental disadvantage in that the quantity of required JavaScript will, in most cases, increase as an application expands and develops. This is particularly problematic with the growth of new JavaScript libraries, polyfills, and outsider code, all of which compete for handling power and frequently must be dealt with before a page's content can be provided. Encounters working with CSR and dependent on massive JavaScript packs should consider using forced code-parting and apathetically burden JavaScript - "serve just what we want, when we want it." Server providing may provide a more adaptive solution to these problems when dealing with interactions with almost little intelligence (Desamsetti, 2016b). A new compiler built on the Rust programming language that uses native compiling and is built on SWC may be found in Next.js 12, which has been updated. We cut our build time from 90 seconds to 30 seconds by changing the version of Next.js we were using.

- **Quickly Respond Revisit**: A Next.js feature called Quick Refresh can be enabled in all Next.js applications running version 9.4 or a more recent version. It provides instant feedback on the changes we make to our React components in milliseconds without causing the state of the part to be lost. The incorporation of SWC into Next.js 12 significantly contributed to improving the refresh rate, resulting in 3 times faster refreshes than older versions. Vercel recommends that we get rid of conditions one at a time and then restart our program after each removal to ensure that the dependency was unnecessary and that we did not disrupt our application in the process of getting rid of it.

A FOCUS ON SERVER COMPONENTS

NextJS is placing a significant amount of emphasis on server components. If we have been keeping up with React version 18, we might already be familiar that React components can now render themselves as client- and server-side components. This capability was introduced in this version of React.

This idea is expanded upon in NextJS 13, which also includes the introduction of Server Components as a default feature in the new app directory. We may already be familiar with the pages folder, where we define all our routes. NextJS employs file-based routing rather than component-based routing to generate URLs. However, in the not-too-distant future, the pages folder will become obsolete. Instead, the app directory will be used automatically (we will learn more about this in the following section).

The file-based routes in this new app directory will automatically be converted into React Server Components. However, first and foremost, we need to understand the distinction between these two: Client-Side components and Server Components.

However, if we wanted to serve the React App as the Server side, we could spin it up against an express-like server and use it in that capacity. Historically, React has been an opinion-free view library on the application's client side. After the introduction of server-side rendering techniques by NextJS, such as getStaticPath and getServer-SideProps, SSR-based pages perform significantly better and are helpful in the long term (Krill, 2016). It eliminates all of those network latency concerns, in which end users are generally required to wait for an API to retrieve data, and during this time frame, they are forced to deal with unpleasant loaders and spinners.

As a result, the NextJS team has deployed React Server Components by default, drawing from a wide variety of sources of inspiration and drawing in general on the experience of developers. When those sites are rendered, the server will only provide content prefetched in HTML, and only a minimal amount of Javascript will be used. This ensures that caching is improved as well as overall performance. Eliminating all of the data fetching React hooks and NextJS methods like getServerSideProps(), getStaticProps(), and getInitialPath(), amongst others, is another benefit of using React Server Components. In the following paragraph, we will acquire further knowledge on this subject.

SIMPLIFYING THE PROCESS OF OBTAINING DATA

The method to retrieve data in our React applications was fundamentally altered by NextJS version 13. Suppose we have been working with Next for a considerable amount of time. In that case, we may be already familiar with the various methods for retrieving data, such as getServerSideProps(), getStaticProps(), getInitialPath(), and so on, because we have already discussed As of NextJS version 13, these methods are no longer recommended because they are considered anti-patterns. Instead, NextJS 13 emphasizes just requesting data on the server side, and it forbids the usage of any hooks, such as useEffect(), to fetch data on the client side under any circumstances.

The retrieval of data from the server has many advantages, including but not limited to the following: Our sensitive data, such as API keys, hash functions, staging/development URLs, and so on, can be stored on the server, where it will remain safe and will not be exposed to the end user in any way, even if they gain access to the client side. As a result of the server sending pre-rendered material to the client, we are reducing the number of communication protocols between the client and the server. This results in a performance improvement. As a result of the pages being

generated in advance, there are now fewer loaders and spinners, which is a highly apparent shift. The server can retrieve data from various APIs, and then just that data is delivered to the client, which reduces the number of network calls made by the client (Gutlapalli, 2017c).

By utilizing the simple old retrieve() Web API and passing it along as a prop to wherever it is required, we can save significant time and effort by utilizing React Server Components. Because everything would be generated on the server, the page to which we supplied our props would only be rendered once the API response has been registered. This is because everything would be generated on the server.

The supplied feature would function faultlessly because the given component would be generated on the server and transmitted to the client. On the client side, it is optional for us to fetch any API. On the other hand, there may be some circumstances in which a component on the client side is necessary for us (Carlos, 2018). Cases in which we are employing React hooks, altering state values directly on the user interface (UI), providing feedback to the end users, and so on. In each one of these scenarios, the use-client notation can be written directly on top of a React Component. It would communicate to NextJS that we only want this to be rendered locally on the client's machine.

MAKING APPLICATIONS THE NEW PAGES DIRECTORY

The NextJS ecosystem's new app directory is a hot topic. We can still construct an excellent old pages directory and put all our pages there, but we will miss out on NextJS version 13's latest and most significant capabilities. NextJS version 13 adds an app directory for page and route setup. As noted before, all app directory pages are SSR only and require use-client on the client side. App directory has comparable properties to pages folder, such as,

The above folder structure inside the new app directory maps to the same URL as the previous pages directory:

Renders Content. JSX at
localhost:3000/dashboard/content
localhost:3000/dashboard/analytics (Renders Analytics. js)

However, the app directory has introduced specific files suitable for placement in the primary folder (in this case, outside the dashboard folder) or route-wise/folder-wise.

The particular files are: Each route needs a separate component file, page. TSX, to be publicly available. Layout.tsx—This component file could be a generic Layout file like the base one with Header and Footer components. Layout file will not re-render UI while children state can. Loading.tsx—This optional file can conditionally render a page part as loading. Like Loading.tsx, error.tsx can generate a page section to conditionally render our error status and resolve it if a condition is fulfilled. template.tsx is identical to the layout.tsx but not optimized for performance; therefore, it does not store and cache state or re-render every time. In head.tsx, we can keep meta tags previously stored in the React component under the <head></head> tags.

ONLINE STREAMING

The capabilities of NextJS version 13 are comparable to those of React version 18. Streaming, widely considered one of the most valuable additions to React18, is incorporated into the process. On standard SSR sites, we need to wait for the entirety of the page to be created on the server before we can see it being rendered on the screen (Deming et al., 2018). This is the case in almost all cases. Naturally, this can be rectified by introducing client-side rendering and constructing a page section by section as the data is collected from the server (Gutlapalli et al., 2019). This is a straightforward solution to the problem (Gutlapalli, 2016a). However, rendering on the client side presents new challenges, including increased loaders and spinners, an inferior experience for the end user.

Only by employing Streaming in SSR pages can we resolve this issue. With this method, our end users would not have to wait for a longer length of time, but they would start seeing some skeleton or basic UI until the complete page loads (Mandapuram et al., 2020). React server components would begin "streaming" and transmitting data as soon as something is generated on the server. This behavior can be thought of as being comparable to how client-side rendering behaves; however, in this instance, it is SSR sites that are delivering data to the client through the use of Streaming.

This feature is incorporated into NextJS's new app directory beginning with version 13 of the framework. By default, the new app directory would generate SSR pages, as was initially mentioned in the points addressed above, which we already know (Lal & Ballamudi, 2017). Our page can be considered a "stream" if we use the Suspense APIs in the following ways:

Although the Dashboard component will be built in the code shown above once it has data from the server, until that time, React will suspend the component's rendering and will instead display the Loading User Dashboard. We can substitute this straightforward text with a more intricate user interface (UI) or a screen skeleton (Gutlapalli, 2016b). The Suspense API will automatically supply the UI when we receive the server data. Naturally, this style is superior to server-side rendering; nevertheless, it also makes significant advancements to the standard NextJS SSR pages.

API CHANGE

NextJS 13 improves several surface APIs:

- If familiar with NextJS, the Image component is the preferred API for rendering images, replacing the native img tag. NextJS 13's Image component ships less javascript to the client and improves performance.
- NextJS introduces its typeface system in version 13. The font display and other parameters and strategies would automatically optimize our font for better rendering performance and First Contentful Paint (FCP) score. We can utilize it as Google Fonts is where @next/font gets its fonts, so make sure they are public.
- Next/link: Minor changes to the <Link/> API are planned. NextJS 13 simplifies the Link tag, eliminating the need for the <a /> tag. Instead, NextJS automatically adjusts.

NEXT JS API ROUTES

The JavaScript package React is the foundation for the fantastic production-ready framework called Next.js. It enables us to construct basic static websites, server-side rendered applications, or a combination of the two because of its high adaptability (Lal, 2016). How Next.js manages its routing system is one of its many impressive features.

When we bootstrap an app with create-next-app, a folder named "pages" is automatically created for us. All that needs to be done to construct routes is to add a file to this folder. Another folder with the name API can be found inside of this one. A file created within the API will be viewable via the path /api/... However, a page will not be delivered to us at this time.

This particular API endpoint is located on the server side of our application, and its primary purpose is to return data rather than web pages. If we so choose, we can write the back-end code for this application using Node.js. If we are coming from the perspective of a front-end developer, then we could have some reservations about this (Lal, 2015). One of the features of Next.js is the ability to seamlessly move from writing code in Node.js on the server to writing code in React on the client in a single minute (Thaduri & Lal, 2020).

Uses of /api routes

One of the more helpful metaphors for a Next.js API route is that of an intermediary for our application. Now for a small quantity of metaphor! Take, for example, a person named John who is interested in writing a letter to his close buddy Jane (Vega, 2017). Once Jane has the letter in her possession, she will promptly send a response to John. Nevertheless, how exactly are the letters transported from one location to another? Of course, it is the mailman or mailwoman! The Postman or Postwoman is the one who plays the role of the Middleman. This individual is in charge of the logic involved in getting the letters to where they are supposed to go, which is analogous to what we do with the Next.js API routes (Gutlapalli, 2017a). We frequently send a request and some data to a particular API route. This is the equivalent of dropping our letters into the mailbox.

After that, behind the scenes, the post will be subjected to operations such as sorting and processing before ultimately being transmitted to the other end (for example, an external API). When the individual receives the letter and responds to it, the original sender will finally open the response, also known as the reply. Therefore, if some of our API routes also send requests, we might inquire about something along these lines. There are a few different justifications that could support our decision to go ahead with this.

- We can access our environment variables safely with process.env on the server side.
- We can hide our interaction with some external providers by keeping it out of the client.
- Maybe we have cookies we want to handle on the server

We redirect client requests to our API route api/... with a body of data if needed. Using these routes, our app may safely communicate with our client and serverless requirements. Next.js API routes are only available from the exact origin unless enabled with middleware like CORS. Set up our project directory next-API-routes with Next.js using the following command (Kirill, 2018).

npx create-next-app next-api-routes

In their most basic form, the routes of our API are merely functions. Each API route has a single default exported function associated with it. In this section, we are also importing our posts that have been hardcoded (Michele, 2017). The process is given two parameters to work with, denoted by the names req and res, and defined as follows.req - request object based on the incoming HTTP message

res - The response object based on the HTTP response of the server

The following paragraph will further delve into the req and res objects. We will be comfortable with these if we have previous experience working with Node.js and Express. At this point, we indicate that the operation was successful by sending back a response code of 200, and we can chain this status with the () response function to transmit a JSON representation of our posts (Mandapuram et al., 2018). Given that our list is hardcoded and imported, we have every reason to believe it would be successful; for this reason, the status code is 200.

Next.js provides middleware built into API routes to parse the request.

- req.body - request body where we send some data to our route
- req.cookies - the Cookie object sent with the request
- req.query - URL query string object of the request

We will be able to see examples of some of the items we can access on the request, along with the results of those examples. If we log the request object there, we will notice that a significant quantity of data has been returned to the console. Remember that we cannot see this logged in the browser because it is now executing on the server. After all, the browser cannot communicate with the server. Instead, it will be in the terminal that we are using (Desamsetti & Mandapuram, 2017).

Let us have a look at the response that was given now. As we have discussed, the API routes we use for our app frequently serve as middleware between our client and any external services. The flow of things generally works like this.

- Send a request to the API route
- Do something with the request
- Finish by returning a response

The following will be expected if we are familiar with Node.js and Express. Next.js includes several methods on the res object that allow us to modify how a response is returned (Gutlapalli, 2017b). Let us go ahead and investigate them. to control how we return a response. As we have seen in the examples before this one, we can string them together to return a status code and some data. res.status(500).json({ message: "Bad request" }), allowing us to control our responses better.

INTERNAL TOOLING

Webpack is an asset bundler that bundles HTML, JavaScript, picture, and video files into a JavaScript bundle. Longtime React ecosystem leader Webpack (Bodepudi et al., 2019). Create-react-app, a popular React template, leverages Webpack for JavaScript packaging. Webpack is excellent, but the JavaScript ecosystem has built more minor, more efficient bundlers. Such bundlers include Turbopack (Thodupunori & Gutlapalli, 2018). NextJS 13 will use Turbopack instead of Webpack (Elyse, 2018). Turbopack's core is developed in Rust to support its Rust-based future. Turborepo bundles are better than Webpack in many ways. We do not have to worry about how NextJS bundles our files internally, and NextJS developers would see a slight speedup in compilation.

CONCLUSION

A framework such as next.js comes with many helpful features and optimizations built right in, and optimization is not a one-time effort but rather an ongoing process constantly being improved upon. The Vercel team developed Next.js, as several developers have stated, to simplify the process of developing web applications and assist in deploying the product more quickly. However, in this paper, we attempted to convey our experience with next.js while developing online apps, specifically when developing a badminton-based web app, and the many pieces of knowledge gained from that journey. We hope that reading this paper has enlightened us in some way. The only industry in which technology is used significantly by professionals, such as players, coaches, and teams, is sports. The technology needed for everyday sports aspirants is a long way off. It is not because technology is prohibitively expensive; instead, the problem is that only so many technically savvy people are involved in it just now. In terms of scale, there is a significant market opening for those individuals who are capable of developing software for an

REFERENCES

Alam, I. (2019). Best practices to increase the speed of next.js apps. *Stack Overflow discussions blog post.*

Bodepudi, A., Reddy, M., Gutlapalli, S. S., & Mandapuram, M. (2019). Voice Recognition Systems in the Cloud Networks: Has It Reached Its Full Potential?. *Asian Journal of Applied Science and Engineering*, *8*(1), 51–60. https://doi.org/10.18034/ajase.v8i1.12

Carlos**,** S. R. (2018). *React Cookbook: Create Dynamic Web Apps with React Using Redux, Webpack, Node. js, and GraphQL*. Birmingham, GB: Packt Publishing, Limited, https://www.proquest.com/docview/2136058999/6E1794AB166546D3PQ/4

Chen, S., Thaduri, U. R., & Ballamudi, V. K. R. (2019). Front-End Development in React: An Overview. *Engineering International*, *7*(2), 117–126. https://doi.org/10.18034/ei.v7i2.662

Conolly, S. (2018). Performance difference between next.js and react.js. *Log Rocket Series of articles.*

Dekkati, S., & Thaduri, U. R. (2017). Innovative Method for the Prediction of Software Defects Based on Class Imbalance Datasets. *Technology & Management Review*, *2*, 1–5. https://upright.pub/index.php/tmr/article/view/78

Dekkati, S., Lal, K., & Desamsetti, H. (2019). React Native for Android: Cross-Platform Mobile Application Development. *Global Disclosure of Economics and Business*, *8*(2), 153-164. https://doi.org/10.18034/gdeb.v8i2.696

Deming, C., Dekkati, S., & Desamsetti, H. (2018). Exploratory Data Analysis and Visualization for Business Analytics. *Asian Journal of Applied Science and Engineering*, *7*(1), 93–100. https://doi.org/10.18034/ajase.v7i1.53

Desamsetti, H. (2016a). A Fused Homomorphic Encryption Technique to Increase Secure Data Storage in Cloud Based Systems. *The International Journal of Science & Technoledge*, *4*(10), 151-155.

Desamsetti, H. (2016b). Issues with the Cloud Computing Technology. *International Research Journal of Engineering and Technology (IRJET)*, *3*(5), 321-323.

Desamsetti, H., & Mandapuram, M. (2017). A Review of Meta-Model Designed for the Model-Based Testing Technique. *Engineering International*, *5*(2), 107–110. https://doi.org/10.18034/ei.v5i2.661

Elyse, G. (2018). *Isomorphic Web Applications: Universal Development with React.* New York, NY: Manning Publications Co. LLC, https://www.proquest.com/docview/2547047705/5637E4705A9345E8PQ/3

Gutlapalli, S. S. (2016a). An Examination of Nanotechnology's Role as an Integral Part of Electronics. *ABC Research Alert*, *4*(3), 21–27. https://doi.org/10.18034/ra.v4i3.651

Gutlapalli, S. S. (2016b). Commercial Applications of Blockchain and Distributed Ledger Technology. *Engineering International*, *4*(2), 89–94. https://doi.org/10.18034/ei.v4i2.653

Gutlapalli, S. S. (2017a). Analysis of Multimodal Data Using Deep Learning and Machine Learning. *Asian Journal of Humanity, Art and Literature*, *4*(2), 171–176. https://doi.org/10.18034/ajhal.v4i2.658

Gutlapalli, S. S. (2017b). The Role of Deep Learning in the Fourth Industrial Revolution: A Digital Transformation Approach. *Asian Accounting and Auditing Advancement*, *8*(1), 52–56. Retrieved from https://4ajournal.com/article/view/77

Gutlapalli, S. S. (2017c). An Early Cautionary Scan of the Security Risks of the Internet of Things. *Asian Journal of Applied Science and Engineering*, *6*, 163–168. Retrieved from https://ajase.net/article/view/14

Gutlapalli, S. S., Mandapuram, M., Reddy, M., & Bodepudi, A. (2019). Evaluation of Hospital Information Systems (HIS) in terms of their Suitability for Tasks. *Malaysian Journal of Medical and Biological Research*, *6*(2), 143–150. https://doi.org/10.18034/mjmbr.v6i2.661

Kirill, K. (2018). *Next. js Quick Start Guide : Server-Side Rendering Done Right.* Packt Publishing, Limited. https://www.proquest.com/docview/2110389651/418F103A5C3845E7PQ/1

Krill, P. (2016). Next step after Node.js: Framework for 'universal' JavaScript apps. *JavaWorld* San Francisco. *https://www.proquest.com/docview/1835177901/6C5B7D762F1042A4PQ/37*

Krill, P. (2017). Next.js 2.0 plays better with React and JavaScript. *InfoWorld.com*, https://www.proquest.com/docview/1881728502/418F103A5C3845E7PQ/2

Krill, P. (2018). Next.js 7 framework compiles faster, supports WebAssembly. *InfoWorld.com*, https://www.proquest.com/docview/2110389651/35E649C9787B4630PQ/15

Lal, K. (2015). How Does Cloud Infrastructure Work?. *Asia Pacific Journal of Energy and Environment*, *2*(2), 61-64. https://doi.org/10.18034/apjee.v2i2.697

Lal, K. (2016). Impact of Multi-Cloud Infrastructure on Business Organizations to Use Cloud Platforms to Fulfill Their Cloud Needs. *American Journal of Trade and Policy*, *3*(3), 121–126. https://doi.org/10.18034/ajtp.v3i3.663

Lal, K., & Ballamudi, V. K. R. (2017). Unlock Data's Full Potential with Segment: A Cloud Data Integration Approach. *Technology &Amp; Management Review*, *2*, 6–12. https://upright.pub/index.php/tmr/article/view/80

Lal, K., Ballamudi, V. K. R., & Thaduri, U. R. (2018). Exploiting the Potential of Artificial Intelligence in Decision Support Systems. *ABC Journal of Advanced Research*, *7*(2), 131-138. https://doi.org/10.18034/abcjar.v7i2.695

Mandapuram, M., Gutlapalli, S. S., Bodepudi, A., & Reddy, M. (2018). Investigating the Prospects of Generative Artificial Intelligence. *Asian Journal of Humanity, Art and Literature*, *5*(2), 167–174. https://doi.org/10.18034/ajhal.v5i2.659

Mandapuram, M., Gutlapalli, S. S., Reddy, M., Bodepudi, A. (2020). Application of Artificial Intelligence (AI) Technologies to Accelerate Market Segmentation. *Global Disclosure of Economics and Business 9*(2), 141–150. https://doi.org/10.18034/gdeb.v9i2.662

Michele, B. (2017). *React Design Patterns and Best Practices: Build Modular Applications That Are Easy to Scale Using the Most Powerful Components and Design Patterns That React Can Offer You Right Now*. Packt Publishing, Limited. https://www.proquest.com/docview/2136029018/D8BE03DC87784BB7PQ/13

Reddy, M., Bodepudi, A., Mandapuram, M., & Gutlapalli, S. S. (2020). Face Detection and Recognition Techniques through the Cloud Network: An Exploratory Study. *ABC Journal of Advanced Research*, *9*(2), 103–114. https://doi.org/10.18034/abcjar.v9i2.660

Thaduri, U. R., & Lal, K. (2020). Making a Dynamic Website: A Simple JavaScript Guide. *Technology & Management Review*, *5*, 15–27. https://upright.pub/index.php/tmr/article/view/81

Thaduri, U. R., Ballamudi, V. K. R., Dekkati, S., & Mandapuram, M. (2016). Making the Cloud Adoption Decisions: Gaining Advantages from Taking an Integrated Approach. *International Journal of Reciprocal Symmetry and Theoretical Physics*, *3*, 11–16. https://upright.pub/index.php/ijrstp/article/view/77

Thodupunori, S. R., & Gutlapalli, S. S. (2018). Overview of LeOra Software: A Statistical Tool for Decision Makers. *Technology & Management Review*, *3*(1), 7–11.

Vega, C. (2017). Client-side vs. server-side rendering: Why it is not all black and white, *Free Code Camp.*