DEPI - Program

DevOps Track

Technical Report:

Todo List App CI/CD and Deployment Architecture

Graduation project

First patch

Yat 142 – Alexandria

Under supervision:

Eng/ Mostafa Roshdy

Table of Contents:

1.Team members (Team 3):

- Ahmed Samir
- Mahmoud Akrm
- Ali Mohamed Rezq
- Ziad Hesham
- Ibrahim Elsaid Abd Elkader
- Ahmed Kamel

2.overview:

The Todo List App is a full-stack application designed to help users manage their tasks with a clean, intuitive interface. The application is developed using React for the front end and NodeJS for the back end, ensuring a smooth, dynamic user experience. It is deployed on a Kubernetes (K8s) cluster, leveraging modern containerization and orchestration technologies to provide scalability, reliability, and continuous delivery. This report delves into the application architecture, Docker setup, CI/CD pipeline stages, Kubernetes cluster configuration, monitoring, and future enhancements.

3. Application Architecture:

The Todo List App architecture follows a decoupled design where the frontend and backend operate independently but communicate via APIs, ensuring flexibility in development, testing, and deployment.

3.1 **Frontend**

- Framework: React (JavaScript/TypeScript)
- Features:
  - User Interface (UI): The UI is designed to be responsive, providing users with an optimal experience across various devices (mobile, tablet, desktop). React hooks are used extensively for managing state and side effects.

    - Task Management: Users can add, update, delete, and mark tasks as complete. The app offers real-time task filtering and sorting based on task status or deadlines.
    - Routing: React Router is employed for seamless navigation between different views (e.g., task list, settings, user profiles).
    - State Management: React's context API or a state management library (like Redux) is used to manage the application's global state, making it easier to pass data across different components.

3.2 **Backend**

- Framework: NodeJS (Express)
- API Design:
    - RESTful APIs: The backend provides several RESTful endpoints for creating, reading, updating, and deleting tasks. Authentication endpoints also exist to manage user sessions.
    - Database: MongoDB is used for storing user data and task information. The backend interacts with the database using Mongoose for data modeling and schema management.
    - Middleware: Authentication and validation are handled using middleware functions. For example, JSON Web Tokens (JWT) are used to secure the API,

ensuring that only authenticated users can interact with it.

- Error Handling: The backend includes robust error handling and logging to provide useful insights during runtime and help with debugging.

4. Containerization (Docker)

To ensure that the application runs in any environment without compatibility issues, both the frontend and backend are containerized using Docker. This allows for consistent environments from development to production.

4.1 Frontend Dockerfile

The frontend is built and served from a Docker container that contains:

- Base Image: A lightweight NodeJS image is used as the base. This ensures the app can build and run in any system that supports Docker.
- Dependency Installation: The necessary dependencies (like React, React Router, etc.) are installed via npm or yarn.
- Build Process: The Dockerfile includes a build step to optimize the React application for production. This includes minification of JavaScript, inlining critical CSS, and optimizing static assets.
- Serving the App: After building the app, a lightweight server (e.g., nginx) is used to serve the static assets. This

ensures fast, efficient delivery of the frontend application.

4.2 Backend Dockerfile

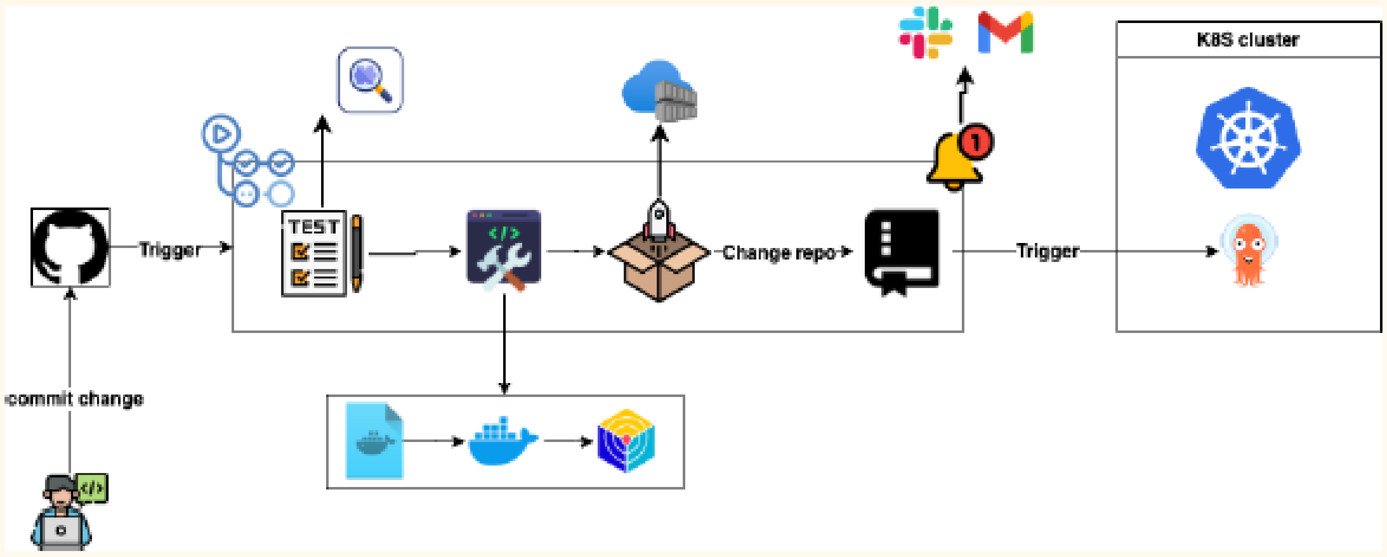
The backend is also containerized to ensure consistency and reliability:

- Base Image: A minimal NodeJS image is chosen to reduce the container size and speed up deployment times.
- Dependency Installation: All necessary modules and packages (e.g., Express, Mongoose) are installed during the image build process.
- Environment Variables: Sensitive data (like database URLs, API keys) are injected into the container via environment variables, following best security practices.
- Port Exposure: The backend exposes the required port (usually 3000 or 8080) for communication with the frontend or external services.

Once both Docker images are built, they are pushed to Docker Hub, making them easily accessible for deployment.

5. CI/CD Pipeline

- The CI/CD pipeline ensures that changes made to the codebase are automatically tested, built, and deployed, leading to faster iterations and fewer manual steps in the deployment process. The pipeline uses GitHub Actions for the CI part and ArgoCD for orchestrating continuous delivery.



- **5.1 Triggering Events**
- GitHub Integration: The pipeline is triggered whenever a developer pushes changes or submits a pull request (PR) to the main branch in the GitHub repository. This allows for seamless code integration and deployment.
- Branching Strategy: A branching strategy such as Git Flow or GitHub Flow is employed, ensuring that only stable, tested code is merged into the main branch, while feature branches or bug fixes are tested in isolation.
- **5.2 CI Stage (Continuous Integration)**
- Automated Testing: Once the code is pushed, GitHub Actions runs a series of automated tests, including:
- Unit Tests: Test individual components and functions to ensure they behave as expected.
- Integration Tests: Ensure that different components (e.g., frontend and backend) communicate correctly.

- Linting and Formatting Checks: Tools like ESLint and Prettier are used to enforce code quality and consistency.
- Containerization: After passing the tests, the pipeline proceeds to build the Docker images for both frontend and backend components.
- Image Security: The images are scanned using Trivy for known vulnerabilities, ensuring that no insecure dependencies are included in the production environment.

- **5.3 CD Stage (Continuous Delivery)**
- Orchestration with ArgoCD: After the Docker images are successfully built and scanned, ArgoCD pulls the updated images from Docker Hub and deploys them to the Kubernetes cluster.
- Blue/Green Deployments: A deployment strategy such as blue/green or rolling updates ensures minimal downtime and allows quick rollback in case of issues.
- Deployment Verification: Post-deployment, smoke tests are run to verify the health and functionality of the application.
- Notifications: If any stage of the pipeline fails, notifications are sent to developers via Slack, Gmail, or other integrated messaging tools to ensure quick troubleshooting and resolution.

6. Kubernetes Cluster Structure

The Kubernetes cluster provides a scalable and highly available environment for running the application. Its design ensures that the app can handle varying loads, scale horizontally, and recover from failures automatically.

6.1 Node Structure

- Master and Worker Nodes: The cluster consists of a master node (control plane) responsible for scheduling and managing workloads and worker nodes where the actual application containers (pods) run.
- Pod Management: Each service (frontend, backend) runs inside its respective pods, which are managed by Kubernetes Deployments. These deployments ensure that the right number of pod replicas are always running.
- Service Discovery: Kubernetes Services expose the pods, allowing the frontend to communicate with the backend and vice versa.

6.2 Scaling and Load Balancing

- Horizontal Pod Autoscaler (HPA): The cluster uses the HPA to automatically scale the number of pods based on CPU/memory usage, ensuring the app can handle increased traffic without manual intervention.
- Load Balancing: A built-in Kubernetes LoadBalancer or Ingress ensures that traffic is distributed evenly across multiple pod replicas.

6.3 Network Policies

- Internal Communication: Network policies enforce communication rules between services, ensuring that only authorized services can communicate with each other. For instance, the frontend should only talk to the backend, and the backend should only talk to the database.
- External Access: External access to the app is managed using Ingress controllers, which route traffic to the appropriate services.

6.4 Persistent Storage

- **Persistent Volumes (PVs):** For data that needs to be retained beyond the lifecycle of a pod (such as user information or task data), Kubernetes Persistent Volumes are used, ensuring data persistence and resilience.

6.5 Deployment

- Kubernetes (K8s) Cluster:
- The Kubernetes cluster pulls the Docker image from the container registry.
- The application is deployed to the K8s cluster, where it is run in a production-like environment.

6.6 Monitoring and Alerts:

- Monitoring tools (such as Prometheus or Grafana) continuously check the application's health.
- Any critical issues trigger an alert that is sent to the team via email, Slack, or other messaging platforms.

- **Triggers and Flow**
- 1. Commit Change: A developer commits a change in the GitHub repository. This action triggers the CI pipeline.
- 2. CI Phase: Automated Tests: Once the CI tool picks up the commit, it runs predefined tests.
- Build Docker Image: After successful testing, the code is packaged into a Docker image.
- 3. CD Phase: Deploy to Kubernetes: The Docker image is pulled and deployed to the Kubernetes cluster.
- Alerts: Any failures or issues in the deployment phase result in a notification being sent to the team through channels like Slack or Gmail.
- 4. Automated Deployment:
- Changes are deployed automatically to the Kubernetes cluster without manual intervention once all tests pass.

7. Monitoring and Future Enhancements

7.1 Current Monitoring Setup

- **Prometheus and Grafana Integration:** As part of future enhancements, **Prometheus** will be integrated to collect metrics from the Kubernetes cluster and the running application (e.g., CPU usage, memory consumption, request latency).
- **Grafana Dashboards: Grafana** will be used to visualize the metrics collected by Prometheus, providing a clear overview of the system's health.

- **Alerting:** Custom alerts will be set up based on thresholds, ensuring that the team is notified of any critical issues before they impact the end-users.

7.2 Terraform for Infrastructure as Code (IaC)

- **Infrastructure as Code: Terraform** will be used to define and provision the Kubernetes infrastructure. This will ensure that the cluster setup is repeatable, scalable, and easy to manage.
- **Automated Provisioning:** With Terraform, changes to the infrastructure (e.g., adding more nodes, configuring new services) can be versioned and applied automatically, reducing the risk of manual errors and improving operational efficiency.

---

8. Conclusion

The **Todo List App** employs a modern, robust architecture that leverages containerization, Kubernetes orchestration, and CI/CD pipelines to deliver a highly available, scalable, and easily maintainable application. With ongoing improvements such as monitoring and infrastructure automation, the system is set to become even more resilient and capable of handling future growth. By using industry best practices, this setup ensures fast delivery, minimized downtime, and improved developer productivity.