SAHIL DHONDIRAM DHANAVADE

sahildhanavade769@gmail.com | (+91) 9527552188

@sahildhanawadeofficial

/in/sahildhanavade

SKILLS

- **Python**
- **HTML, CSS, Javascript**
- **PHP**
- **MySQL**
- **Node js**
- **React js**
- **Next js**
- **MongoDb**
- **Git GITHUB**
- **Docker**
- **Kubernetes**
- **CICD Pipeline**
- **Argocd**
- **Grafana**
- **ShellScript**
- **NGINX**
- **AWS :- S3bucket, Cloudfront, aws global accelerator, load balancers(NLB,ALB), target group, route53, etc**
- **Google Cloud :- Running kubernetes cluster on GCloud using It’s free credits**

EDUCATION

- MCA (Master in computer Application) (currently pursuing)
- Savitribai Phule Pune University (B.com) 6.4 CGPA
- XII (kendriya Vidyalaya) (CBSE) (Commerce Stream) | School 78.4%
- X (kendriya Vidyalaya) (CBSE) | School 78.4%

ACHIEVEMENTS / HOBBIES

● Self-taught coding over the past three years and developed *storechoose.com*, an e-commerce store builder (full details mentioned in the resume projects section).

● Grew a YouTube channel(@sahildhanawade) to 12,100 subscribers by creating and sharing my cycling stunt videos.

Projects Made By Me (All the work you see here is done solely by me.)

- **Business Website**:- https://jdbinterior.netlify.app
- **Education website for MCQs** :- https://1markers.vercel.app
  - **Developed an MCQ-based learning platform** using **Next.js** for a seamless front-end and back-end integration.
  - Designed a user-friendly interface enabling students to **study MCQs, take tests**, and analyze their performance via a dedicated **Performance tab**.
  - Implemented secure **user authentication and session management** using **NextAuth**.
  - Utilized **MongoDB** for efficient data storage, ensuring reliable and scalable management of test data and user performance metrics.
  - Hosted the platform on **Vercel**.
- **Ecommerce Website** :- https://upmystandard.vercel.app

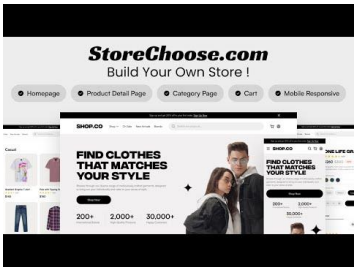
- **Developed a dynamic e-commerce platform** using **Next.js** for the front-end and back-end integration.
- **Designed an innovative product card UI**, allowing users to view product variants (e.g., size, color) and dynamically fetch images without navigating to a separate product page.
- **Integrated a reviews feature** for user feedback directly within the product card.
- **Built a comprehensive admin panel** enabling store owners to:
    - **Add and edit products** with custom option sets, such as size and color for t-shirts or processor, RAM, and storage for mobile phones.
    - **User Management** :- Manage user roles and **assign specific access** to employees for tasks like adding or updating products.
- **Implemented dynamic category creation**, where categories and corresponding filters are automatically added to the navigation bar and side panel as new products are added to the database.
- **Utilized MongoDB** for scalable and efficient data management, ensuring seamless product and user data handling.
- Hosted the platform on **Vercel**, ensuring high performance, scalability, and global accessibility.

❖ StoreChoose.com A Store Builder:- https://storechoose.com

Fully developed and ready to launch, currently not live due to GKE(Google Kubernetes Cluster) hosting costs.

How to Use StoreChoose.com (Hindi Video Tutorial):

https://youtu.be/5v1SFy4WqNo?si=xRL50wg-ucfRRCTG



 Overview:

- Developed **Storechoose.com**, a platform that enables users to create and customize their own e-commerce websites and

sell products online. The platform offers a **pay-as-you-go** pricing model where users purchase bandwidth for their stores, and bandwidth is deducted based on incoming traffic. Unlike other platforms, Storechoose.com charges **0% commission** on sales, payment gateway fees applicable (Stripe, Razorpay, Paytm, etc.).

- The platform competes directly with Shopify, Dukan, and Digital Showroom app etc by offering a more flexible pricing model and efficient use of cloud technologies.

 Key Features:

- **Zero Commission**: Storechoose.com does not take any commission from sales; users only pay for bandwidth usage.
- **Customizable E-commerce Stores**: Users can easily create and manage their e-commerce store using a user-friendly interface.
- **Bandwidth Deduction**: Each store has its own bandwidth allocation, and as traffic increases, bandwidth is automatically deducted from the user's purchased quota.

 Core Technologies:

- **Backend**:
    - **Node.js** for server-side operations.
    - **Next.js** (version 14) for the storefront.
    - **MongoDB** for data storage, managing user accounts, store configurations, product catalogs, and bandwidth usage tracking.
- **Caching**:
    - **Dragonfly** is used as the in-memory caching solution within **Kubernetes** cluster for efficient request handling.
    - **ioredis** npm package is used for interacting with Dragonfly.
- **SSL Management**:

        - **CertManager** in Kubernetes, integrated with **Let's Encrypt**, automates the issuance of SSL certificates for new store domains (e.g., sahil.storechoose.store).

- **CI/CD Pipeline**:
    - **DockerHub** hosts container images for microservices.
    - **ArgoCD** automatically syncs the changes in the Git repository (containing Kubernetes YAMLs) with the live Kubernetes environment. When code is pushed, a new image is built and tagged with the **Git commit hash**, pushed to DockerHub, and deployed in Kubernetes using ArgoCD.

- **Microservices & Key Components**:
    - **StorechooseApp**:
        - Primary microservice hosting the main platform at **storechoose.com**, responsible for user accounts, store creation, billing, and bandwidth purchase.
    - **Upmystandardnext14**:
        - Frontend microservice responsible for serving the individual e-commerce stores created by users (e.g., sahil.storechoose.store,xyzdomain.com). Built using **Next.js**, it handles rendering and delivery of the store pages.
        - Paytm and Razorpay payment gateway integrated.
        - Shiprocket and Shipstation integrated for shipping.
    - **Storeuiproxy**:
        - A **Node.js**-based HTTP proxy that intercepts incoming requests for individual store domains (e.g., sahil.storechoose.store,xyzdomain.com).
        - **Caching**: Checks Dragonfly cache for the requested data. If the data is present, it serves it directly from the cache. Otherwise, it proxies the request to **Upmystandardnext14** (storefront service) and caches the response.

    - **TTL Management**: It monitors the **time-to-live (TTL)** of cached data. If the TTL is about to expire, the data is refreshed before serving new requests to avoid stale data.

    - **Race Conditions Handling**: Handles concurrent requests by ensuring that multiple incoming requests for the same data don't result in redundant cache refresh operations.

- **SSLEnable**:

    - Responsible for dynamically creating **SSL certificates** for store domains (sahil.storechoose.store,xyzdomain.com) using **Let's Encrypt. HTTP01 challenge** is used for generating SSL certificates for custom domains attached to any store (Eg:- xyzdomain.com). **DNS01 challenge configured with Route53** is used for wildcard SSL certificates of domain (storechoose.store,storechoose.com)

    - Runs a periodic job every 30 minutes to check for new domains, create SSL certificates, and apply them.

- **TrafficAndUsageStreamRedis**:

    - This microservice tracks the bandwidth usage for stores based on the data cached in Dragonfly(redis streams).

    - For each incoming request, **Storeuiproxy** logs the number of bytes served and stores this information in Dragonfly (using redis streams for it), keyed by the **Store ID**.

    - Every 5 minutes, **TrafficAndUsageStreamRedis** aggregates this data and updates MongoDB with the total bandwidth consumed by each store.

- **TrafficAndUsageStream**:

    - Similar to the above but focused on tracking bandwidth for images and other media files served via **AWS CloudFront**.

        - CloudFront logs the amount of data served for each media request (known as standard logs). **TrafficAndUsageStream** parses these logs, identifies the **Store ID** from the file names (media files are stored with Store IDs), and updates the corresponding store's bandwidth usage in MongoDB.

- **Caching Architecture**:
    - **Dragonfly Caching**:
        - Storeuiproxy checks **Dragonfly** for cached responses before proxying requests to the storefront service (Upmystandardnext14).
        - If data is cached, it is served immediately, reducing response time and bandwidth usage.
        - **TTL Management**: Storeuiproxy checks the **TTL** for cached data on each request for the data to ensure it's refreshed just before expiration, minimizing downtime and race conditions.
- **Bandwidth Tracking & Management**:
    - **For Cached Data**:
        - Storeuiproxy records the size of each response (in bytes) for every request served and logs this in Dragonfly(redis streams), tagged with the **Store ID**.
        - **TrafficAndUsageStreamRedis** periodically aggregates these logs and updates the total bandwidth used by each store in MongoDB.
    - **For Media Files (via CloudFront)**:
        - **AWS CloudFront** is used to serve static media files (such as images stored in **S3 buckets**). Each file is named with the **Store ID**, so when CloudFront serves a file, it logs the data usage, which is processed by **TrafficAndUsageStream(micro service)** to calculate the bandwidth consumed by the store.
        - The calculated bandwidth is then deducted from the store’s quota in MongoDB.

 Deployment Architecture:

- **Global Load Balancing**:
    - **AWS Global Load Balancer** is used to distribute traffic globally. It forwards requests to an **AWS Network Load Balancer (NLB hosted in a specific region)**, which then forwards **TCP traffic** to the **NGINX proxy using TCP streams (running on ec2)** within a **Route Group** attached to the Network load balancer (NLB).
    - **NGINX (running on ec2)** forwards traffic to **Kubernetes** cluster running in **Google Cloud Platform (GCP)**. This setup ensures flexibility for moving to other cloud providers or bare-metal setups in the future.
- **SSL Termination in Kubernetes**:
    - SSL termination is handled within the Kubernetes cluster to maintain flexibility. **AWS's Global Load Balancer** forwards traffic using **TCP (Layer 4)** to support this configuration, as **GCP's Global Load Balancer** only forwards **HTTPS (Layer 7)** traffic, which would interfere with in-cluster SSL termination.

 Next-Auth package updated:

Separate Database is created for each Store according to **StoreId (eg:- 84af75bf-4ede-4ca7-9519-9d8f13e9ff51)**

To dynamically connect to **multiple databases** based on **storeId** in our (**Upmystandardnext14 a micro service** to serve store frontends) Next.js application with NextAuth.js, i initially used the standard MongoDBAdapter function. This approach, however, resulted in a significant problem: every time a request was made, it created a separate connection for each database, which led to numerous open connections, eventually overwhelming the system.

To solve this, I decided to switch to using the **MongoDB Data API** instead of direct connections. The MongoDB Data API allows us to interact with MongoDB databases via HTTP, making it a more efficient and scalable solution for our use case.

The updated **MongoDBAdapter function** now uses the MongoDB Data API instead of mongoDb connection to interact with different databases to handle authentication:

1. Dynamically route requests to the correct database based on storeId.

2. Eliminate the need for multiple open connections by relying on the stateless HTTP-based Data API.

3. Ensure that the application can scale without hitting connection limits or performance bottlenecks.

This shift from direct database connections to the MongoDB. Data API not only addresses the issue of open connections but also optimizes the overall performance and scalability of the application, especially when managing multiple databases dynamically.

Now our import for MongoDBAdapter function in **[…nextauth] file** looks like below I am importing it from our **servercomponents Folder** from within our application

```
import { MongoDBAdapter } from "@/servercomponents/MongoDBAdapter"


// import { MongoDBAdapter } from "@auth/mongodb-adapter"
```

- **Kubernetes Setup**:

    - **Multi-Cloud Kubernetes Cluster**: Deployed on **Google Cloud Platform (GCP)** using Kubernetes for orchestration and **ArgoCD** for GitOps-based continuous deployment.

    - **Microservices**: The Kubernetes cluster runs several microservices, each in separate deployments:

    - **StorechooseApp** (core platform)

    - **Upmystandardnext14** (storefront service)

    - **Storeuiproxy** (proxy and caching layer)

    - **SSLEnable** (SSL certificate management)

- **TrafficAndUsageStreamRedis** (bandwidth tracking and deduction for API requests)
- **TrafficAndUsageStream** (bandwidth Deduction service for media files served by CloudFront)

 CI/CD Pipeline:

- Code changes trigger an automated pipeline:
  - Github Actions trigger a new **Docker image** to be built and pushed to **DockerHub** and update the appropriate Yaml file in the repository for Yaml files.
  - The commit hash is used for versioning of Docker image.
  - **ArgoCD** automatically syncs with the **YAML configuration** repository and updates the running Kubernetes deployments, ensuring seamless and continuous integration and delivery.

 Scalability & Flexibility:

- This architecture allows **easy migration** to other cloud platforms (like Azure or AWS) or even **bare-metal servers**.
- **Multi-cloud strategy** leverages free credits on **GCP** while using **AWS** for global load balancing and CloudFront for media delivery.
- With this approach we can host multiple kubernetes cluster in different regions all across the world minimizing latency for users as we are caching data in dragonfly(database for caching) running within kubernetes. AWS Global Accelerator will load balance the traffic to different Network load balancers(NLB)=>RouteGroup(Nginx running on ec2 forwarding TCP traffic to kubernetes cluster hosted on GCloud or any other service provider or bare metal). **But in some cases like user placing an order** the order details have to go to mongoDb database. And if we are running our mongoDb cluster in a single region the latency may increase in this scenario. To address this issue we will **enable Global Cluster Configuration** option in mongoDb (**it enables Low-Latency Reads and Writes from anywhere in the world**

 Successfully Completed Storechoose.com :

I successfully completed **storechoose.com**, a project that took over a year to evolve from an idea to a fully functional solution.

Initially, my approach involved using **Vercel APIs** to create a new Project for each store created, based on the GitHub repository of **(upmystandardnext14)**. The admin panel was integrated into the e-commerce website itself. However, this approach proved to be **non-scalable**.

To improve scalability and flexibility, I learned **Kubernetes** and separated the admin panel's logic from the e-commerce website. This led to the creation of **(storechooseapp)** with some more added features, which made creating and managing stores much more efficient.

To address security concerns, I developed **(SSLEnable)**, a microservice written in **Node.js**. This service generates **SSL certificates** using **cert-manager** and **Let's Encrypt** for verified domains (by checking A records).

For caching, I built **(storeuiproxy)**, another Node.js microservice. All requests to store domains first route through **storeuiproxy**, which uses the **http-proxy** npm package to proxy requests to **(upmystandardnext14)**. It caches the responses in **Dragonfly** (a multithreaded, Redis-like caching database) for 15 minutes. Subsequent requests for the same URL (e.g., https://mystore.com) are served directly from the cache, significantly improving performance.

To implement a **pay-as-you-go** solution, I developed two additional microservices:

- **TrafficAndUsageStreamRedis**: Tracks and deducts bandwidth for API requests.
- **TrafficAndUsageStream**: Tracks and deducts bandwidth for media files served via **CloudFront**.

This architecture ensures scalability, security, and optimized performance while supporting a flexible pricing model for the platform.

 Pricing :

Our **basic plan** charges **₹150 for 1GB of bandwidth**. Based on our calculations:

- When a user visits a store, around **700 KB** of JavaScript and CSS files are loaded.
- We estimate that a user will explore **40 images**, each compressed to **20 KB**, which totals **800 KB** (40 × 20).
- This means a user's browser loads around **1.5 MB** of data per visit.
- Javascript files, CSS files and Images gets cached in the user's browser for over 1 year of time period so if the user's refreshes the page or come later after some time or some days **it will not result in cosuming another 1.5MB of bandwidth**

If **1 GB (1024 MB)** is divided by **1.5 MB**, we can serve approximately **682 users** per GB. Rounding this up, we assume **700 users**.

If the **conversion rate** (users making a purchase) is **1%**, then **7 customers** will make a purchase. Dividing ₹150 (the cost of 1GB) by 7 gives a **cost of ₹21 per conversion** for the store owner.

Running Costs

The **minimum cost** for running the infrastructure includes:

1. **Kubernetes Nodes** (N2 or N instances with 8 cores and 24GB RAM): ₹20,000 per month.
2. **Other Costs** (AWS Global Accelerator, Network Load Balancer, RouteGroup, and EC2 running NGINX TCP streams): ₹10,000 per month.
3. **S3 Bucket and CloudFront** costs are not fixed like above two they are relative to the traffic we get, for low traffic in our initial days they will mostly fall in free tier

So, the **total monthly cost** is approximately **₹30,000**.

Performance

- Each **storeuiproxy(serving requests from cache dragonfly database running within kubernetes)** microservice pod can handle **60 requests per minute**.

- With **5 pods of storeuiproxy running**, we can handle **300 requests per minute**.

Break-Even Point

To cover our monthly cost of ₹30,000 (200Gb of recharge X Rs150) total requests to handle = (200Gb X 680 requests per Gb = 1,36,000 requests) :

- We need to handle 1,36,000 requests per month, **4,533 requests per day**, which is around **3 requests per minute**.

- Since our infrastructure can handle **300 requests per minute**, achieving the break-even point is well within capacity.