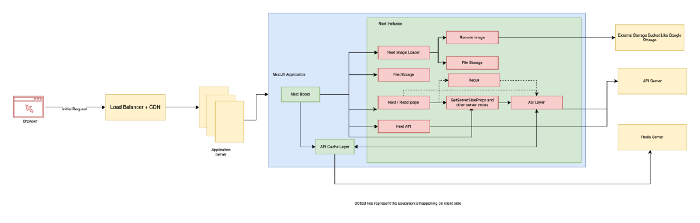
**System Design of Voyce.me**

This document outlines the application’s architecture design, which is organized into several sections as follows:

1. Initial Request
2. Next React Application — The API call
3. Authentication and use of security layer
4. Load time and SEO
5. Deployment Architecture

.



The layout is divided into three sections:

1. Network layer — How the request will react to the application
2. Next Application instance, the SSR
3. External Services

**1. Network layer**

When a user requests a website, this section discusses how the request will travel to the server and which protocol we will utilise.

When a user requests the application, the request will be sent to a load balancer; if the request is for loading website HTML content, the request will be sent to the application server; if the request is for loading static content such as images uploaded from the admin panel, build files, fonts, etc., the request will be sent to the CDN, and the content will be loaded from the CDN.

In this way, we will be able to deliver all static content from the edge location, reducing the static request load on the application server in this way.

In both cases, we have used HTTP/3.

**2. Next Application instance, the SSR**

This section describes the application code and how it will work; all requests will go through the next-boot cache custom server, which will cache any incoming request; the reason we’re using next-boost is that it’s a high-performance NextJs cache library that runs on a SQLite database; as a result, the throughput is very high, and it’s easy to configure this layer; the requests are mainly made up of five different types of requests.

1. The request for the page,  
   This is mostly the SSR page request, in which the next application loads all of the material and generates an HTML version of the page requested. If we need dynamic data to be loaded alongside the HTML, we must use NextJs implementations, namely getServerSideProps.
2. This request will be made when a user changes from one page to another, and if the second page includes a getServerSideProps, then this request will be made. Next-boost will not cache this request, thus we’ll need our own cache layer for these code, which may be a redis cache.
3. Next API requests are requests to our NextJs API definitions, and we usually don’t need to cache this layer unless there is a specific use case, because the platform will make the request most of the time from the browser level.
4. Next Image — For every picture loading on the website, we use NextImage. NextImage is a loader that will transcode the image into a certain size and format to minimise the transmission time over the internet. We could enable next-boost cache and specify most webp and avif format images here.
5. File Request, This route will not be required unless we need to serve dynamic static content; as previously stated, all static content will be moved to CDN, so this route will not be required; however, if we need to serve a sitemap that is very dynamic and automatically updates on a regular basis, we may need to store it alongside the website itself; in these cases, we can use this route.

**3. External service**

This will include, for example, all external services that we use.  
The database engine, redis, and other Google cloud services

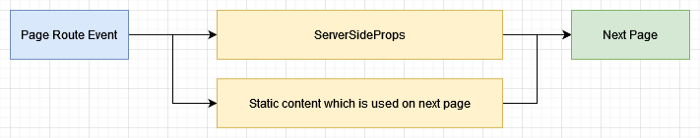
**Initial Request**

Steps in relation to the browser:

1. The HTML markup  
   1. Once the user enters the website url, the browser sends the first initial page request to the server.  
   2. The load balancer will receive the request.  
   3. The load balancer will route the request to the custom server NextApplication.  
   4. If the file has already been cached, the custom server checks for it and returns the cached data.  
   5. If not, load the next application and then run the Next Server executable like GetServerSideProps, then load the complete next application, and finally convert this page to HTML  
   6. Cache the HTML file that has been parsed  
   7. Return the data to the client
2. Subsequent calls  
   1. The HTML markup is downloaded and loaded into the browser.  
   2. After that, it sends a request to load all essential files, such as CSS, JS, Fonts, Images, and so on.  
   3. The load balancer will receive these requests.  
   4. The load balancer then sends this request to the CDN and to the storage bucket where the files are kept.  
   5. We can build a subdomain and connect the CDN to the Storage Bucket directly, bypassing the Application LoadBalancer.  
   6. The react side of the website will initialise on the client side once all of these have been loaded.  
   7. If any frontend API calls are needed for dynamic loading, react will make the call and save the data in redux.



1. Loading next page  
   1. When a user moves from page A to page B, the JavaScript required to initialise the second page will be pre-loaded, and an API request to get the getServerSideProps will be made.  
   2. If the API is successful, the second page will be displayed, along with all other dynamic JavaScript content.  
   3. In addition to JavaScript, the system will make a request to load the images and other static content needed for page B.

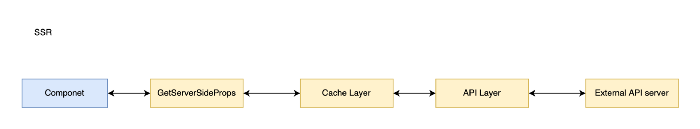


**Next React Application — The API call**

There are mainly three types of API calls,

**SSR API calls**

The reason we should optimise SSR API call (GetServerSideProps) is that if you are careless, GetServerSideProps can be a single point of failure for your application. That is, for every page visit that has a direct link or frontend routing, the next js will automatically call an API to get the getServerSideProps, which means that if 1000 people visit the website, we are unnecessarily increasing the server load



All getServerSideProps requests will first go to a cache layer, which is based on redis cache, and check whether the API is already cached or not; if the API is cached, we can simply return the data; if the API is not cached, the request will travel to an API layer, which will call an external API and retrieve the data; finally, the request will travel backward and send to the frontend. By using this method, we only need to process one single request;

**Redux based API calls**

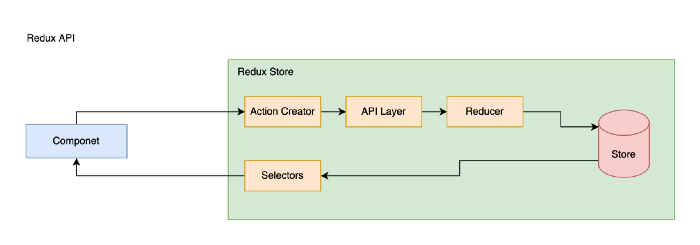
To comprehend this section, we must first pose two questions:  
\* Why is Redux required?  
\* Why can’t we make the API call using react query or another library?

The answer to this issue is that Redux is used for more than simply API calls; it’s also used to handle the full application state, as well as api calls that need to be cached and modified.

For example, in a page, we may need to load all of the content created by all users, and we may want to sort the items based on the user’s selection. In this case, we can load the complete data to the frontend and sort the data directly from the redux store before fetching to the components.

The following structure can be utilised with Redux.  
(The state is divided into three sections.)

1. Auth — To save all authentication information
2. Entities — To store all server representation data so that we can add and alter both the local state and the server at the same time.
3. UI— To keep track of all UI-related states.



To make an API call, the components must all go via their respective action creators, who will decide whether or not to make the API calls. If the data is available, we won’t have to make the API call again.

If the data isn’t available, the action creator will send an API event, which the API Layer will intercept and use to perform the API call and execute the appropriate reducers.

The reducer will work with the data in the store and write it to the output.

The component will use selectors to read data from redux, which are defined within the store; we can then change the data before returning to the component.

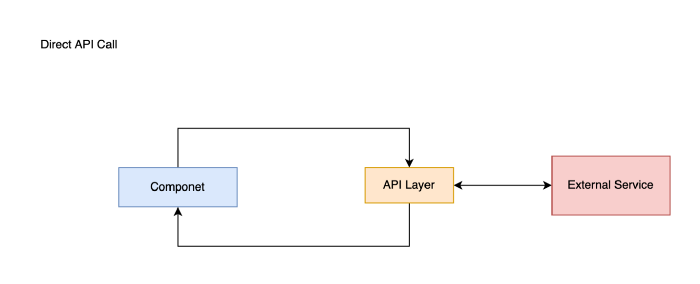
Another benefit of a redux-based approach is that we may preload the API before the component that will consume it is loaded. All we have to do is call the action creator.

* *Redux has the power of a centralised state, which means that any changes to the data will automatically update all listeners.*

**Direct / Cached API calls**

Why are direct API calls required?  
It is not suggested to make all API calls through Redux unless we need to store the data in Redux and want to reuse it. We don’t need to make the API calls through Redux unless we need to store the data in Redux and want to reuse it.

We’ll use straight API calls in these circumstances.



**Authentication and use of Security Layer**

Saving authentication information directly on cookie or localStorage is a typical mistake people do with single page applications, but the problem with this technique is that people will be able to teal this information very quickly.

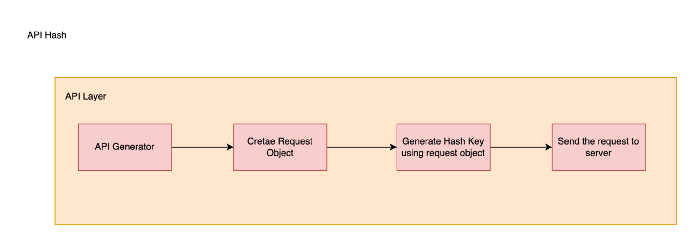
To fix this problem, the best solution is to save the data in encrypted localStorage. However, if we encrypt data with a save key for all devices, people will be able to copy local storage or cookie information from one browser to another, and the other browser running the same application will be able to decrypt the data.This is yet another issue to resolve.

To fix this difficulty, we’ll need to utilise a safe method of storing data that can only be viewed by the browser which it’s encrypted. To accomplish this, we must first generate a unique key for each browser and then use that key to decode. To accomplish this, we can utilise the fingerprint key from the browser as an encryption key and preserve the data.

**The Security / API Layer**

This is not a requirement for the frontend, but it is required to protect the entire API from being exploited. Normally, when a person receives an auth token, they will be allowed to submit requests to all of the routes that the token may access.

To address this problem, we can set up a secure environment in which the client and server share a hash key for each request.



The API Layer is made up of four steps, the first of which is the API Generator, which creates the request url and defines the request parameters.

The second step is to create a request object with all of these arguments; typically, the request object will include the fields listed below.

1. Url
2. Method
3. Body
4. Header
5. User-agent / browser fingerprint
6. Authorization token

In the next step, by using the request object, we will create a hash key using a secure key and add the field into the header of actual request,

The final step is to send the data to the server and use the same logic to generate a hash key there.

People won’t be able to utilise a token leak to attack the system since each request’s hash key will be unique for all browsers and users, and only the application will be able to generate this hash key using secure key, thus even if there is a token leak, people won’t be able to exploit the system.

**Load time and SEO**

The major reason for utilising NextJs is to have default support for SEO; in this case, we’re looking at ways to increase SEO and load time.

1. Third party libraries  
   To load any third-party scripts, use the NextJs script tag, and use a different strategy to load the required libraries before the content loads; all other libraries can be loaded only when needed or after the page has loaded.
2. Opt-out NON SSR pages and codes  
   This is critical because when we build an application, we load all of the components without considering whether we need this code to load on its own or not. Use dynamic load with SSR: false, if a page doesn’t need to be SSR, such as authorised pages
3. Eye on build size  
   Always keep an eye on the build size; if we don’t pay attention to it early on, it will be very difficult to do so later on when we have a lot of features, thus while designing a page, use bundle analyzer and next build to see how it affects the build size.  
   Use the bundle-wizard npx library to check the platform’s page-by-page load.  
   It’s also crucial to look at the size of CSS files.
4. Caching of the SSR page, static content, and getServerSideProps  
   Use next-boost to cache the full application markup and server side props, as we explained in detail in the initial request section.
5. Properly sized image and prefetch  
   If the application fails to load images, it’s critical to utilise NextImage or another loader that uses a lazy loading technique to load the images. This reduces the load greatly, but it’s not enough to speed up the system.  
   Preloading the images that are visible in the viewport (First Fold) is also vital; this will considerably reduce the LCP.  
   It’s also vital to load photos that are precisely required to load. For example, if we have a 200x200px image canvas, loading an image that is 1000x1000px wastes internet bandwidth; instead, we load images that are exactly required to load.
6. Use of schema engine  
   This is a crucial idea in SEO because Google uses Rich results to identify, categorise, and display websites to visitors. Use proper schema for all of the elements specified on the page, and consult schema.org for further information.
7. Use dynamic import with lazy import  
   For all conditional rendering items, use dynamic loading. This will prevent all unneeded codes from being packed together.  
   If you want a library to be dynamically loaded, you can use the lazy loading technique and a conditional import statement within the component.
8. Opt-out from Link / Router prefetching  
   By default, NextJs preloads all of the js for pages that have a link to them from the current page. This is done to speed up client-side routing.  
   This is a fantastic feature, but it may have a negative impact on page performance.  
   If you have a website with a number of links in the footer and none of them are on the initial page, we don’t need to preload these pages because the chances of people clicking these links are low.  
   As a result, on Link, mention these items as follows:  
   prefetch=false
9. To get the full assessment for the website, use the Chrome Lighthouse tab and make the suggested changes.
10. Use of AMP  
    NextJs has built-in support for AMP, which is a great method to increase page speed by converting high-value pages to AMP and making them load much faster.

**Deployment Architecture**

It is critical to cache the website, API, and images using next-boost, as we explained in the first part.

Next, with a solid cache and cache revalidation logic, run Boost on sqlite3.  
To improve platform performance in terms of FCP, use the CDN and next-boost cache layer instead of serving all content directly from the NextApplication.