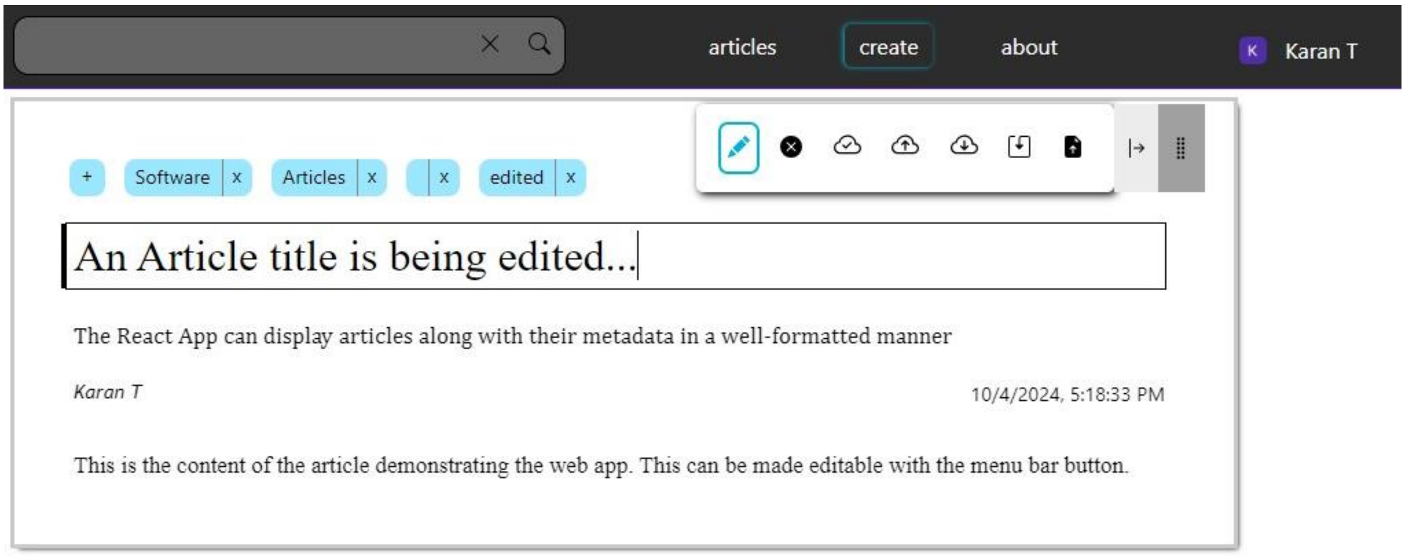
Articles-Showcase

A **MERN Stack** website for users to create, edit, and share articles. The built-in editor provides features to work and save your progress across sessions. The built-in search engine offers search functionalities across the entire collection of articles.



An Article title is being edited...

The React App can display articles along with their metadata in a well-formatted manner

Karan T

10/4/2024, 5:18:33 PM

This is the content of the article demonstrating the web app. This can be made editable with the menu bar button.

API Documentation:

Architecture

The app's architecture, includes **4 tiers - server, front-end, database, and usage analytics**. The components are built based on **REST and WS3** principles. The app's server is implemented using **NodeJS**, which facilitates building non-blocking asynchnorous, event-driven, single-threaded JS applications for high scalability, and is built on the V8 JS engine, supporting native C++ code integration. The server includes a **HTTPS/3** RESTful API. The front-end is built using **ReactJS**, which faciltates building stateful modular components. The database is implemented using **MongoDB Atlas**, a NoSQL DBMS. A lucene search engine is implemented using MongoDB Atals Search. The server is dockerized and deployed in a serverless **Azure Container App** instance. The server was also deployed using **Azure Kubernetes Service (AKS)** with multiple replicas across availability zones and regions. **Usage data** including API requests, performance metrics, and client hints are collected and stored in an Atlas database to be used for **analytics** purposes.

Authentication

The app supports **Google SSO** authentication to ensure that the articles published by a user cannot be edited, or deleted by other users. Users can browse, create, or edit articles even without logging in. Once they are ready, they can log in to save their **sessions**, so that they can continue later, to edit their published articles, or to publish new ones. The built-in editor provides options to save the session in the browser's local storage as well as in the cloud server. The server can track separate sessions for each article.

Security and Privacy

The Node.js server cannot be accessed without **HTTPS**. Also, the Node.js server provides dedicated APIs to manage additional TLS sessions on top of the existing TLS session within the HTTPS protocol, using **SHA-256** (asymmetric), and **AES-256-GCM** (symmetric) **encryptions** along with **key rotation** during every request.
This mechanism ensures privacy by allowing the client and the server to encrypt their communication payloads, thus preventing data leaks while transmitting packets through public or vulnerable networks. This ensures protection from packet sniffing attacks such as man-in-the-middle attacks. The key rotation avoids replay attacks.

The web app doesn't use **cookies** to store personal data such as session tokens, symmetric encryption keys, user info provided by the OAuth2 identity provider, etc... This ensures protection against CSRF (Cross-site request forgery) attacks.

The app has no **third-party dependencies**, and the entire app is built from scratch, thus offering protection from third-party vulnerabilities. This also provides reliability against third-party deprecations.

Google SSO is performed using **OIDC with PKCE flow**, in which the user is redirected to Google SSO provider's consent screen, where the users can securely provide their consent, after which the SSO provider (Identity Provider [IdP]) would send an authorization code to the app through a url query parameter. The app server submits the code along with a secret, and a code verifier (both of which are required to get authorized by the IdP) to the IdP to get the user info (ID Token). The secret, and the code verifier are never revealed publicly, thus preventing packet sniffers from utilizing the authorization code. In addition the IdP's reponse is verified by the nonce provided by the IdP, thus preventing the authorization code, or the IdP's response getting tampered while the packets are in transit through public networks. The ID Token provided by the IdP is a **JWT (JSON Web Token)**. The JWT is split based on the delimiting '.' characters, to get the base64 encoded user claims.

Performance and Caching

The app has **no third-party dependencies**, thus reducing the size of the built scripts to lesser than 200 KB. So, when users open the website, network load and latency are reduced while downloading the web app. The server combines server-side rendering (SSR) with client-side rendering (CSR) to reduce the initial page load time and FCP. **Code splitting**, implemented through lazy loading minimizes the network latency during page load. Built script chunks are named with a hash of their contents, thus allowing the browsers to **cache** the chunk files.

The React UI is optimized with **hooks** such as useMemo, useCallback, and APIs such as memo to make functionalities like dragging, editing, drawing, etc... seamless and responsive. These hooks provide mechanisms for caching rendered components, thus preventing unnecessary rerenders even when the state of the components or the states of any of the components' ancestors change.

The UI provides efficiency and reduces latency while accessing data using different **caching** mechanisms. It accesses the recently cached data during the same session through an **in-memory cache**, while accessing the earlier or past sessions' cached data through a more persistent **local storage**. It stores large files like images in the browsers' **IndexedDB** which is also a persistent storage.

The app utilizes the **performance API** of client agents, to ensure that the users are provided with the optimal performance. Whenever the users experience any issues like lack of responsiveness, latency in resource fetching, or inconvenient layout-shifts, the app reports, and logs the event. The app also **logs** every request made by the users along with **client-hints** including details about the user's device (memory, os, ip), client-agent (browser), network connection (bandwidth, RTT, ECT), preferences (color , language), etc... These reports and logs are analysed to fix issues, and improve the user experience.

Design Patterns and Coding Standards

Global and common variables such as session info, and in-memory cache are made available to the consumer codes through singleton classes and Context API. Custom hooks such as useApi and useDrag improve the **separation of concerns**. The useApi hook separates the common and repeated patterns across API calls such as attaching session tokens, encryption and decryption, and TLS handshakes. The useDrag hook maintains the states of the positions of every draggable component concerning the whole app layout. The singleton class TLS maintains and offers a single TLS session in addtion to the one within the HTTPS protocol, across the entire app.

An **in-memory data store** is implemented as a **singleton class** with static variables. It includes publish and subscribe functionalities. Singleton classes make the same benefits of state variables, to be available outside of custom hooks or function components. They also remove **prop drilling**. The data store is used to implement an in-memory cache.

The Node.js server utilizes separate **middlewares** to perform encryption and decryption of payloads, and authentication mechanisms at the entry and exit points of the packets, thus acting as a **firewall**. Common and CRUD operations are modularized into separate middlewares and reused across different resource types. **Express app**'s built-in functions are **modified** to serve custom purposes. This avoids replacing function calls with different functions, or middlewares at each of their occurrences across the code. The singleton class UserSessions maintains the session information of every user.

DBMS

The entire dataset is **replicated** across 3 nodes to improve availability, and partition tolerance. The database collections are maintained by unique indexes to provide fast access. and implement schema constraints. The Lucene-based **search engine** and **aggregation pipelines** allow customizable search and data processing queries. **Transaction** management is done for every query. **Write and read concerns** ensure reliability, consistency, and availability of data in real-time across regions. **NoSQL** database allows flexible schema, to support improvements in app functionalities. Request logs and **usage metrics** are collected and stored in a seperate NoSQL database. IndexedDB, local storage, and in-memory data stores are used to manage data in clients' devices, based on the required data persistency, performance, speed, and efficiency.

Deployment and Containerization

During the build, the react app is **bundled** into different minified chunks based on the locality of reference. React's lazy loading API improves page load times through **code splitting**. Docker's optimization features such as **bind mounts, and cache mounts** are used to manage dependencies in seperate build stages, thus reducing cache invalidations during the react build step. **Environment variables** are passed to the app in **runtime** using the browsers' window API. The window API is initialized with an env object using a self-destructing **script tag**, which is provided along with the initial **shell of SSR**.

The source code is committed to the GitHub repo, where a **GitHub actions workfow** rebuilds a new docker image and pushes it to **Azure Container Registry**, after which, the workflow redeploys the **Azure Container App** as a serverless instance to publish the committed app to the end-users.

K8s cluster was created and tested using **minikube**. The cluster was deployed and tested across regions and availability zones using **Azure AKS** by configuring the **nodeSelector** field of the deployment manifest.
The **deployment controller** uses **rolling update** strategy, and includes a **readiness probe** to ensure that the probes are ready to listen to incoming requests. The k8s secret manifest for environment variables is automatically **generated** by the JS code './js_scripts/createK8sSecretManifest.js', which **transpiles** the .env file to .yaml format, while encoding each value to **base64** to use in the manifest.

The folders **./.bat** and ./js_scripts contain scripts to perform development and deployment operations such as starting, updating, and running minikube node, building, and running Docker containers, and committing, and pushing to GitHub. These files simplify the development workflow by replacing frequently used complex command sequences with simple one line commands.

API Specification (Application Programming Interface Specification)

User Authentication

For resources that require authentication, session token must be provided. Once a user logs in and submits the authorization code, a session token will be provided to the user. It can either be attached to the body of the requests that access protected resources, or to the 'Authorization' header as the keyword 'bearer' followed by the session token.

/api/user/login

To perform Google OAuth2 login. Once logged in, you will be redirected to the Articles home page

which will submit the authorization code to the following API "api/user/getGoogleOAuth2Claims".

Methods Allowed : GET

Response :

302 : Redirection to the Google OAuth2 consent screen

500 : Internal server error

/api/user/getGoogleOAuth2Claims

To submit the Google Oauth2 authorization code to get the user claims and a session token

Methods Allowed : POST

Content-Type : application/json

Body :

accessToken : The access token received from the Google Oauth2 IdP

Response :

200 : The claims have been successfully retrieved from the IdP

data :

session : A JSON object with session information

userInfo : A JSON object containing the requested user claims

sessionToken : A string to identify the user during the session.

[NOTE: This is different from the session IDs which are associated with the TLS sessions]

207 : Request successfull. But error while storing the claims in the service provider's database

data : Same as the data for the status code 200.

401 : Invalid accessToken

500 : Internal server error

Manage user's edit sessions

/api/user/saveCloudSession

To save an editing session in the cloud, so that the session can be continued later

Methods Allowed : POST/PUT

Authentication : required

Content-Type : application/json

Body :

sessionToken : The session token provided during authentication

cloudSession : The JSON object representing the editing session

Response :

200 : The session has been successfully saved in cloud

400 : cloudSession object field required

401 : Authorization credentials required

422 : Invalid sessionToken

500 : Internal server error

/api/user/loadCloudSession

To fetch the saved editing sessions from the cloud

Methods Allowed : POST

Authentication : required

Content-Type : application/json

Body :

sessionToken : The session token provided during authentication

Response :

200 : The session has been successfully retrieved from cloud

data :

cloudSession : A JSON object representing the saved sessions

401 : Authorization credentials required

422 : Invalid sessionToken

500 : Internal server error

Article

/api/article/search

To search through the articles

Methods Allowed : POST

Content-Type : application/json

Body :

searchSpecs : A JSON object to use as the query for $search stage in MongoDB Atlas aggregation pipeline.

Response :

200 : Search results have been successfully processed. (This also includes empty search results)

data :

searchResults : An array of articles

400 : searchSpecs body field required and it must be a JSON object

422 : Unprocessible entity. Refer the response statusText and data for details

500 : Internal server error

/api/article/getOne

To fetch an article by its id

Methods Allowed : POST

Content-Type : application/json

Body :

id : The unique ID of the article

Response :

200 : The article has been successfully identified

data :

article : The fetched article

400 : id body field required and it must be a string

401 : Unauthorized request. Please provide authorization credentials.

422 : Invalid id

500 : Internal server error

/api/article/update

To update or upsert an article

Methods Allowed : POST/PUT

Authentication : required

Content-Type : application/json

Body :

article : The new article document

upsert : Boolean specifying the whether insertion allowed

Response :

200 : The article has been successfully created

data :

upsertedId : The id of the newly inserted document if an upsert was performed

400 : article body field required and it must be a JSON object

422 : Unprocessible entity. Refer the response statusText and data for details

409 : An article with the title already exists. Change the title of the article.

500 : Internal server error

/api/article/delete

To delete an article

Methods Allowed : POST/DELETE

Authentication : required

Content-Type : application/json

Body :

id : The id of the article to be deleted

Response :

200 : The article has been successfully deleted

400 : id field required and it must be a string

422 : Invalid id

409 : There is no article with the specified id

500 : Internal server error

TLS - Transport Layer Security

To manage additional TLS sessions, on top of the TLS session within the HTTPS protocol.

/api/tls/handshake

To establish a TLS session through TLS handshake.

Methods Allowed : POST

Content-Type : application/json

Body :

encryption : The encryption method used. It must be set to "public" for this client hello.

payload : 256 byte AES key encrypted using the public key of the TLS certificate

Response : The server hello

200 : Session has been successfully established

data :

sessionId : A newly created generated unique session id

ciphertext : An empty JSON object encrypted using the AES key

iv : Initialization Vector

authTag : AES-256-GCM auth tag for the ciphertext

Symmetric Encryption

/api/ (Any resource)**

- This is a general specification applicable across all the APIs

To use the AES key after the TLS handshake

Methods Allowed : POST/PUT/DELETE

Content-Type : application/json

Body :

- **sessionId :** The session ID generated during the TLS handshake
- **iv :** AES-256-GCM Initialization Vector for the payload
- **authTag :** AES-256-GCM auth tag for the payload
- **payload :** The actual body for the underlying API, but encrypted using the AES key

Response :

- **419 :** The TLS session has expired. Please initiate a new TLS handshake.
- **** :** Refer the specification of the underlying API to understand the status code
  - **data :**
    - **sessionId :** A newly created generated unique session id
    - **ciphertext :** The actual response data of the underlying API, but encrypted using the AES key
    - **iv :** AES-256-GCM Initialization Vector for the ciphertext
    - **authTag :** AES-256-GCM auth tag for the ciphertext