**Project Report: Real-Time Chat Application Development using Spring Boot, Angular, Mongo DB, Docker & CI/CD**

This report summarizes the step-by-step development of a real-time chat application, including its core functionalities, containerization with Docker, and the establishment of an automated Continuous Integration/Continuous Delivery (CI/CD) pipeline using GitLab CI/CD. It also details the various issues encountered throughout the process and their respective solutions.

**1. Project Overview**

The project involved building a real-time chat application with a multi-component architecture:

* **Angular Frontend:** The client-side application for user interaction and displaying messages.
* **Java Spring Boot Backend:** The server-side application providing REST APIs for user management and handling WebSocket connections for real-time chat.
* **MongoDB Database:** A NoSQL database for persisting user data and chat messages.

The ultimate goal was to develop these components, containerize them using Docker, and then automate their build and deployment process to Docker Hub using GitLab CI/CD.

**1. MongoDB Setup Instructions**

Before running the backend, a MongoDB instance was required.

* **Options Provided:**
  + **Local MongoDB:** Recommended for quick local development, involving downloading MongoDB Community Server, creating the /data/db directory, and starting mongod.
  + **MongoDB Atlas (Cloud-based):** An alternative for a cloud-hosted database, requiring signing up for a free tier cluster, configuring network access, creating a database user, and obtaining a connection string.

**1.2 Spring Boot Backend Setup**

The backend was built using Spring Boot with Java.

* **Project Creation:**
  + Initiated the project using Spring Initializr (<https://start.spring.io/>) as a Maven Project with Java 17, Spring Boot 3.3.1, com.example group, chat artifact.
  + Key dependencies added: Spring Web, Spring Data MongoDB, Spring WebSocket, and Lombok.
* **pom.xml Updates:**
  + The generated pom.xml was reviewed and confirmed to include the necessary dependencies (spring-boot-starter-web, spring-boot-starter-data-mongodb, spring-boot-starter-websocket, sockjs-client, stomp-websocket, lombok) and Java 17 compatibility.
* **Main Application Class (ChatApplication.java):**
  + The standard Spring Boot entry point was kept as generated.
* **Model Classes (Message.java, User.java):**
  + Message.java: Defined fields for id, sender, content, and timestamp, mapped to a messages collection in MongoDB. Lombok's @Data was used.
  + User.java: Defined fields for id, username, and password, mapped to a users collection. (Note: Passwords were stored in plain text for simplicity in this learning phase, with a warning about hashing in production).
* **Repository Interfaces (MessageRepository.java, UserRepository.java):**
  + MongoRepository was extended for basic CRUD operations.
  + Custom methods like findAllByOrderByTimestampAsc() for MessageRepository and findByUsername() for UserRepository were added for specific queries.
* **Controller Classes (AuthController.java, ChatController.java):**
  + AuthController.java: Handled /api/auth/register (user registration, checking for unique usernames) and /api/auth/login (user authentication).
  + ChatController.java: Exposed /api/messages for fetching chat history (REST GET) and handled WebSocket messages via @MessageMapping for sending messages (/app/chat.sendMessage) and adding users (/app/chat.addUser), broadcasting them to /topic/public.
* **WebSocket Configuration (WebSocketConfig.java):**
  + Configured WebSocket endpoints (/ws with SockJS fallback) and message broker prefixes (/app for application destinations, /topic for public subscriptions).
* **application.properties Setup:**
  + Configured server.port=8080.
  + Initially set spring.data.mongodb.host=localhost and spring.data.mongodb.port=27017 for local MongoDB. Later adjusted for Docker Compose and MongoDB Atlas.
* **Build and Run Instructions:**
  + mvn clean install to build the JAR.
  + mvn spring-boot:run to start the application.

**1.3 Angular Frontend Setup**

The frontend was built using Angular.

* **Project Creation:**
  + Node.js and Angular CLI were installed (npm install -g @angular/cli).
  + A new Angular project was created using ng new chat-frontend --no-standalone --routing=false --style=css.
* **Tailwind CSS Setup:**
  + Installed tailwindcss, postcss, autoprefixer.
  + Initialized with npx tailwindcss init -p.
  + Configured tailwind.config.js to scan src/\*\*/\*.{html,ts}.
  + Added Tailwind directives (@tailwind base, etc.) to src/styles.css.
* **WebSocket Client Libraries:**
  + Installed sockjs-client and @stomp/stompjs for WebSocket communication.
* **Services (auth.service.ts, chat.service.ts):**
  + AuthService: Handled HTTP requests for user register and login to the backend.
  + ChatService: Managed WebSocket connection (using SockJS and STOMP), sending messages, and receiving real-time messages. Also provided a method to fetch historical messages via REST.
* **Module (app.module.ts):**
  + Imported BrowserModule, FormsModule, HttpClientModule.
  + Declared AppComponent.
  + Provided ChatService and AuthService.
* **Component (app.component.ts):**
  + Implemented OnInit and OnDestroy lifecycle hooks.
  + Managed login/registration state, current user, message list, and new message content.
  + Subscribed to ChatService for real-time messages and handled scrollToBottom for message display.
* **HTML Template (app.component.html):**
  + Structured the UI for login/register forms and the chat interface.
  + Used Angular directives (\*ngIf, \*ngFor, [(ngModel)], [ngClass]) and Tailwind CSS for styling.
* **CSS (app.component.css):**
  + Left mostly empty, relying on Tailwind. Included custom scrollbar styles.
* **Run and Verification:**
  + ng serve --open to start the frontend.
  + Verification involved registering users, logging in from multiple tabs, and observing real-time message exchange and scrolling.

**1.4 Conclusion & Verification**

A functional real-time chat application was running locally, demonstrating user authentication, message persistence, and real-time communication via WebSockets.

**2. Enhanced User Presence**

Developing "who's online" feature, showing active users in the chat.

**2.1 Backend Updates**

The Spring Boot backend was updated to detect and broadcast user connection/disconnection events.

* **WebSocketEventListener.java:**
  + Created a new class annotated with @Component to listen for SessionConnectedEvent and SessionDisconnectEvent.
  + On connect, it retrieved the username from the WebSocket session (stored by ChatController.addUser) and broadcasted a "joined!" message to /topic/public.
  + On disconnect, it retrieved the username and broadcasted a "left!" message to /topic/public.
  + SimpMessageSendingOperations was injected to facilitate broadcasting.
* **ChatController.java:**
  + No significant changes were needed, as the existing addUser method already stored the username in the session attributes, which the listener then utilized.
* **Rebuild and Run:**
  + The backend was stopped, rebuilt (mvn clean install), and rerun (mvn spring-boot:run) to incorporate the new event listener.

**2.2 Frontend Updates**

The Angular frontend was updated to consume and display the user presence messages.

* **app.component.ts Updates:**
  + A new onlineUsers: Set<string> was added to store unique online usernames.
  + The ChatService subscription logic was enhanced to:
    - Add users to onlineUsers when a "joined!" message (from another user) was received.
    - Remove users from onlineUsers when a "left!" message was received.
  + The current user was added to their onlineUsers set immediately upon successful login.
  + The onlineUsers set was cleared on logout.
* **app.component.html Updates:**
  + A new div section was added to display the "Online Users" list, looping through onlineUsers and showing a presence indicator.
* **Run and Verify:**
  + The Angular development server was restarted (ng serve --open).
  + Verification involved logging in with multiple users in separate tabs, observing the "Online Users" list updating dynamically as users joined or left, and confirming that standard chat messages still functioned.

**2.3 Conclusion & Verification**

Successfully added real-time user presence tracking, making the chat application more interactive and robust.

**3. Containerization with Docker & Docker Compose**

Here we have focused on packaging the application components into Docker containers and orchestrating them with Docker Compose.

**3.1 Backend Dockerization**

* **backend/Dockerfile:**
  + A multi-stage Dockerfile was created:
    - **build stage:** Used eclipse-temurin:17-jdk-jammy to install Maven, copy pom.xml and source code, and build the Spring Boot JAR (mvn clean install -DskipTests).
    - **Final stage:** Used eclipse-temurin:17-jre-jammy (JRE-only for smaller image), copied the built JAR from the build stage, exposed port 8080, and defined the ENTRYPOINT to run the JAR (java -jar chat-app.jar).
    - **Crucial Fix:** The JAR file name in COPY --from=build was corrected to match the actual Maven output (chat-0.0.1-SNAPSHOT-0.0.1-SNAPSHOT.jar).
* **application.properties Update:**
  + The MongoDB connection details in backend/src/main/resources/application.properties were updated to connect to the MongoDB service by its Docker Compose service name (mongodb) and explicit authentication details:
  + spring.data.mongodb.host=mongodb
  + spring.data.mongodb.port=27017
  + spring.data.mongodb.database=chatdb
  + spring.data.mongodb.username=chatuser
  + spring.data.mongodb.password=chatpassword
  + spring.data.mongodb.authentication-database=admin
* **Build and Run (Individual Docker Image):**
  + docker build -t chat-backend .
  + docker run -p 8080:8080 -d --name chat-backend-container chat-backend
  + Verification: docker ps, docker logs chat-backend-container, and API calls to http://localhost:8080/api/messages.

**3.2 Frontend Dockerization**

* **frontend/chat-frontend/Dockerfile:**
  + A multi-stage Dockerfile was created:
    - **build stage:** Used node:18-alpine, installed Angular CLI, copied package.json, ran npm ci, copied source code, and built the Angular app for production (ng build --configuration production --output-path /app/dist/frontend --verbose).
    - **Final stage:** Used nginx:1.23.4-alpine, copied the built Angular app from /app/dist/frontend to Nginx's web root (/usr/share/nginx/html).
    - **Critical Fix:** Ensured mkdir -p /usr/share/nginx/html/assets was run in the Dockerfile to create the necessary directory before entrypoint.sh tries to write to it.
* **entrypoint.sh:**
  + A shell script (entrypoint.sh) was created to be executed at container startup. Its primary role was to dynamically inject the backend's BASE\_URL (passed as an environment variable from Docker Compose) into config.json file located in the Angular app's assets directory. It then started Nginx in the foreground.
* **nginx.conf:**
  + Custom Nginx configuration was used to serve the Angular application, handle try\_files for Angular routing, and specifically configure the /assets/config.json location to prevent caching, ensuring fresh configuration on restarts.
* **Angular App Updates (config.service.ts, app.module.ts, auth.service.ts, chat.service.ts):**
  + **config.service.ts:** New service to load config.json at application startup (using APP\_INITIALIZER) and provide the BASE\_URL to other services. It also included a fallback for local development (when config.json might not exist).
  + **app.module.ts:** Configured APP\_INITIALIZER to call ConfigService.loadConfig() before the Angular application bootstrapped.
  + **auth.service.ts & chat.service.ts:** Modified to inject ConfigService and retrieve API\_BASE\_URL dynamically from it, instead of using hardcoded values.

**3.3 Docker Compose Setup**

A docker-compose.yml file was created at the root of the project to orchestrate all three services.

* **mongodb Service:**
  + Used mongo:6.0.4 image.
  + Mapped port 27017.
  + Used a named volume (mongodb\_data) for persistent data.
  + **Crucial Fix for Authentication:** Initial MongoDB user creation via MONGO\_INITDB\_ROOT\_USERNAME/PASSWORD proved unreliable due to replica set requirements. The final, robust solution involved:
    - Overriding the command to mongod --auth --replSet rs0 --bind\_ip\_all.
    - Mounting a custom mongo-init.js script to /docker-entrypoint-initdb.d/ which manually initiated the replica set and created the chatuser in the admin database with readWrite access to chatdb.
    - Increased start\_period in healthcheck to allow the replica set initiation and user creation to complete.
* **backend Service:**
  + Configured to build from ./backend context.
  + Mapped port 8080.
  + depends\_on: Configured to wait for mongodb to be service\_healthy.
  + environment: Provided SPRING\_DATA\_MONGODB\_URI for connectivity.
* **frontend Service:**
  + Configured to build from ./frontend/chat-frontend context (corrected path).
  + Mappped host port 4200 to container port 80 (Nginx).
  + environment: Passed BASE\_URL (http://backend-service:8080) to entrypoint.sh.
  + depends\_on: Configured to wait for backend to start.
* **Volumes:**
  + Defined a named volume mongodb\_data for MongoDB persistence.
* **Overall Docker Compose Operations:**
  + docker compose build
  + docker compose up -d
  + docker compose logs -f for monitoring.
  + docker compose down for stopping and removing.

**4. CI/CD Pipeline Setup**

The final phase involved setting up Continuous Integration and Continuous Delivery for the Dockerized application using various CI/CD tools.

**4.1 GitLab CI/CD Pipeline Setup**

* A .gitlab-ci.yml file was created at the root of the GitLab repository.
* It defined global variables for Docker registry, username, password, and image names (swathiraok/chat-backend:latest, swathiraok/chat-frontend:latest).
* The pipeline used a single stage: build\_and\_push\_images.
* Two combined jobs were created: build\_and\_push\_backend\_image and build\_and\_push\_frontend\_image. Each job:
  + Used a docker:20.10.24 image with docker:20.10.24-dind service.
  + Logged into Docker Hub using CI/CD variables for credentials.
  + Navigated to the respective application directory (backend or frontend/chat-frontend).
  + Executed docker build to create the Docker image with a specific tag.
  + Immediately followed with docker push to upload the image to Docker Hub.
  + Used rules to trigger only on pushes to the main branch affecting relevant paths.
  + The frontend job was configured to needs the backend job to ensure correct order of execution.

**4.2 GitHub Actions CI/CD Pipeline Setup**

If we were to deploy this project using GitHub Actions, we would define two separate workflow files (.yml files) in the .github/workflows/ directory: one for the backend and one for the frontend.

* **backend-ci.yml (for backend/ changes):**
  + **Trigger:** On push to main branch, specifically when changes occur within the backend/\*\* path.
  + **Job:** A build-and-push job running on ubuntu-latest.
  + **Steps:**
    1. actions/checkout@v4: To clone the repository.
    2. docker/login-action@v3: To log in to Docker Hub using GitHub Secrets (${{ secrets.DOCKER\_USERNAME }} and ${{ secrets.DOCKER\_PASSWORD }}).
    3. **Build Docker Image:** Execute docker build --file ./backend/Dockerfile --tag your-docker-username/chat-backend:latest ./backend.
    4. **Push Docker Image:** Execute docker push your-docker-username/chat-backend:latest.
* **frontend-ci.yml (for frontend/chat-frontend/ changes):**
  + **Trigger:** On push to main branch, specifically when changes occur within the frontend/chat-frontend/\*\* path.
  + **Job:** A build-and-push job running on ubuntu-latest.
  + **Steps:**
    1. actions/checkout@v4: To clone the repository.
    2. docker/login-action@v3: To log in to Docker Hub using GitHub Secrets.
    3. **Build Docker Image:** Execute docker build --file ./frontend/chat-frontend/Dockerfile --tag your-docker-username/chat-frontend:latest ./frontend/chat-frontend.
    4. **Push Docker Image:** Execute docker push your-docker-username/chat-frontend:latest.
  + **Dependencies:** While not explicitly shown in simple GitHub Actions workflows (they run independently by default unless needs is used), for a full deployment, a separate deployment workflow could needs both build workflows to complete before deploying the combined application.

**4.3 Jenkins CI/CD Pipeline Setup**

Deploying this project with Jenkins would typically involve setting up a Jenkins server (self-hosted or cloud-hosted) and configuring a "Pipeline" job using a Jenkinsfile stored in your repository.

* **Jenkins Environment Setup:**
  + A Jenkins Master node with Docker installed and configured.
  + Potentially Jenkins Agent nodes for distributed builds.
  + Plugins for Docker, Git, Pipeline, and credentials management would be installed.
* **Jenkinsfile (Declarative Pipeline):**
  + A Jenkinsfile would be placed in the root of the repository.
  + **Pipeline Definition:**
  + pipeline {
  + agent any // Or a specific agent label (e.g., agent { label 'docker-host' })
  + stages {
  + stage('Checkout') {
  + steps {
  + git 'YOUR\_GIT\_REPOSITORY\_URL' // Clone the repository
  + }
  + }
  + stage('Build and Push Backend') {
  + steps {
  + script {
  + // Log in to Docker Hub using Jenkins credentials
  + withCredentials([usernamePassword(credentialsId: 'docker-hub-creds', passwordVariable: 'DOCKER\_PASSWORD', usernameVariable: 'DOCKER\_USERNAME')]) {
  + sh "docker login -u ${DOCKER\_USERNAME} -p ${DOCKER\_PASSWORD} docker.io"
  + // Navigate to backend directory and build
  + sh "cd backend && docker build -t your-docker-username/chat-backend:latest ."
  + sh "docker push your-docker-username/chat-backend:latest"
  + }
  + }
  + }
  + }
  + stage('Build and Push Frontend') {
  + steps {
  + script {
  + withCredentials([usernamePassword(credentialsId: 'docker-hub-creds', passwordVariable: 'DOCKER\_PASSWORD', usernameVariable: 'DOCKER\_USERNAME')]) {
  + sh "docker login -u ${DOCKER\_USERNAME} -p ${DOCKER\_PASSWORD} docker.io"
  + // Navigate to frontend directory and build
  + sh "cd frontend/chat-frontend && docker build -t your-docker-username/chat-frontend:latest ."
  + sh "docker push your-docker-username/chat-frontend:latest"
  + }
  + }
  + }
  + }
  + // Further stages for deployment (e.g., to Kubernetes/Compose)
  + // stage('Deploy') {
  + // steps {
  + // sh "docker compose -f docker-compose.prod.yml up -d" // Example for Docker Compose deployment
  + // }
  + // }
  + }
  + }
  + **Credentials:** Docker Hub credentials (docker-hub-creds) would be securely stored in Jenkins (e.g., as "Username with password" credential type).
  + **Triggers:** Jenkins jobs can be configured to poll the SCM for changes or receive webhooks from GitLab/GitHub to trigger builds automatically.

**4.4 Frontend Deployment on Render**

The Angular frontend will also be deployed as a Render "Static Site" or "Web Service" (if using Nginx directly). Given our setup, deploying it as a **Web Service** that runs Nginx is the most suitable.

* **Prerequisites:**
  + Your chat-frontend Docker image is successfully pushed to Docker Hub.
  + The backend service is deployed and its URL is known (e.g., https://chat-backend-service.onrender.com).
* **Steps on Render:**
  + **Log in to Render:** Go to <https://render.com/> and log in.
  + **New Web Service:** Click "New" -> "Web Service".
  + **Connect Git Repository:** Connect your GitLab (or GitHub) repository where your frontend/chat-frontend/Dockerfile resides.
  + **Configuration:**
    - **Name:** chat-frontend-service (or a similar descriptive name).
    - **Region:** Choose the same region as your backend.
    - **Branch:** main.
    - **Root Directory:** frontend/chat-frontend (This tells Render where to find your Dockerfile).
    - **Build Command:** docker build -t chat-frontend .
    - **Start Command:** /bin/bash /entrypoint.sh (This executes your custom entrypoint script).
    - **Port:** 80 (Nginx listens on port 80 inside the container).
    - **Instance Type:** Choose a suitable instance type.
  + **Environment Variables:**
    - BASE\_URL: The URL of your deployed backend service (e.g., https://chat-backend-service.onrender.com). This BASE\_URL will be picked up by your entrypoint.sh script and injected into config.json for the Angular app.
  + **Create Web Service:** Click "Create Web Service". Render will then pull your code, build the Docker image, and deploy it.

**4.5 MongoDB Atlas Integration on Render**

For production, it's best to use a managed database like MongoDB Atlas.

* **Configuration:** The MongoDB Atlas connection string (e.g., mongodb+srv://...) is provided as an environment variable (SPRING\_DATA\_MONGODB\_URI) directly to the **backend** Render service. Render handles the secure injection of this variable, allowing the Spring Boot application to connect to your cloud MongoDB instance.

**4.6 Issues Faced and Solutions (from GitLab CI/CD Experience)**

Several challenges arose during the CI/CD setup, primarily due to the isolated nature of CI/CD job environments and specific Docker image availability on GitLab's shared runners. These lessons are transferable to other CI/CD platforms.

* **Issue 1: CI/CD Configuration Not Recognized / Pipeline Not Triggering.**
  + **Cause:** The .gitlab-ci.yml file not being correctly pushed to the repository or subtle YAML syntax errors causing validation failures. Initial restrictive rules also prevented jobs from triggering.
  + **Solution:** Ensured the .gitlab-ci.yml file was correctly pushed to the main branch. Utilized CI/CD Lint tools for syntax validation. Simplified rules to trigger consistently. Explicitly quoted echo commands in script blocks ('echo "..."') to adhere to YAML string formatting, preventing Incorrect type errors.
* **Issue 2: Docker Image Pull Failures (manifest unknown).**
  + **Cause:** CI/CD runners failed to pull specific Docker base image tags from Docker Hub (e.g., docker:20.10.16-alpine, docker:latest-git, docker:latest-dind), indicating they were unstable, deprecated, or not consistently available.
  + **Solution:** Systematically updated the image and services definitions in the .gitlab-ci.yml to use more explicit, widely available, and stable Docker image versions (e.g., docker:20.10.24 for the job image and docker:20.10.24-dind for the Docker-in-Docker service).
* **Issue 3: "An image does not exist locally with the tag" Error During docker push.**
  + **Cause:** This was a critical issue stemming from the isolated nature of CI/CD job environments. An image built in one job (build\_image) was not automatically available to a subsequent job (push\_image) because the first container was terminated.
  + **Solution (Crucial Fix):** Combined the build and push steps for each Docker image into a *single job*. This ensures that the docker build command successfully creates the image, and the subsequent docker push command operates on the image still present within the same job's container environment. Debugging commands like docker images, --no-cache, and --progress=plain were added to docker build commands to provide verbose output and confirm image creation within the job. This principle of combining build/push is highly applicable to Jenkins and GitHub Actions if not handled implicitly by their runners.
* **Issue 4: Docker Hub Not Showing Latest Images After Pipeline Passes.**
  + **Cause:** A direct consequence of Issue 3. If the docker push command failed internally (due to the image not being present locally), Docker Hub would not show updates, even if the job itself technically "passed" without a fatal error.
  + **Solution:** Resolved once the build and push steps were combined into a single job, allowing the docker push to operate on a locally present image.
* **Issue 5: Persistent Configuration Synchronization Issues.**
  + **Cause:** A recurring problem was ensuring that the latest CI/CD configuration (e.g., .gitlab-ci.yml) was actually applied to the remote repository. Older versions of the file would sometimes continue to trigger pipelines despite local changes.
  + **Solution:** Required precise, explicit confirmation of copying the *entire* updated configuration content and overwriting the existing file in the remote repository's web editor, followed by a new commit to the main branch. This ensured the CI/CD system always ran the intended configuration.

**4.7. Conclusion**

The project successfully delivered a real-time chat application with a client-server architecture, utilizing Angular for the frontend, Spring Boot for the backend, and MongoDB for data persistence. All application components were containerized using Docker and orchestrated for local development using Docker Compose.

The Continuous Integration/Continuous Delivery (CI/CD) pipeline was automated with GitLab CI/CD, enabling automatic building and pushing of Docker images to Docker Hub upon every push to the main branch. This automation significantly streamlines the development workflow, ensuring consistent builds and efficient deployments. The lessons learned, particularly regarding job isolation and environment consistency, are fundamental to setting up effective CI/CD across various platforms like GitHub Actions and Jenkins.

While the project established a strong foundation, future enhancements could include:

* Deployment to cloud platforms (e.g., Render, Kubernetes) for production.
* Implementing more robust authentication (e.g., JWT).
* Integrating an external message broker (like Apache Kafka) for large-scale real-time message handling.









