กรมการขนส่งทางบก
Department of Land Transport

คู่มือสำหรับผู้ดูแลระบบ (Technical Manual)

โครงการพัฒนาระบบเทคโนโลยีสารสนเทศบริหารจัดการขนส่งสินค้าทางถนนเพื่อช่วยสนับสนุนและยกระดับมาตรฐานผู้ประกอบการขนส่งสินค้าทางถนน (DLT-TMS) ของประเทศไทย

คณะวิศวกรรมศาสตร์ มหาวิทยาลัยเกษตรศาสตร์

DLT TMS

สารบัญ

สารบัญ (ต่อ)

สารบัญ (ต่อ)

สารบัญ (ต่อ)

สารบัญ (ต่อ)

สารบัญรูป

สารบัญรูป (ต่อ)

สารบัญตาราง

สารบัญตาราง (ต่อ)

บทนำ

โครงการพัฒนาระบบเทคโนโลยีสารสนเทศบริหารจัดการขนส่งสินค้าทางถนนเพื่อช่วยสนับสนุนและยกระดับมาตรฐานผู้ประกอบการขนส่งสินค้าทางถนนของประเทศไทย (DLT-TMS) เป็นโครงการที่มุ่งเน้นการนำเทคโนโลยีสารสนเทศสมัยใหม่มาใช้ในการบริหารจัดการระบบขนส่งสินค้าทางถนน ให้มีประสิทธิภาพมากยิ่งขึ้น โดยในกระบวนการพัฒนา (Development) และการดำเนินงาน (Operation) ของระบบ ทีมพัฒนาได้เลือกใช้เทคโนโลยีหลักหลายอย่างเพื่อให้ระบบมีความยืดหยุ่น ปลอดภัย และง่ายต่อการดูแลรักษา

เทคโนโลยีที่นำมาใช้มีดังนี้

1. **Kubernetes** คือ ระบบจัดการ container ที่ช่วยจัดสรรทรัพยากรและควบคุมการทำงานของ container ต่างๆ ในระบบให้เป็นไปอย่างอัตโนมัติและมีประสิทธิภาพ ทำให้การปรับขยายระบบ (Scaling) หรือการอัปเดตแอปพลิเคชันทำได้ง่ายและรวดเร็ว
2. **Container และ Image** เป็นเทคโนโลยีที่ช่วยแยกสภาพแวดล้อมการทำงานของแต่ละแอปพลิเคชันออกจากกัน ทำให้สามารถพัฒนา ทดสอบ และนำแอปพลิเคชันขึ้นใช้งานได้อย่างรวดเร็วและปลอดภัย การจัดการ Container ภายในระบบก็จะใช้ Kubernetes เป็นตัวควบคุมหลัก
3. **Microservice** หรือระบบบริการย่อย คือแนวคิดการออกแบบซอฟต์แวร์ให้แต่ละฟังก์ชันหรือบริการภายในระบบแยกออกจากกันอย่างชัดเจน ทำให้สามารถพัฒนาและปรับปรุงแต่ละส่วนได้อย่างอิสระ ลดผลกระทบเวลาเกิดข้อผิดพลาดในบางส่วนของระบบและช่วยให้ระบบมีความทนทาน (Resilient) สูงขึ้น
4. **ArgoCD** เป็นเครื่องมือสำหรับการตรวจสอบและจัดการกระบวนการส่งมอบโค้ด (Continuous Delivery) ไปยังระบบปฏิบัติการจริง ช่วยให้การนำโค้ดเวอร์ชันใหม่ขึ้นระบบเป็นไปอย่างปลอดภัยและอัตโนมัติ ลดความผิดพลาดที่อาจเกิดจากการอัปเดตด้วยมือ

รายงานฉบับนี้จึงถูกจัดทำขึ้นเพื่อเป็นคู่มือสำหรับการใช้งานระบบในแต่ละส่วน พร้อมทั้งอธิบายแนวปฏิบัติที่เหมาะสมในการจัดการข้อมูล รวมถึงการดูแลและบริหารจัดการเครื่องแม่ข่าย (Server) เพื่อให้มั่นใจว่าระบบจะทำงานได้อย่างมีประสิทธิภาพและปลอดภัยสูงสุด เหมาะสำหรับผู้ดูแลระบบและผู้พัฒนาที่ต้องการเข้าใจโครงสร้างและการทำงานของ DLT-TMS อย่างละเอียด

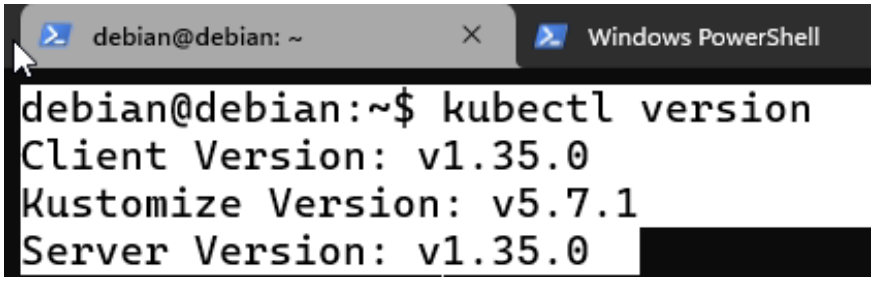
1. Kubernetes

Kubernetes หรือที่มักเรียกย่อว่า K8s คือแพลตฟอร์มโอเพนซอร์สสำหรับการบริหารจัดการ container ที่ได้รับความนิยมอย่างแพร่หลายในระดับสากล จุดเด่นของ Kubernetes คือสามารถจัดสรรทรัพยากรให้กับ container ต่างๆ ได้อย่างมีประสิทธิภาพ ควบคุมการทำงาน การขยายขนาด (Scaling) การอัปเดต รวมถึงการฟื้นฟูบริการเมื่อเกิดปัญหาโดยอัตโนมัติ ช่วยให้การบริหารระบบที่มีหลายแอปพลิเคชันหรือบริการย่อยเป็นไปอย่างง่ายดายและยืดหยุ่น นอกจากนี้ Kubernetes ยังรองรับการทำงานแบบกระจายศูนย์ (Distributed) และสามารถทำงานร่วมกับเครื่องแม่ข่ายหลายเครื่องพร้อมกันได้อีกด้วย จึงเหมาะกับองค์กรที่ต้องการระบบที่มีความเสถียรและขยายตัวได้ตามความต้องการ

1.1. ผลการติดตั้ง Kubernetes บนเครื่องแม่ข่าย

ทางที่ปรึกษาได้ดำเนินการติดตั้งระบบ Kubernetes บนเครื่องแม่ข่ายที่ใช้ระบบปฏิบัติการ Debian 12 รวมถึงติดตั้งและใช้งาน kind (Kubernetes IN Docker) เพื่อสร้างคลัสเตอร์ Kubernetes ในสภาพแวดล้อมการทดสอบอย่างมีประสิทธิภาพ โดยการเลือกใช้ Debian 12 เป็นฐานของเครื่องแม่ข่ายช่วยให้ระบบมีความเสถียรและปลอดภัย ส่วน kind ช่วยให้สามารถจำลองคลัสเตอร์ Kubernetes ได้อย่างรวดเร็วและง่ายดาย

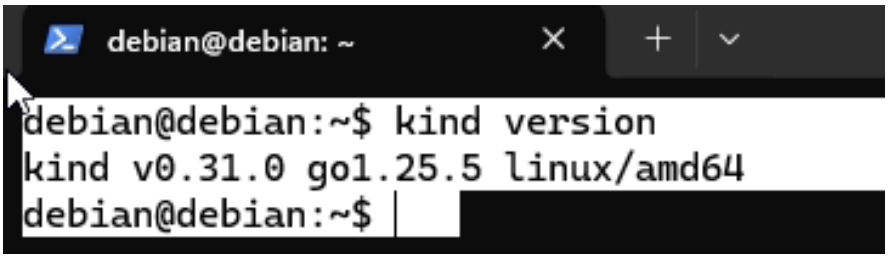
ที่ปรึกษาได้ทำการติดตั้ง Kubernetes และ Kind software ลงบนเครื่องแม่ข่ายของระบบ DLT-TMS



รูปที่ 1-1 ผลการตรวจสอบเวอร์ชัน Kubernetes

ภาพแสดงผลการตรวจสอบเวอร์ชันของ Kubernetes บนเครื่องแม่ข่าย (**รูปที่ 1-1**) โดยใช้คำสั่ง kubectl version ซึ่งแสดงรายละเอียดทั้ง Client Version, Kustomize

Version และ Server Version เพื่อยืนยันว่าระบบ Kubernetes ถูกติดตั้งและพร้อมใช้งานอย่างถูกต้อง



รูปที่ 1-2 ผลการตรวจสอบเวอร์ชัน Kind

ภาพแสดงผลการตรวจสอบเวอร์ชันของโปรแกรม Kind (Kubernetes in Docker) (**รูปที่ 1-2**) ซึ่งเป็นเครื่องมือสำหรับสร้างและจำลอง Kubernetes cluster ภายใน Docker environment โดยใช้คำสั่ง kind version เพื่อแสดงรายละเอียดของเวอร์ชันที่ติดตั้งอยู่ในระบบ พร้อมทั้งข้อมูลสถาปัตยกรรมของระบบปฏิบัติการ (เช่น linux/amd64) และเวอร์ชันของภาษา Go ที่ใช้ในการพัฒนาโปรแกรม ทั้งนี้การตรวจสอบดังกล่าวมีวัตถุประสงค์เพื่อยืนยันว่าเครื่องมือ Kind ได้รับการติดตั้งอย่างถูกต้องและสามารถนำไปใช้ในการสร้างสภาพแวดล้อมสำหรับการทดสอบและพัฒนา Kubernetes cluster ได้อย่างเหมาะสมและมีประสิทธิภาพก่อนนำไปใช้งานร่วมกับระบบจริงในโครงการ DLT-TMS ต่อไป

2. เทคโนโลยี Container และ Image สำหรับระบบปฏิบัติการ (Containerization)

ในสถาปัตยกรรมระบบ DLT-TMS เทคโนโลยี Container ถือเป็นหัวใจสำคัญที่ช่วยให้แอปพลิเคชันสามารถทำงานได้อย่างอิสระ มีความปลอดภัย และสามารถถ่ายโอน (Portability) ไปยังเครื่องแม่ข่ายต่างๆ ได้โดยปราศจากข้อผิดพลาดที่เกิดจากความแตกต่างของสภาพแวดล้อม (Environment configuration)

2.1. แนวคิดและการทำงานของ Container

Container คือการจำลองสภาพแวดล้อมการทำงานของซอฟต์แวร์ที่แยกออกจากระบบปฏิบัติการหลัก (Isolated environment) โดยภายใน Container จะบรรจุแอปพลิเคชัน โค้ดที่จำเป็น ไฟล์คอนฟิกูเรชัน รวมถึงไลบรารีที่ต้องใช้ในการประมวลผลเอาไว้ในหน่วยเดียวกัน การใช้งาน Container ช่วยลดปัญหาความเหลื่อมล้ำระหว่างสภาพแวดล้อมการพัฒนา (Development), การทดสอบ (Staging) และการใช้งานจริง (Production) ส่งผลให้ระบบทำงานได้อย่างเสถียรและสามารถขยายขนาดได้ตามปริมาณงานที่เกิดขึ้นจริง

2.2. การจัดการ Image ในฐานะไฟล์ต้นแบบ

Image คือไฟล์ต้นแบบ (Immutable Template) ที่กำหนดโครงสร้างของ Container โดยภายใน Image จะประกอบไปด้วยไฟล์ระบบปฏิบัติการขนาดเล็ก แอปพลิเคชัน และส่วนประกอบที่จำเป็นทั้งหมดสำหรับการทำงาน Image จะถูกกำหนดเวอร์ชัน (Versioning) เพื่อความแม่นยำในการระบุรุ่นของซอฟต์แวร์ ทำให้มั่นใจได้ว่าทุกครั้งที่มีการนำ Image ไปสร้างเป็น Container ระบบจะคงคุณสมบัติและความสามารถเดิมไว้เสมอโดยไม่มีการเปลี่ยนแปลง

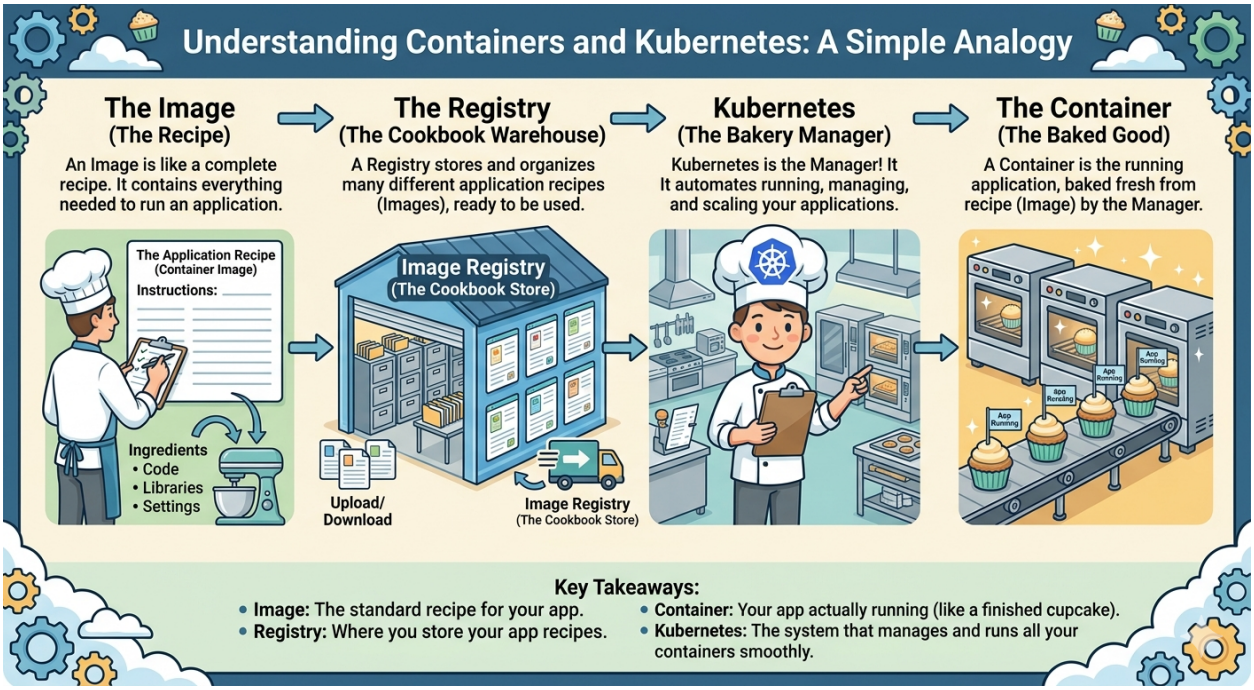
2.3. บทบาทของ Image Registry ในการจัดเก็บและบริหารจัดการ

Image Registry คือระบบจัดเก็บ Image ที่ได้รับการอนุมัติ (Validated artifacts) เพื่อเป็นศูนย์กลางในการเข้าถึงสำหรับคลัสเตอร์ Kubernetes ภายในระบบ DLT-TMS ได้เลือกใช้ GitLab Image Registry ซึ่งมีระบบรักษาความปลอดภัยและการบริหารจัดการสิทธิ์ในการเข้าถึง (Access Control) ทำให้ทีมพัฒนาสามารถบริหารจัดการเวอร์ชันของ

แอปพลิเคชันได้อย่างเป็นระเบียบ และรองรับการดึง Image ไปใช้งานในสภาวะที่มีการขยายตัว (Scale out) ได้อย่างรวดเร็ว

2.4. กลไกการประมวลผลของ Image บน Kubernetes

เมื่อระบบมีการอัปเดตเวอร์ชันซอฟต์แวร์ผ่านกระบวนการ Continuous Delivery (CD) Kubernetes จะทำหน้าที่ตรวจสอบการเปลี่ยนแปลงของ Image ภายใน Registry จากนั้นจึงดึง (Pull) Image เวอร์ชันล่าสุดมาสร้างเป็น Container Instance ใหม่ภายใน Pod โดยอัตโนมัติ กระบวนการนี้ถูกควบคุมผ่านคำสั่งที่ระบุไว้ใน Deployment Manifest ซึ่งช่วยให้สามารถสลับเปลี่ยน (Rolling Update) หรือย้อนกลับ (Rollback) เวอร์ชันของแอปพลิเคชันได้อย่างราบรื่นและลดผลกระทบต่อการให้บริการ



รูปที่ 2-1 ภาพประกอบคำอธิบาย Container, Image, Kubernetes แบบเข้าใจง่าย

รูปที่ 2-1 แสดงให้เห็นความสัมพันธ์ระหว่าง Container, Image และ Kubernetes โดยเริ่มจาก Image ซึ่งเป็นไฟล์ต้นแบบที่บรรจุข้อมูลทุกอย่างที่จำเป็นสำหรับแอปพลิเคชัน เมื่อ Kubernetes รับคำสั่งจะดึง Image จาก Image registry แล้วสร้าง Container ขึ้นมาเพื่อรันบนคลัสเตอร์ การไหลของข้อมูลในภาพจะช่วยให้เข้าใจว่าแต่ละส่วนทำหน้าที่อะไร และเชื่อมโยงกันอย่างไรในกระบวนการทำงานของระบบ

ที่ปรึกษาได้ติดตั้งระบบสร้าง image และนำ Image ลงมาใช้เป็น pod เพื่อสร้างการปฏิบัติการของระบบเต็มรูปแบบโดยจะได้เป็นรายการ pod รูปที่ 2-2

```
debian@debian:~$ kubectl get pod
NAME                                    READY   STATUS             RESTARTS         AGE
cloudflared-55f5d4f69f-rclmd            0/1     CrashLoopBackOff   28418 (2s ago)   61d
dlt-connect-frontend-59bcb5f465-m78pm   1/1     Running            0                2d5h
dlt-cronjob-54d9d4944d-8hqhj            1/1     Running            0                8d
dlt-datapool-7bd5b75cf7-fvkfv           1/1     Running            0                6h10m
dlt-event-builder-54657f8c5f-dxqkg      1/1     Running            0                6h12m
dlt-event-consumer-845b8df4ff-fn2z5     0/1     Error              86 (8m4s ago)    10h
dlt-mongodb-0                           1/1     Running            0                21d
dlt-mongodb-1                           1/1     Running            0                21d
dlt-mongodb-2                           1/1     Running            0                21d
dlt-platform-api-6b5759bf-ph2wf         1/1     Running            0                87m
dlt-platform-api-6b5759bf-sgd9l         1/1     Running            1 (29m ago)      88m
dlt-platform-api-6b5759bf-tshqw         1/1     Running            0                87m
dlt-postgres-0                          1/1     Running            1 (61d ago)      62d
dlt-report-5476589bfc-jgqfk             1/1     Running            0                3h42m
dlt-socket-api-75668fb87-sgg2d          1/1     Running            8 (9h ago)       7d13h
mongodb-init-replicaset-fzq2l           0/1     Completed          0                21d
position-publisher-787957455-s7g27      1/1     Running            0                12h
redis-0                                 1/1     Running            1 (61d ago)      62d
```

รูปที่ 2-2 ข้อมูล Pod บนระบบ DLT-TMS

ภาพแสดงรายการ Pod ซึ่งเป็นหน่วยการทำงานของแอปพลิเคชันภายใน Kubernetes Cluster ของระบบ DLT-TMS โดยแต่ละ Pod แทนบริการย่อย (Microservice) ที่ถูกติดตั้งและดำเนินการอยู่ในระบบ ทั้งนี้ข้อมูลที่ปรากฏประกอบด้วยชื่อ Pod สถานะความพร้อมใช้งาน สถานะการทำงาน จำนวนครั้งในการรีสตาร์ท และระยะเวลาการทำงานของแต่ละ Pod

ภาพดังกล่าวสะท้อนให้เห็นถึงการทำงานของบริการต่าง ๆ ภายในระบบที่มีการประสานการทำงานร่วมกันเป็นระบบเดียว ภายใต้สถาปัตยกรรมแบบ Microservices ซึ่งแต่ละบริการมีหน้าที่เฉพาะและสามารถทำงานร่วมกันได้อย่างมีประสิทธิภาพ โดยรายละเอียดเชิงลึกของการทำงานในแต่ละบริการจะอธิบายเพิ่มเติมในหัวข้อที่ 7 ต่อไป

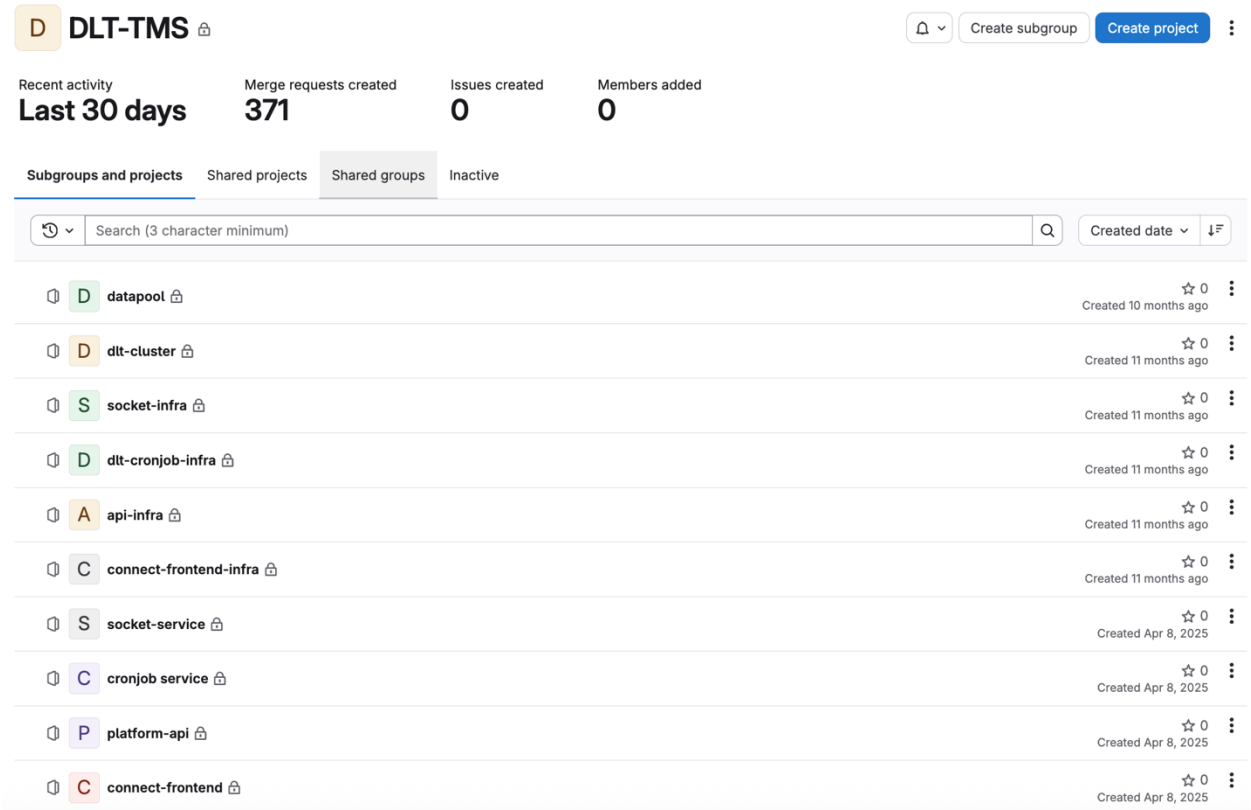
2.5. สถาปัตยกรรมความสัมพันธ์ระหว่างระบบ (Container, Image, และ Kubernetes)

ความสัมพันธ์ระหว่างองค์ประกอบทั้งสามประการนี้ถือเป็นโครงสร้างพื้นฐานที่สำคัญของ DLT-TMS โดยเริ่มจาก Image ที่ทำหน้าที่เป็นพิมพ์เขียว (Blueprint) ของระบบ จากนั้น Kubernetes จะทำหน้าที่เป็นระบบควบคุม (Orchestrator) ในการสั่งการให้ Container ทำงานตามจำนวนที่กำหนดไว้ในแต่ละ Pod เพื่อให้บริการรองรับแก่ผู้ใช้งาน ทั้งนี้ การออกแบบสถาปัตยกรรมแบบนี้ช่วยให้ระบบสามารถรองรับภาระงาน (Load) ที่เปลี่ยนแปลงตลอดเวลาได้อย่างยืดหยุ่นและเป็นระบบ

3. Gitlab repository

GitLab เป็นแพลตฟอร์มที่ใช้สำหรับบริหารจัดการซอร์สโค้ดและการพัฒนาโปรเจกต์แบบครบวงจร โดยมีระบบควบคุมเวอร์ชัน (Version Control) ด้วย Git ทำให้สามารถติดตามและจัดการการเปลี่ยนแปลงของโค้ดได้อย่างมีประสิทธิภาพ นอกจากนี้ GitLab ยังมีฟีเจอร์สำหรับการทำงานร่วมกัน เช่น การสร้างและรับรอง Pull Request, การบริหาร Issue, Pipeline สำหรับการ CI/CD และการจัดการสิทธิ์การเข้าถึงโค้ดในแต่ละโปรเจกต์ ซึ่งช่วยให้ทีมพัฒนาสามารถทำงานร่วมกันอย่างเป็นระบบและปลอดภัย

ทางที่ปรึกษาได้ทำการสร้าง Repository หรือที่เก็บซอร์สโค้ดทั้งหมดที่ Gitlab โดยแต่ละ Repository จะถูกแบ่งเป็น microservice และ config ระบบสำหรับ argocd application configuration



รูปที่ 3-1 ภาพแสดง repository ในโปรเจค DLT-TMS

ภาพแสดงการจัดเก็บโครงสร้างซอร์สโค้ดและข้อมูลคอนฟิกูเรชันของโครงการ DLT-TMS บนแพลตฟอร์ม GitLab โดยมีการแบ่งจัดหมวดหมู่ Repository ออกเป็นส่วนงานย่อย

ตามแนวทาง Microservices Architecture และ Infrastructure-as-Code เพื่อความคล่องตัวในการบริหารจัดการซอร์สโค้ด ดังนี้:

- **Application Repositories (Microservices):** เป็นแหล่งจัดเก็บซอร์สโค้ดสำหรับพัฒนาบริการย่อยต่าง ๆ ของระบบ เช่น platform-api, connect-frontend, socket-service, และ cronjob-service ซึ่งช่วยให้ทีมพัฒนาสามารถปรับปรุงแต่ละส่วนงานได้อย่างเป็นอิสระและมีความปลอดภัยในการควบคุมเวอร์ชัน
- **Infrastructure Repositories (Configuration):** เป็นแหล่งจัดเก็บไฟล์คอนฟิกูเรชันของระบบ Kubernetes (Kubernetes Manifests) เช่น api-infra, socket-infra, connect-frontend-infra, และ dlt-cronjob-infra เพื่อใช้ในกระบวนการทำงานแบบ GitOps ร่วมกับ ArgoCD ในการบริหารจัดการสถานะของคลัสเตอร์โดยอัตโนมัติ

การบริหารจัดการผ่าน Repository ที่แบ่งแยกส่วนประกอบอย่างชัดเจนเช่นนี้ ช่วยให้การกำกับดูแลการเปลี่ยนแปลง (Change Management) เป็นไปอย่างโปร่งใส มีระบบตรวจสอบย้อนหลัง (Audit Trail) ที่แม่นยำ และช่วยให้การทำ Continuous Integration และ Continuous Delivery (CI/CD) ภายในโครงการ DLT-TMS มีประสิทธิภาพสูงสุดตามมาตรฐานการพัฒนาระบบเทคโนโลยีสารสนเทศสมัยใหม่

4. ArgoCD และ Continuous Delivery (GitOps)

Argocd เป็นเครื่องมือที่ช่วยในการจัดการและควบคุมการ Deploy แอปพลิเคชันบน Kubernetes แบบอัตโนมัติ โดยใช้แนวคิด GitOps คือการใช้ Git repository เป็นแหล่งข้อมูลหลักสำหรับสถานะของระบบ เมื่อมีการเปลี่ยนแปลง configuration หรือโค้ดใน Git Argocd จะตรวจจับและทำการ Sync สถานะของคลัสเตอร์ให้ตรงกับที่กำหนดไว้ใน repository โดยอัตโนมัติ ช่วยลดความผิดพลาดจากการทำงานแบบ manual และเพิ่มความโปร่งใสในการบริหารจัดการระบบ

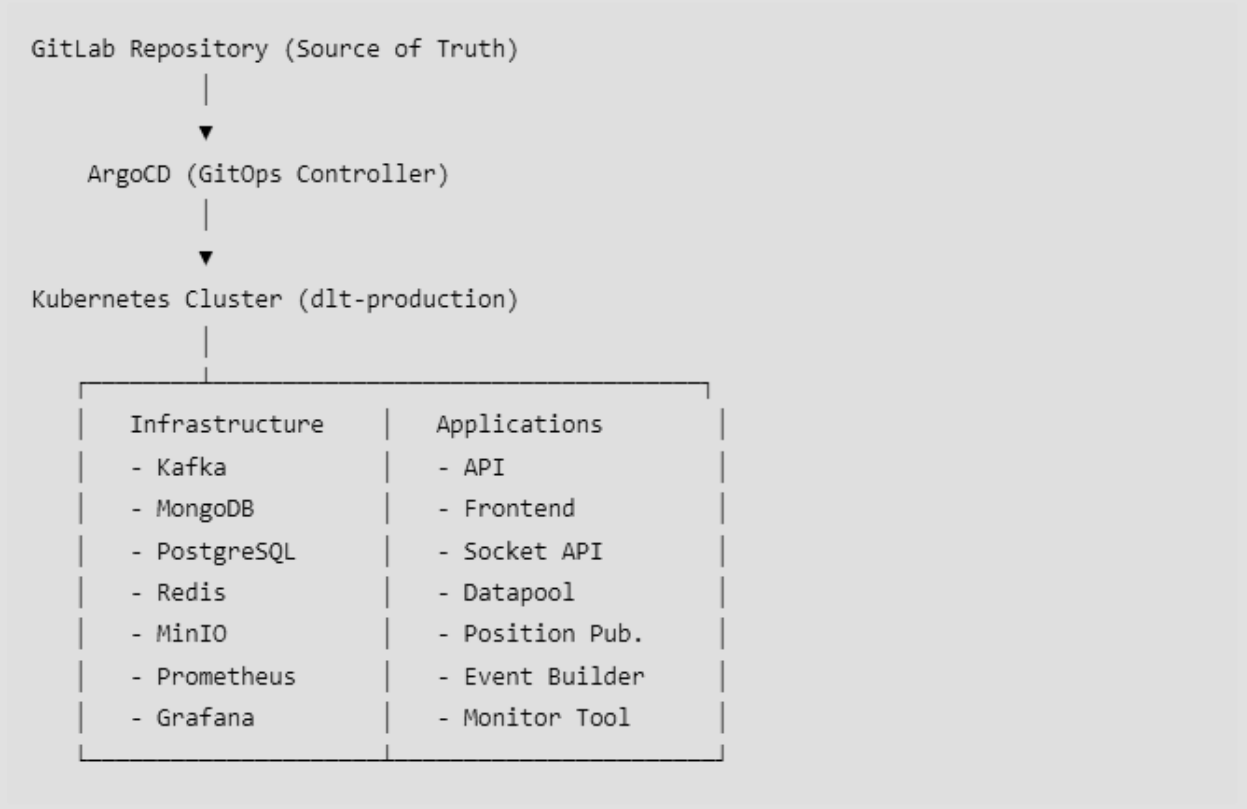
ส่วน Continuous Delivery (CD) คือกระบวนการที่ช่วยให้โค้ดหรือแอปพลิเคชันสามารถส่งต่อไปยัง production ได้อย่างต่อเนื่องและรวดเร็ว โดยอาศัยการทดสอบและ Deploy แบบอัตโนมัติ เมื่อทีมพัฒนา push โค้ดหรือ configuration ใหม่เข้า repository ระบบ CD จะทำงานตั้งแต่การทดสอบจนถึงการ Deploy ไปยังคลัสเตอร์ Kubernetes ทำให้สามารถอัปเดตแอปพลิเคชันได้บ่อยและปลอดภัย ลดความเสี่ยงจากการ Deploy แบบ manual และตอบโจทย์การพัฒนาแบบ agile ได้อย่างมีประสิทธิภาพ

4.1. ภาพรวม

ระบบบริหารจัดการขนส่ง DLT TMS ของกรมการขนส่งทางบกได้นำ ArgoCD มาใช้เป็นเครื่องมือหลักสำหรับการ deploy และดูแลแอปพลิเคชันบน Kubernetes โดยยึดแนวคิด GitOps เป็นศูนย์กลางของกระบวนการจัดการระบบ กล่าวคือ สถานะทั้งหมดของระบบ—including configuration, source code และ manifests ต่าง ๆ—จะถูกจัดเก็บไว้ใน Git repository ซึ่งถือเป็น single source of truth หรือแหล่งข้อมูลหลักที่ทุกฝ่ายยึดตาม ArgoCD จะทำหน้าที่ตรวจสอบและซิงค์สถานะของ cluster ให้สอดคล้องกับข้อมูลใน Git repository แบบอัตโนมัติ ทำให้มั่นใจได้ว่าแอปพลิเคชันใน environment ต่าง ๆ มีความถูกต้องและสอดคล้องกับ version ใน repository อยู่เสมอ

แนวคิด GitOps เป็นการนำหลักการของ version control มาประยุกต์ใช้กับการบริหารระบบ infrastructure และ deployment โดยทุกการเปลี่ยนแปลงจะผ่านการ commit และ review ใน Git ก่อน เมื่อมีการปรับปรุงหรืออัปเดตไฟล์ configuration หรือ source code ArgoCD จะตรวจจับการเปลี่ยนแปลงเหล่านั้นและดำเนินการ deploy หรือ sync ให้ Kubernetes cluster มีสถานะตรงกับที่ระบุไว้ใน repository โดยอัตโนมัติ ข้อดีของแนวทางนี้คือช่วยลดความผิดพลาดจากการ deploy แบบ manual เพิ่มความโปร่งใสในการตรวจสอบ และสามารถย้อนดูประวัติการเปลี่ยนแปลงได้อย่างชัดเจน ส่งผลให้การบริหารจัดการระบบ

มีประสิทธิภาพ ยืดหยุ่น และตอบโจทย์การพัฒนาแบบ agile ได้อย่างสมบูรณ์ โดยทางระบบ DLT-TMS ได้ออกแบบ GitOps ตาม**รูปที่ 4-1**



รูปที่ 4-1 ภาพรวมการทำงาน แนวคิด GitOps

จาก**รูปที่ 4-1** นั้นทางระบบจะมี Gitlab Repository ของ Service ต่างๆ ข้างล่าง เมื่อมีแผนผังนี้แสดงกระบวนการบริหารจัดการซอฟต์แวร์และโครงสร้างพื้นฐานโดยยึดแนวคิด GitOps เป็นหัวใจสำคัญในการปฏิบัติการ (Operational Workflow) ของโครงการ DLT-TMS โดยมีลำดับขั้นตอนดังนี้:

1. **GitLab Repository (Source of Truth):** ทุกการเปลี่ยนแปลงของซอร์สโค้ด และคอนฟิกูเรชันทั้งหมดจะถูกจัดเก็บและผ่านกระบวนการตรวจสอบ (Code Review/Version Control) ณ แหล่งข้อมูลอ้างอิงหลักเพียงแหล่งเดียวคือ GitLab Repository
2. **ArgoCD (GitOps Controller):** ระบบ ArgoCD จะทำหน้าที่เป็นตัวกลางในการตรวจจับการเปลี่ยนแปลงที่เกิดขึ้นใน Repository และดำเนินการซิงโครไนซ์

(Synchronization) สถานะของ Kubernetes Cluster ให้เป็นปัจจุบันและตรงกับความต้องการที่ระบุไว้ใน Git โดยอัตโนมัติ

3. **Kubernetes Cluster (dlt-production):** เป็นหน่วยประมวลผลปลายทางที่รันการทำงานของระบบ โดยแบ่งองค์ประกอบการทำงานออกเป็น 2 ส่วนหลัก ได้แก่
    - **Infrastructure:** ระบบโครงสร้างพื้นฐานที่สนับสนุนการทำงานของแอปพลิเคชัน เช่น ระบบฐานข้อมูล (MongoDB, PostgreSQL), ระบบรับส่งข้อความแบบสตรีม (Kafka), ระบบแคช (Redis), รวมถึงระบบเฝ้าระวังและการแสดงผลข้อมูล (Prometheus, Grafana)
    - **Applications:** ส่วนประกอบของซอฟต์แวร์แอปพลิเคชันที่รองรับการใช้งานระบบ DLT-TMS ได้แก่ ระบบ API, หน้าเว็บ (Frontend), บริการ Real-time (Socket API), ระบบรับข้อมูล GPS (Datapool), และระบบประมวลผลเหตุการณ์ (Event Builder)

กระบวนการนี้ช่วยให้การ Deploy หรืออัปเดตระบบปฏิบัติการและสถาปัตยกรรมต่าง ๆ ภายในโครงการ DLT-TMS เป็นไปอย่างมีมาตรฐาน ปลอดภัย และสามารถตรวจสอบย้อนกลับ (Traceability) ได้ในทุกขั้นตอนการดำเนินงาน

4.2. Repositories ที่ใช้

Repository มีความสำคัญอย่างยิ่งในการบริหารจัดการการตั้งค่าและการ Deploy ระบบผ่าน ArgoCD เพราะ Repository ทำหน้าที่เป็นศูนย์กลางในการจัดเก็บไฟล์ configuration, source code และ manifests ต่าง ๆ ของแต่ละ Service ภายในระบบ ทั้งนี้ในแนวคิด GitOps จะถือว่า Repository คือแหล่งข้อมูลหลัก (single source of truth) ที่ทุกฝ่ายต้องอ้างอิงและใช้ในการควบคุมสถานะของระบบ เมื่อมีการเปลี่ยนแปลงใด ๆ ใน Repository เช่น การแก้ไข configuration หรือโค้ด ArgoCD จะตรวจจับและดำเนินการ Sync หรือ Deploy ไปยัง Kubernetes cluster โดยอัตโนมัติ ทำให้มั่นใจได้ว่าระบบ production หรือ environment ต่าง ๆ มีความถูกต้องและสอดคล้องกับ version ที่ผ่านการตรวจสอบและอนุมัติแล้วใน Repository เพิ่มความโปร่งใส สามารถตรวจสอบย้อนหลัง และลดข้อผิดพลาดจากการดำเนินการด้วยมือ

สำหรับการตั้งค่า ArgoCD application จะต้องมีการกำหนด Repository ที่เกี่ยวข้องกับแต่ละส่วนของระบบ เช่น repository ที่เก็บไฟล์ configuration หลัก, repository สำหรับ Kubernetes manifests ของ API service, Frontend และ WebSocket API เป็นต้น การตั้งค่านี้ช่วยให้ ArgoCD สามารถติดตามและควบคุมการเปลี่ยนแปลงของแต่ละ Service ได้อย่างมีประสิทธิภาพและอัตโนมัติ

ตัวอย่าง Repository ที่ใช้งานในระบบ DLT-TMS สามารถดูรายละเอียดได้ใน ตารางที่ 4-1 ซึ่งสรุปชื่อ Repository และหน้าที่ของแต่ละ Repository ไว้อย่างชัดเจน เช่น **configs** ใช้เก็บ config หลัก, app definition และ ingress rules api-infra ใช้เก็บ Kubernetes manifests สำหรับ API service connect-frontend-infra ใช้เก็บ Kubernetes manifests สำหรับ Frontend socket-infra ใช้เก็บ Kubernetes manifests สำหรับ WebSocket API

การจัดการ Repository อย่างมีระบบและสอดคล้องกับแนวคิด GitOps ผ่าน ArgoCD จึงเป็นหัวใจสำคัญของการพัฒนาและดูแลรักษาระบบ DLT-TMS ให้มีความยืดหยุ่น ปลอดภัย และสามารถตรวจสอบย้อนกลับได้ในทุกขั้นตอน

ตารางที่ 4-1 รายชื่อและการทำของ Repositories ในระบบ DLT-TMS

ชื่อ Repository	หน้าที่
configs	เก็บ config หลัก, apps definition, ingress rules
api-infra	Kubernetes manifests สำหรับ API service
connect-frontend-infra	Kubernetes manifests สำหรับ Frontend
socket-infra	Kubernetes manifests สำหรับ WebSocket API
dlt-datapool-infra	Kubernetes manifests สำหรับ Datapool service
position-publisher-infra	Kubernetes manifests สำหรับ Position Publisher
dlt-eventbuilder-infra	Kubernetes manifests สำหรับ Event Builder

ตารางดังกล่าวสรุปการจัดแบ่งโครงสร้างและหน้าที่ของ Repositories บน GitLab ซึ่งทำหน้าที่เป็นคลังจัดเก็บซอร์สโค้ดและไฟล์คอนฟิกูเรชันของระบบ โดยจำแนกตามประเภทการใช้งานเพื่อให้สอดคล้องกับแนวคิดโครงสร้างพื้นฐานในรูปแบบโค้ด (Infrastructure as Code) ดังนี้

- **Repository configs:** ทำหน้าที่เป็นศูนย์กลางการจัดเก็บคอนฟิกูเรชันหลักของระบบ (Main Configurations), การกำหนดรูปแบบการทำงานของแอปพลิเคชัน (Apps Definition) และกฎการจัดการเส้นทางการเข้าถึง (Ingress Rules) เพื่อควบคุมภาพรวมการทำงานของแพลตฟอร์ม DLT-TMS
- **Infrastructure Repositories (กลุ่ม *-infra):** ได้แก่ api-infra, connect-frontend-infra, socket-infra, dlt-datapool-infra, position-publisher-infra และ dlt-eventbuilder-infra ทำหน้าที่จัดเก็บ Kubernetes Manifests ของบริการย่อย (Microservices) แต่ละส่วน เพื่อใช้เป็นแหล่งอ้างอิงสถานะระบบ

ที่ถูกต้อง (Single Source of Truth) สำหรับให้ ArgoCD นำไปปรับใช้ (Sync) กับสภาพแวดล้อมการทำงานจริง (Kubernetes Cluster) ให้สอดคล้องกันอย่างมีประสิทธิภาพ

การจัดสรร Repository ในลักษณะนี้ช่วยให้การบริหารจัดการการเปลี่ยนแปลง (Change Management) มีความชัดเจน ทีมพัฒนาสามารถตรวจสอบไฟล์คอนฟิกูเรชันเฉพาะส่วนได้โดยไม่ต้องแก้ไขโครงสร้างหลักของระบบ ทั้งยังช่วยลดความเสี่ยงที่เกิดจากการตั้งค่าระบบผิดพลาดอันเนื่องมาจากการปรับเปลี่ยนด้วยวิธีอื่นที่ไม่ผ่านกระบวนการ GitOps

4.3. Kubernetes Cluster (Kind)

Kubernetes cluster คือ กลุ่มของเครื่องคอมพิวเตอร์ (nodes) ที่ทำงานร่วมกันเพื่อรันและบริหารจัดการ containerized applications โดยมี node หลักที่เรียกว่า control-plane ทำหน้าที่ควบคุมและจัดการ cluster ในขณะที่ worker nodes จะทำหน้าที่รันแอปพลิเคชันจริง ระบบนี้ช่วยให้สามารถปรับขนาด บำรุงรักษา และตรวจสอบแอปพลิเคชันได้อย่างมีประสิทธิภาพ
Kind (Kubernetes in Docker) เป็นเครื่องมือที่ช่วยให้สามารถสร้างและจัดการ Kubernetes cluster ได้อย่างรวดเร็วและง่ายดายบนเครื่อง development หรือ server จริง โดยจะรัน cluster อยู่ภายใน Docker container เหมาะสำหรับการทดสอบหรือจำลองสภาพแวดล้อม Kubernetes โดยไม่ต้องติดตั้งบนเครื่องจริงทั้งหมด

4.4. Port Mapping

ตารางที่ 4-2 ตารางแสดงผลการเชื่อมต่อพอร์ตบนเครื่องแม่ข่าย

Port บน Host	Port ใน Container	Protocol	หน้าที่
80	80	TCP	HTTP traffic
443	443	TCP	HTTPS traffic

Port mapping ใน Kubernetes คือกระบวนการกำหนดพอร์ตของเครื่องแม่ข่าย (Host) ให้เชื่อมต่อกับพอร์ตภายในคอนเทนเนอร์ (Container) เพื่อให้บริการที่รันอยู่ภายในคอนเทนเนอร์สามารถรับและส่งข้อมูลผ่านพอร์ตที่กำหนดไว้ได้ ตัวอย่างเช่น หากต้องการให้ผู้ใช้เข้าถึงเว็บแอปพลิเคชันผ่านอินเทอร์เน็ต อาจกำหนดให้พอร์ต 80 และ 443 บนเครื่องแม่ข่ายถูกแมปไปยังพอร์ตเดียวกันภายในคอนเทนเนอร์ ซึ่งเป็นมาตรฐานสำหรับการให้บริการเว็บ

สำหรับศูนย์เทคโนโลยีสารสนเทศและการสื่อสาร (ศทส) ของกรมการขนส่งทางบก ได้กำหนดให้สามารถเข้าถึงเครื่องแม่ข่ายได้เฉพาะพอร์ต 80 และ 443 เท่านั้น เพื่อเพิ่มความปลอดภัยในการเข้าถึงระบบและลดความเสี่ยงจากการโจมตีผ่านพอร์ตอื่น ๆ

- **พอร์ต 80 (HTTP):** ใช้สำหรับรับส่งข้อมูลแบบไม่เข้ารหัสผ่านโปรโตคอล HTTP เหมาะกับเว็บไซต์ทั่วไปที่ไม่ต้องการความปลอดภัยสูง แต่ข้อมูลสามารถถูกดักจับหรือแก้ไขได้
- **พอร์ต 443 (HTTPS):** ใช้สำหรับรับส่งข้อมูลแบบเข้ารหัสผ่านโปรโตคอล HTTPS ให้ความปลอดภัยและความเป็นส่วนตัวของข้อมูล เหมาะสำหรับเว็บไซต์ที่ต้องการความน่าเชื่อถือและปกป้องข้อมูลผู้ใช้ เช่น การทำธุรกรรมออนไลน์หรือการเข้าสู่ระบบ

การเปิดแค่สองพอร์ตนี้ช่วยให้ระบบมีความปลอดภัยมากขึ้น เพราะลดโอกาสที่ผู้ไม่หวังดีจะเข้าถึงบริการหรือข้อมูลสำคัญจากพอร์ตอื่น ๆ ที่ไม่ได้ใช้งาน

4.5. Persistent Storage (Volume Mounts)

การจัดเก็บข้อมูลถาวรหรือ Volume Mounts คือกระบวนการที่ใช้ในการเชื่อมต่อพื้นที่จัดเก็บข้อมูลจากเครื่องแม่ข่าย (Host Path) เข้ากับคอนเทนเนอร์ (Container Path) เพื่อให้ข้อมูลภายในบริการต่าง ๆ เช่น ฐานข้อมูล MongoDB, PostgreSQL หรือ Redis สามารถถูกเก็บรักษาและเข้าถึงได้อย่างต่อเนื่อง แม้คอนเทนเนอร์จะถูกรีสตาร์ทหรือหยุดทำงาน ข้อมูลจะไม่สูญหายและสามารถเรียกใช้งานได้ตลอดเวลา

วิธีการนี้ช่วยให้การจัดการข้อมูลมีความปลอดภัยและเสถียร เหมาะสำหรับระบบที่ต้องการความน่าเชื่อถือในการเก็บข้อมูล เช่น บริการฐานข้อมูลที่ต้องการบันทึกข้อมูลสำคัญไว้ในระยะยาว

ตารางที่ 4-3 pathfile การสำรองข้อมูลของ container ในเครื่องแม่ข่าย

Host Path	Container Path	บริการ
/mnt/data/mongodb	/mongodb	MongoDB database
/mnt/data/postgres	/postgres	PostgreSQL database
/mnt/data/redis	/redis	Redis cache
/mnt/data/kafka0	/kafka0	Kafka broker 0
/mnt/data/kafka1	/kafka1	Kafka broker 1
/mnt/data/kafka2	/kafka2	Kafka broker 2
/mnt/data/minio	/minio	MinIO object storage

ตารางแสดงรายละเอียดการจัดสรรพื้นที่จัดเก็บข้อมูลถาวร (Volume Mounts) เพื่อเชื่อมต่อข้อมูลระหว่างเครื่องแม่ข่าย (Host) และคอนเทนเนอร์ (Container) ภายในระบบ DLT-TMS โดยมีวัตถุประสงค์หลักเพื่อรักษาความต่อเนื่องของข้อมูล (Data Persistence) แม้ในกรณีที่คอนเทนเนอร์จะหยุดทำงานหรือมีการเริ่มการทำงานใหม่ (Restart) ข้อมูลสำคัญที่เกี่ยวข้องกับระบบฐานข้อมูลและบริการรับส่งข้อความจะถูกจัดเก็บไว้บนพื้นที่จัดเก็บของเครื่องแม่ข่ายอย่างปลอดภัย โดยมีรายละเอียดการจับคู่ตำแหน่งไฟล์ (Path Mapping) ดังนี้

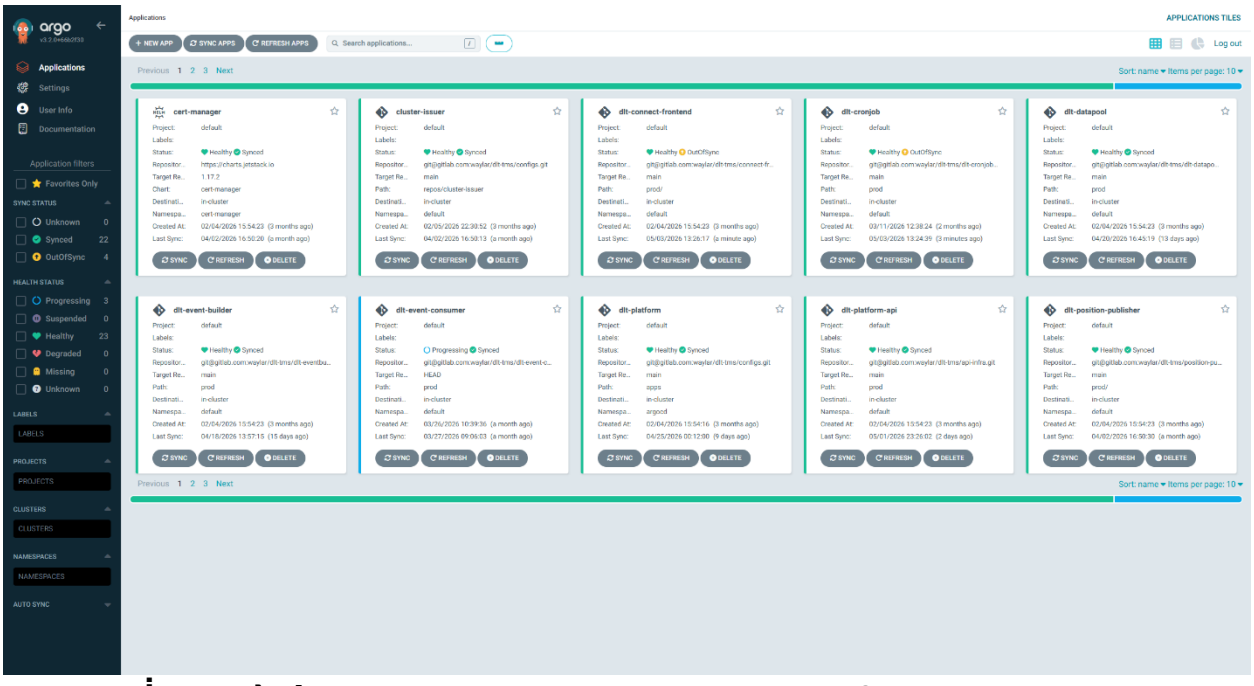
- **ฐานข้อมูลเชิงสัมพันธ์และ NoSQL:** มีการเชื่อมต่อตำแหน่ง /mnt/data/mongodb ไปยัง /mongodb สำหรับ MongoDB และ /mnt/data/postgres ไปยัง /postgres สำหรับ PostgreSQL เพื่อรองรับการจัดเก็บข้อมูลที่มีโครงสร้างและไม่มีโครงสร้างอย่างมีประสิทธิภาพ
- **ระบบแคชและ Message Broker:** มีการจัดสรรพื้นที่ /mnt/data/redis สำหรับ Redis Cache และ /mnt/data/kafka0-2 สำหรับ Kafka Broker ทั้ง 3 โหนด เพื่อรองรับการประมวลผลข้อมูลแบบเรียลไทม์ (Streaming Data) ที่มีความเร็วสูง
- **ระบบจัดเก็บไฟล์:** มีการเชื่อมต่อ /mnt/data/minio ไปยัง /minio เพื่อใช้งานในฐานะ Object Storage สำหรับจัดเก็บไฟล์เอกสารและสื่อต่างๆ ในระบบ

การกำหนดค่าดังกล่าวไม่เพียงแต่ช่วยเพิ่มเสถียรภาพให้กับข้อมูล (Data Reliability) แต่ยังช่วยให้ผู้ดูแลระบบสามารถบริหารจัดการการสำรองข้อมูล (Backup) และการกู้คืนข้อมูล (Recovery) จากเครื่องแม่ข่ายได้โดยตรง ซึ่งถือเป็นแนวปฏิบัติที่ดีในการดูแลรักษาระบบฐานข้อมูลระดับองค์กร

4.6. การเข้าถึง ArgoCD UI

สามารถเข้าถึง Argocd ได้ที่ URI: https://argocd-tms.dlt.go.th/ โดยตัวระบบจะมีการตรวจสอบและป้องกันการเข้าถึง โดยชุดรหัสการเข้าถึงจะจัดทำให้ผู้ที่เกี่ยวข้องสามารถเข้าถึงได้เท่านั้นจะไม่ถูกเขียนไว้ในเอกสารนี้

Argocd จะเป็น Tool ที่ใช้ในการตรวจสอบการทำงานของระบบทั้งหมดบนโครงการ DLT-TMS โดยจะมีหน้าดังรูปที่ 4-2



รูปที่ 4-2 หน้าต่าง Argocd แสดงรายการ application ในโครงการ DLT-TMS

ภาพแสดงอินเทอร์เฟซหลักของ ArgoCD (ArgoCD UI) ซึ่งทำหน้าที่เป็นเครื่องมือบริหารจัดการการดีพลอย (Deployment Management) ของโครงการ DLT-TMS โดยภายในหน้าต่างแสดงผลประกอบด้วยรายละเอียดของแอปพลิเคชันย่อยต่าง ๆ ที่ระบบดูแลอยู่ (Application Tiles) ข้อมูลในแต่ละส่วนประกอบด้วย

- **สถานะความพร้อม (Health Status):** แสดงสถานะความสมบูรณ์ของแต่ละแอปพลิเคชัน (Healthy, Progressing, หรือ OutOfSync) เพื่อให้ผู้ดูแลระบบสามารถตรวจสอบ ความผิดปกติของบริการได้ทันที

- **ข้อมูลแหล่งอ้างอิง (Configuration Source):** แสดงที่อยู่ของ Git Repository และ Path ของ Manifest ไฟล์ที่แอปพลิเคชันนั้น ๆ กำลังใช้งานจริง
- **ฟังก์ชันการจัดการ (Operational Controls):** ระบบได้จัดเตรียมเครื่องมือสำหรับการซิงโครไนซ์สถานะระบบ (Sync), การรีเฟรชข้อมูลล่าสุด (Refresh), และการจัดการแอปพลิเคชัน (Delete) เพื่อความสะดวกในการบริหารจัดการในสถานการณ์จริง

การใช้ ArgoCD UI ช่วยให้การบริหารจัดการระบบที่มีความซับซ้อนและมีจำนวนบริการย่อยหลายส่วนเป็นไปอย่างเป็นระบบ ช่วยเพิ่มประสิทธิภาพในการติดตามสถานะการ Deploy และช่วยลดขั้นตอนการตรวจสอบสถานะด้วยคำสั่งผ่าน Terminal (Command Line) แบบเดิม ทำให้การปฏิบัติการด้านระบบเทคโนโลยีสารสนเทศของโครงการ DLT-TMS มีความโปร่งใสและตรวจสอบสถานะได้อย่างเป็นปัจจุบัน (Real-time Monitoring)

4.7. App of Apps Pattern

ระบบนี้ใช้ **"App of Apps"** pattern ซึ่งเป็น best practice ของ ArgoCD โดยแนวคิดของ App of Apps คือการสร้าง Application หลักที่ดูแลและจัดการการ deploy แอปพลิเคชันย่อยหลายตัวพร้อมกันอย่างอัตโนมัติ ตัว Application หลักจะชี้ไปที่โฟลเดอร์ที่มีไฟล์ config ของแต่ละแอปพลิเคชันย่อย เมื่อสั่ง deploy Application หลัก ArgoCD จะอ่านไฟล์ในโฟลเดอร์นั้นและสร้างแอปพลิเคชันย่อยทั้งหมดให้อัตโนมัติ ทำให้การจัดการและซิงค์แอปพลิเคชันจำนวนมากในระบบง่ายขึ้นและมีความเป็นระบบมากขึ้น เหมาะกับการใช้งานในระบบที่มีบริการหลายส่วนและต้องการการ deploy ที่รวดเร็วและมีประสิทธิภาพ

โดยที่ปรึกษาได้ทำการสร้าง Root application ที่ Repositories ชื่อว่า config โดยจะมีไฟล์ชื่อว่า dlt-platform.yml โดยโครงสร้างของตัวโปรแกรมจะใช้คำสั่ง ข้างล่าง

```
kubectl apply -f dlt-platform.yaml
```

โดยตัวระบบ Argocd จะทำการอ่านไฟล์และทำการดึง yaml ไฟล์อื่นๆออกมาจัดตั้งเป็น application

4.8. App of Apps Pattern ทำงานอย่างไร

App of Apps Pattern คือแนวทางปฏิบัติที่ได้รับความนิยมใน ArgoCD ซึ่งออกแบบมาเพื่อให้สามารถบริหารจัดการและดีพลอยแอปพลิเคชันจำนวนมากได้อย่างมีประสิทธิภาพ โดยแนวคิดหลักคือการสร้าง "Application หลัก" (root application) ขึ้นมาเพียงตัวเดียว

ซึ่งจะทำหน้าที่เป็นตัวกลางในการอ้างอิงไฟล์คอนฟิกของแอปพลิเคชันย่อยๆ ที่จัดเก็บไว้ในโฟลเดอร์ (เช่น apps/) เมื่อสั่ง deploy Application หลักนี้ ArgoCD จะเข้าไปอ่านไฟล์คอนฟิกทั้งหมดในโฟลเดอร์ดังกล่าว แล้วสร้างแอปพลิเคชันย่อย (child applications) ให้โดยอัตโนมัติทุกตัวในระบบ ทำให้ผู้ดูแลสามารถจัดการและซิงโครไนซ์แอปพลิเคชันย่อยหลายตัวได้พร้อมกัน ลดความซับซ้อนและความผิดพลาดที่อาจเกิดขึ้นจากการตั้งค่าทีละแอปพลิเคชัน เหมาะกับระบบที่มีบริการหลายส่วนและต้องการความรวดเร็วในการขยายหรือปรับเปลี่ยนระบบ

ขั้นตอนการขึ้นระบบด้วย dlt-platform.yaml จะเริ่มจากการสร้าง Application หลัก ชื่อ dlt-platform ซึ่งสามารถสร้างได้ด้วยคำสั่งเดียวคือ

kubectl apply -f dlt-platform.yaml

หลังจากรันคำสั่งนี้ ArgoCD จะอ่านค่าในไฟล์ dlt-platform.yaml ซึ่งจะระบุ path ไปยังโฟลเดอร์ apps/ ใน repository ที่กำหนดไว้ จากนั้น ArgoCD จะเข้าไปอ่านไฟล์ yaml ของแต่ละแอปพลิเคชันย่อยภายในโฟลเดอร์นี้ และสร้าง Application ย่อยในระบบให้อัตโนมัติทุกตัวดัง**รูปที่** 4-3 โดยไม่ว่าจะกี่ตัวก็ตาม ช่วยให้การดีพลอยและซิงค์ระบบทั้งแพลตฟอร์มทำได้โดยง่ายและมีความเป็นระบบ ไม่ต้องสร้าง Application ทีละตัว ลดขั้นตอนและความซับซ้อนในการดูแลระบบขนาดใหญ่

```
dlt-platform (Application หลัก)
      │
      │  ชี้ไปที่ folder: apps/
      │
      ├── api.yaml          → สร้าง Application: dlt-platform-api
      ├── frontend.yaml     → สร้าง Application: dlt-connect-frontend
      ├── kafka.yaml        → สร้าง Application: kafka
      ├── postgres.yaml     → สร้าง Application: postgres
      ├── redis.yaml        → สร้าง Application: redis
      ├── mongodb.yaml      → สร้าง Application: mongodb
      ├── minio.yaml        → สร้าง Application: minio
      ├── cert-manager.yaml → สร้าง Application: cert-manager
      ├── ingress.yaml      → สร้าง Application: ingress
      └── ... และอื่น ๆ
```

รูปที่ 4-3 ภาพโครงสร้าง apps in dlt-platoform argocd

แผนผังนี้แสดงลำดับชั้นความสัมพันธ์และการทำงานของ **App of Apps Pattern** ซึ่งเป็นแนวทางปฏิบัติมาตรฐาน (Best Practice) ในการบริหารจัดการแอปพลิเคชันจำนวนมากผ่าน ArgoCD โดยมีกลไกการทำงานดังนี้:

1. **Root Application (dlt-platform):** เป็นแอปพลิเคชันหลักที่ทำหน้าที่เป็นจุดเริ่มต้น (Entry Point) ในการสั่งการ ArgoCD ให้รับทราบถึงโครงสร้างทั้งหมดของระบบ โดย Root Application จะอ้างอิงตำแหน่งไปยังโฟลเดอร์ apps/ ใน Git Repository
2. **Application Discovery:** เมื่อคำสั่งถูกสั่งการ (ผ่าน kubectl apply) ArgoCD จะดำเนินการอ่านไฟล์คอนฟิกูเรชัน (YAML) ทั้งหมดที่อยู่ภายในโฟลเดอร์ apps/ โดยอัตโนมัติ
3. **Child Applications:** ระบบจะแปลงไฟล์คอนฟิกูเรชันย่อยแต่ละไฟล์ให้เป็นแอปพลิเคชันอิสระภายใต้การกำกับดูแลของ ArgoCD เช่น api.yaml จะถูกเปลี่ยนสถานะเป็นแอปพลิเคชัน dlt-platform-api, frontend.yaml เป็น dlt-connect-frontend รวมถึงบริการโครงสร้างพื้นฐานอื่น ๆ เช่น Kafka, PostgreSQL, Redis, MinIO, Cert-manager และ Ingress

กระบวนการนี้ช่วยให้การบริหารจัดการระบบที่มีความซับซ้อนสูง (High Complexity) เป็นไปอย่างเป็นระเบียบและสามารถควบคุมสถานะ (State Management) ของระบบทั้งแพลตฟอร์มได้ด้วยไฟล์คอนฟิกูเรชันเพียงชุดเดียว ช่วยลดความผิดพลาดที่อาจเกิดขึ้นจากการจัดการทีละแอปพลิเคชัน และเพิ่มประสิทธิภาพในการ Deploy ระบบทั้งระบบให้มีความสอดคล้อง (Consistency) อยู่เสมอ

สรุปคือ App of Apps pattern จะช่วยให้การบริหารจัดการแอปจำนวนมากใน ArgoCD เป็นระเบียบ ง่ายต่อการขยายหรือปรับเปลี่ยนระบบ และเมื่อใช้ร่วมกับไฟล์ dlt-platform.yaml ก็สามารถขึ้นระบบและซิงค์แอปพลิเคชันทั้งหมดได้ด้วยคำสั่งเดียว สะดวกและลดความผิดพลาดในการดีพลอยอย่างมาก โดยสามารถดูตัวอย่างโค้ด dlt-platform.yml ได้ที่**รูปที่ 4-4**

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: dlt-platform
  namespace: argocd
spec:
  source:
    repoURL: git@gitlab.com:dlt/dlt-tms/configs.git
    targetRevision: main
    path: apps           # อ่านทุกไฟล์ใน folder apps/
  destination:
    server: https://kubernetes.default.svc
    namespace: argocd
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

รูปที่ 4-4 โค้ดส่วนของ dlt-platform.yaml

ภาพแสดงเนื้อหาภายในไฟล์ dlt-platform.yaml ซึ่งเป็นไฟล์คอนฟิกูเรชันหลักที่กำหนดรูปแบบการดีพลอยแอปพลิเคชันภายใต้สถาปัตยกรรมแบบ **App of Apps** โดยมีส่วนประกอบสำคัญในการทำงานดังนี้:

- **Metadata และ Namespace:** กำหนดชื่อแอปพลิเคชันหลักเป็น dlt-platform ภายใต้ Namespace argocd เพื่อทำหน้าที่เป็นศูนย์กลางการควบคุม
- **Source Configuration:** ระบุแหล่งข้อมูลที่ repoURL ไปยัง git@gitlab.com:dlt/dlt-tms/configs.git และกำหนด Path ไปยังโฟลเดอร์ apps/ เพื่อให้ ArgoCD อ่านและประมวลผลไฟล์คอนฟิกูเรชันย่อย (Child Applications) ทั้งหมดที่อยู่ภายในโฟลเดอร์ดังกล่าวโดยอัตโนมัติ
- **Destination:** ระบุปลายทางคือ Kubernetes Cluster ปัจจุบัน (https://kubernetes.default.svc) เพื่อให้การซิงโครไนซ์สถานะเกิดขึ้นภายในคลัสเตอร์เดียวกัน
- **SyncPolicy (Automated/Self-Heal):** การตั้งค่า automated: true ร่วมกับ prune: true และ selfHeal: true ช่วยให้ระบบ ArgoCD สามารถปรับปรุงสถานะของแอปพลิเคชันย่อยทั้งหมดให้ตรงกับ Git Repository ได้ทันทีแบบเรียลไทม์

และสามารถกู้คืนสถานะการตั้งค่า (Configuration Drift) ให้กลับมาถูกต้องตามที่กำหนดไว้ใน Git ได้โดยอัตโนมัติ

ไฟล์คอนฟิกูเรชันชุดนี้เปรียบเสมือน "คำสั่งแม่บท" ที่ช่วยให้ทีมดูแลระบบสามารถขยายขนาดหรือติดตั้งบริการใหม่ ๆ เข้าสู่โครงการ DLT-TMS ได้อย่างรวดเร็ว โดยอาศัยเพียงการอัปเดตไฟล์ใน Git Repository ระบบก็จะดำเนินการปรับปรุงการดีพลอยให้สอดคล้องกันทั่วทั้งแพลตฟอร์มอย่างแม่นยำ

4.9. ArgoCD Application

รายการ ArgoCD Application ทั้งหมดที่อยู่ใน configs/apps/ แบ่งออกเป็น 2 กลุ่ม คือ

- Application Services เป็นส่วนของระบบที่ปรึกษาจัดทำขึ้น
  หน้าที่: เป็นตัวขับเคลื่อนการทำงานของระบบ เช่น การรับส่งข้อมูล API, การประมวลผลตำแหน่ง GPS, และระบบประมวลผลเหตุการณ์ต่างๆ
  ความสำคัญ: เป็นส่วนที่มี "ความเป็นเจ้าของ" สูงสุด เพราะเป็นผลงานการเขียนโค้ดเพื่อแก้ปัญหาให้กรมการขนส่งทางบกโดยเฉพาะ
- Infrastructure Services เป็นส่วนของระบบที่ที่ปรึกษานำระบบมาใช้
  หน้าที่: เป็นโครงสร้างพื้นฐาน เช่น ฐานข้อมูล (MongoDB, PostgreSQL), ตัวเก็บแคช (Redis), ระบบรับส่งข้อความ (Kafka), และระบบเฝ้าระวัง (Prometheus/Grafana)
  ความสำคัญ: เป็นการการันตีความเสถียรและความปลอดภัย โดยใช้มาตรฐานสากลแทนการเขียนระบบเหล่านี้ขึ้นใหม่

4.9.1. Application Services

Application Services คือกลุ่มบริการที่ **"ทีมที่ปรึกษาพัฒนาขึ้น" (Custom-built Services)** เพื่อตอบสนองตรรกะทางธุรกิจของกรมการขนส่งทางบกโดยเฉพาะ

- **ลักษณะสำคัญ:** เป็นซอฟต์แวร์ประยุกต์ที่ออกแบบมาเพื่อจัดการข้อมูลการขนส่ง ระบบสมาชิก และบริการสนับสนุนผู้ใช้งาน
- **หน้าที่ใน ArgoCD:** ArgoCD ทำหน้าที่ติดตามและ Deploy ซอร์สโค้ดจาก Git Repository ของทีมพัฒนาโดยตรง เพื่อให้มั่นใจว่าฟีเจอร์ใหม่หรือการแก้ไขบั๊กต่างๆ จะถูกส่งมอบไปยัง Production อย่างถูกต้องและปลอดภัย

- **ตัวอย่างบริการ:** เช่น dlt-platform-api (ตัวจัดการคำสั่งระบบ), dlt-connect-frontend (หน้าจอแสดงผลข้อมูลสำหรับผู้ใช้งาน), และ dlt-event-consumer (ระบบประมวลผลข้อมูลเหตุการณ์เฉพาะของขนส่ง)

Application Services จะเป็นส่วนของระบบที่ทางที่ปรึกษาได้จัดทำขึ้นเพื่อสำหรับระบบ DLT-TMS

ตารางที่ 4-4 ส่วนระบบที่มีการพัฒนาบนระบบ DLT-TMS

Application Name	Config Source Repo	หน้าที่
dlt-platform-api	api-infra.git	REST API หลักของระบบ
dlt-connect-frontend	connect-frontend-infra.git	Web Frontend (Next.js)
dlt-socket-api	socket-infra.git	WebSocket Server (Real-time)
dlt-datapool	dlt-datapool-infra.git	รับข้อมูล GPS จากอุปกรณ์
dlt-position-publisher	position-publisher-infra.git	Publish ตำแหน่งยานพาหนะไปยัง Kafka
dlt-event-builder	dlt-eventbuilder-infra.git	สร้าง Event สำหรับแจ้งเตือนและรายงาน
dlt-cronjob	dlt-cronjob-infra.git	CronJob งานประจำของระบบ
dlt-system-cronjob	dlt-cronjob-infra.git	CronJob ระบบ (ใช้ repo เดียวกับ dlt-cronjob)
dlt-report	dlt-report-infra.git	บริการสร้างรายงาน

ตารางนี้สรุปรายการบริการซอฟต์แวร์ประยุกต์ (Application Services) ที่ทีมที่ปรึกษาได้พัฒนาขึ้นเพื่อรองรับการดำเนินงานตามภารกิจของโครงการ DLT-TMS โดยแสดงความสัมพันธ์ระหว่างชื่อแอปพลิเคชัน (Application Name) กับแหล่งเก็บคอนฟิกูเรชัน (Config Source

Repo) ที่ใช้งานบน ArgoCD เพื่อให้เกิดความชัดเจนในการบริหารจัดการซอร์สโค้ดและสถานะการทำงาน ดังนี้

- **โครงสร้างการจัดการ:** ทุกแอปพลิเคชันถูกกำหนดค่าผ่าน Git Repository เฉพาะตัว (*-infra.git) ซึ่งบรรจุ Kubernetes Manifests ที่ระบุพารามิเตอร์การทำงานไว้อย่างละเอียด
- **ความหลากหลายของบริการ:** รายการในตารางครอบคลุมตั้งแต่บริการสนับสนุนการแสดงผล (Frontend), บริการจัดการข้อมูลสื่อสารหลัก (REST API, WebSocket), ไปจนถึงบริการสนับสนุนภารกิจเฉพาะด้าน เช่น ระบบรับข้อมูลตำแหน่ง GPS (Datapool, Position Publisher), ระบบสร้างเหตุการณ์ (Event Builder), และระบบงานตามกำหนดเวลา (CronJob)
- **การเชื่อมโยงระบบ:** การแยก Repository ของแต่ละบริการช่วยให้ทีมพัฒนาสามารถบริหารจัดการวงจรชีวิตของซอฟต์แวร์ (Software Development Life Cycle) ได้อย่างอิสระ สามารถปรับปรุงแก้ไขบริการใดบริการหนึ่งได้โดยไม่ส่งผลกระทบต่อบริการอื่น ช่วยเพิ่มความยืดหยุ่นและความปลอดภัยในการปฏิบัติการในสภาวะการใช้งานจริง

การกำหนดโครงสร้างตามตารางนี้ถือเป็นกลไกสำคัญที่ช่วยให้ ArgoCD สามารถควบคุมและซิงโครไนซ์สถานะของแต่ละบริการใน Kubernetes Cluster ให้มีความถูกต้องและพร้อมใช้งานอยู่เสมอตามสถานะล่าสุดที่ระบุไว้ใน GitLab

4.9.2. Infrastructure Services

Infrastructure Services คือกลุ่มบริการที่ **"ทีมที่ปรึกษานำระบบมาตรฐานมาติดตั้งและปรับแต่ง" (Integration & Configuration)** เพื่อใช้เป็นฐานราก (Foundation) ในการรัน Application Services ข้างต้นให้ทำงานได้อย่างมีประสิทธิภาพและเสถียร

- **ลักษณะสำคัญ:** เป็นซอฟต์แวร์มาตรฐานระดับอุตสาหกรรม (Industry Standard Tools) ที่นำมาติดตั้งลงใน Kubernetes Cluster โดยที่ปรึกษาทำหน้าที่กำหนดค่าพารามิเตอร์ (Tuning) ให้เหมาะสมกับภาระงานของโครงการ
- **หน้าที่ใน ArgoCD:** ArgoCD ทำหน้าที่จัดการและซิงโครไนซ์ไฟล์คอนฟิกูเรชัน (Manifests) ของเครื่องมือเหล่านี้ เพื่อให้โครงสร้างพื้นฐานมีสถานะที่พร้อมใช้งานตลอดเวลาและเป็นไปตามมาตรฐานความปลอดภัยที่กำหนด
- **ตัวอย่างบริการ:** เช่น cert-manager (สำหรับจัดการ SSL/TLS), ingress-nginx (สำหรับจัดการ Traffic), และระบบฐานข้อมูล/แคชต่างๆ อย่าง mongodb, redis, kafka ที่ถูกติดตั้งในรูปแบบของ Pod ภายในคลัสเตอร์

ส่วน Infrastructure Services จะเป็นส่วนของ Application ที่มีผู้พัฒนาไว้แล้วและผ่านการใช้งานในองค์กรณ์ระดับโลกทางที่ปรึกษาได้นำ Application ต่างๆ เหล่านี้มาใช้งานตามการออกแบบสถาปัตยกรรมระบบ

ตารางที่ 4-5 ส่วนของระบบโครงสร้างพื้นฐานของระบบ DLT-TMS

ไฟล์ใน apps/	Application Name	หน้าที่ของระบบ
mongodb.yaml	mongodb	ระบบฐานข้อมูล NoSQL สำหรับจัดเก็บข้อมูลแบบเอกสาร
postgres.yaml	postgres	ระบบฐานข้อมูลเชิงสัมพันธ์ (Relational Database) สำหรับจัดเก็บข้อมูลโครงสร้าง
redis.yaml	redis	ระบบแคชข้อมูล (In-memory Cache) และระบบ Message Broker สำหรับการสื่อสารระหว่างบริการ
kafka.yaml	kafka	ระบบรับส่งข้อมูลแบบสตรีม (Distributed Streaming Platform) สำหรับประมวลผลข้อมูลขนาดใหญ่
minio.yaml	minio	ระบบจัดเก็บไฟล์แบบ Object Storage ที่เข้ากันกับ S3
prometheus.yaml	prometheus	ระบบมอนิเตอร์และเก็บข้อมูลเมตริกต่างๆ ของคลัสเตอร์
grafana.yaml	grafana	ระบบแสดงแดชบอร์ดและวิเคราะห์ข้อมูลเมตริก
loki.yaml	loki	ระบบจัดเก็บและค้นหาข้อมูล Log แบบกระจายศูนย์
promtail.yaml	promtail	ระบบเก็บรวบรวมและส่งต่อ Log ไปยัง Loki
cert-manager.yaml	cert-manager	ระบบบริหารจัดการใบรับรองดิจิทัล (Certificate) อัตโนมัติ
cluster-issuer.yaml	cluster-issuer	กำหนดผู้ออกใบรับรองระดับคลัสเตอร์ (Let's Encrypt, Self-signed)
ingress.yaml	ingress	กำหนดเส้นทางเข้าใช้งาน (Ingress Rules) สำหรับ Production

ingress-uat.yaml	ingress-uat	กำหนดเส้นทางเข้าใช้งาน (Ingress Rules) สำหรับ UAT
ingress-controller.yaml	ingress-controller-rollout	ควบคุมและจัดการ NGINX Ingress Controller ในคลัสเตอร์
kube-state-metric.yaml	kube-state-metrics	บริการรวบรวมและส่งข้อมูลเมตริกของอ็อบเจ็กต์ใน Kubernetes
gitlab-secret.yaml	—	Kubernetes Secret สำหรับเข้าใช้งาน GitLab Registry (ไม่ใช่ของ ArgoCD App)

ตารางนี้แสดงรายการบริการสนับสนุนและเครื่องมือโครงสร้างพื้นฐาน (Infrastructure Services) ที่ถูกติดตั้งลงใน Kubernetes Cluster เพื่อทำหน้าที่เป็นระบบสนับสนุนการทำงาน (Foundation) ให้กับแอปพลิเคชันหลัก โดยทุกบริการจะถูกกำหนดค่าผ่านไฟล์ในโฟลเดอร์ apps/ เพื่อให้ ArgoCD บริหารจัดการสถานะได้โดยอัตโนมัติ โดยจำแนกหน้าที่ของระบบได้ดังนี้:

- **ระบบจัดการข้อมูลและแคช:** ประกอบด้วยระบบฐานข้อมูลมาตรฐาน ได้แก่ mongodb (NoSQL), postgres (Relational Database) และ redis (In-memory Cache/Message Broker) ซึ่งเป็นรากฐานสำคัญในการจัดเก็บและส่งผ่านข้อมูลระหว่างบริการ
- **ระบบประมวลผลและจัดเก็บข้อมูลขนาดใหญ่:** การใช้งาน kafka สำหรับระบบสตรีมมิ่งข้อมูล (Distributed Streaming) และ minio สำหรับทำหน้าที่เป็น Object Storage ที่รองรับโปรโตคอล S3 ทำให้ระบบสามารถรองรับข้อมูลที่มีปริมาณมหาศาลได้อย่างมีประสิทธิภาพ
- **ระบบเฝ้าระวังและการวิเคราะห์ข้อมูล (Observability Stack):** เป็นกลุ่มเครื่องมือที่ใช้ติดตามสถานะสุขภาพของระบบและวิเคราะห์ปัญหา ได้แก่ prometheus (เก็บข้อมูลเมตริก), grafana (แดชบอร์ดวิเคราะห์), loki (ระบบจัดเก็บ Log) และ promtail (ตัวส่งต่อ Log) ซึ่งเป็นส่วนสำคัญในการบำรุงรักษาระบบให้มีเสถียรภาพ (Uptime) สูง
- **ระบบความปลอดภัยและการจัดการเส้นทาง (Security & Networking):** ได้แก่ cert-manager และ cluster-issuer สำหรับจัดการใบรับรอง SSL/TLS อัตโนมัติ และกลุ่มบริการ ingress สำหรับควบคุมช่องทางการเข้าถึงระบบทั้งในสภาพแวดล้อม Production และ UAT เพื่อความปลอดภัยระดับสูงสุด
- **เครื่องมือสนับสนุนการจัดการคลัสเตอร์:** เช่น kube-state-metrics สำหรับรวบรวมข้อมูลสถานะภายในคลัสเตอร์ และการจัดการ gitlab-secret เพื่อการเชื่อมต่อกับแหล่งซอร์สโค้ดอย่างปลอดภัย

การจัดการ Infrastructure ผ่านโครงสร้าง GitOps ใน ArgoCD ตามรายการข้างต้น ช่วยให้ทีมผู้ดูแลระบบสามารถรักษามาตรฐานการตั้งค่า (Configuration Consistency) และลดระยะเวลาในการติดตั้งระบบสนับสนุนใหม่ได้อย่างรวดเร็ว

4.10. การ Route Traffic และ Ingress Rule

การกำหนดเส้นทางการรับส่งข้อมูล (Route Traffic) หรือการกำหนดกฎ Ingress (Ingress rule) บน NGINX Ingress Controller คือกลไกสำคัญที่ใช้ควบคุมว่าคำขอจากภายนอกจะถูกส่งต่อเข้ามายังบริการต่าง ๆ ภายในคลัสเตอร์ Kubernetes อย่างไร โดย Ingress จะทำหน้าที่เป็นตัวกลาง ตรวจสอบคำขอ (Request) ที่ได้รับ และจับคู่กับ Hostname หรือ Path ตามที่กำหนดไว้ในแต่ละกฎ เมื่อพบว่าตรงตามเงื่อนไข จะทำการส่งต่อคำขอนั้นไปยัง Service ที่เกี่ยวข้อง ส่งผลให้สามารถบริหารจัดการการเข้าถึงระบบ Application ได้อย่างมีประสิทธิภาพและปลอดภัยยิ่งขึ้น

จากตัวอย่างในตารางด้านล่าง จะเห็นว่ามีการกำหนด Ingress Name ว่า **tms-ingress** ซึ่งระบุ Hostname เป็น **tms.dlt.go.th** โดยจะรับคำขอและส่งต่อไปยัง Service ชื่อ **tms-connect-frontend** ที่ Port 3000 พร้อมระบุ TLS Secret สำหรับการเข้ารหัสข้อมูล (dlt-tls-cert) และกำหนดมาตรการด้านความปลอดภัยเพิ่มเติม เช่น การเปิดใช้งาน ModSecurity และ CSP Header ทั้งนี้ โครงสร้างดังกล่าวช่วยให้แต่ละ Application ถูกจัดการเส้นทางการเข้าถึงได้อย่างเหมาะสมและรองรับความต้องการขององค์กร สามารถดูรายการ Hostname ที่ทำการ Ingress กับ Service บนเครื่องแม่ข่าย DLT-TMS ได้ที่**ตารางที่ 4-6**

ตารางที่ 4-6 Hostname และการ Ingress service เข้ากับ Kubernetes application

Ingress Name	Hostname	Service	Port	TLS Secret
tms-ingress	tms.dlt.go.th	tms-connect-frontend	3000	dlt-tls-cert
tms-socket-ingress	socket-tms.dlt.go.th	dlt-socket-svc	4004	dlt-socket-tls-cert
datapool-ingress	data-pool-tms.dlt.go.th	dlt-datapool-api-svc	7788	dlt-cert-tls
dlt-grafana-ingress	grafana.dlt.go.th	grafana	80	dlt-cert-tls

tms-minio-ingress	minio-tms.dlt.go.th	minio-svc	9000	dlt-minio-cert
prometheus-debug	debug.waylar.net	prometheus	9090	— (ไม่มี TLS)

ตารางดังกล่าวแสดงรายละเอียดการตั้งค่า **Ingress Rules** ซึ่งเป็นกลไกสำคัญในการบริหารจัดการการเข้าถึงแอปพลิเคชันจากภายนอกสู่ภายใน Kubernetes Cluster ของโครงการ DLT-TMS โดยมีองค์ประกอบหลักในการทำงานดังนี้

- **การจัดการเส้นทาง (Traffic Routing):** แต่ละ Ingress Name จะถูกจับคู่กับ Hostname (Domain Name) ที่กำหนดไว้ เพื่อให้ Ingress Controller สามารถแยกแยะและส่งต่อคำขอ (Request) ไปยัง Service ปลายทางที่ถูกต้องได้อย่างแม่นยำ
- **การระบุพอร์ตสื่อสาร (Port Management):** ตารางระบุพอร์ตปลายทางที่แต่ละบริการเปิดใช้งาน ทำให้การสื่อสารข้อมูลของแต่ละแอปพลิเคชัน (เช่น REST API, WebSocket, หรือแดชบอร์ด Grafana) ดำเนินไปอย่างเป็นสัดส่วน
- **การรักษาความปลอดภัย (SSL/TLS Security):** มีการกำหนด TLS Secret ให้กับเกือบทุกบริการ ซึ่งเป็นการบังคับใช้มาตรฐานการเข้ารหัสข้อมูลระหว่างการรับส่ง (Encryption in Transit) เพื่อความปลอดภัยของข้อมูลตามมาตรฐานภาครัฐ โดยระบบจะเรียกใช้ใบรับรองดิจิทัลที่ระบุไว้ในการยืนยันตัวตนของแต่ละ Hostname

การจัดการ Ingress ในลักษณะนี้ ช่วยให้โครงการ DLT-TMS มีระบบเครือข่ายที่ยืดหยุ่น สามารถขยายบริการเพิ่มได้ง่ายผ่านการกำหนดค่าเพียงจุดเดียว (Single Entry Point) โดยที่ยังคงรักษาความปลอดภัยและความเป็นส่วนตัวของข้อมูลตามข้อกำหนดของกรมการขนส่งทางบกไว้อย่างครบถ้วน

4.11.การจัดการความมั่นคงปลอดภัยด้วย SSL/TLS

การประยุกต์ใช้ ModSecurity บน NGINX Ingress Controller การเพิ่มชั้นความปลอดภัยให้แก่ระบบด้วย ModSecurity เป็นส่วนหนึ่งของกลยุทธ์การรักษาความมั่นคงปลอดภัย (Security Hardening) ของโครงการ โดย ModSecurity จะทำหน้าที่เป็นเกราะป้องกัน (WAF) ในการตรวจสอบการสื่อสารผ่านโปรโตคอล HTTP/HTTPS ทั้งขาเข้าและขาออก โดยมีความสามารถเด่นดังนี้:

1. **การตรวจจับภัยคุกคามแบบเชิงรุก:** ระบบสามารถวิเคราะห์รูปแบบคำขอ (Request) ที่ผิดปกติเพื่อป้องกันการโจมตีในรูปแบบ SQL Injection และ Cross-site Scripting (XSS) ได้อย่างรวดเร็ว
2. **มาตรฐานการป้องกันตามแนวทาง OWASP:** การนำ OWASP Core Rule Set (CRS) มาประยุกต์ใช้ ช่วยให้ระบบมีเกณฑ์การตัดสินใจด้านความปลอดภัยที่เป็นมาตรฐานสากล สามารถตรวจจับและปิดกั้นการเข้าถึงที่มุ่งประสงค์ร้ายต่อข้อมูลหรือการพยายามเจาะระบบได้อย่างครอบคลุม
3. **ความพร้อมต่อการปฏิบัติงาน:** การตั้งค่าทั้งหมดถูกรวมอยู่ในคอนฟิกูเรชันของ Ingress Controller ช่วยให้การบริหารจัดการด้านความปลอดภัยเป็นไปอย่างเป็นระบบ (Infrastructure as Code) และสามารถรองรับการขยายตัวของระบบโดยคงไว้ซึ่งระดับความปลอดภัยที่เท่าเทียมกันทุกจุด

4.12. การป้องกันภัยคุกคามด้วย Web Application Firewall (WAF)

ในสถาปัตยกรรมระบบ DLT-TMS การนำ **Web Application Firewall (WAF)** มาใช้งานถือเป็นการเพิ่มชั้นความปลอดภัยที่เข้มงวดที่สุดในการกรองการสื่อสารระหว่างเครือข่ายภายนอกและเซิร์ฟเวอร์หลัก โดย WAF ทำหน้าที่เสมือน "พนักงานรักษาความปลอดภัยที่ตรวจค้นสัมภาระ" ก่อนให้บุคคลเข้าสู่พื้นที่อาคาร

หลักการทำงานและความสำคัญ:

- **การคัดกรองข้อมูลระดับแอปพลิเคชัน (Layer 7 Filtering):** WAF ไม่ได้ดูเพียงแค่เลขที่อยู่ไอพี (IP Address) เหมือน Firewall ทั่วไป แต่จะทำการ "แกะ" และ "อ่าน" เนื้อหาในแพ็กเก็ตข้อมูล HTTP/HTTPS เพื่อตรวจสอบว่ามีคำสั่งหรือรูปแบบข้อมูลที่มุ่งร้ายแฝงมาหรือไม่
- **การป้องกันช่องโหว่มาตรฐาน (OWASP Top 10):** ระบบสามารถตรวจจับและสกัดกั้นการโจมตีที่พบบ่อยได้โดยอัตโนมัติ อาทิ:

- **SQL Injection (SQLi):** ป้องกันการส่งคำสั่งเข้าไปทำลายหรือดึงข้อมูลจากฐานข้อมูล
- **Cross-Site Scripting (XSS):** ป้องกันการส่งสคริปต์อันตรายเข้าไปฝังในหน้าเว็บเพื่อขโมยข้อมูลผู้ใช้งาน

- **ความยืดหยุ่นในการปรับแต่ง:** การใช้งาน WAF ร่วมกับระบบ Ingress Controller ช่วยให้สามารถตั้งค่ากฎการป้องกัน (Ruleset) เฉพาะเจาะจงสำหรับแต่ละบริการย่อย (Microservices) ได้ ทำให้ระบบมีความปลอดภัยที่สมดุลกับประสิทธิภาพการทำงาน

```
annotations:
  nginx.ingress.kubernetes.io/enable-modsecurity: "true"
  nginx.ingress.kubernetes.io/enable-owasp-core-rules: "true"
```

รูปที่ 4-5 โค้ดในส่วนการเพิ่มการป้องกันสำหรับ เพื่อรองรับการทดสอบการเจาะระบบมาตรฐาน OWASP บน Nginx Ingress

ดัง**รูปที่ 4-5** ที่ปรึกษาได้ทำการเปิดการป้องกันระบบ และแก้ไขหลังจากมีการทดสอบระบบบนมาตราฐาน OWASP ทั้งนี้ทั้งนั้นจะมีบางช่องทางที่ทางที่ปรึกษาเห็นควรว่าจะไม่มีการเปิดการป้องกันเนื่องจากส่งผลต่อระบบและการใช้งานของระบบเนื่องจากมีการเรียกใช้ API อื่นๆ ทั้งนี้ทั้งนั้นทางที่ปรึกษามีการระบุ Scope API ที่ใช้ไว้ในตัวระบบแล้วซึ่งจะอยู่ในรายงานการประเมินและแก้ไขช่องโหว่

4.13. การเฝ้าระวังและการวิเคราะห์สถานะระบบ (Observability: Monitoring & Logging)

เพื่อให้การปฏิบัติการระบบ DLT-TMS มีความต่อเนื่องและสามารถรับมือกับปัญหาที่อาจเกิดขึ้นได้อย่างทันท่วงที ทีมที่ปรึกษาได้ออกแบบสถาปัตยกรรมด้านการเฝ้าระวังและวิเคราะห์สถานะระบบ (Observability Stack) โดยใช้เครื่องมือมาตรฐานสากลเพื่อสร้างระบบที่สามารถตรวจสอบได้ทุกสถานะ (Full-stack Monitoring) ดังนี้:

4.13.1. ระบบการจัดเก็บข้อมูลตัวชี้วัด (Prometheus Monitoring)

Prometheus ทำหน้าที่เป็นระบบรวบรวมข้อมูลเชิงตัวเลข (Metrics) แบบอนุกรมเวลา (Time Series) โดยมีรายละเอียดทางเทคนิคดังนี้:

- **กลไกการทำงาน:** ใช้วิธีการ Scrape ข้อมูลผ่าน HTTP ตามช่วงเวลาที่กำหนด (Scrape Interval) จากเป้าหมาย (Targets) เช่น Pod, Node และ Service ต่างๆ
- **การตั้งค่า (Configuration):** กำหนดแหล่งข้อมูลผ่าน ConfigMap เพื่อดึงข้อมูลจาก Kubernetes API Server, Node (ผ่าน cAdvisor) และ Pod ที่ระบุ annotation prometheus.io/scrape: "true"
- **การบริหารจัดการ:** ติดตั้งในรูปแบบ Deployment ภายใต้ Namespace monitoring โดยจำกัดการเข้าถึงภายใน Cluster ด้วย Service ประเภท ClusterIP พร้อมทั้งกำหนด Resource Limits ตามขนาดความเหมาะสมของคลัสเตอร์เพื่อป้องกันการใช้ทรัพยากรเกินจำเป็น

4.13.2. ระบบการจัดการข้อมูลบันทึกส่วนกลาง (Loki Centralized Logging)

Loki ทำหน้าที่เป็นระบบรวมศูนย์ข้อมูล Log (Log Aggregation) ที่ออกแบบมาเพื่อรองรับข้อมูลปริมาณมาก (High Scalability):

- **ประสิทธิภาพ:** รองรับการบีบอัดข้อมูล Log เพื่อประหยัดพื้นที่จัดเก็บ และใช้ภาษา **LogQL** ในการสืบค้นข้อมูลที่รวดเร็วและยืดหยุ่น
- **การบริหารจัดการ:** ติดตั้งในรูปแบบ Single Binary เพื่อความสะดวกในการดูแลรักษา โดยมีการกำหนด Resource Limits (CPU: 100m-1000m, Memory: 128Mi-1256Mi) และกำหนด Retention Period ตามความเหมาะสมของนโยบายการจัดเก็บข้อมูล

4.13.3. ระบบตัวแทนรวบรวมข้อมูลบันทึก (Promtail Agent)

Promtail ทำหน้าที่เป็น Log Agent ที่ติดตั้งในรูปแบบ DaemonSet บนทุก Node ภายในคลัสเตอร์:

- **การทำงาน:** ตรวจสอบและดึงข้อมูล Log จากทุก Pod แบบเรียลไทม์ผ่าน /var/log/pods
- **การเชื่อมต่อ:** ส่งต่อข้อมูลบันทึกไปยัง Loki ผ่าน Push URL (http://loki:3100/loki/api/v1/push) โดยใช้ค่าคอนฟิกูเรชันมาตรฐานผ่าน Helm Chart เพื่อให้ง่ายต่อการบริหารจัดการและอัปเดตเวอร์ชัน

4.13.4. การบูรณาการข้อมูลผ่านแดชบอร์ด (Grafana Visualization)

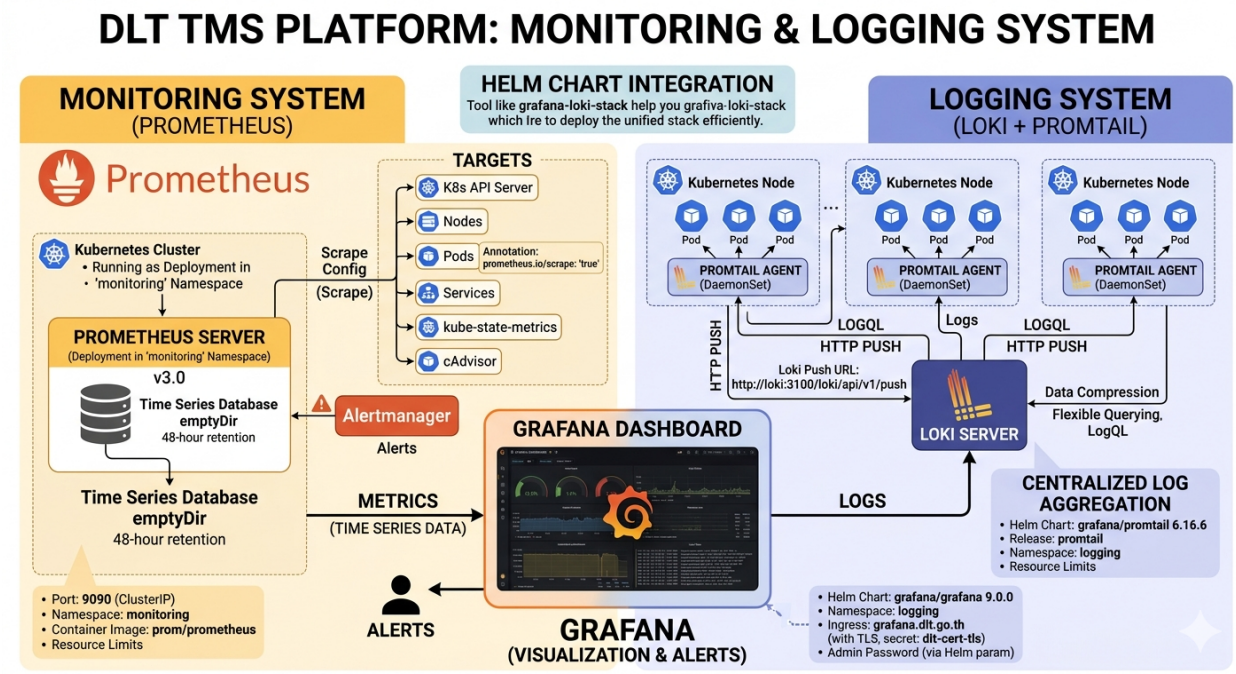
Grafana ทำหน้าที่เป็นศูนย์กลางการแสดงผลข้อมูลจากทั้ง Prometheus และ Loki:

- **การแสดงผล:** สร้าง Dashboards เพื่อวิเคราะห์สถานะสุขภาพของระบบและประสิทธิภาพของ Microservices
- **ความปลอดภัย:** รองรับการเชื่อมต่อผ่าน HTTPS (TLS) ผ่าน Ingress grafana.dlt.go.th โดยใช้ใบรับรองดิจิทัล dlt-cert-tls เพื่อความปลอดภัยในการเข้าใช้งาน
- **การแจ้งเตือน:** กำหนดค่าการแจ้งเตือน (Alerting) เพื่อรายงานเหตุการณ์ผิดปกติผ่านช่องทางสื่อสารที่กำหนด ช่วยให้ผู้ดูแลระบบแก้ไขปัญหาได้แบบเชิงรุก

4.13.5. การบริหารจัดการระบบด้วย Helm Chart (Configuration Management)

เพื่อให้การจัดการ Observability Stack เป็นไปอย่างเป็นระบบ ทีมที่ปรึกษาได้นำการบริหารจัดการแบบ **Infrastructure as Code (IaC)** ผ่าน Helm Chart มาใช้:

- **ความเป็นมาตรฐาน:** ใช้ Chart มาตรฐาน (เช่น grafana-loki-stack หรือแยก Chart รายส่วน) เพื่อกำหนดเวอร์ชัน Namespace และ Resource Limits ให้เป็นมาตรฐานเดียวกันทั้งระบบ
- **ข้อดีของการจัดการ:** การใช้ Helm ช่วยให้การติดตั้ง อัปเดต และย้ายระบบ (Migration) สามารถทำได้อย่างรวดเร็ว แม่นยำ และลดความผิดพลาดในการตั้งค่า (Human Error) ที่อาจเกิดขึ้นในกรณีการติดตั้งด้วยตนเอง



รูปที่ 4-6 รูปแผนผังระบบ monitoring loggin บนระบบ DLT-TMS

ภาพแผนผังนี้แสดงกระบวนการทำงานแบบอัตโนมัติของ **Cert-manager** ซึ่งทำหน้าที่เป็นศูนย์กลางในการบริหารจัดการใบรับรองดิจิทัล (SSL/TLS Certificate) ภายใน Kubernetes Cluster ของโครงการ DLT-TMS โดยมีขั้นตอนหลักตามมาตรฐานโปรโตคอล ACME ดังนี้:

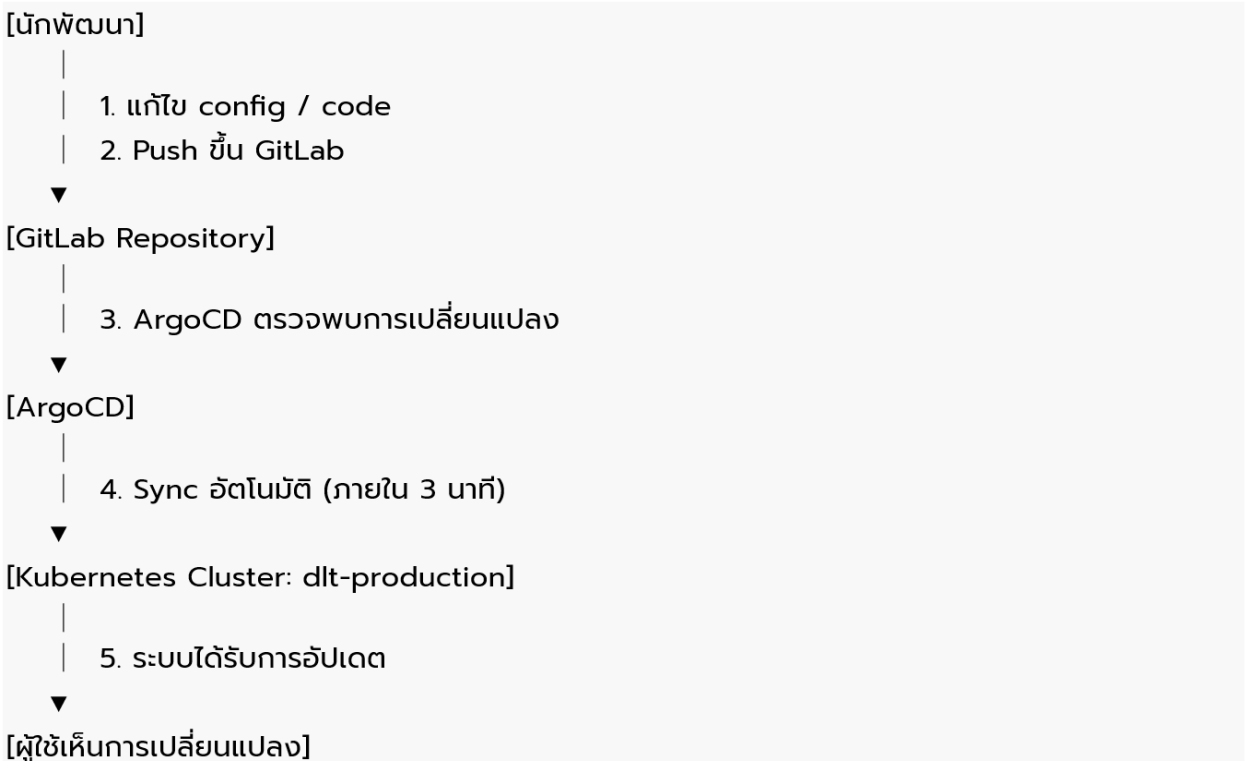
1. **การร้องขอใบรับรอง (Certificate Request):** เมื่อมีการประกาศใช้ (Deploy) Ingress Resource หรือ Certificate Resource ในระบบ Cert-manager จะตรวจพบความต้องการใช้งานใบรับรองดิจิทัลสำหรับ Hostname นั้นๆ โดยอัตโนมัติ
2. **การพิสูจน์ความเป็นเจ้าของ (ACME Challenge):** Cert-manager จะประสานงานกับผู้ให้บริการออกใบรับรอง (เช่น Let's Encrypt) ผ่านโปรโตคอล ACME เพื่อทำกระบวนการตรวจสอบ (Challenge) ว่าระบบที่ร้องขอเป็นเจ้าของโดเมนเนมนั้นจริง
3. **การออกและติดตั้งใบรับรอง (Issuance & Secret Injection):** เมื่อผ่านการตรวจสอบแล้ว ผู้ให้บริการจะออกใบรับรองดิจิทัลและส่งกลับมายัง Cert-manager ซึ่งจะนำใบรับรองไปจัดเก็บในรูปแบบ **Kubernetes Secret**
4. **การนำไปใช้งาน (Integration with Ingress):** ใบรับรองที่จัดเก็บใน Secret จะถูกเรียกใช้งานโดย NGINX Ingress Controller เพื่อสร้างการเชื่อมต่อที่ปลอดภัยผ่านโปรโตคอล HTTPS (TLS) สำหรับบริการต่างๆ ในโครงการ

5. **การต่ออายุอัตโนมัติ (Automated Renewal):** เมื่อใบรับรองใกล้ครบกำหนดอายุ Cert-manager จะเข้าสู่กระบวนการร้องขอใบรับรองใหม่โดยอัตโนมัติ (Renew) ช่วยลดความเสี่ยงที่ระบบจะหยุดชะงักจากการลืมต่ออายุใบรับรอง

5. คู่มือการทำงานของ Argocd

ระบบ **DLT TMS** ใช้แนวคิด **GitOps** — ทุกการเปลี่ยนแปลงในระบบต้องเริ่มจาก Git เสมอ ห้ามแก้ไข cluster โดยตรง

ในระบบ **DLT TMS** มีการนำแนวทาง **GitOps** มาใช้เป็นหลักในการจัดการและควบคุมการเปลี่ยนแปลงต่าง ๆ ภายในระบบ โดยแนวคิด GitOps คือ การกำหนดให้ทุกการปรับปรุงแก้ไข หรืออัปเดตทรัพยากรในระบบ ไม่ว่าจะเป็นการตั้งค่าคอนฟิกหรือการปรับปรุงโค้ด จะต้องดำเนินการผ่านระบบควบคุมเวอร์ชันอย่าง Git เท่านั้น กล่าวคือ ทุกอย่างต้องแก้ไขที่ Repository ของ Git ก่อน แล้วจึงให้ระบบอัตโนมัติ เช่น ArgoCD ทำหน้าที่นำการเปลี่ยนแปลงนั้นไปประยุกต์ใช้กับ Cluster จริง ห้ามเข้าไปแก้ไขหรือปรับแต่ง Kubernetes Cluster โดยตรงเด็ดขาด เพื่อป้องกันความผิดพลาด ลดความสับสน และทำให้สามารถติดตามย้อนกลับประวัติการเปลี่ยนแปลงทั้งหมดได้อย่างโปร่งใสและตรวจสอบได้ เป็นการสร้างมาตรฐานและความปลอดภัยในการดูแลโครงสร้างพื้นฐานของระบบแบบมืออาชีพ โดยกระบวนการแก้ไขจะเป็นไปตาม**รูปที่ 5-1**



รูปที่ 5-1 แผนผังการอัพเดท Application ตาม GitOps

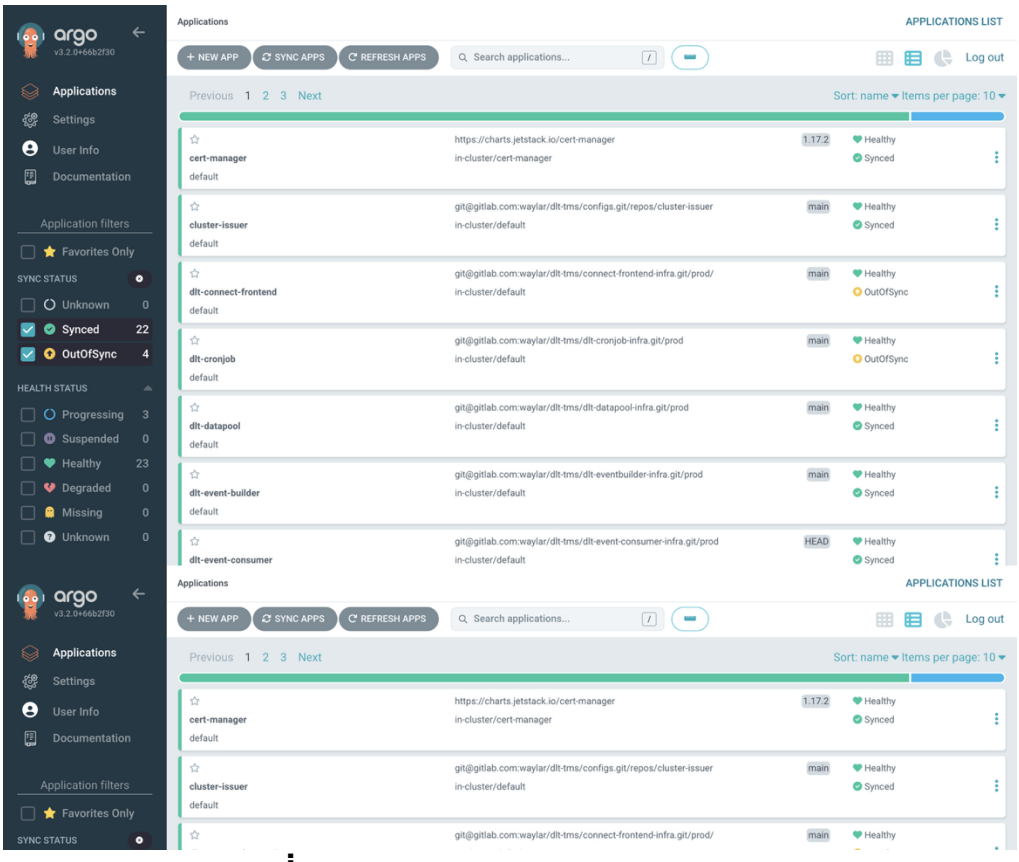
แผนผังนี้แสดงถึงการไหลของข้อมูล (Workflow) ในการปรับปรุงซอฟต์แวร์ โดยมีลำดับขั้นตอนที่เป็นมาตรฐานดังนี้

1. **การแก้ไขสถานะผ่าน Git (Development & Versioning):** นักพัฒนาเริ่มต้นการทำงานโดยการแก้ไขไฟล์กำหนดค่า (Config) หรือซอร์สโค้ดในพื้นที่ทำงานส่วนตัว เมื่อเสร็จสิ้นจะดำเนินการ **Push** ข้อมูลเหล่านั้นขึ้นสู่ **GitLab Repository** ซึ่งทำหน้าที่เป็น "แหล่งความจริงเพียงหนึ่งเดียว" (Single Source of Truth) ของระบบทั้งหมด
2. **การเฝ้าสังเกตการณ์ (Continuous Observation): ArgoCD** จะทำหน้าที่เป็น Controller ที่คอยเฝ้าสังเกตการณ์ (Watch) การเปลี่ยนแปลงใน GitLab Repository อย่างต่อเนื่อง หากพบว่ามีความแตกต่างระหว่างสถานะใน Git กับสถานะที่เกิดขึ้นจริงใน Cluster ระบบจะทำการแจ้งเตือนหรือเตรียมการ Sync โดยอัตโนมัติ
3. **การประยุกต์ใช้สถานะโดยอัตโนมัติ (Automated Reconciliation/Sync):** เมื่อ ArgoCD ตรวจพบความแตกต่าง ระบบจะทำการสั่งการไปยัง **Kubernetes Cluster** เพื่อปรับเปลี่ยนสถานะ (Sync) ให้ตรงกับที่กำหนดไว้ใน Git โดยใช้ระยะเวลาดำเนินการภายใน 3 นาที (โดยประมาณ) ซึ่งลดความเสี่ยงจากการที่ผู้ดูแลระบบต้องเข้าไปจัดการผ่าน Command Line ด้วยตนเอง
4. **การตรวจสอบความสำเร็จ (State Verification):** เมื่อกระบวนการ Sync เสร็จสิ้น ระบบ Kubernetes จะอัปเดตสถานะของ Container/Pod ใหม่ทั้งหมด ทำให้ระบบให้บริการด้วยเวอร์ชันล่าสุดที่ผ่านการทดสอบและรับรองจาก Git แล้ว ซึ่งส่งผลให้ผู้ใช้ปลายทางได้รับฟีเจอร์หรือการแก้ไขปัญหาที่อัปเดตล่าสุด

5.1. การเข้าใช้งาน ArgoCD Interface

ผู้ดูแลระบบสามารถเข้าถึงหน้าจอบริหารจัดการ ArgoCD ได้ผ่านช่องทางที่กำหนด:

1. เปิดโปรแกรมเว็บเบราว์เซอร์ และเข้าสู่ URL: https://argocd-tms.dlt.go.th
2. ทำการยืนยันตัวตน (Authentication) ด้วย Username และ Password ที่ได้รับอนุญาตตามนโยบายรักษาความมั่นคงปลอดภัยขององค์กร



รูปที่ 5-2 ภาพ UI Argocd Production

ภาพหน้าจอโปรแกรม ArgoCD ที่ปรากฏในรูปที่ 5-2 เป็นเครื่องมือหลักสำหรับผู้ดูแลระบบ (System Administrator) ในการกำกับดูแลสถานะของระบบแบบ **Centralized Dashboard** โดยมีองค์ประกอบการทำงานที่สำคัญที่สื่อออกมาจากภาพ ดังนี้:

- **รายการแอปพลิเคชัน (Application List):** บริเวณกลางภาพแสดงรายชื่อแอปพลิเคชันทั้งหมดที่อยู่ภายใต้การจัดการของ ArgoCD เช่น cert-manager, cluster-issuer, dlt-connect-frontend เป็นต้น ซึ่งแสดงให้เห็นว่าระบบบริหารจัดการทุกอย่าง (ทั้ง Infrastructure และ Application) ผ่าน GitOps อย่างเป็นระบบ
- **สถานะการซิงโครไนซ์ (Sync Status):** คอลัมน์สถานะแสดงให้เห็นว่าแอปพลิเคชันอยู่ในสภาวะใด เช่น **Synced** (สถานะใน Cluster ตรงกับใน Git) หรือ **OutOfSync** (มีสิ่งที่ต้องอัปเดตหรือมีการเปลี่ยนแปลงเกิดขึ้น) ซึ่งเป็นตัวบ่งชี้ว่าระบบมีการจัดการการเปลี่ยนแปลงอย่างโปร่งใส
- **สถานะสุขภาพของระบบ (Health Status):** แสดงสถานะ **Healthy** สำหรับแอปพลิเคชันที่ทำงานได้ตามปกติ ช่วยให้ผู้ดูแลระบบตรวจสอบสุขภาพของบริการต่างๆ ได้ทันทีจากหน้าจอเดียวโดยไม่ต้องเข้า Command Line

- **การเชื่อมต่อกับแหล่งต้นทาง (Git Source):** ในแต่ละรายการแอปพลิเคชัน จะระบุที่มาของข้อมูล (Repository Path) เช่น git@gitlab.com:waylar/dlt-tms/... ซึ่งยืนยันว่าระบบมีการอ้างอิงสถานะจาก Git ตามแนวทาง GitOps ที่กำหนดไว้

5.2. การบริหารจัดการข้อมูลความปลอดภัย (Credential Management)

คำสั่ง kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d เป็นวิธีการบริหารจัดการเพื่อเรียกคืนข้อมูลรหัสผ่านเริ่มต้นสำหรับบัญชีผู้ดูแลระบบ (Admin) ในระบบ ArgoCD โดยมีรายละเอียดการทำงานดังนี้

- **กลไกการเรียกข้อมูล:** คำสั่งดังกล่าวเป็นการติดต่อกับ Kubernetes API เพื่อดึงข้อมูลความลับ (Secret) ที่ระบุชื่อ argocd-initial-admin-secret ภายในเนมสเปซ argocd โดยใช้เครื่องมือ jsonpath เพื่อเจาะจงเฉพาะข้อมูลรหัสผ่านที่จัดเก็บอยู่ในรูปแบบฐานเลข 64 (Base64-encoded)
- **การถอดรหัสและการนำไปใช้:** ข้อมูลที่ดึงได้จะถูกส่งผ่านไปสู่กระบวนการถอดรหัส (Base64 Decoding) เพื่อเปลี่ยนสถานะข้อมูลให้กลับมาอยู่ในรูปแบบข้อความปกติ (Plain Text) ซึ่งผู้ดูแลระบบสามารถนำไปใช้ในการยืนยันตัวตนเพื่อเข้าสู่ระบบครั้งแรกได้อย่างรวดเร็ว
- **นัยสำคัญทางปฏิบัติ:** วิธีการดังกล่าวช่วยเพิ่มประสิทธิภาพในการปฏิบัติงานโดยลดขั้นตอนที่ซับซ้อนในการเข้าถึงไฟล์กำหนดค่า โดยเฉพาะในกรณีการตั้งค่าระบบใหม่หรือการทดสอบระบบ

ข้อควรระวังด้านความมั่นคงปลอดภัย: เนื่องจากรหัสผ่านดังกล่าวเป็นข้อมูลสำคัญของระบบ ผู้ปฏิบัติงานควรใช้งานคำสั่งนี้ภายใต้สภาพแวดล้อมที่ปลอดภัยและจำกัดสิทธิ์การเข้าถึงเครื่องแม่ข่ายตามนโยบายความมั่นคงปลอดภัยสารสนเทศของโครงการอย่างเคร่งครัด

```
kubectl -n argocd get secret argocd-initial-admin-secret \
  -o jsonpath="{.data.password}" | base64 -d
```

รูปที่ 5-3 คำสั่งในการดูรหัสผ่านเข้าระบบ Argocd UI

5.3. การตรวจสอบสถานะระบบ (Observability)

ก่อนที่จะสามารถใช้คำสั่ง argocd app list เพื่อตรวจสอบสถานะของแอปพลิเคชันในระบบ Argocd ได้นั้น ผู้ดูแลระบบจำเป็นต้องเข้าเครื่องแม่ข่าย (server) ที่ติดตั้ง Argocd อยู่ก่อน โดยทั่วไปจะใช้โปรแกรม SSH เช่น ssh เพื่อเชื่อมต่อไปยังเครื่องแม่ข่ายผ่าน command line ตัวอย่างเช่น ssh user@hostname เมื่อเข้าสู่ระบบแล้วจึงสามารถรันคำสั่งที่เกี่ยวข้องกับ Argocd ได้อย่างถูกต้องและปลอดภัย

การเข้าเครื่องแม่ข่ายถือเป็นขั้นตอนสำคัญเพื่อความปลอดภัย เพราะช่วยให้มั่นใจว่า. การดำเนินการต่างๆ ถูกควบคุมโดยผู้มีสิทธิ์ที่ได้รับอนุญาตเท่านั้น หลังจากเข้าสู่เครื่องแม่ข่ายแล้วจึงสามารถใช้ argocd app list เพื่อแสดงรายการแอปพลิเคชัน พร้อมสถานะและสุขภาพของแต่ละแอปในระบบ Argocd ได้อย่างมีประสิทธิภาพ

โดยเมื่อเข้าถึงเครื่องแม่ข่ายสามาถใช้คำสั่งดัง**รูปที่ 5-4** ได้ และจะได้ผลลัพธ์ดัง**ตารางที่ 5-1**

```
argocd app list
```

รูปที่ 5-4 คำสั่งตรวจสอบ Argocd app บนเครื่องแม่ข่าย

ตารางที่ 5-1 ผลลัพธ์การค้นหา Argocd app list

STATUS	HEALTH	ความหมาย	ต้องทำอะไร
Synced	Healthy	ปกติ ทุกอย่าง OK	ไม่ต้องทำอะไร
OutOfSync	Healthy	มีการเปลี่ยนแปลงใน Git รอ sync	รอ (~3 นาที) หรือ sync มือ
Synced	Degraded	Deploy แล้วแต่ app มีปัญหา	ดู logs ของ app นั้น
OutOfSync	Degraded	มีปัญหาและยังไม่ได้ sync	ตรวจสอบด่วน
Unknown	Unknown	ArgoCD ไม่สามารถตรวจสอบได้	ตรวจสอบ ArgoCD เอง

ตารางดังกล่าวสรุปเกณฑ์การวินิจฉัยสถานะของแอปพลิเคชันผ่านคำสั่ง argocd app list โดยแบ่งออกเป็นสองมิติหลัก คือ **Sync Status** (สถานะการซิงโครไนซ์กับ Git) และ **Health Status** (สถานะความพร้อมใช้งานของแอปพลิเคชัน) เพื่อใช้เป็นแนวทางในการตรวจสอบและแก้ไขปัญหา (Troubleshooting) ดังนี้

- **กลุ่มสถานะปกติ (Synced & Healthy):** เป็นสถานะที่ระบบทำงานถูกต้องสมบูรณ์ (Desired State) ข้อมูลในคลัสเตอร์ตรงกับที่กำหนดไว้ใน Git Repository โดยไม่ต้องมีการดำเนินการใดๆ เพิ่มเติม
- **กลุ่มสถานะรอการดำเนินการ (OutOfSync & Healthy):** บ่งชี้ว่ามีการปรับปรุงแก้ไขไฟล์คอนฟิกูเรชันใน Git แล้ว แต่ระบบอยู่ในระหว่างกระบวนการรอซิงค์อัตโนมัติ (Reconciliation Loop) หรือผู้ดูแลสามารถสั่ง "Sync มือ" เพื่อปรับสถานะให้เป็นปัจจุบันทันที
- **กลุ่มสถานะต้องเฝ้าระวัง (Degraded):**
    - **Synced & Degraded:** แสดงว่าระบบได้รับไฟล์คอนฟิกูเรชันล่าสุดแล้ว แต่ตัวแอปพลิเคชันภายใน (เช่น Pod) ไม่สามารถรันได้ตามปกติ จำเป็นต้องตรวจสอบ **Application Logs** เพื่อหาสาเหตุของข้อผิดพลาด
    - **OutOfSync & Degraded:** เป็นสถานะวิกฤตที่สุด บ่งชี้ว่าระบบเกิดความผิดปกติในระดับการ Deploy หรือไฟล์คอนฟิกูเรชันมีปัญหา จำเป็นต้อง **ตรวจสอบด่วน (Urgent Troubleshooting)** เพื่อกู้คืนสถานะการทำงาน
- **กลุ่มสถานะไม่ทราบแน่ชัด (Unknown):** เกิดจากกรณีที่ ArgoCD ไม่สามารถสื่อสารกับคลัสเตอร์ได้ หรือระบบจัดการขาดการเชื่อมต่อ ทำให้ไม่สามารถประเมินสถานะใดๆ ได้ จำเป็นต้องตรวจสอบการทำงานของตัวระบบ ArgoCD เอง

5.4. การตรวจสอบสถานะการทำงานของ Pods

คำสั่ง kubectl get pods -n default ใช้สำหรับตรวจสอบสถานะการทำงาน (Runtime Status) ของหน่วยประมวลผลขนาดเล็กที่สุดภายใน Kubernetes คลัสเตอร์ (Pods) ที่ติดตั้งอยู่ในเนมสเปซ default โดยระบบจะแสดงข้อมูลที่สำคัญดังนี้

- **READY:** แสดงจำนวน Container ที่ทำงานได้ตามเงื่อนไข (Ready State) เทียบกับจำนวน Container ทั้งหมดภายใน Pod
- **STATUS:** สถานะปัจจุบันของ Pod เช่น Running (พร้อมปฏิบัติงาน), Pending (กำลังรอจัดสรรทรัพยากร), หรือ CrashLoopBackOff (เกิดความผิดพลาดและพยายามรีสตาร์ทตัวเอง)
- **RESTARTS:** จำนวนครั้งที่ Pod หรือ Container ภายในเกิดความผิดพลาดและต้องเริ่มต้นการทำงานใหม่ ซึ่งเป็นตัวบ่งชี้สำคัญถึงความเสถียรของแอปพลิเคชัน
- **AGE:** ระยะเวลาสะสมที่ Pod เริ่มทำงานบนคลัสเตอร์

```
kubectl get pods -n default

NAME                          READY   STATUS    RESARTS   AGE
dlt-platform-api-xxxxx          1/1     Running   0         2d
tms-connect-frontend-xxxxx      1/1     Running   0         2d
```

รูปที่ 5-5 คำสั่งดูสถานะ Pod

ภาพนี้แสดงผลลัพธ์จากการใช้คำสั่ง kubectl get pods -n default เพื่อตรวจสอบสถานะการทำงาน (Runtime Status) ของแอปพลิเคชันที่ติดตั้งอยู่ภายใน Kubernetes Cluster ในเนมสเปซ default โดยมีองค์ประกอบของข้อมูลที่สำคัญดังนี้:

- **NAME:** แสดงชื่อเฉพาะของ Pod ซึ่งเป็นหน่วยประมวลผลพื้นฐาน เช่น dlt-platform-api-xxxxx (บริการส่วนหลังบ้าน) และ tms-connect-frontend-xxxxx (บริการหน้าบ้าน) โดยส่วนท้ายของชื่อ (xxxxx) คือ Hash ของ Deployment ที่ระบบสร้างขึ้นอัตโนมัติ
- **READY:** แสดงความพร้อมของคอนเทนเนอร์ภายใน Pod (Container Readiness) ตัวอย่างเช่น 1/1 หมายถึงภายใน Pod มี 1 คอนเทนเนอร์ และพร้อมให้บริการแล้วทั้ง 1 คอนเทนเนอร์
- **STATUS:** สถานะปัจจุบันของ Pod ซึ่งสถานะ Running ยืนยันว่าแอปพลิเคชันกำลังปฏิบัติงานตามปกติและได้รับจัดสรรทรัพยากรจากคลัสเตอร์เรียบร้อยแล้ว
- **RESTARTS:** จำนวนครั้งที่คอนเทนเนอร์มีการเริ่มต้นทำงานใหม่ (Restart) ค่าเป็น 0 ในภาพถือเป็นสถานะที่เหมาะสมที่สุด (Optimal) บ่งบอกถึงความเสถียรของแอปพลิเคชันที่ไม่เคยเกิดการหยุดชะงัก
- **AGE:** ระยะเวลาสะสมที่ Pod เริ่มทำงานบนคลัสเตอร์ ซึ่งในภาพคือ 2d (2 วัน) สะท้อนถึงเสถียรภาพและความต่อเนื่องในการให้บริการ (High Availability)

5.5. การอัปเดตระบบและการใช้ Image Versioning

การอัปเดตระบบดำเนินการผ่านกลไก **Declarative Deployment** โดยผู้ดูแลระบบจะทำการแก้ไขไฟล์ kustomization.yaml เพื่อระบุ newTag ของ Docker Image เวอร์ชันใหม่ เมื่อมีการ Push code ขึ้นสู่ Repository ระบบ ArgoCD จะดำเนินการปรับสถานะของ Cluster ให้สอดคล้องกับค่าที่ระบุไว้ใหม่ (Rolling Update) โดยอัตโนมัติ

```
kubectl get pods -n default | grep -E "mongo|postgres|redis"
kubectl get pods -n kafka
```

รูปที่ 5-6 การใช้คำสั่ง grep -E (Extended Regular Expression)

คำอธิบายภาพ

คำอธิบาย: คำสั่งนี้ใช้ grep -E (Extended Regular Expression) เพื่อดึงเฉพาะ Pods ที่เกี่ยวข้องกับระบบฐานข้อมูลออกมาจากรายการ Pods ทั้งหมด ช่วยให้ผู้ดูแลระบบตรวจสอบความพร้อมของฐานข้อมูลได้อย่างรวดเร็วโดยไม่ต้องดูรายการ Pods จำนวนมาก

คำอธิบาย: เนื่องจากระบบ Kafka มักมีการทำงานแบบคลัสเตอร์ที่ซับซ้อน การตรวจสอบเฉพาะเนมสเปซ kafka จึงช่วยให้เห็นสถานะของ Broker ทุกตัวได้ชัดเจนว่าอยู่ในสถานะ Running พร้อมให้บริการการรับส่งข้อมูลหรือไม่

5.6. การตรวจสอบข้อมูลบันทึกการทำงานของ Pod (Application Log Analysis via ArgoCD UI)

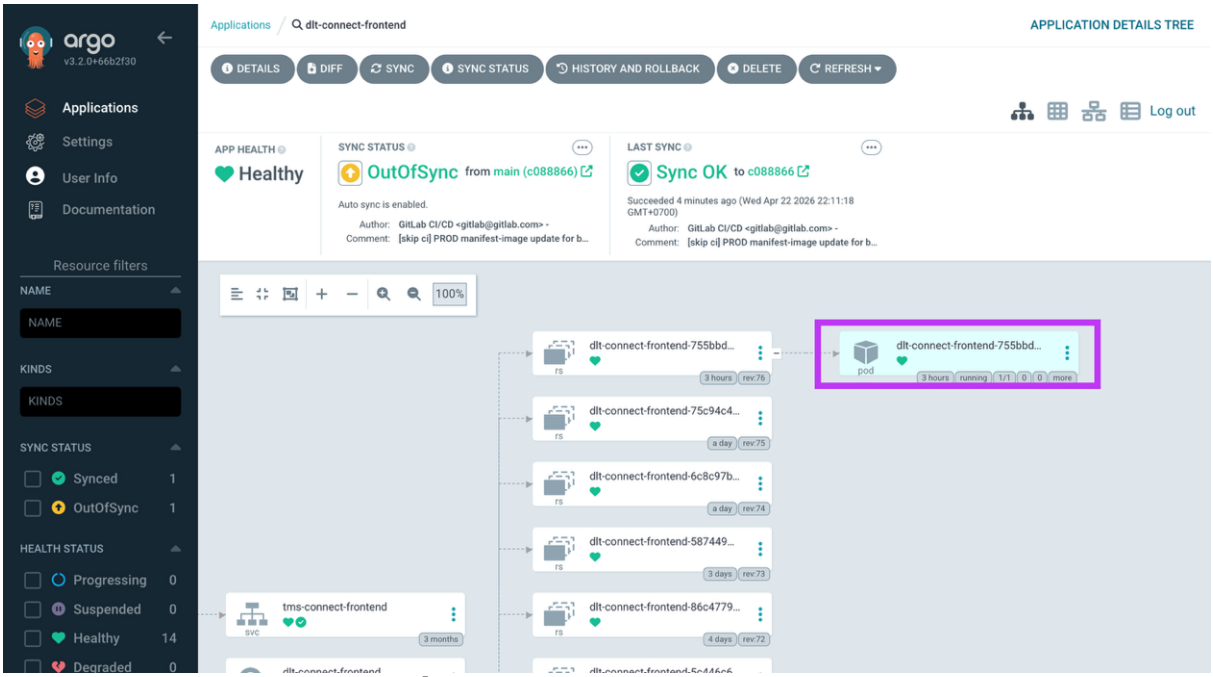
ในกรณีที่แอปพลิเคชันหรือบริการภายในระบบ DLT-TMS เกิดข้อผิดพลาดหรือทำงานผิดปกติ (Degraded State) ผู้ดูแลระบบสามารถตรวจสอบข้อมูลบันทึกการทำงาน (Logs) เพื่อวิเคราะห์หาสาเหตุของปัญหา (Root Cause Analysis) ผ่านหน้าจอ ArgoCD UI ได้โดยตรง โดยมีขั้นตอนการดำเนินงานดังนี้:

ขั้นตอนการเข้าถึงข้อมูลบันทึก:

1. **การเลือกบริการเป้าหมาย:** เข้าสู่ ArgoCD UI และเลือกแอปพลิเคชันที่ต้องการตรวจสอบในรายการแอปพลิเคชันทั้งหมด
2. **การระบุหน่วยปฏิบัติงาน:** ภายในรายละเอียดของแอปพลิเคชัน จะปรากฏรายการ Pods ที่เกี่ยวข้อง ให้เลือก Pod ที่มีสถานะผิดปกติหรือ Pod ที่ต้องการตรวจสอบข้อมูลการทำงาน
3. **การเข้าถึงส่วนแสดงผล Log:** เมื่อคลิกที่ Pod ดังกล่าว ให้เลือกแท็บ **"Logs"** ซึ่งระบบจะแสดงข้อมูลบันทึกการทำงานแบบเรียลไทม์ (Real-time Stream) จาก Container ภายใน Pod นั้นๆ

ประโยชน์เชิงเทคนิคและการสนับสนุนการปฏิบัติงาน:

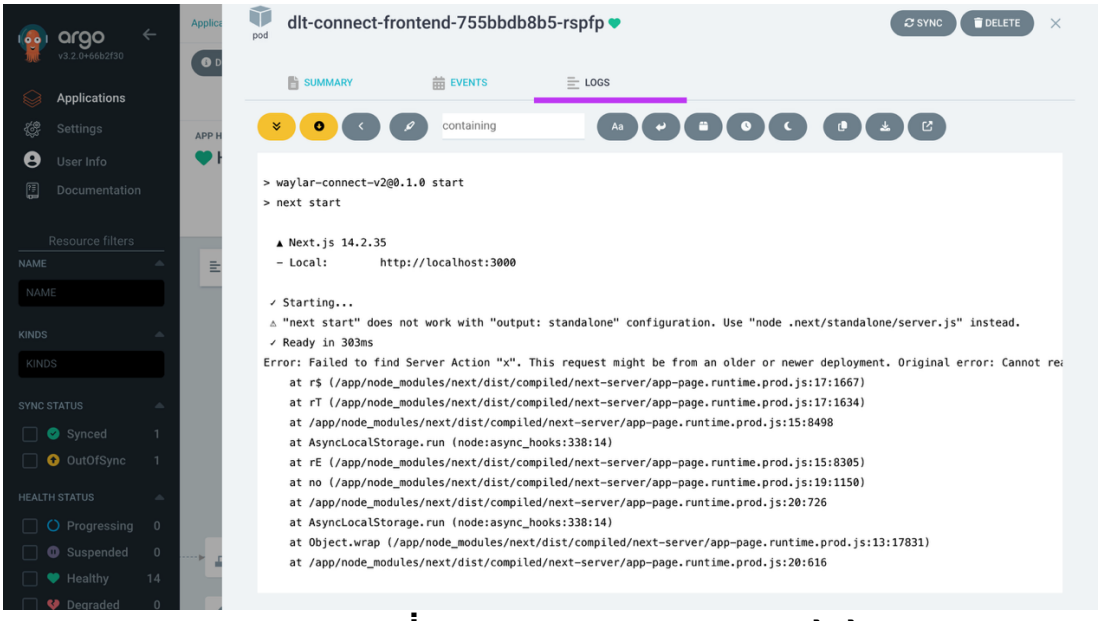
- **Centralized Debugging:** การดู Log ผ่าน ArgoCD UI ช่วยลดความจำเป็นในการเข้าถึงเครื่องแม่ข่าย (Server) ผ่าน Command Line โดยตรง ช่วยให้ผู้ปฏิบัติงานสามารถวิเคราะห์ปัญหาได้จากส่วนกลางอย่างสะดวกรวดเร็ว
- **Container-level Filtering:** ระบบอนุญาตให้ผู้ดูแลระบบสามารถเลือกกรอง Log เฉพาะเจาะจงราย Container (ในกรณีที่ Pod ประกอบด้วยหลาย Container) ซึ่งช่วยเพิ่มความละเอียดแม่นยำในการระบุจุดบกพร่อง
- **Historical Traceability:** ระบบสามารถแสดงประวัติ Log ย้อนหลังได้ในระดับหนึ่ง ช่วยให้ผู้ดูแลระบบสามารถตรวจสอบเหตุการณ์ที่เกิดขึ้นก่อนเกิดข้อผิดพลาด (Pre-incident context) ทำให้การวางแผนแก้ไขปัญหามีความถูกต้องและรวดเร็วยิ่งขึ้น



รูปที่ 5-7 ภาพ Pod ของระบบหน้าบ้าน

ภาพการระบุตำแหน่งทรัพยากร (Resource Identification)

- แสดงผังความสัมพันธ์ (Application Details Tree) ของแอปพลิเคชัน dlt-connect-frontend ซึ่งประกอบด้วยส่วนประกอบต่าง ๆ เช่น rs (ReplicaSet) และ pod
- การคลิกเลือก Pod ที่ต้องการตรวจสอบ (กรอบสีม่วง) จะช่วยให้ผู้ดูแลระบบเจาะจงลงไปถึงหน่วยประมวลผลที่อาจกำลังประสบปัญหา ทำให้ทราบสถานะสุขภาพ (Health) และข้อมูลเบื้องต้นของ Pod นั้น ๆ ได้ทันที



รูปที่ 5-8 ภาพแสดง Log ระบบหน้าบ้าน

ภาพการวิเคราะห์ข้อมูลบันทึกการทำงาน (Log Stream Analysis)

- แสดงหน้าต่างบันทึกการทำงาน (Logs) ของแอปพลิเคชันที่ดึงข้อมูลโดยตรงจาก Container
- ในภาพแสดงข้อความบันทึกของ Framework (Next.js) ซึ่งรวมถึงขั้นตอนการ Start ระบบ และข้อมูล Error Stack Trace เมื่อเกิดปัญหา "Failed to find Server Action"
- **ความสำคัญเชิงวิศวกรรม:** ข้อมูลที่ปรากฏในหน้านี้เปรียบเสมือน "กล่องดำ" ของระบบที่บันทึกร่องรอยการทำงานทั้งหมด ทำให้ผู้พัฒนาระบบสามารถตรวจสอบต้นตอของปัญหา (Root Cause) ได้อย่างแม่นยำ ว่าข้อผิดพลาดเกิดขึ้นที่ระดับโครงสร้างของซอฟต์แวร์ (Application Layer) หรือการตั้งค่า (Configuration) ในจุดใด

5.7. ขั้นตอนการเพิ่มแอปพลิเคชันใหม่เข้าสู่ระบบ (Application Onboarding Procedure)

ในกรณีที่โครงการมีความต้องการขยายขอบเขตการให้บริการ โดยการเพิ่มบริการใหม่ (เช่น Web Application หรือ Microservice ใหม่) ผู้ดูแลระบบต้องปฏิบัติตามขั้นตอนมาตรฐานภายใต้แนวทาง **GitOps** เพื่อให้มั่นใจว่าทุกทรัพยากรที่ถูกนำเข้าสู่คลัสเตอร์สามารถตรวจสอบได้และมีความปลอดภัย โดยมีรายละเอียดดังนี้

5.7.1.การจัดเตรียมโครงสร้างข้อมูล (Application Manifest Configuration)

การจัดเตรียมข้อมูลเริ่มต้นสำหรับแอปพลิเคชันใหม่ มีขั้นตอนการดำเนินงานดังนี้:

1. **Repository Synchronization:** ดึงข้อมูลล่าสุด (Pull) จาก Git Repository ที่จัดเก็บข้อมูลการกำหนดค่า (Config Repository) ลงสู่เครื่องแม่ข่าย เพื่อป้องกันความขัดแย้งของข้อมูล (Merge Conflicts)
2. **Resource Definition:** จัดทำไฟล์นิยามทรัพยากร (Resource Manifest) ในรูปแบบ **YAML** โดยจัดเก็บไว้ภายในโฟลเดอร์ apps/ ของ Git Repository หลัก เพื่อเป็น "แบบแปลน" (Blueprint) ให้ ArgoCD นำไปใช้กำหนดสถานะที่ต้องการ (Desired State) ให้กับคลัสเตอร์

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: my-new-app          # ← เปลี่ยนชื่อ
  namespace: argocd
spec:
  project: default
  source:
    repoURL: git@gitlab.com:dlt/dlt-tms/my-new-app-infra.git  # ← เปลี่ยน repo
    targetRevision: main
    path: prod              # ← folder ที่เก็บ manifests
  destination:
    server: https://kubernetes.default.svc
    namespace: default      # ← namespace ที่จะ deploy
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
    syncOptions:
      - CreateNamespace=true
```

รูปที่ 5-9 โค้ดตัวอย่างสำหรับการสร้าง argocd application

รูปที่ 5-9 แสดงถึงไฟล์นิยามทรัพยากรที่ใช้ในการสั่งการให้ ArgoCD เข้าไปจัดการแอปพลิเคชันภายใน Kubernetes โดยไฟล์นี้ทำหน้าที่เป็น "คำสั่งการ" (Instruction Set) เพื่อให้ระบบทราบว่าต้องนำซอฟต์แวร์จากแหล่งใด มาติดตั้งลงในส่วนใดของคลัสเตอร์ ดังนี้

- **การระบุตัวตนและเป้าหมาย (metadata & destination):** ส่วนนี้กำหนดชื่อของแอปพลิเคชันในระบบ ArgoCD และระบุปลายทาง (Destination) ใน Kubernetes Cluster ว่าต้องนำไปติดตั้งใน namespace: default เพื่อความเป็นระเบียบ
- **การเชื่อมต่อกับแหล่งต้นทาง (source):** นี่คือหัวใจสำคัญของการทำงานแบบ GitOps โดยระบุว่าต้องดึง "พิมพ์เขียว" (Manifests) มาจาก Git Repository ใด (repoURL), Branch ไหน (targetRevision: main), และอยู่ในโฟลเดอร์ใด (path: prod)
- **นโยบายการซิงโครไนซ์อัตโนมัติ (syncPolicy)**

- **Automated & SelfHeal:** เปิดใช้งานคุณสมบัติ "รักษาตัวเอง" (Self-Healing) หากมีใครเข้าไปแก้ไขคลัสเตอร์ด้วยมือ ระบบจะตรวจพบและดึงค่าจาก Git มาเขียนทับให้ตรงกับความจริงโดยอัตโนมัติ
- **Prune:** ยอมให้ระบบลบทรัพยากรเก่าที่ไม่จำเป็นออกเมื่อมีการอัปเดต ช่วยรักษาความสะอาดของคลัสเตอร์

- **ความยืดหยุ่น (syncOptions):** ตัวเลือก CreateNamespace=true ช่วยให้ ArgoCD สามารถสร้าง Namespace ปลายทางให้โดยอัตโนมัติหากยังไม่มีอยู่จริง ทำให้ลดขั้นตอนการเตรียมระบบด้วยมือไปได้มาก

5.7.2. การลงทะเบียนแหล่งที่มาของแอปพลิเคชัน (Repository Integration & Authentication)

ในระบบ GitOps ของ DLT-TMS การที่ ArgoCD จะสามารถดึงข้อมูล Manifests (YAML) เพื่อนำไป Deploy แอปพลิเคชันได้นั้น ระบบจำเป็นต้องมีการสร้าง **"ช่องทางการสื่อสารที่ปลอดภัย" (Secure Communication Channel)** กับ Git Repository ต้นทางเสียก่อน โดยใช้คำสั่ง argocd repo add เป็นตัวกำหนดเงื่อนไขการเข้าถึง ดังนี้:

Bash
```
argocd repo add git@gitlab.com:dlt/dlt-tms/my-new-app-infra.git \
  --ssh-private-key-path ./argocd_shared
```

กลไกการทำงานและความปลอดภัยเชิงเทคนิค:

การระบุตัวตนผ่าน SSH (SSH-based Authentication): การใช้แฟล็ก --ssh-private-key-path เป็นการกำหนดให้ ArgoCD ใช้กุญแจส่วนตัว (Private Key) ในการพิสูจน์ตัวตนกับ GitLab วิธีนี้มีความปลอดภัยสูงเนื่องจากเป็นการสื่อสารแบบเข้ารหัสผ่านช่องทาง SSH ซึ่งเป็นมาตรฐานในการจัดการซอร์สโค้ดระดับองค์กร

การจำกัดสิทธิ์การเข้าถึง (Least Privilege Principle): การเพิ่ม Repository ทีละแห่งช่วยให้ผู้ดูแลระบบสามารถกำหนดสิทธิ์ (Permission) แยกตามแต่ละแอปพลิเคชันได้ หากแอปพลิเคชันใดไม่จำเป็นต้องเข้าถึง Repository อื่น ก็ไม่ต้องอนุญาตสิทธิ์เพิ่มเติม ช่วยลดความเสี่ยงจากการเข้าถึงข้อมูลเกินจำเป็น

นัยสำคัญต่อการออกแบบระบบ:

การออกแบบให้ ArgoCD แยกการจัดการ Repository เป็นส่วนหนึ่งของ Application Onboarding มีประโยชน์สำคัญในด้าน **Repository Isolation** (การแยกโปรเจกต์ซอร์สโค้ดออกจาก Infrastructure Repository หลัก) และ **Scalability** (รองรับการขยายตัวของระบบในอนาคตได้อย่างอิสระโดยไม่กระทบต่อการทำงานเดิม)

5.7.3. การบันทึกและผลักดันการเปลี่ยนแปลงเข้าสู่ระบบควบคุมเวอร์ชัน (Version Control Operations)

คำสั่งข้างล่างจะเป็นการอธิบายขั้นตอนการนำไฟล์ที่สร้างไว้ในขั้นตอนก่อนหน้า (เช่น apps/my-new-app.yaml) ไป commit และ push ขึ้นไปยัง GitLab repository โดยใช้คำสั่ง git ซึ่งประกอบด้วย:

- git add apps/my-new-app.yaml – ใช้สำหรับเพิ่มไฟล์ apps/my-new-app.yaml เข้าไปยัง staging area ของ git เพื่อเตรียม commit
- git commit -m "feat: add my-new-app to ArgoCD" – ทำการ commit ไฟล์ที่ถูกเพิ่มเข้า staging area พร้อมใส่ข้อความอธิบายการเปลี่ยนแปลงในครั้งนี้
- git push origin main – ส่ง (push) การ commit ที่ทำไว้ขึ้นไปยัง branch main บน GitLab repository

ขั้นตอนนี้จะทำให้ไฟล์ my-new-app.yaml ที่สร้างขึ้นถูกรวมเข้าไปอยู่ใน repository และสามารถนำไปใช้งานกับ ArgoCD ได้ต่อไป

รายละเอียดกระบวนการปฏิบัติงาน:

- **การเตรียมข้อมูลเพื่อบันทึกสถานะ (Staging Area Management):** คำสั่ง git add ทำหน้าที่กำหนดรายการไฟล์ (File Indexing) ที่มีการเปลี่ยนแปลงเข้าสู่พื้นที่เตรียมการ (Staging Area) เพื่อรอการตรวจสอบและยืนยันความถูกต้อง ก่อนที่จะถูกบันทึกเป็นประวัติการทำงานของระบบ
- **การบันทึกสถานะและประวัติการเปลี่ยนแปลง (Commit Operation):** คำสั่ง git commit เป็นการสร้างจุดตรวจสอบ (Checkpoint) ของระบบ โดยบันทึกไฟล์ที่จัดเตรียมไว้พร้อมระบุคำอธิบายการเปลี่ยนแปลง (Commit Message) ซึ่งเป็นมาตรฐานที่สำคัญในการตรวจสอบย้อนหลัง (Audit Trail) และเพิ่มความโปร่งใสในการปรับเปลี่ยนทรัพยากรของระบบ
- **การผลักดันการเปลี่ยนแปลงสู่แหล่งข้อมูลกลาง (Remote Synchronization):** คำสั่ง git push ดำเนินการส่งข้อมูลที่บันทึกไว้ไปยังแหล่งเก็บข้อมูลกลาง (Centralized

Repository) บน GitLab ซึ่งเป็นศูนย์กลางที่ ArgoCD จะทำหน้าที่สำรวจความเปลี่ยนแปลง (Polling) เพื่อนำข้อมูลไปประมวลผลและปรับสถานะให้สอดคล้องกับโครงสร้างพื้นฐานใน Kubernetes คลัสเตอร์โดยอัตโนมัติ

นัยสำคัญในการบริหารจัดการระบบ: การดำเนินการตามขั้นตอนเหล่านี้ถือเป็นหัวใจสำคัญของการทำงานในรูปแบบ **GitOps Workflow** ซึ่งเปลี่ยนจากการจัดการทรัพยากรด้วยคำสั่งแบบ Manual เป็นการจัดการผ่านประวัติการเปลี่ยนแปลงที่ตรวจสอบได้ (Declarative Infrastructure) ซึ่งจะช่วยลดข้อผิดพลาดในการตั้งค่า (Human Error) และสร้างความเสถียรภาพให้กับบริการของระบบ DLT-TMS ในระยะยาว

5.7.4. การซิงโครไนซ์สถานะระบบ (Automated & Manual Synchronization)

ในกระบวนการทำงานแบบ GitOps ระบบ ArgoCD จะทำหน้าที่เป็นตัวกลางในการตรวจสอบความสอดคล้อง (Reconciliation Loop) ระหว่างสถานะที่กำหนดไว้ใน Git Repository และสถานะที่เกิดขึ้นจริงใน Kubernetes Cluster โดยมีรายละเอียดการปฏิบัติงานดังนี้:

1. การซิงโครไนซ์อัตโนมัติ (Automated Synchronization): ระบบ ArgoCD ได้รับการกำหนดค่าให้เฝ้าสังเกตการณ์ (Polling) ความเปลี่ยนแปลงใน GitLab Repository อย่างต่อเนื่อง โดยภายหลังการผลักดันการเปลี่ยนแปลง (Push) เข้าสู่ Repository ระบบจะดำเนินการตรวจสอบความแตกต่างและปรับปรุงสถานะการติดตั้ง (Deployment) ให้เป็นปัจจุบันโดยอัตโนมัติ ภายในระยะเวลาประมาณ 3 นาที เพื่อลดภาระงานของผู้ดูแลระบบ. และป้องกันความผิดพลาดจากการตั้งค่าด้วยมือ

2. การซิงโครไนซ์ด้วยคำสั่งเฉพาะ (Manual Synchronization): ในกรณีที่จำเป็นต้องดำเนินการอัปเดตระบบทันทีโดยไม่ต้องรอรอบการตรวจสอบอัตโนมัติ ผู้ดูแลระบบสามารถใช้คำสั่งเพื่อบังคับการซิงโครไนซ์ (Force Sync) ได้ดังนี้:

```
argocd app sync dlt-platform   # sync parent app ให้รับ child ใหม่
```

รูปที่ 5-10 คำสั่ง Sync Arcocd App

คำอธิบายเชิงเทคนิค
คำสั่ง: argocd app sync dlt-platform

วัตถุประสงค์: เป็นการส่งคำสั่ง (Command) เพื่อบังคับให้ ArgoCD ดำเนินการ "ดึงข้อมูลล่าสุด" (Pull Latest State) จาก Git Repository ที่ระบุไว้ในไฟล์ Manifests ของแอปพลิเคชันที่ชื่อ dlt-platform แล้วนำไปปรับใช้ (Deploy/Sync) ใน Kubernetes คลัสเตอร์ทันที

สถานการณ์ที่ต้องใช้งาน: ในภาพระบุว่า: # sync parent app ให้รับ child ใหม่ หมายความว่าการทำงานในโครงการ DLT-TMS มีโครงสร้างแบบ Hierarchical (Parent-Child Application) ซึ่งการซิงค์ที่ตัว Parent แอปพลิเคชัน จะช่วยให้แอปพลิเคชันย่อย (Child App) ได้รับการปรับปรุงสถานะหรือตรวจพบการเปลี่ยนแปลงที่เกิดขึ้นกับ Component ลูกข่าย ทั้งหมดพร้อมกัน

ความจำเป็น: โดยปกติ ArgoCD จะทำงานแบบอัตโนมัติ (Automated) ตามรอบเวลา แต่ในกรณีที่คุณเพิ่ง Onboarding แอปพลิเคชันใหม่, ทำการแก้ไข Hotfix เร่งด่วน, หรือต้องการยืนยันว่าโครงสร้างลูกข่าย (Child Apps) ทั้งหมดอยู่ในสถานะล่าสุดพร้อมกัน คำสั่งนี้คือวิธีที่รวดเร็วที่สุดในการบังคับให้ระบบทำ Sync

5.7.5. การตรวจสอบความสำเร็จในการติดตั้ง (Deployment Verification)

ภายหลังกระบวนการซิงโครไนซ์สถานะระบบ (Synchronization) ผู้ดูแลระบบ. ต้องดำเนินการตรวจสอบความสมบูรณ์ของระบบ (System Integrity Verification) เพื่อยืนยันว่าแอปพลิเคชันได้รับการ Deploy เข้าสู่ Kubernetes Cluster อย่างถูกต้องและพร้อมสำหรับการให้บริการ โดยแบ่งขั้นตอนการตรวจสอบออกเป็น 2 ระดับ ดังนี้:

1. การตรวจสอบสถานะในระดับ ArgoCD (Control Plane Verification)

เป็นการตรวจสอบสถานะการทำงานผ่าน ArgoCD เพื่อยืนยันว่าสถานะการกำหนดค่า (Desired State) กับสถานะการติดตั้งจริง (Actual State) มีความสอดคล้องกัน:

Bash
argocd app get my-new-app

เป้าหมาย: เพื่อประเมินสถานะภาพรวม (Health Status) ของแอปพลิเคชัน หากผลลัพธ์ปรากฏเป็น Healthy และ Synced แสดงว่า ArgoCD ได้ส่งมอบทรัพยากรเข้าสู่ระบบสำเร็จแล้ว

2. การตรวจสอบสถานะในระดับ Kubernetes (Data Plane Verification)

เป็นการตรวจสอบสถานะการทำงานจริงของหน่วยประมวลผลภายในคลัสเตอร์ เพื่อยืนยันว่า Pod ของแอปพลิเคชันกำลังปฏิบัติงานอยู่จริง:

Bash
kubectl get pods -n default | grep my-new-app

เป้าหมาย: เพื่อยืนยันว่า Pod ของแอปพลิเคชันมีสถานะเป็น Running และค่า READY แสดงสถานะ 1/1 (หรือตามจำนวน Container ที่กำหนด) ซึ่งถือเป็นหลักฐานเชิงประจักษ์ว่าแอปพลิเคชันเริ่มทำงานได้อย่างสมบูรณ์

```
argocd app get my-new-app
kubectl get pods -n default | grep my-new-app
```

รูปที่ 5-11 คำสั่งในการตรวจสอบ Argocd application และ Kubernetes Pod

5.8. การปรับปรุงแอปพลิเคชันและกลไกการอัปเดตเวอร์ชัน (Application Update & Versioning Strategy)

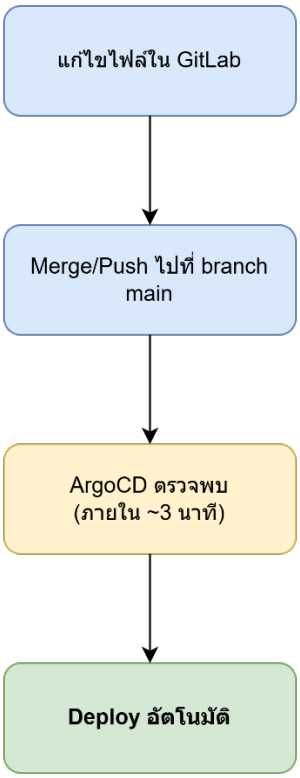
ในแต่ละ Apps จะมีการเรียกใช้ docker image จาก gitlab image registry โดยแต่ละ image จะมีการระบุ version ของ โปรแกรม Image คือโปรแกรมที่ทำสำเร็จรูปแล้วให้ argocd สัง Kubernetes โดยใช้ image นั้นๆ เพื่อรันได้เลย โดยที่จะทำการแก้ไขชื่อ image เพื่อ และ push code update เพื่อให้ argocd สั่งรัน application เวอร์ชั่นใหม่ได้
รายละเอียดของ image จะอธิบายในบทที่ 7

5.8.1. กระบวนการปรับปรุงแอปพลิเคชัน (Update Procedure)

ระบบดังกล่าวได้รับการออกแบบให้ดำเนินการอัปเดตผ่าน Git เท่านั้น ไม่ควรดำเนินการแก้ไขใด ๆ โดยตรงบน Kubernetes การดำเนินการอัปเดตระบบผ่าน GitOps flow จะเน้นการบริหารจัดการซอร์สโค้ดและไฟล์กำหนดค่าการปรับใช้ (deployment configuration) ทั้งหมดผ่าน Git repository เป็นหลัก โดยเมื่อมีการแก้ไขโค้ดหรือไฟล์กำหนดค่า เช่น deployment.yaml ใน repository และดำเนินการ push ขึ้น GitLab ระบบ ArgoCD จะตรวจสอบและนำการเปลี่ยนแปลงดังกล่าวไปดำเนินการ deploy อัตโนมัติบน Kubernetes cluster โดยไม่จำเป็นต้องเข้าไปปรับเปลี่ยนหรือสั่ง deploy ผ่าน Kubernetes โดยตรง ทั้งนี้ แนวทางดังกล่าวช่วยให้สามารถติดตามประวัติการเปลี่ยนแปลงได้อย่างชัดเจนและลดความเสี่ยงจากความผิดพลาดที่อาจเกิดจากการดำเนินการด้วยตนเอง

- เริ่มต้นโดยการแก้ไขไฟล์ใน repository เช่น deployment.yaml เพื่ออัปเดต image tag หรือปรับปรุง configuration อื่น ๆ ที่เกี่ยวข้อง
- ดำเนินการ push การเปลี่ยนแปลงขึ้น GitLab
- ArgoCD จะซิงโครไนซ์การเปลี่ยนแปลงจาก GitLab ไปยัง Kubernetes โดยอัตโนมัติ
- สามารถตรวจสอบสถานะการ deploy ได้โดยใช้คำสั่ง argocd app get [app-name] หรือ kubectl get pods -n [namespace]

การดำเนินการอัปเดตผ่าน GitOps flow มีข้อดี คือ ความโปร่งใสในการบริหารจัดการระบบ โดยทุกการเปลี่ยนแปลงสามารถตรวจสอบและตรวจทานย้อนหลังได้จากประวัติบน Git อีกทั้งยังช่วยลดความเสี่ยงจากความผิดพลาดที่อาจเกิดขึ้นจากการ deploy ด้วยวิธี manual หรือเกิด configuration drift นอกจากนี้ แนวทางนี้ยังส่งเสริมให้การทำงานเป็นทีมมีประสิทธิภาพและเป็นระบบมากยิ่งขึ้น โดยกำหนดให้ Git เป็นศูนย์กลางของกระบวนการ deploy ทั้งหมด ดังรูปที่ 5-12



รูปที่ 5-12 กระบวนการอัพเดทโค้ดและระบบปฏิบัติการผ่าน Argocd

5.8.2. ตัวอย่างการปรับปรุงเวอร์ชันของแอปพลิเคชัน (Example: Image Tag Update)

ตัวอย่างขั้นตอนการอัพเดทโค้ดส่วนของ API คือระบบจะทำการสร้าง image version ใหม่ผู้ดูแลระบบสามารถไปแก้ไข version file kustomization บน repositories api-infra โดยเมื่อมีการแก้ไข code newTag เป็น image ของโปรแกรมเวอร์ชั่นใหม่แล้ว Argocd จะทำการเช็ค commit update หากมีการอัพเดทเกิดขึ้นตัวระบบจะดึง image เวอร์ชั่นใหม่ที่มีฟีเจอร์ใหม่มาแสดง

รูปที่ 5-13 ภาพ File Kustomization Config ของระบบหลังบ้านสำหรับ Argocd

5.8.3. การสั่งการซิงโครไนซ์ด้วยตนเอง (Manual Synchronization)

ตามกลไกการทำงานปกติ ระบบ ArgoCD จะมีการตรวจสอบการเปลี่ยนแปลง (Polling/Webhooks) ภายใน Git Repository โดยอัตโนมัติ เพื่อนำสถานะล่าสุดจาก Git ไปปรับปรุงใน Kubernetes คลัสเตอร์ อย่างไรก็ตาม ในสถานการณ์ที่ต้องการปรับปรุงระบบอย่างเร่งด่วน (เช่น การแก้ไขข้อผิดพลาดในภาวะวิกฤต หรือการเร่ง Deploy ฟีเจอร์ใหม่) ผู้ดูแลระบบสามารถข้ามขั้นตอนการรอรอบการทำงานอัตโนมัติได้โดยการสั่งการซิงโครไนซ์ด้วยตนเอง (Manual Sync) ผ่าน Command Line Interface (CLI) ดังนี้:

คำสั่งสำหรับการซิงโครไนซ์ทันที:

Bash
argocd app sync [app-name]

- **กระบวนการทำงาน:** เมื่อดำเนินการคำสั่งดังกล่าว ArgoCD จะเข้าสู่สถานะ **Force Reconciliation** โดยระบบจะทำการดึงข้อมูลนิยามแอปพลิเคชัน (Manifests) เวอร์ชันล่าสุดจาก Git Repository มาประมวลผลและนำไป Deploy ลงในคลัสเตอร์ทันที โดยไม่รอรอบระยะเวลาการตรวจสอบปกติ (Poll Interval)
- **ประโยชน์เชิงปฏิบัติ:**
    - **ลดเวลาในการรอ (Reduced Lead Time):** ช่วยให้สถานะของแอปพลิเคชันเป็นปัจจุบันตามการเปลี่ยนแปลงใน Git ได้ในทันที
    - **การจัดการสถานะวิกฤต (Incident Response):** ช่วยให้ผู้ดูแลระบบสามารถแก้ไขหรือย้อนกลับ (Rollback) การตั้งค่าได้อย่างรวดเร็วในกรณีที่เกิดปัญหา
    - **การตรวจสอบสถานะ (Immediate Verification):** ช่วยยืนยันว่าโครงสร้างการตั้งค่าใหม่ที่ผลักดันขึ้นไปนั้นได้รับการตอบสนองจากระบบอย่างถูกต้องและทันท่วงที

5.8.4. การตรวจสอบความสำเร็จในการติดตั้ง (Deployment Verification)

ภายหลังกระบวนการซิงโครไนซ์สถานะระบบ (Synchronization) ผู้ดูแลระบบต้องดำเนินการตรวจสอบกระบวนการอัปเดตเวอร์ชันของแอปพลิเคชัน เพื่อยืนยันว่า Kubernetes ได้ดำเนินการปรับเปลี่ยนสถานะ (Rolling Update) เป็นไปอย่างถูกต้องและไม่มีข้อผิดพลาด โดยมีขั้นตอนการตรวจสอบ ดังนี้:

1. การติดตามสถานะการ Deploy (Rollout Status Tracking)

คำสั่งนี้ใช้สำหรับติดตามความคืบหน้าของการทำ Rolling Update ของ Deployment ในคลัสเตอร์:

Bash
kubectl rollout status deployment/dlt-platform-api -n default

- **เป้าหมาย:** ระบบจะแสดงผลลัพธ์จนกว่าการอัปเดต Pod ทั้งหมดจะเสร็จสิ้น (Successfully rolled out) หากมีปัญหาในระหว่างการอัปเดต คำสั่งนี้จะระบุสาเหตุที่ทำให้กระบวนการหยุดชะงัก

2. การตรวจสอบการเกิดใหม่ของ Pods (Pod Lifecycle Observation)

คำสั่งนี้ใช้สำหรับเฝ้าสังเกตการณ์การสร้าง Pod ใหม่และการยกเลิก Pod เดิมในแบบเรียลไทม์:

Bash
kubectl get pods -n default -w

- **เป้าหมาย:** การใช้แฟล็ก -w (Watch) ช่วยให้ผู้ดูแลระบบเห็นเหตุการณ์ (Event) ที่เกิดขึ้นในคลัสเตอร์ทันที เช่น สถานะ ContainerCreating, Running, หรือ Terminating ซึ่งเป็นหลักฐานเชิงประจักษ์ว่า Kubernetes กำลังดำเนินการเปลี่ยนผ่านสู่เวอร์ชันใหม่ตามคำสั่งของ ArgoCD

นัยสำคัญในการบริหารจัดการปัญหา (Troubleshooting Procedures)

การตรวจสอบผ่าน rollout status และการติดตามแบบ Watch ถือเป็นแนวทางปฏิบัติที่ดีที่สุด (Best Practice) ในการบริหารจัดการแอปพลิเคชัน เพราะช่วยให้ผู้ดูแลระบบมั่นใจได้ว่า:

- **ความต่อเนื่องในการให้บริการ (High Availability):** สามารถยืนยันได้ว่าระบบมีการทยอยเปลี่ยน Pod โดยไม่ทำให้บริการหยุดชะงัก
- **ความถูกต้องของเวอร์ชัน (Deployment Accuracy):** ยืนยันได้ว่า Image Tag ใหม่ถูกนำมาใช้งานจริงอย่างถูกต้อง

5.9. การตรวจสอบ Log ผ่าน ArgoCD UI (Log Analysis)

การดู log ของแต่ละแอปพลิเคชันหรือ service บนระบบ DLT-TMS สามารถทำได้ผ่านหน้าเว็บของ ArgoCD UI ซึ่งช่วยให้ผู้ดูแลระบบและผู้พัฒนาสามารถตรวจสอบสถานะและปัญหาของ service ต่าง ๆ ได้อย่างสะดวก ไม่จำเป็นต้องใช้คำสั่งผ่าน command line ดังนี้

1. เข้าสู่ระบบ ArgoCD UI ผ่านเบราว์เซอร์ โดยใช้ URL ที่องค์กรกำหนด (เช่น [URL]>) และเข้าสู่ระบบด้วยบัญชีผู้ใช้ที่ได้รับอนุญาต
2. หลังจากเข้าสู่ระบบแล้ว จะเห็นรายการแอปพลิเคชันทั้งหมดที่ถูก deploy และจัดการผ่าน ArgoCD
3. เลือกแอปพลิเคชันที่ต้องการดู log เช่น dlt-platform-api, tms-connect-frontend หรือ service อื่น ๆ ที่ต้องการตรวจสอบ

4. เมื่อคลิกเข้าไปในแอปพลิเคชัน จะเห็นรายละเอียดสถานะของแอปพลิเคชันนั้น ๆ รวมถึงรายชื่อ pod ที่เกี่ยวข้อง
5. เลือก pod ที่ต้องการดู log จากนั้นคลิกเมนู "Logs" หรือ "View Logs" ระบบจะแสดง log แบบ realtime ของ pod นั้น ๆ สามารถเลื่อนดูย้อนหลัง หรือเลือกดูเฉพาะบางช่วงเวลาได้ตามต้องการ
6. นอกจากนี้ ArgoCD UI ยังสามารถแสดง log ของ pod หลายตัวพร้อมกัน หรือเลือก filter เฉพาะ container ภายใน pod ได้อีกด้วย ช่วยให้การตรวจสอบปัญหาและการ debug ระบบทำได้อย่างรวดเร็วและเป็นระบบมากขึ้น

ข้อดีของการใช้ ArgoCD UI ในการดู log คือไม่ต้องจำคำสั่ง kubectl ให้ยุ่งยาก ลดความผิดพลาดในการเลือก namespace หรือ deployment และเหมาะสำหรับผู้ใช้งานที่ชอบการทำงานผ่านหน้าจอกราฟิก ไม่ถนัด command line อีกทั้งยังสามารถดูสถานะและ log ของทุกแอปพลิเคชันในระบบได้จากศูนย์กลางจุดเดียว

ทั้งนี้ การดู log ผ่าน ArgoCD UI จะช่วยสนับสนุนการตรวจสอบและดูแลระบบ DLT-TMS ให้มีความต่อเนื่องและปลอดภัย สามารถแก้ไขปัญหาได้อย่างมีประสิทธิภาพ

5.10. การแก้ไขปัญหาที่พบบ่อย (Troubleshooting & Incident Response)

ในการบริหารจัดการระบบผ่านแนวทาง GitOps และ ArgoCD ผู้ดูแลระบบอาจพบความผิดปกติในระหว่างกระบวนการ Deploy หรือการรันแอปพลิเคชัน เพื่อให้การกู้คืนระบบ (Recovery) เป็นไปอย่างรวดเร็วและมีประสิทธิภาพ จึงได้รวบรวมแนวทางการแก้ไขปัญหาที่พบบ่อย (Common Pitfalls) และขั้นตอนการตอบสนอง ดังนี้:

1. ข้อผิดพลาดเกี่ยวกับสถานะการทำงานของ Pods (Pod Lifecycle Errors)

เมื่อ Pod ไม่สามารถเข้าสู่สถานะ Running ได้อย่างสมบูรณ์ ให้ดำเนินการวิเคราะห์ตามสถานะความผิดพลาดดังนี้:

- **CrashLoopBackOff:** บ่งชี้ว่าแอปพลิเคชันภายใน Container มีการสั่งหยุดทำงานซ้ำๆ
    - **แนวทางแก้ไข:** ตรวจสอบ Logs ผ่าน ArgoCD UI หรือคำสั่ง kubectl logs เพื่อค้นหา Runtime Error ในซอร์สโค้ด (เช่น การเชื่อมต่อฐานข้อมูลล้มเหลว หรือการตั้งค่า Config ไม่ถูกต้อง)

- **ImagePullBackOff:** บ่งชี้ว่า Kubernetes ไม่สามารถดาวน์โหลด Container Image มาใช้งานได้
    - **แนวทางแก้ไข:** ตรวจสอบความถูกต้องของชื่อ Image และ Tag ในไฟล์ YAML อีกครั้ง รวมถึงตรวจสอบสิทธิ์การเข้าถึง (Pull Secret) และการเชื่อมต่อกับ GitLab Container Registry
- **OOMKilled (Out of Memory):** บ่งชี้ว่าแอปพลิเคชันใช้หน่วยความจำเกินกว่าที่กำหนดไว้ใน Limit
    - **แนวทางแก้ไข:** ปรับปรุงค่า resources.limits ในไฟล์ YAML Manifests ให้เหมาะสมกับปริมาณงานจริงของแอปพลิเคชัน

2. ข้อผิดพลาดเกี่ยวกับความสอดคล้องของสถานะ (Synchronization Mismatch)

- **OutOfSync:** แอปพลิเคชันในคลัสเตอร์ไม่ตรงกับสิ่งที่ระบุใน Git Repository
    - **แนวทางแก้ไข:** ตรวจสอบว่ามีการผลักดัน (Push) การเปลี่ยนแปลงขึ้นสู่ Branch หลักที่ถูกต้องหรือไม่ หรือหากต้องการบังคับอัปเดตทันที ให้ใช้คำสั่ง argocd app sync [app-name] เพื่อบังคับให้ระบบปรับสถานะ (Force Reconciliation)
- **Config Synchronization Fail:** เกิดจากความผิดพลาดในการระบุรูปแบบไฟล์ YAML (Syntax Error)
    - **แนวทางแก้ไข:** ตรวจสอบการจัดวางบรรทัด (Indentation) และโครงสร้างไฟล์ YAML ตามมาตรฐาน Kubernetes เพื่อให้ ArgoCD สามารถประมวลผลไฟล์ได้ถูกต้อง

5.10.1. ปัญหา: Application สถานะ OutOfSync

อธิบายปัญหา: เมื่อแอปพลิเคชันใน ArgoCD แสดงสถานะ OutOfSync หมายความว่า configuration ที่กำหนดไว้ใน Git repository ไม่ตรงกับสถานะจริงของระบบ Kubernetes หรือมีการเปลี่ยนแปลงที่ไม่ได้ sync กับต้นทาง เช่น มีการแก้ไข resource ผ่าน kubectl โดยตรง หรือ commit ใหม่ใน Git ยังไม่ได้ deploy เข้าสู่ cluster ส่งผลให้เกิดความไม่ตรงกันระหว่าง source กับ target ซึ่งอาจทำให้แอปพลิเคชันทำงานผิดพลาดหรือขาดความสอดคล้องกับ version ที่ควรจะเป็น

การแก้ไขเบื้องต้น:

1. ตรวจสอบรายละเอียด OutOfSync ในหน้า ArgoCD UI ว่า resource ใดบ้างที่มีปัญหา
2. กดปุ่ม **Sync** หรือ **Refresh** เพื่อให้ ArgoCD ดึง configuration ล่าสุดจาก Git repository แล้วปรับสถานะของ resource ใน cluster ให้ตรงกับ source
3. หากมี error ในการ sync ให้ตรวจสอบ log หรือ error message ที่แสดงในหน้า ArgoCD เพื่อแก้ไขตามคำแนะนำ เช่น แก้ไขไฟล์ YAML ใน Git หรือแก้ไข resource ที่ขัดแย้งกัน
4. กรณีที่มีการเปลี่ยนแปลงผ่าน command line (kubectl) แนะนำให้ revert กลับไปรอการ sync ผ่าน ArgoCD เพื่อรักษาความสอดคล้องของระบบ

การแก้ไขเบื้องต้นนี้ช่วยให้ระบบกลับมาสอดคล้องกับเวอร์ชันใน Git และลดปัญหาจากการ deploy ที่ไม่ตรงกัน โดยควรปฏิบัติตามแนวทางการจัดการผ่าน ArgoCD เป็นหลักเพื่อความปลอดภัยและความต่อเนื่องของระบบ

5.10.2. ปัญหา: Pod สถานะ CrashLoopBackOff

อาการนี้บ่งชี้ว่าแอปพลิเคชันเริ่มทำงานแต่เกิดข้อผิดพลาดและถูกหยุดการทำงานซ้ำๆ (Restart Loop)

- **ขั้นตอนการตรวจสอบ:**

  Bash

  ```
  # ตรวจสอบรายละเอียดสถานะและเหตุการณ์ล่าสุดของ Pod
  kubectl describe pod <pod-name> -n default

  # ตรวจสอบบันทึกการทำงานของ Container ในรอบก่อนหน้าที่ล้มเหลว
  kubectl logs <pod-name> -n default --previous
  ```

- **การแก้ไข:** วิเคราะห์สาเหตุจาก Logs (เช่น Config ผิดพลาด หรือขาด Database Connection) แล้วดำเนินการแก้ไขใน Git Repository ตามมาตรฐาน GitOps เพื่อให้ ArgoCD ทำการซิงโครไนซ์สถานะที่ถูกต้องอีกครั้ง

5.10.3. ปัญหา: ImagePullBackOff ดึง Docker image ไม่ได้

สาเหตุส่วนใหญ่มักเกิดจากข้อมูลรับรอง (Credentials) ของ GitLab Registry หมดอายุ หรือมีการระบุชื่อ Image/Tag ผิดพลาด

สาเหตุ: GitLab registry credentials หมดอายุ หรือ image ไม่มีอยู่

วิธีแก้:

```
# ตรวจสอบ secret
kubectl get secret gitlab -n default

# อัปเดต credentials (ถ้าหมดอายุ)
kubectl delete secret gitlab -n default
kubectl create secret docker-registry gitlab \
  --docker-server=registry.gitlab.com \
  --docker-username=<username> \
  --docker-password=<new-token> \
  --docker-email=<email> \
  -n default
```

5.10.4. ปัญหา: เว็บไซต์เข้าไม่ได้ (tms.dlt.go.th)

การแก้ไขปัญหาในระดับการให้บริการ (Service Level) ให้ดำเนินการตามลำดับขั้นดังนี้:

ตรวจสอบ Ingress Controller: kubectl get pods -n ingress-nginx (ยืนยันว่าตัวนำเข้าข้อมูลทำงานปกติ)
ตรวจสอบ Ingress Rules: kubectl get ingress -n default (ยืนยันการตั้งค่า Routing)
ตรวจสอบ Frontend Pod: kubectl get pods -n default | grep frontend (ยืนยันสถานะของแอปพลิเคชัน)
ตรวจสอบ SSL/TLS: kubectl get certificate -n default (ยืนยันสถานะใบรับรองความปลอดภัย)

ขั้นตอนตรวจสอบ:

```
# ขั้น 1: ตรวจสอบ Ingress Controller
kubectl get pods -n ingress-nginx
# ขั้น 2: ตรวจสอบ Ingress rules
kubectl get ingress -n default
# ขั้น 3: ตรวจสอบ Frontend pod
kubectl get pods -n default | grep frontend
# ขั้น 4: ตรวจสอบ SSL certificate
kubectl get certificate -n default
```

5.10.5. ปัญหา: ArgoCD UI เข้าไม่ได้

วิธีแก้:

```
# ตรวจสอบ argocd pods
kubectl get pods -n argocd
# ถ้า pod ไม่ Running ให้รอ หรือดู logs
kubectl logs -n argocd deploy/argocd-server
# เปิด port forward ใหม่
kubectl port-forward svc/argocd-server -n argocd 8080:443
```

5.10.6. ปัญหา: Cluster หายไปทั้งหมด / ต้องสร้างใหม่

ในกรณีวิกฤตที่คลัสเตอร์เสียหายจนต้องติดตั้งใหม่ (Re-provisioning):

1. **การกำจัดทรัพยากรที่เสียหาย:** kind delete cluster --name dlt-production
2. **การติดตั้งระบบใหม่:** ดำเนินการผ่านสคริปต์มาตรฐานที่เตรียมไว้: sh ./setup-infra.sh

คำเตือนด้านความปลอดภัย: แม้การติดตั้งคลัสเตอร์ใหม่จะทำให้สถานะระบบกลับมาเริ่มต้นใหม่ แต่ข้อมูลในฐานข้อมูลจะยังคงอยู่ตราบใดที่ข้อมูลใน Persistent Volume (เช่น /mnt/data/) ยังคงมีความสมบูรณ์ ผู้ดูแลระบบควรสำรองข้อมูลสำคัญก่อนดำเนินการทำลายคลัสเตอร์ทุกครั้ง

```
# ลบ cluster เก่า
kind delete cluster --name dlt-production
# สร้างใหม่ทั้งหมด
sh ./setup-infra.sh
```

คำเตือน: ข้อมูลใน Database จะยังอยู่ถ้า folder `/mnt/data/` ยังมีอยู่

5.11. รายการบริการและโครงสร้างแอปพลิเคชันภายในระบบ (System Component Inventory)

เพื่อให้การบริหารจัดการทรัพยากรและการติดตามสถานะการดำเนินงานของระบบเป็นไปอย่างเป็นระบบ (Systematic Resource Management) จำเป็นต้องมีการจัดทำรายการแอปพลิเคชันทั้งหมดที่รันอยู่บนโครงสร้างพื้นฐาน Kubernetes ของโครงการ DLT-TMS ตารางต่อไปนี้สรุปรายละเอียดที่สำคัญของแต่ละแอปพลิเคชัน รวมถึงบทบาทหน้าที่หลักและการจัดเก็บข้อมูลเพื่อใช้เป็นข้อมูลอ้างอิงในการตรวจสอบสถานะและการบำรุงรักษาอย่างต่อเนื่อง

5.11.1. Application Services

ตารางที่ 5-2 ส่วนงานหลัก

ชื่อใน ArgoCD	URL ที่เข้าได้	Namespace	หน้าที่
dlt-platform-api	api-tms.dlt.go.th	default	REST API หลัก
dlt-connect-frontend	tms.dlt.go.th	default	หน้าเว็บ
dlt-socket-api	socket-tms.dlt.go.th	default	WebSocket realtime
dlt-datapool	—	default	จัดการ data pool
dlt-position-publisher	—	default	ส่งข้อมูลตำแหน่ง
dlt-event-builder	—	default	สร้าง events

ตารางนี้สรุป "ส่วนงานหลัก" (Core Business Logic) ของระบบ DLT-TMS ที่ติดต่อกับผู้ใช้งานหรือทำหน้าที่ประมวลผลข้อมูลโดยตรง:

- **จุดประสงค์:** เพื่อแสดงว่าในระบบมีกี่ Service ที่รันอยู่ โดยแบ่งตามชื่อที่ตั้งใน ArgoCD, ช่องทางที่ผู้ใช้หรือระบบอื่นเข้าถึง (URL), เนมสเปซที่สังกัด และหน้าที่หลัก
- **สิ่งที่น่าสนใจ:** มีทั้งส่วนที่เปิดให้เข้าถึงผ่าน Browser/API (เช่น dlt-platform-api, dlt-connect-frontend) และส่วนงานเบื้องหลัง (Background Worker) ที่เน้นการประมวลผลข้อมูล (เช่น dlt-datapool, dlt-position-publisher) ซึ่งแสดงถึงสถาปัตยกรรมแบบ Microservices ที่ชัดเจน

5.11.2. Infrastructure Services

ตารางที่ 5-3 ส่วนงานสนับสนุน

ชื่อใน ArgoCD	Namespace	หน้าที่	หมายเหตุ
kafka	kafka	Message queue	3 brokers
mongodb	default	Document database	prune=false
postgres	default	Relational database	prune=false
redis	default	Cache & sessions	prune=false
minio	minio	Object storage	prune=false
ingress	default	Routing rules	—
ingress-controller-rollout	ingress-nginx	NGINX config	—
cluster-issuer	default	SSL issuer	Let's Encrypt
cert-manager	cert-manager	Certificate manager	v1.17.2

ตารางนี้สรุป "ส่วนงานสนับสนุน" (Supporting/Middleware Services) ที่จำเป็นต่อการทำงานของแอปพลิเคชันหลัก:

- **จุดประสงค์:** เพื่อแสดงรายการซอฟต์แวร์สนับสนุนที่ติดตั้งลงบนคลัสเตอร์ ซึ่งมีความสำคัญต่อความเสถียรของระบบ
- **องค์ประกอบหลักที่สำคัญ:**
  - **Data/Messaging Layer:** มีทั้ง kafka (Message Queue), mongodb และ postgres (Databases), redis (Cache) ซึ่งเป็นมาตรฐานสำหรับการสร้างระบบงานที่มีประสิทธิภาพสูง
  - **Storage Layer:** มี minio สำหรับทำ Object Storage

- **Network & Security Layer:** มี ingress สำหรับจัดการทราฟฟิก และ cluster-issuer สำหรับจัดการใบรับรอง SSL (Let's Encrypt) เพื่อความปลอดภัย

- **หมายเหตุ**

5.11.3. Monitoring & Logging

ตารางที่ 5-4 เครื่องมือที่ทีมใช้ในการเฝ้าระวังสุขภาพของระบบ

ชื่อใน ArgoCD	Namespace	เข้าถึงได้จาก	หน้าที่
prometheus	monitoring	port-forward 9090	เก็บ metrics
grafana	logging	port-forward 3000	Dashboard
loki	logging	ผ่าน Grafana	เก็บ logs
promtail	logging	—	ดึง logs จาก pods

ตารางนี้สรุปเครื่องมือที่ทีมใช้ในการ "เฝ้าระวังสุขภาพของระบบ" เพื่อให้ทราบว่าบริการต่างๆ ทำงานปกติหรือไม่:

- **Prometheus:** ทำหน้าที่ "วัดผล" (Metrics) เช่น อัตราการใช้งาน CPU/RAM ของ Pod ต่างๆ
- **Grafana:** คือ "หน้าจอแสดงผล" (Dashboard) ที่ดึงข้อมูลจาก Prometheus มาแสดงให้ผู้ดูแลเห็นเป็นกราฟที่อ่านง่าย
- **Loki:** ทำหน้าที่เป็น "คลังเก็บประวัติการทำงาน" (Logs) เพื่อให้ย้อนกลับมาดูได้ว่าเกิดข้อผิดพลาดอะไรขึ้น
- **Promtail:** เปรียบเสมือน "ตัวส่งข้อมูล" ที่คอยดึง Logs จาก Pods ต่างๆ ส่งไปเก็บไว้ที่ Loki

5.12. การตรวจสอบ Minifests ของระบบ

ผู้พัฒนาสามารถตรวจสอบ Minifest และ Environment ที่ใช้ใน Microservice ได้ที่ Repositories เหล่านี้

ตารางที่ 5-5 ผังการจัดเก็บไฟล์ Config

Repository name	Describe microservice
api-infra	Kubernetes manifests สำหรับ API service
connect-frontend-infra	Kubernetes manifests สำหรับ Frontend
socket-infra	Kubernetes manifests สำหรับ WebSocket API
dlt-datapool-infra	Kubernetes manifests สำหรับ Datapool service
position-publisher-infra	Kubernetes manifests สำหรับ Position Publisher
dlt-eventbuilder-infra	Kubernetes manifests สำหรับ Event Builder

ตารางนี้เป็น "ผังการจัดเก็บไฟล์ Config" ซึ่งเป็นหัวใจสำคัญของ GitOps ที่โครงการนี้ใช้:

- **หลักการ:** ระบบนี้มีการแยกส่วน (Decoupling) อย่างชัดเจน โดยแต่ละ Microservice จะมี **Repository แยกเป็นของตัวเอง** (เช่น api-infra, socket-infra) เพื่อให้ ArgoCD เข้ามาอ่านค่าไฟล์ Manifests ไปสร้างหรืออัปเดตระบบใน Kubernetes
- **ประโยชน์:**
    - **ลดความผิดพลาด:** แก้ไข Config ของ Service หนึ่ง จะไม่กระทบกับ Service อื่น
    - **ง่ายต่อการติดตาม:** ถ้าต้องการรู้ว่าหน้าเว็บ (Frontend) มีการตั้งค่าอย่างไร ก็ไปดูที่ repository connect-frontend-infra ได้ทันที
    - **มาตรฐาน:** แสดงถึงมาตรฐานการทำงานในระดับมืออาชีพที่แยก Code ส่วนงาน (Application Code) ออกจาก Code ส่วนโครงสร้าง (Infrastructure Code) อย่างชัดเจน

6. Microservice

ระบบโครงการพัฒนาระบบเทคโนโลยีสารสนเทศบริหารจัดการขนส่งสินค้าทางถนน ได้จัดทำระบบย่อยๆ เป็นมาประกอบกันเป็นระบบใหญ่ หรือเรียกว่า ไมโครเซอร์วิส (Microservice) โดยจะแบ่งเป็นเซอร์วิสดังนี้

- ระบบหน้าบ้าน (Frontend) โดยจะเรียกว่า: dlt-connect-frontend
- ระบบหลังบ้าน (Backend) โดยจะเรียกว่า: dlt-platform-api
- ระบบส่งข้อมูล Real time (Socket) โดยจะเรียกว่า: dlt-socket-api
- ระบบรับข้อมูล GPS (data pool) โดยจะเรียกว่า: dlt-datapool
- ระบบจัดการกิจกรรม (event builder) โดยจะเรียกว่า: dlt-event-builder
- ระบบจดการตามกำหนดเวลา (Cronjob) โดยจะเรียกว่า: dlt-cronjob

โดยในบทนี้จะกล่าวไปยังระบบ Microservice และ การจัดการ Repository และปัญหาที่อาจจะพบบ่อย เพื่อให้สามารถที่จะเข้าใจและดูแลระบบได้อย่างมีประสิทธิภาพ

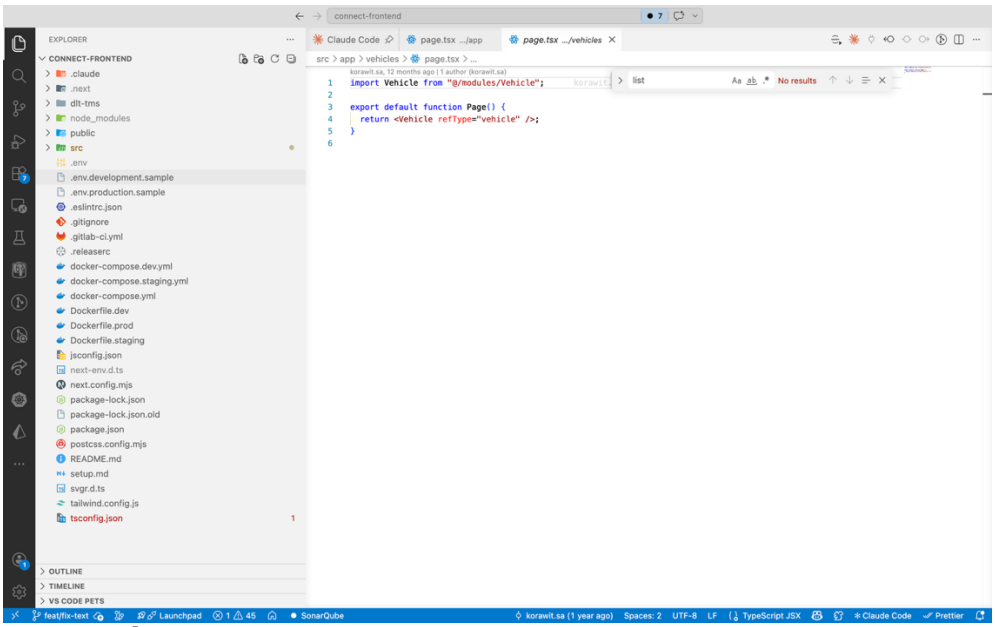
6.1. ระบบหน้าบ้าน (Frontend)

ทางที่ปรึกษาได้ออกแบบระบบหน้าบ้านโดยใช้ระบบ Nextjs ซึ่ง Next.js เป็นเฟรมเวิร์กสำหรับการพัฒนาเว็บแอปพลิเคชันด้วยภาษา JavaScript บนพื้นฐานของ React โดย Next.js จะช่วยให้การสร้างเว็บที่มีความเร็วสูง รองรับการแสดงผลแบบ server-side (SSR) และ static site generation (SSG) เป็นไปได้อย่างง่ายและมีประสิทธิภาพ นอกจากนี้ Next.js ยังมีระบบจัดการ routing อัตโนมัติ การปรับแต่ง SEO และการ deploy ที่สะดวก ทำให้เหมาะกับการสร้างเว็บที่ต้องการความเร็วและประสิทธิภาพ เช่น เว็บหน้าบ้านของระบบไมโครเซอร์วิสในโครงการนี้

โดยระบบหน้าบ้านในโครงการพัฒนาระบบเทคโนโลยีสารสนเทศบริหารจัดการขนส่งสินค้าทางถนน (DLT-TMS) เป็นส่วนสำคัญในการแสดงผลสำหรับให้ผู้ใช้งานได้เข้าใช้ระบบ, กรอกข้อมูลสำคัญ และ แสดงข้อมูลสำคัญในงานขนส่งสินค้าทางถนน

โครงสร้างระบบโดยรวม:

```
Frontend (Next.js) → Backend API → Database
                    ↕️
              WebSocket Server (Real-time)
```



รูปที่ 6-1 ภาพแสดงโครงสร้างของ Repositories Frontend

6.1.1. รายการเทคโนโลยีหลัก (Technology Stack)

ตารางที่ 6-1 ตารางเทคโนโลยีที่ใช้ในบริการหน้าบ้าน

Layer	Technology
Framework	Next.js 14, React 18, TypeScript 5
State Management	React Context API, TanStack React Query 5
UI	Ant Design 5, Tailwind CSS, SASS
Maps	Google Maps, HERE Maps, Longdo Maps
Charts	ECharts, Chart.js, AmCharts 4/5
Real-time	Socket.IO 4
Auth	NextAuth.js (Google OAuth, LINE OAuth)
i18n	next-intl, i18next
Analytics	Google Analytics 4

ตารางนี้สรุปองค์ประกอบทางเทคโนโลยีทั้งหมดที่ใช้ในการพัฒนาระบบหน้าบ้าน เพื่อให้เห็นภาพรวมของเครื่องมือและไลบรารีที่เลือกใช้ในการขับเคลื่อนประสิทธิภาพของระบบ

6.1.2. รายการตัวแปรสภาพแวดล้อม (Environment Variables)

ตารางที่ 6-2 ค่าคอนฟิกูเรชันที่จำเป็นสำหรับระบบ

Variable	คำอธิบาย
NEXT_PUBLIC_ENDPOINT_URL	Backend API base URL
NEXT_PUBLIC_SOCKET_API_URL	WebSocket server URL
NEXT_PUBLIC_GOOGLE_MAP_API_KEY	Google Maps API key
NEXT_PUBLIC_LONGDO_API_KEY	Longdo Maps API key
GOOGLE_CLIENT_ID	Google OAuth client ID
GOOGLE_CLIENT_SECRET	Google OAuth secret
NEXTAUTH_URL	Frontend URL (สำหรับ NextAuth)
NEXTAUTH_SECRET	Session secret key
NEXT_PUBLIC_GOOGLE_ANALYTICS_ID	Google Analytics ID

ตารางนี้แสดงรายการค่าคอนฟิกูเรชันที่จำเป็นสำหรับระบบ เพื่อใช้ในการเชื่อมต่อกับบริการภายนอกและการตั้งค่าความปลอดภัยของแอปพลิเคชันอย่างเป็นระบบ

6.1.3. ขั้นตอนการนำงานขึ้นระบบ

ทางที่ปรึกษาได้ออกแบบ Automation Deployment เพื่อช่วยให้นักพัฒนาสามารถนำซอฟแวร์ขึ้นระบบได้ง่ายโดยมีลำดับการทำงานดังนี้

การ Deploy

ระบบใช้ **GitLab CI/CD** แบบ GitOps โดยมี 2 repository แยกกัน:

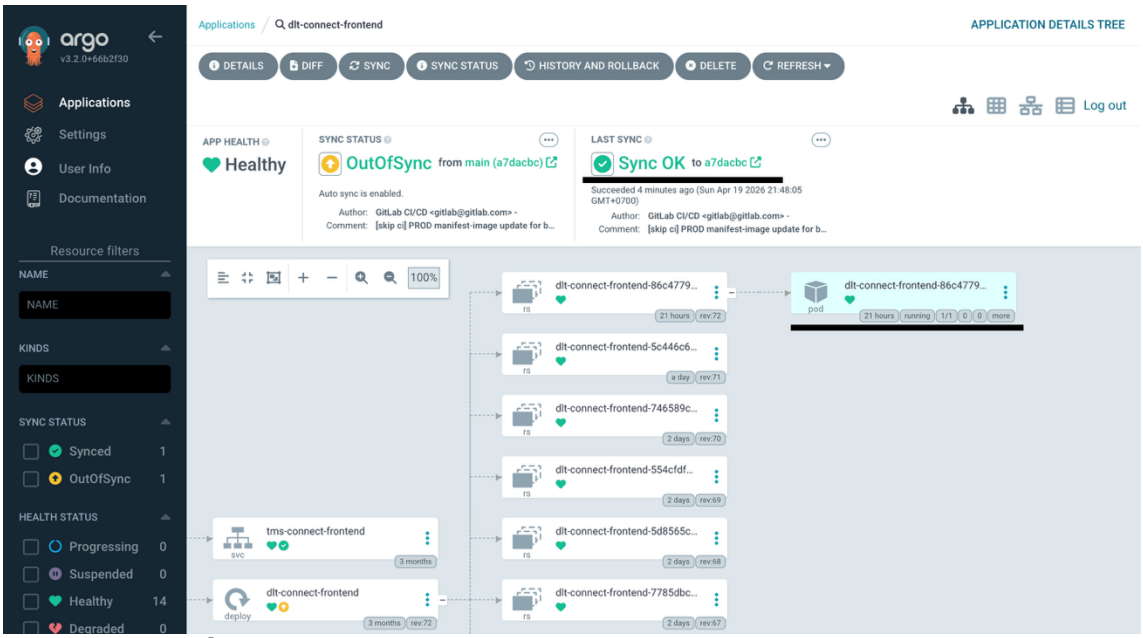
ตารางที่ 6-3 รูปแบบของ Reposition

Repository	หน้าที่
connect-frontend	Source code ของ Frontend
connect-frontend-infra	Kubernetes manifests (Kustomize) สำหรับ Staging และ Production

เมื่อ CI build Docker image เสร็จแล้ว จะ push image ไปยัง GitLab Container Registry แล้วไป อัปเดต image tag ใน connect-frontend-infra เพื่อให้ Kubernetes ดึง image ใหม่ขึ้นมารันผ่า argocd app

ภาพรวม Pipeline

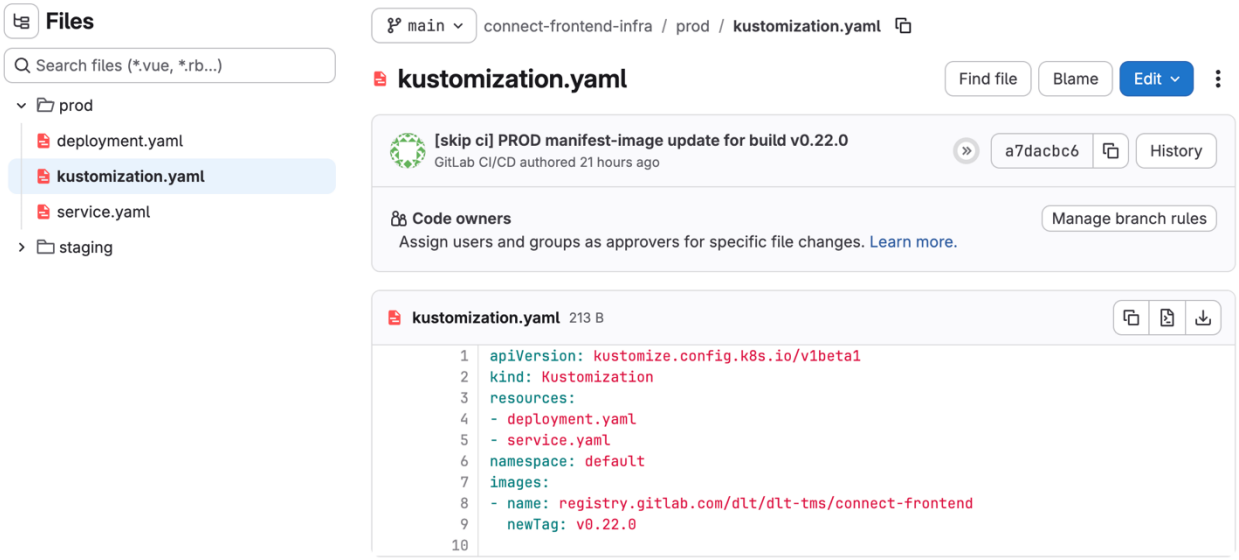
Push Code / Tag เข้า main branch with commit prefix like “feat: deploy fix vehicle register”
|
▼
[prod: build]
Build Docker Image from main branch
Push ไปยัง GitLab Registry
|
▼
[prod: deploy]
Clone connect-frontend-infra
kustomize edit set image → <new tag> ปรับ docker image version
Push commit กลับไปที่ infra repo
|
▼
[Stage: notification]
แจ้งผล Success / Failure



รูปที่ 6-2 ภาพแสดงการทำงานของระบบ Frontend ตามปกติ

ภาพดังกล่าวแสดงการตรวจสอบสถานะการทำงานของแอปพลิเคชัน dlt-connect-frontend ผ่านระบบ ArgoCD ซึ่งสะท้อนถึงการปฏิบัติงานจริงตามแนวทาง **GitOps** โดยมีรายละเอียดเชิงเทคนิคที่สำคัญดังนี้:

- **สถานะสุขภาพของระบบ (App Health):** แสดงสถานะ "Healthy" ซึ่งยืนยันว่าหน่วยประมวลผล (Pod) และองค์ประกอบทั้งหมดของแอปพลิเคชันกำลังทำงานได้อย่างถูกต้องตามที่กำหนด
- **การจัดการสถานะการซิงโครไนซ์ (Sync Status):** แสดงข้อมูลล่าสุดของการซิงค์ระบบกับ Repository หลัก (Main Branch) โดยระบบจะเปรียบเทียบสถานะจริงในคลัสเตอร์ (Live State) กับสิ่งที่ระบุไว้ใน Git (Desired State) แบบอัตโนมัติ ช่วยให้มั่นใจได้ว่าระบบมีการอัปเดตเวอร์ชันถูกต้องและสอดคล้องกันเสมอ
- **โครงสร้างต้นไม้ของทรัพยากร (Details Tree):** แสดงลำดับชั้นความสัมพันธ์ของทรัพยากร (Resource Hierarchy) ตั้งแต่ระดับ Service, Deployment, ReplicaSet ไปจนถึงระดับ Pod ช่วยให้ผู้ดูแลระบบสามารถตรวจสอบความเชื่อมโยงและค้นหาจุดที่เกิดข้อผิดพลาด (Root Cause Analysis) ได้อย่างรวดเร็วในกรณีที่ระบบเกิดปัญหา
- **ประวัติและการย้อนกลับ (History and Rollback):** ระบบมีการเก็บประวัติการเปลี่ยนแปลง (Revision History) ซึ่งช่วยให้ทีมผู้ดูแลสามารถดำเนินการ Rollback กลับไปสู่เวอร์ชันก่อนหน้า (Last Known Good Configuration) ได้ทันทีหากการอัปเดตระบบในปัจจุบันเกิดความไม่เสถียร



รูปที่ 6-3 Gitlab runner update image ที่ connect-frontend-infra

ภาพแสดงไฟล์ kustomization.yaml ซึ่งทำหน้าที่เป็นตัวจัดการทรัพยากร (Resource Orchestration) ในกระบวนการ Deploy ของแอปพลิเคชัน โดยไฟล์นี้ระบุการเชื่อมโยงไปยัง deployment.yaml และ service.yaml รวมถึงทำหน้าที่สำคัญในการทำ **Image Tagging (v0.22.0)** เพื่อกำหนดเวอร์ชันของ Docker Image ที่ต้องการติดตั้งลงบน Kubernetes Cluster ซึ่งช่วยให้กระบวนการอัปเดตเวอร์ชันซอฟต์แวร์มีความเป็นอัตโนมัติ แม่นยำ และสามารถตรวจสอบย้อนหลังได้ตามมาตรฐาน GitOps

6.1.4. ปัญหาที่พบบ่อย

Auth / Login

ตารางที่ 6-4 ปัญหาที่พบบ่อย

อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
Login ไม่ได้	NEXTAUTH_SECRET ไม่ถูกต้อง	ตรวจสอบค่าใน .env
Google/LINE OAuth error	Client ID/Secret ไม่ตรง	ตรวจสอบ OAuth credentials
Session หมดเร็วผิดปกติ	NEXTAUTH_URL ไม่ตรง domain	ตั้งค่าให้ตรงกับ URL จริง

Maps ไม่แสดง

ตารางที่ 6-5 ปัญหาที่พบบ่อย

อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
Google Maps ว่างเปล่า	API Key ไม่ถูกต้อง หรือ domain ไม่ได้ whitelist	ตรวจสอบ Google Cloud Console
HERE Maps error	API Key หมดอายุหรือ quota เต็ม	ตรวจสอบ HERE Developer Portal

Real-time ไม่อัปเดต

ตารางที่ 6-6 ปัญหาที่พบบ่อย

อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
ข้อมูลไม่อัปเดต live	WebSocket disconnect	ตรวจสอบ `NEXT_PUBLIC_SOCKET_API_URL`

Build / Deploy

ตารางที่ 6-7 ปัญหาที่พบบ่อย

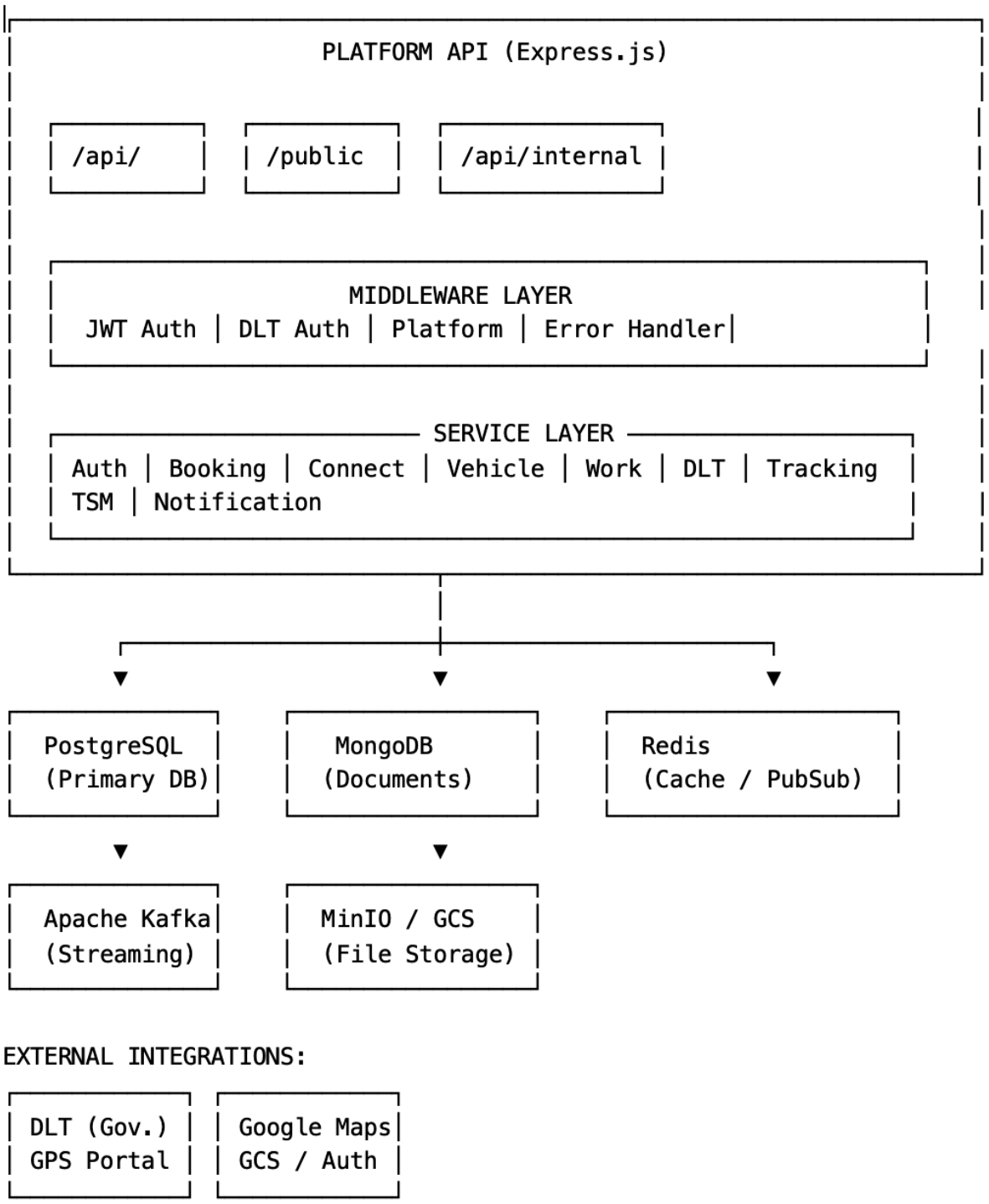
อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
`npm install` fail	Peer dependency conflict	ใช้ `--legacy-peer-deps`
Build error TypeScript	Type mismatch	รัน `npm run lint` ก่อน build

6.2. ระบบหลังบ้าน (หลังบ้าน)

ระบบหลังบ้านของเราใช้ Node.js และ Express ในการพัฒนา API routing โดย Node.js เป็นแพลตฟอร์มที่ช่วยให้สามารถสร้างเซิร์ฟเวอร์และจัดการกับการรับส่งข้อมูลได้อย่างมีประสิทธิภาพ ส่วน Express เป็น framework ที่ช่วยให้การสร้างและจัดการ routing ของ API เป็นเรื่องง่ายและเป็นระบบมากขึ้น สามารถกำหนดเส้นทางแต่ละ API ได้อย่างชัดเจน ทำให้การเชื่อมต่อระหว่าง client และ server มีความรวดเร็วและปลอดภัย เหมาะกับการใช้งานในระบบที่ต้องการความเสถียรและขยายตัวได้ในอนาคต

โดยระบบปฏิบัติการหลังบ้านมีหน้าที่ในการบันทึกข้อมูลลงฐานข้อมูล, ประมวลผลข้อมูลและนำมาแสดงผล, ดึงข้อมูลจากฐานข้อมูลมาแสดงผล เป็นต้น
ภายในระบบ Backend Platform API นั้นจะใช้การจัดการโค้ดแบบ MVC (model view controller)

โครงสร้าง MVC (Model-View-Controller) ในระบบ Backend Platform API แบ่งออกเป็น 3 ส่วนสำคัญ ได้แก่ Model, View และ Controller ซึ่งแต่ละส่วนมีหน้าที่เฉพาะเจาะจงและทำงานร่วมกันอย่างเป็นระบบ โดย Model ทำหน้าที่จัดการข้อมูล รวมถึงการเชื่อมต่อและสื่อสารกับฐานข้อมูลโดยตรง ส่วน Controller ทำหน้าที่เป็นตัวกลางในการรับคำขอจาก client ผ่านเส้นทาง routing เช่น /api/ , /public , /api/internal จากนั้นประมวลผลคำขอเหล่านั้น และเรียกใช้ Model เพื่อจัดการข้อมูล ก่อนจะส่งผลลัพธ์ไปยัง View ซึ่ง View มีหน้าที่นำข้อมูลที่ผ่านการประมวลผลไปจัดรูปแบบและส่งกลับไปให้ client ตามรูปแบบที่กำหนดไว้อย่างเหมาะสม
โครงสร้างนี้สะท้อนถึงภาพรวมของระบบที่มีการแบ่ง routing ของ API ออกเป็นส่วนต่าง ๆ อย่างชัดเจน พร้อมทั้งมีชั้น Middleware ที่ทำหน้าที่ตั้งแต่การจัดการด้านความปลอดภัย การตรวจสอบสิทธิ์ผู้ใช้ (Authentication) ไปจนถึงการจัดการข้อผิดพลาดต่าง ๆ ก่อนที่คำขอจะถูกส่งต่อไปยัง Controller เพื่อดำเนินการต่อ ดังนั้นการใช้แนวคิด MVC นี้ ช่วยให้ระบบมีความเป็นระเบียบ สามารถแยกความรับผิดชอบของแต่ละส่วนอย่างชัดเจน ทำให้สะดวกต่อการขยาย ปรับปรุง และดูแลรักษาได้ง่ายในอนาคต



รูปที่ 6-4 ภาพองค์ประกอบโดยรวมของระบบหลังบ้าน

6.2.1. Tech Stack

Table 6-8 Tech Stack

Layer	Technology	รายละเอียด
Runtime	Node.js 18 (LTS)	ใช้ bullseye-slim base image
Web Framework	Express.js 4.18.1	REST API, Middleware pipeline
Primary Database	PostgreSQL + Postgis	Relational data, Sequelize ORM 6.19.0
Document Database	MongoDB	Mongoose 6.8.4, ข้อมูล NoSQL
Cache / PubSub	Redis	Session cache, Socket.io emitter
Message Queue	Apache Kafka	Event streaming, kafka-node 5.0.0
Object Storage	MinIO	ไฟล์เอกสาร, รูปภาพ
Authentication	JWT RS256	jsonwebtoken 9.0.2, bcryptjs 2.4.3
Real-time	Socket.io + Redis Emitter	@socket.io/redis-emitter 5.0.0
Geospatial	Turf.js 7.2.0	คำนวณ geofence, ระยะทาง
PDF Generation	pdf-lib + pdfkit + pdfmake	สร้างเอกสาร PDF
Excel/CSV	xlsx + json2csv	Export รายงาน
Image Processing	Sharp 0.32.0	ปรับขนาดและแปลงรูป
HTTP Client	Built-in + Google SDK	เรียก external APIs
Observability	OpenTelemetry 0.25.0	Distributed tracing → Jaeger
Logging	Morgan	HTTP request logging

6.2.2. โมดูลหลักของระบบ

ตารางที่ 6-9 API v1 Modules Core หลัก

โมดูล	Route Prefix	ฟีเจอร์หลัก
Auth	/api/v1/auth	Login, logout, refresh token, JWT management
Booking	/api/v1/booking	งานขนส่ง, คำสั่งซื้อ
Vehicle	/api/v1/vehicle	ลงทะเบียนรถ,
DLT Auth	/api/v1/dlt/auth	Authentication กับระบบ DLT
TSM	/api/v1/tsm	Thai Transportation Service integration
Upload	/api/v1/upload	อัปโหลดไฟล์/รูปภาพ
Public	/public/api/v1	Endpoints ไม่ต้อง authenticate
Place	/api/v1/place	สถานที่

ตารางที่ 6-10 API v2 Modules Integation API

โมดูล	Route Prefix	ฟีเจอร์หลัก
DLT Manage	/api/v2/dltManage	จัดการข้อมูล DLT, ใบอนุญาต, ตรวจสอบ
Driving License	/api/v2/drivingLicense	ตรวจสอบและจัดการใบขับขี่
Invite	/api/v2/invite	ระบบเชิญผู้ใช้เข้าองค์กร
Provinces	/api/v2/provinces	ข้อมูลจังหวัด/อำเภอ/ตำบลไทย
TMS	/api/v2/tms	Transportation Management System features

ตารางที่ 6-11 Service Layer Modules

Service	ไฟล์หลัก	ฟีเจอร์
Auth	`src/service/auth/`	JWT issuance, token validation, session
Booking	`src/service/booking/` (123KB)	Core booking logic, billing, invoice
Connect	`src/service/connect/` (1.3MB)	B2B connections, dispatch, analytics
Vehicle	`src/service/vehicle/`	Fleet management, maintenance
Backoffice	`src/service/backoffice/` (47 files)	Admin operations
DLT	`src/service/dlt/` + `src/serviceV2/dlt-gps-service/`	DLT integration
TSM	`src/service/tsm.js` (21KB)	Thai transport service
Report	`src/service/report/`	Dashboard, analytics, exports
Notification	`src/utils/notification.js`	Nofitication
Storage	`src/service/minio.js` + `src/service/storage.js`	File storage

6.2.3. Environment Variable

บทนี้จะกล่าวถึงตัวแปรต่างๆ ที่เป็นตัวแปลหลักของระบบใช้ในการเชื่อมต่อระบบอื่นๆ

Application

Variable	คำอธิบาย
APP_PORT	Port ที่ Express server ฟัง (เช่น 4004)
NODE_ENV	สภาพแวดล้อม: development หรือ production

Database (PostgreSQL)

Variable	คำอธิบาย
DB_HOST	Host ของ PostgreSQL server
DB_PORT	Port (ปกติ 5432)
DB_USER	ชื่อผู้ใช้ PostgreSQL
DB_PASSWORD	รหัสผ่าน PostgreSQL
DB_NAME	ชื่อฐานข้อมูล

MongoDB

Variable	คำอธิบาย
MONGO_URI	MongoDB connection URI (เช่น mongodb://localhost:27017)
MONGO_DB	ชื่อฐานข้อมูล MongoDB

Redis

Variable	คำอธิบาย
REDIS_HOST	Host ของ Redis server
REDIS_PORT	Port (ปกติ 6379)
REDIS_AUTH	Password สำหรับ Redis (ถ้ามี)

Kafka

Variable	คำอธิบาย
KAFKA_BROKERS	รายการ broker (เช่น localhost:9094)

Object Storage (MinIO)

Variable	คำอธิบาย
MINIO_ENDPOINT	Host ของ MinIO server
MINIO_PORT	Port (ปกติ 9000)
MINIO_ACCESS_KEY	Access key สำหรับ MinIO
MINIO_SECRET_KEY	Secret key สำหรับ MinIO

Authentication

Variable	คำอธิบาย
PRIVATE_KEY_FILE	path ของ RSA private key (.pem หรือ .key)
PUBLIC_KEY_FILE	path ของ RSA public key

DLT (กรมการขนส่งทางบก)

Variable	คำอธิบาย
DLT_GPS_USERNAME	Username สำหรับ login ระบบ DLT GPS
DLT_GPS_PASSWORD	Password สำหรับ DLT GPS
DLT_GPS_AUTO_LOGIN	เปิด/ปิด auto-login (true/false)
DLT_GPS_GRAPHQL_AUTH_ENDPOINT	Endpoint สำหรับ authenticate กับ DLT
DLT_GPS_GRAPHQL_ENDPOINT	Endpoint หลักของ DLT GraphQL API

Static Resources ภาพต่าง ๆ

Variable	คำอธิบาย
RESOURCE_BASE_URL	Base URL สำหรับ static resource files

6.2.4. ขั้นตอนการนำงานขึ้นระบบ

ทางที่ปรึกษาได้ออกแบบ Automation Deployment เพื่อช่วยให้นักพัฒนาสามารถนำซอฟแวร์ขึ้นระบบได้ง่ายโดยมีลำดับการทำงานดังนี้

Merge Merge Request ไป main

GitLab → Project → Merge Requests → เลือก MR ที่ approved → Merge

Pipeline บน main จะรัน job make-release ในทันที

make-release สร้าง Git Tag อัตโนมัติ

make-release ใช้ semantic-release วิเคราะห์ commit messages ตาม Conventional

Commit prefix	ผลลัพธ์	ตัวอย่าง tag
fix:	Patch bump (x.x.+1)	v1.2.1
feat:	Minor bump (x.+1.0)	v1.3.0
feat!: / BREAKING CHANGE:	Major bump (+1.0.0)	v2.0.0

ส่วนของ plugins ที่ใช้งาน มีดังนี้: @semantic-release/changelog – ปลั๊กอินนี้จะช่วยอัปเดตไฟล์ CHANGELOG.md ให้อัตโนมัติ โดยจะเพิ่มรายการการเปลี่ยนแปลงของแต่ละเวอร์ชันใหม่เข้าไป @semantic-release/git – ใช้สำหรับ commit ไฟล์ที่ถูกเปลี่ยน (เช่น CHANGELOG.md) และ push tag เวอร์ชันใหม่กลับไปที่ repository @semantic-release/gitlab – ปลั๊กอินนี้ใช้สร้าง GitLab Release ให้โดยอัตโนมัติหลังจากที่ปล่อยเวอร์ชันใหม่

หมายเหตุ: การจะให้ระบบ push tag กลับไปที่ repository ได้ จำเป็นต้องมี GITLAB_TOKEN ตั้งค่าไว้ใน CI/CD Variables ด้วย เพื่อให้ปลั๊กอินสามารถเข้าถึงสิทธิ์ที่ต้องการใน GitLab

Gitlab runner build-prod สร้าง Docker Image

หลังจาก tag ถูกสร้าง pipeline ใหม่จะเริ่มรันสำหรับ tag นั้น

```
# สิ่งที่ build-prod ทำ (จาก .gitlab-ci.yml):
docker login -u gitlab-ci-token --password-stdin $CI_REGISTRY
docker pull $CI_REGISTRY_IMAGE:latest          # ดึง cache
docker build \
  --cache-from $CI_REGISTRY_IMAGE:latest \
  --tag $CI_REGISTRY_IMAGE:$CI_COMMIT_TAG \    # เช่น :v1.2.0
  --tag $CI_REGISTRY_IMAGE:latest .
docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_TAG
docker push $CI_REGISTRY_IMAGE:latest
```

Image จะถูก push ไปยัง **GitLab Container Registry** ของ project

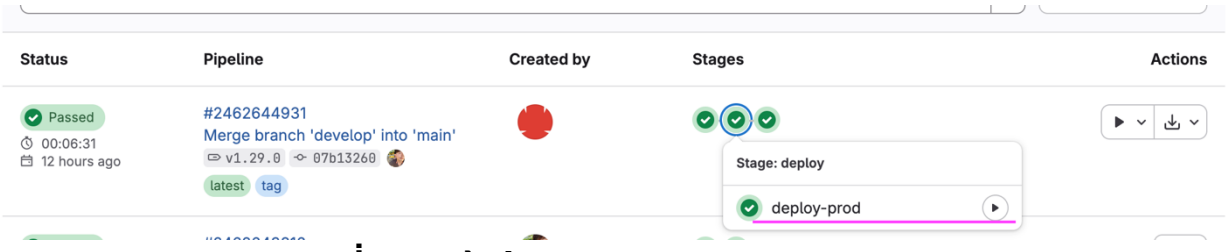
กด Manual Trigger สำหรับ deploy-prod

สำคัญ: job นี้ไม่รันอัตโนมัติ ต้องกดเองทุกครั้ง (when: manual)

วิธีกด:

1. เปิด GitLab → Project → CI/CD → Pipelines
2. หา Pipeline ที่ trigger จาก tag ใหม่ (มีไอคอนป้าย tag สีน้ำเงิน)
3. คลิกเข้าไปดู pipeline detail
4. หา job deploy-prod ที่มีไอคอน ▶ **(Play)** สีส้ม
5. คลิก "Trigger this manual action"

Pipeline จะเริ่มรัน deploy-prod ทันที



รูปที่ 6-5 หน้าต่างการกด deploy บน gitlab

ภาพแสดงการติดตามสถานะการส่งมอบซอฟต์แวร์แบบอัตโนมัติ โดยระบุถึง Pipeline ล่าสุดที่ทำงานสำเร็จ (Status: Passed) ซึ่งมีการผสานรวมโค้ดจากสาขา develop เข้าสู่สาขาหลัก (main) พร้อมแสดงการทำงานของขั้นตอนต่างๆ (Stages) รวมถึงขั้นตอนสำคัญคือ deploy-prod ซึ่งเป็นการยืนยันว่าซอฟต์แวร์ผ่านกระบวนการทดสอบและพร้อมสำหรับการติดตั้งบนสภาพแวดล้อมใช้งานจริงโดยอัตโนมัติ

deploy-prod อัปเดต Kubernetes Manifest

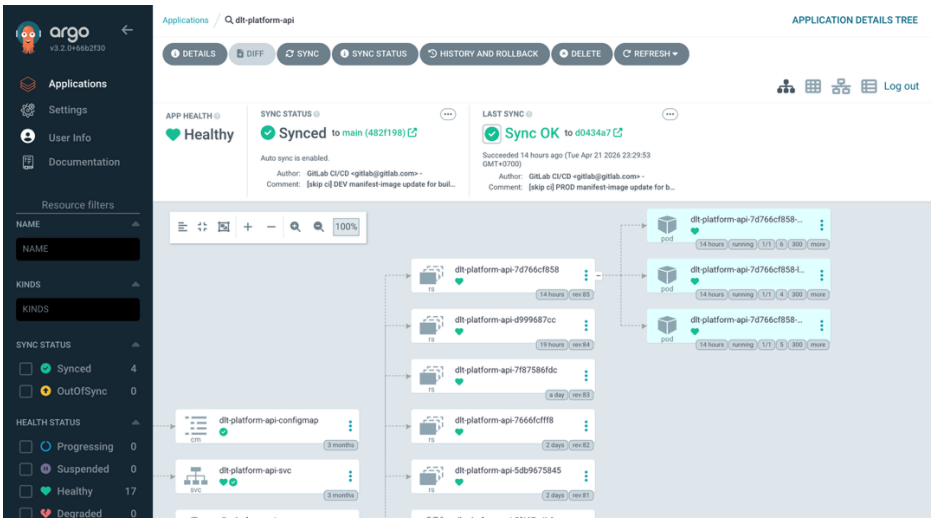
```
# สิ่งที่ deploy-prod ทำ (จาก .gitlab-ci.yml):
git clone https://${USERNAME}:${PAT_TOKEN}@gitlab.com/waylar/dlt-tms/api-infra.git
git checkout -B main
cd api-infra/prod
kustomize edit set image $CI_REGISTRY_IMAGE:$BUILD_VERSION
# → แก้ไข kustomization.yaml ให้ชี้ image ใหม่
git commit -am "[skip ci] PROD manifest-image update for build $BUILD_VERSION"
git push origin main
```

ไฟล์ที่ถูกอัปเดต: `api-infra/prod/kustomization.yaml`

ArgoCD

เมื่อ `api-infra/prod` มีการ commit ใหม่ ArgoCD ที่ monitor repo จะ:

1. ตรวจพบการเปลี่ยนแปลง manifest
2. Apply Kubernetes deployment ใหม่
3. Rolling update pods ทีละตัว
4. ตรวจสอบ health check ก่อน complete



รูปที่ 6-6 หน้าต่าง Argocd ของระบบหลังบ้าน

ภาพดังกล่าวแสดงการตรวจสอบสถานะการทำงานของแอปพลิเคชัน dlt-platform-api ผ่านระบบ ArgoCD ซึ่งสะท้อนถึงการปฏิบัติงานจริงตามแนวทาง **GitOps** โดยมีรายละเอียดเชิงเทคนิคที่สำคัญดังนี้:

- **สถานะสุขภาพของระบบ (App Health):** แสดงสถานะ "Healthy" ซึ่งยืนยันว่าหน่วยประมวลผล (Pod) และองค์ประกอบทั้งหมดของแอปพลิเคชันกำลังทำงานได้อย่างถูกต้องตามที่กำหนด
- **การจัดการสถานะการซิงโครไนซ์ (Sync Status):** แสดงสถานะ "Synced" ซึ่งยืนยันว่าสภาวะของระบบในคลัสเตอร์ (Live State) มีความสอดคล้องสมบูรณ์กับสิ่งที่ระบุไว้ใน Git Repository (Desired State) แบบอัตโนมัติ
- **โครงสร้างต้นไม้ของทรัพยากร (Details Tree):** แสดงลำดับชั้นความสัมพันธ์ของทรัพยากร (Resource Hierarchy) ตั้งแต่ระดับ Service, ConfigMap ไปจนถึงระดับ Deployment และ ReplicaSet ซึ่งช่วยให้ผู้ดูแลระบบสามารถตรวจสอบความเชื่อมโยงและค้นหาจุดที่เกิดข้อผิดพลาดได้อย่างรวดเร็ว

6.2.5. ปัญหาที่พบบ่อย

Authentication/JWT

ตารางที่ 6-12 ปัญหาที่พบบ่อย

อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
401 Unauthorized ทุก request	JWT key files ไม่ถูกต้องหรือหายไป	ตรวจสอบ PRIVATE_KEY_FILE และ PUBLIC_KEY_FILE ใน .env ว่าชี้ถูกไฟล์
Token หมดอายุเร็วผิดปกติ	Timezone ของ server ไม่ตรงกัน	ตรวจสอบ timezone server และ TZ environment variable
Invalid signature error	ใช้ key ผิดคู่ (private/public ไม่ match)	สร้าง key pair ใหม่ด้วย openssl genrsa และ update .env
DLT login ไม่ได้	DLT_GPS credentials หมดอายุหรือผิด	อัปเดต DLT_GPS_USERNAME/DLT_GPS_PASSWORD และ restart

Database (PostgreSQL)

ตารางที่ 6-13 ปัญหาที่พบบ่อย

อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
SequelizeConnectionError	DB_HOST/PORT/USER/PASSWORD ผิด หรือ PostgreSQL ไม่ได้ run	ตรวจสอบค่า env และ pg_isready -h DB_HOST -p DB_PORT
relation does not exist	Migration ยังไม่ได้ run	รัน npm run migrate-up
Migration ค้าง/timeout	Migration ขนาดใหญ่บน production data	ตรวจสอบ lock ด้วย SELECT * FROM pg_locks และปรึกษา developer
too many connections	Connection pool เต็ม	ตรวจสอบ max_connections ของ PostgreSQL และ Sequelize pool config

Database (MongoDB)

ตารางที่ 6-14 ปัญหาที่พบบ่อย

อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
MongoNetworkError	MongoDB ไม่ได้ run หรือ MONGO_URI ผิด	ตรวจสอบ MONGO_URI และ mongosh $MONGO_URI

Redis

ตารางที่ 6-15 ปัญหาที่พบบ่อย

อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
Redis connection refused	Redis ไม่ run หรือ REDIS_HOST ผิด	ตรวจสอบค่า env และรัน redis-cli -h REDIS_HOST ping

Kafka / Event Streaming

ตารางที่ 6-16 ปัญหาที่พบบ่อย

อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
Event ไม่ถูก consume	Kafka broker ไม่ run หรือ topic ไม่มี	ตรวจสอบ broker และ consumer group offset
Kafka connect timeout	`KAFKA_BROKERS` env ผิด	แก้ไข env และ restart service

File Storage (MinIO)

ตารางที่ 6-17 ปัญหาที่พบบ่อย

อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
Upload ไฟล์ไม่ได้	MinIO ไม่ run หรือ credentials ผิด	ตรวจสอบ `MINIO_ENDPOINT`/`MINIO_ACCESS_KEY`/`MINIO_SECRET_KEY`
ไฟล์หาย / 404	Bucket ไม่มี หรือ path ผิด	ตรวจสอบ bucket ผ่าน MinIO console ที่ `http://MINIO_ENDPOINT:9001`

Build / Deploy

ตารางที่ 6-18 ปัญหาที่พบบ่อย

อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
Docker build ล้มเหลว	npm install ไม่สำเร็จในขั้น builder	ตรวจสอบ Node version ใน Dockerfile กับ package.json engines
Out of memory	Heap ไม่พอสำหรับ PDF generation	Production start ใช้ `--max-old-space-size=8192` — ตรวจสอบ `npm start` script
Migration fail บน deploy	DB connection ไม่พร้อม ขณะ init	ตรวจสอบ init container / health check ก่อน run migration

DLT Integration

ตารางที่ 6-19 ปัญหาที่พบบ่อย

อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
DLT GPS data ไม่ update	Auto-login ล้มเหลวหรือ session หมด	ตรวจสอบ DLT_GPS_AUTO_LOGIN=true และ credentials ยังใช้ได้
SOAP service error	DLT SOAP endpoint เปลี่ยนหรือ down	ตรวจสอบ endpoint ใน config/config.js และ DLT announcements

6.3. ระบบส่งข้อมูล Real time (Socket)

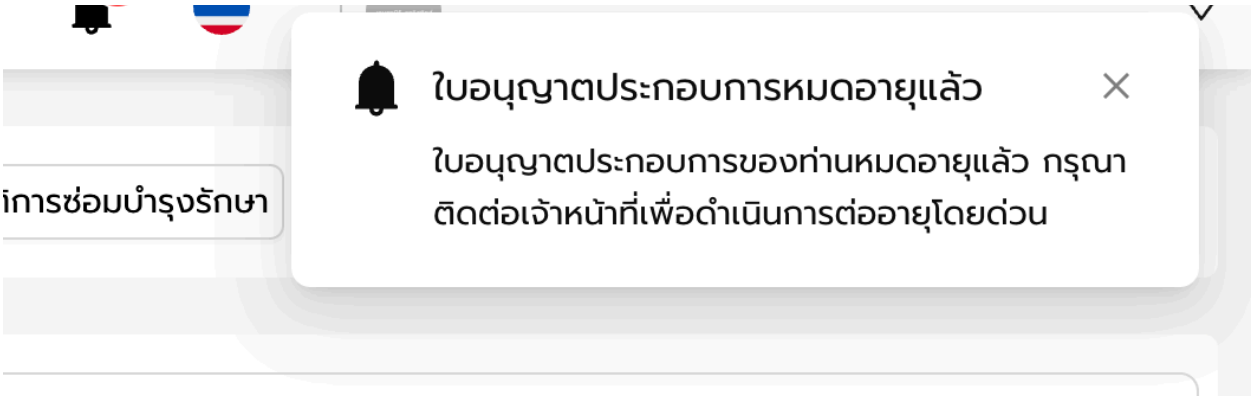
ระบบ **Real-time WebSocket** ของแพลตฟอร์ม DLT มีบทบาทเป็นศูนย์กลางการสื่อสารแบบเรียลไทม์ระหว่าง **client** และระบบหลังบ้าน (backend) โดยใช้ **Socket Service** เป็นกลไกหลักในการเชื่อมต่อแบบ persistent connection ซึ่งรองรับการรับ–ส่งข้อมูลได้ทันที ไม่ต้องร้องขอใหม่ทุกครั้งเหมือน HTTP แบบเดิม ทำให้สามารถอัปเดตข้อมูลได้อย่างรวดเร็ว มีความหน่วงต่ำ และสื่อสารแบบสองทิศทางอย่างมีประสิทธิภาพ

ในการทำงานจริง ระบบ DLT จะรับข้อมูลจาก **Kafka** ซึ่งเป็น message broker ที่ช่วยจัดการข้อมูลแบบ event-driven เช่น **GPS tracking**, **notification**, และ **report** ข้อมูลเหล่านี้จะถูกส่งมายัง Socket Service และ **broadcast** ไปยัง client ที่เชื่อมต่ออยู่ผ่าน **Socket.IO** โดยตรง ลูกค้าจะได้รับข้อมูลแจ้งเตือนแบบเรียลไทม์ เช่น ตำแหน่งรถขนส่ง, การแจ้งเตือนเหตุการณ์สำคัญ และรายงานสถานะต่าง ๆ ที่เกิดขึ้นในระบบ

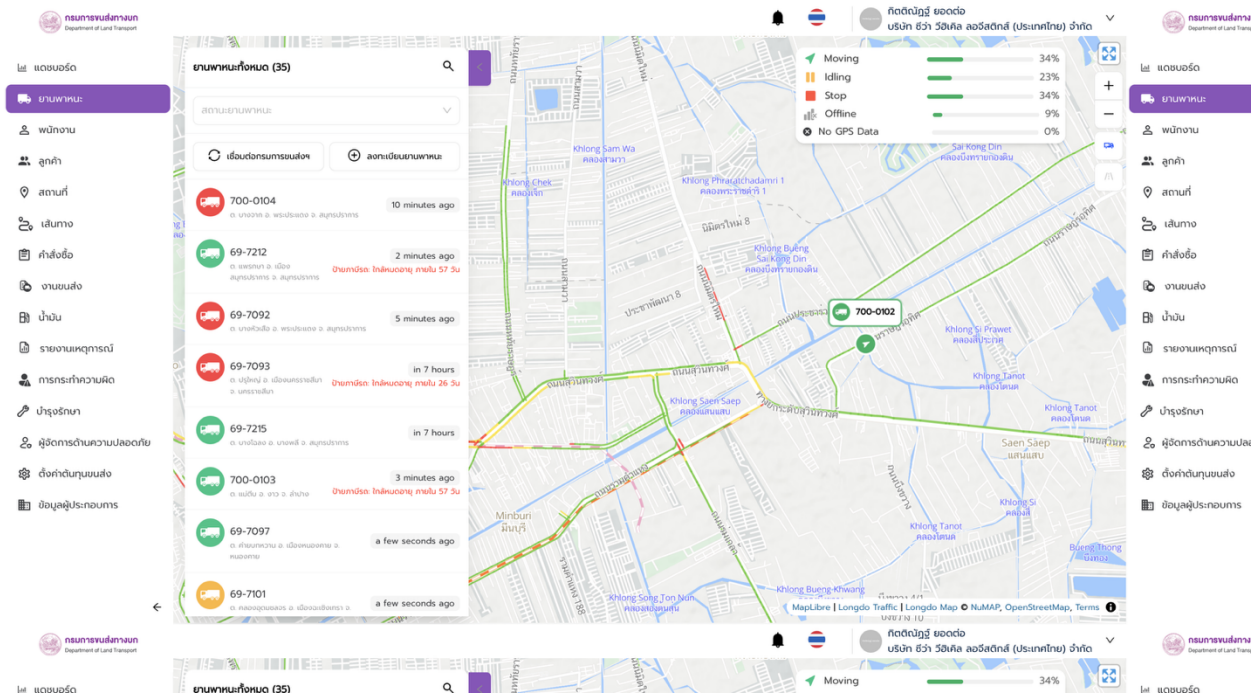
ระบบนี้เหมาะสำหรับการแจ้งเตือนแบบ **realtime notification** ซึ่งช่วยให้ผู้ใช้งานได้รับข้อมูลทันทีที่มีการเปลี่ยนแปลงหรือเกิดเหตุการณ์สำคัญ เช่น การเปลี่ยนตำแหน่ง GPS ของรถขนส่ง, การแจ้งเตือนสถานะ หรือการซิงโครไนซ์ข้อมูลระหว่างผู้ใช้งานกับระบบ backend เพิ่มประสิทธิภาพในการบริหารจัดการขนส่งสินค้าทางถนนและตอบสนองต่อเหตุการณ์ต่าง ๆ ได้อย่างรวดเร็วและถูกต้อง

นอกจากนี้ ระบบยังสามารถขยาย (scalable) เพื่อรองรับผู้ใช้งานจำนวนมากได้ ด้วยการใช้เทคโนโลยีเสริม เช่น message broker และ load balancer เพื่อให้การสื่อสารแบบ real-time มีความเสถียรและต่อเนื่องสำหรับทุก client ที่เชื่อมต่ออยู่

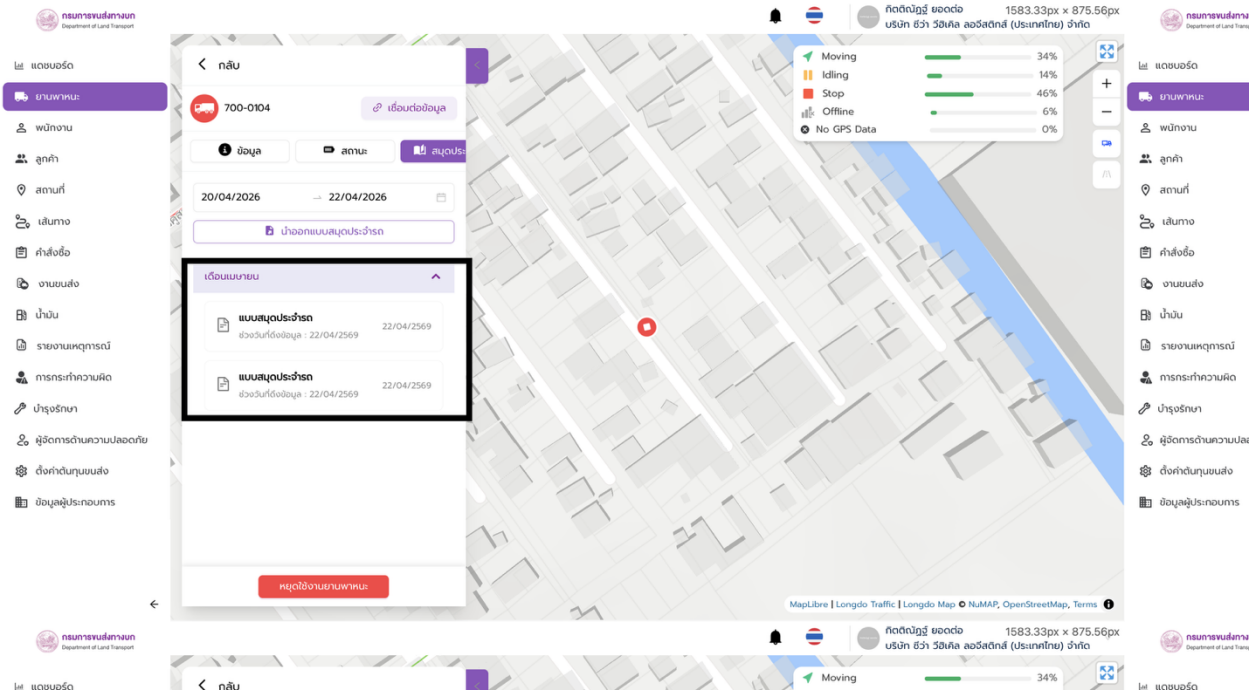
หน้าที่หลังกของ Socket Service คือจะทำหน้าที่เป็น Kafka Consumer โดยการรับข้อความที่จะส่งแจ้งเตือน เมื่อได้รับข้อความก็จะส่งแจ้งเตือนไปยัง Client ต่าง ๆ และอัพเดทตำแหน่งยานภาพหนะบนแผนที่ และแจ้งเตือนเมื่อการทำการสร้างไฟล์ สมุดประจำรถสำเร็จ



รูปที่ 6-7 ภาพการแจ้งเตือนใบอนุญาตประกอบการหมดอายุ



รูปที่ 6-8 ภาพการแสดงตำแหน่งยานพาหนะและสถานะจากบริการ Realtime



รูปที่ 6-9 ภาพสถานะการสร้าง Report สมุดประจำรถสำเร็จ

6.3.1. Tech Stack

ตารางที่ 6-20 Tech Stack

Layer	Technology	Version
Runtime	Node.js	16.x (Alpine)
Web Framework	Express.js	^4.18.1
Real-time	Socket.IO	^4.5.1
Message Broker	Apache Kafka (kafka-node)	^5.0.0
Redis Adapter	@socket.io/redis-adapter	^7.2.0
Redis Emitter	@socket.io/redis-emitter	^5.0.0
Redis Client	redis	^4.3.1
SQL Database	PostgreSQL (pg) + Sequelize ORM	pg ^8.7.3, sequelize ^6.19.0
NoSQL Database	MongoDB (mongodb)	^4.12.1
Authentication	JWT RS256 (jsonwebtoken)	^8.5.1
Security	Helmet.js	^5.1.1
CORS	cors	^2.8.5
Session	express-session	^1.17.3
HTTP Client	Axios	^1.2.6
Utility	Lodash, Moment.js, UUID	latest
Dev Server	Nodemon	^2.0.20
Containerization	Docker (node:16-alpine)	20.10.9
CI/CD	GitLab CI/CD + Kustomize	—

6.3.2. โมดูลหลักของระบบ

จากเอกสาร ระบบ DLT-TMS ถูกออกแบบในลักษณะ Microservices โดยสามารถแบ่งโมดูลหลักออกเป็น 2 กลุ่มใหญ่ ได้แก่ Application Services และ Infrastructure Services

1) Application Services (โมดูลหลักด้านการทำงานของระบบ)

เป็นส่วนที่พัฒนาขึ้นเพื่อรองรับ Business Logic และการใช้งานโดยตรงของผู้ใช้ ประกอบด้วยบริการสำคัญ เช่น

- API Service (dlt-platform-api)

ให้บริการ REST API สำหรับการเชื่อมต่อและแลกเปลี่ยนข้อมูล

- Frontend Service (dlt-connect-frontend)

ส่วนติดต่อผู้ใช้งานผ่านหน้าเว็บ

- Realtime Service (dlt-socket-api)

ให้บริการข้อมูลแบบ Real-time ผ่าน WebSocket

- Datapool Service (dlt-datapool)

รับและจัดการข้อมูล เช่น ข้อมูล GPS จากอุปกรณ์ภายนอก

- Position Publisher (dlt-position-publisher)

ทำหน้าที่ส่งข้อมูลตำแหน่งไปยังระบบอื่น

- Event Builder (dlt-event-builder)

ประมวลผลข้อมูลและสร้างเหตุการณ์ (Events) ตามตรรกะของระบบ

2) Infrastructure Services (โมดูลสนับสนุนระบบ)

เป็นส่วนของโครงสร้างพื้นฐานที่รองรับการทำงานของ Application Services ให้มีความเสถียรและมีประสิทธิภาพ ได้แก่

- Database
  - MongoDB (NoSQL)
  - PostgreSQL (Relational)
- Message Broker
  - Kafka (ระบบรับ-ส่งข้อมูลแบบ Streaming)
- Cache
  - Redis (จัดเก็บข้อมูลชั่วคราว / session)
- Object Storage
  - MinIO (จัดเก็บไฟล์)
- Networking & Security

- Ingress / Ingress Controller (จัดการ routing)
- Cert-manager (จัดการ SSL/TLS Certificate)

6.3.3. Kafka Consumer (Subscribe)

ตารางที่ 6-21 Kafka Consumer

Consumer	ไฟล์	Kafka Topic	ฟีเจอร์
Position Pool	utils/kafkaHelper.js	POSITION_POOL, POSITION_POOL_EMPTY_REFERENCE	รับข้อมูล GPS แล้ว emit ไปยัง /vehicle-tracking, /public-vehicle-tracking, /monitor-transport
Notification	utils/kafkaHelper.js	NOTIFICATION	รับ notification event แล้ว emit ไปยัง /notification
Report Tracking	utils/kafkaHelper.js	REPORT_TRACKING	รับสถานะ report แล้ว emit ไปยัง /report-tracking

6.3.4. Service Layer

ตารางที่ 6-22 Service Layer

Service	ไฟล์	หน้าที่
Notification Service	service/notification.js	สร้าง notification หลายประเภท (GPS event, offense, broadcast, tax expire) บันทึกลง MongoDB

6.3.5. Environment Variables

6.3.6. Application

Variable	คำอธิบาย
APP_PORT	Port ที่ service จะ listen (default: 3008)
SECRET	Secret key สำหรับ express-session

6.3.7. PostgreSQL Database

Variable	คำอธิบาย
DB_HOST	hostname ของ PostgreSQL server
DB_PORT	Port ของ PostgreSQL (default: 5432)
DB_USER	Username สำหรับ connect database
DB_PASSWORD	Password สำหรับ connect database
DB_NAME	ชื่อ database
DB_DIALECT	Dialect สำหรับ Sequelize (ใช้ postgres)

6.3.8. MongoDB

Variable	คำอธิบาย
MONGO_URI	Connection URI ของ MongoDB (รูปแบบ: mongodb://user:pass@host:port/)
MONGO_DB	ชื่อ MongoDB database

6.3.9. Redis

Variable	คำอธิบาย
REDIS_HOST	hostname ของ Redis server
REDIS_PORT	Port ของ Redis
REDIS_AUTH	Password ของ Redis (ถ้ามี)
REDIS_STARTUP_TIMEOUT_MS	Timeout สำหรับ Redis health check ตอน startup (default: 5000)

6.3.10. Apache Kafka

Variable	คำอธิบาย
KAFKA_HOST	hostname ของ Kafka broker
KAFKA_PORT	Port ของ Kafka broker (default: 9092)
KAFKA_BOOTSTRAP	Kafka bootstrap servers (รูปแบบ: host:port) — ถ้าตั้งค่าจะใช้แทน KAFKA_HOST:KAFKA_PORT
KAFKA_GROUP	Consumer group ID สำหรับ position tracking
KAFKA_TOPIC	Kafka topic สำหรับ position (ค่า default: POSITION_POOL)
KAFKA_GROUP_NOTIFICATION	Consumer group ID สำหรับ notification
KAFKA_TOPIC_NOTIFICATION	Kafka topic สำหรับ notification (default: NOTIFICATION)
KAFKA_GROUP_STREAMMAX	Consumer group ID สำหรับ Streammax position
KAFKA_TOPIC_STREAMMAX	Kafka topic สำหรับ Streammax position
KAFKA_STARTUP_TIMEOUT_MS	Timeout สำหรับ Kafka health check ตอน startup (default: 5000)

6.3.11. ขั้นตอนการนำงานขึ้นระบบ

ทางที่ปรึกษาได้ออกแบบ Automation Deployment เพื่อช่วยให้นักพัฒนาสามารถนำซอฟแวร์ขึ้นระบบได้ง่ายโดยมีลำดับการทำงานดังนี้

6.3.12. Merge Merge Request ไป main branch

GitLab → Project → Merge Requests → เลือก MR ที่ approved → Merge
Pipeline บน main จะรัน job make-release ในทันที

6.3.13. make-release สร้าง Git Tag อัตโนมัติ

make-release ใช้semantic-release วิเคราะห์ commit messages ตาม Conventional Commits

Commit prefix	ผลลัพธ์	ตัวอย่าง tag
fix:	Patch bump (x.x.**+1**)	v1.2.1
feat:	Minor bump (x.**+1**.0)	v1.3.0
feat!: / BREAKING:	Major bump (**+1**.0.0)	v2.0.0

6.3.14. Gitlab runner build-prod สร้าง Docker Image

หลังจากถูกติด tag แล้ว Gitlab ก็จะถูกสั่งให้รัน pipeline สำหรับ push image เข้า Gitlab Image Registry

```
# สิ่งที่ build-prod ทำ (จาก .gitlab-ci.yml):
docker login -u gitlab-ci-token --password-stdin $CI_REGISTRY
docker pull $CI_REGISTRY_IMAGE:latest            # ดึง cache
docker build \
  --cache-from $CI_REGISTRY_IMAGE:latest \
  --tag $CI_REGISTRY_IMAGE:$CI_COMMIT_TAG \      # เช่น :v1.2.0
  --tag $CI_REGISTRY_IMAGE:latest .
docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_TAG
docker push $CI_REGISTRY_IMAGE:latest
```

Image จะถูก push ไปยัง **GitLab Container Registry** ของ project

6.3.15. deploy-prod อัปเดต Kubernetes Manifest

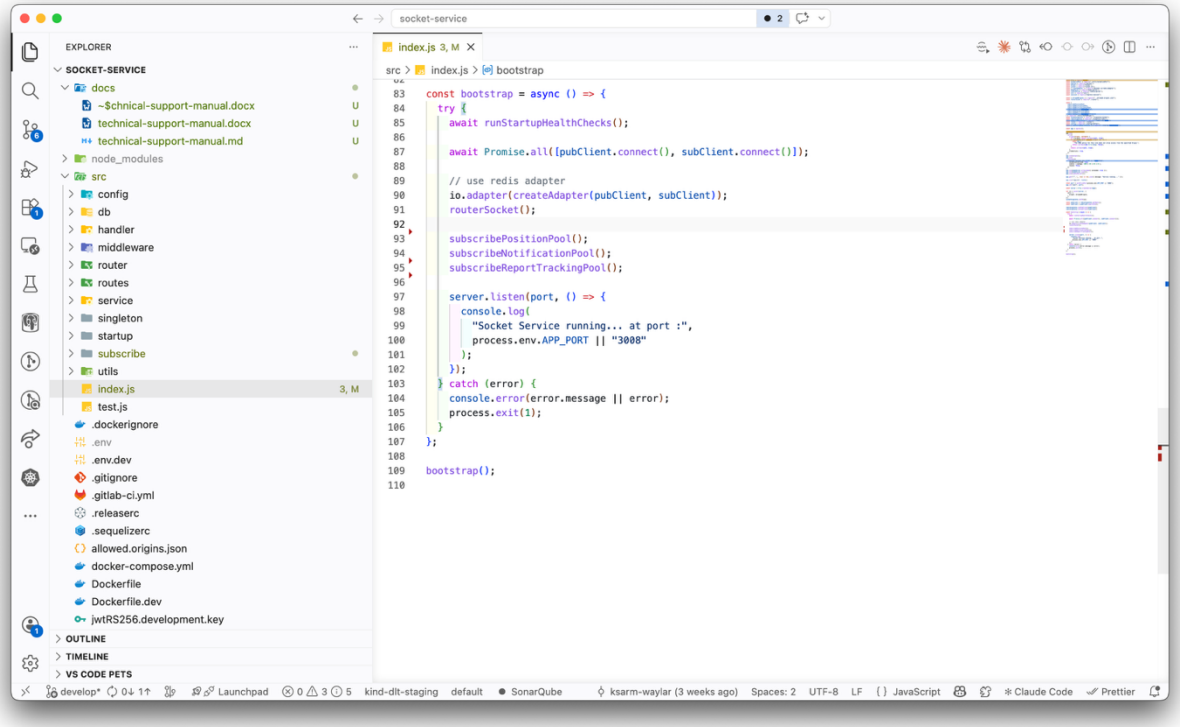
```
# สิ่งที่ deploy-prod ทำ (จาก .gitlab-ci.yml):
git clone https://${USERNAME}:${PAT_TOKEN}@gitlab.com/waylar/dlt-tms/api-infra.git
git checkout -B main
cd api-infra/prod
kustomize edit set image $CI_REGISTRY_IMAGE:$BUILD_VERSION
# → แก้ไข kustomization.yaml ให้ชี้ image ใหม่
git commit -am "[skip ci] PROD manifest-image update for build $BUILD_VERSION"
git push origin main
```

ไฟล์ที่ถูกอัปเดต: `api-infra/prod/kustomization.yaml`

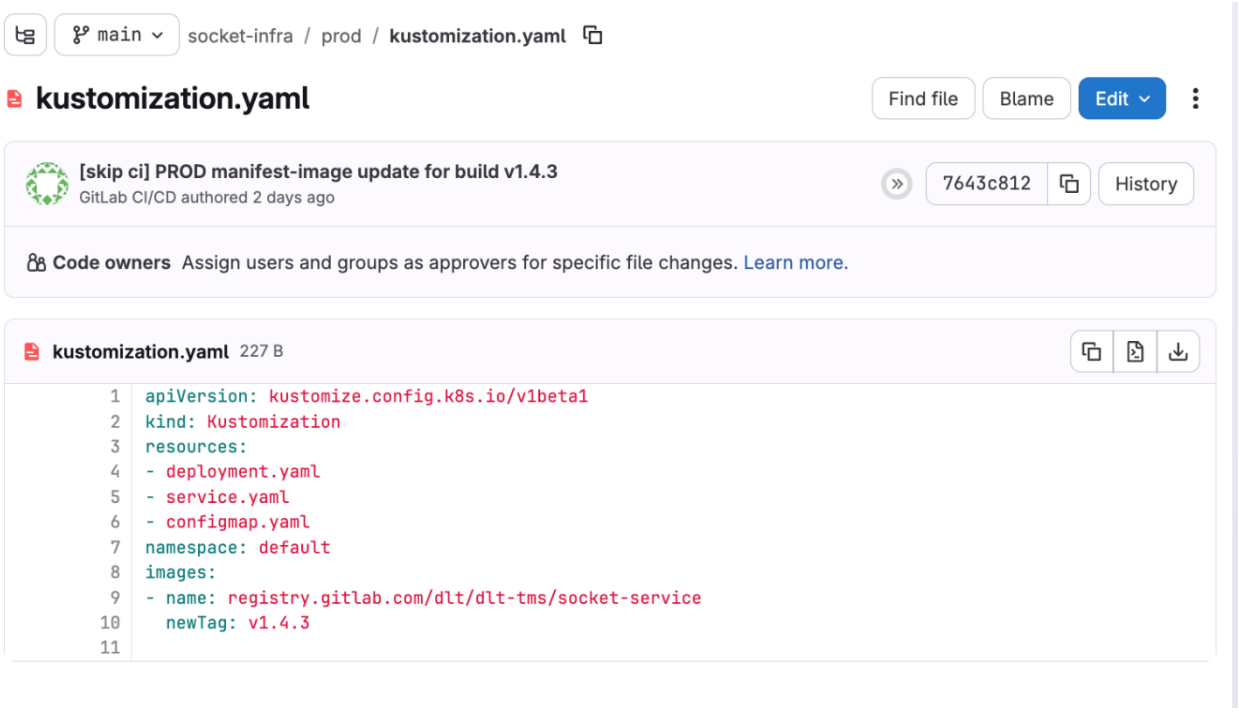
6.3.16. ArgoCD

เมื่อมีการ commit ใหม่ที่ไฟล์ socket-infra/prod/kustomization.yaml ระบบ ArgoCD ซึ่งเฝ้าติดตาม repository ดังกล่าวจะดำเนินการดังนี้:

- ตรวจพบการเปลี่ยนแปลงของไฟล์ manifest
- นำ manifest ที่เปลี่ยนแปลงไป Apply เพื่อเริ่มการ Deploy Kubernetes ใหม่
- ทำการ Rolling update pods ทีละตัว เพื่อให้ระบบยังคงทำงานได้ต่อเนื่อง
- ตรวจสอบ health check ของแต่ละ pod ก่อนจะถือว่าการ deploy เสร็จสมบูรณ์



รูปที่ 6-10 ภาพแสดงโครงสร้างของ Repositories Socket



รูปที่ 6-11 Gitlab runner update image ที่ socket-infra

ภาพแสดงเนื้อหาของไฟล์ kustomization.yaml ภายในคลังข้อมูล socket-infra ซึ่งใช้สำหรับระบุองค์ประกอบทรัพยากร (Resources) ที่จำเป็น เช่น Deployment, Service และ ConfigMap โดยมีส่วนสำคัญคือการกำหนด newTag ของ Docker Image เป็นเวอร์ชัน v1.4.3 เพื่อระบุเวอร์ชันของบริการที่ต้องการติดตั้งบนระบบอย่างชัดเจนและแม่นยำ

6.3.17. ปัญหาที่พบบ่อย

Kafka Consumer

อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
Service crash พร้อม error จาก Kafka ตอน startup	Kafka broker ไม่พร้อม หรือ config ผิด	ตรวจสอบ KAFKA_HOST, KAFKA_PORT หรือ KAFKA_BOOTSTRAP และสถานะ Kafka cluster ที่ argocd
ข้อมูล GPS ไม่ถูกส่งไปยัง client	Consumer group offset ค้าง หรือ topic ไม่มีข้อมูลใหม่	ตรวจสอบ consumer group lag ใน Kafka, ตรวจสอบว่า producer ส่งข้อมูลเข้า topic อยู่ ที่ argocd
Service restart loop เพราะ Kafka error	Consumer ได้รับ error event จาก Kafka แล้วเรียก process.exit(1)	ตรวจสอบ log หา Kafka error message, ตรวจสอบ network connectivity ระหว่าง service กับ Kafka ที่ argocd

Redis

อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
Service ไม่ start พร้อม error Redis health check failed	Redis server ไม่พร้อม หรือ password ผิด	ตรวจสอบ REDIS_HOST, REDIS_PORT, REDIS_AUTH และสถานะ Redis server
Socket.IO ส่ง message ไม่ถึง client บางตัว (multi-instance)	Redis adapter ไม่ทำงาน	ตรวจสอบ pub/sub client ว่า connect สำเร็จหรือไม่ ใน log

MongoDB

อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
Service ไม่ start พร้อม error จาก MongoDB startup check	MONGO_URI หรือ MONGO_DB ผิด, MongoDB ไม่พร้อม	ตรวจสอบ MONGO_URI และ MONGO_DB รวมถึงสถานะ MongoDB server
Notification ไม่ถูกบันทึก	MongoDB connection หลุด, collection ผิด	ตรวจสอบ log และ connectivity ไปยัง MongoDB

Real-time / Notification

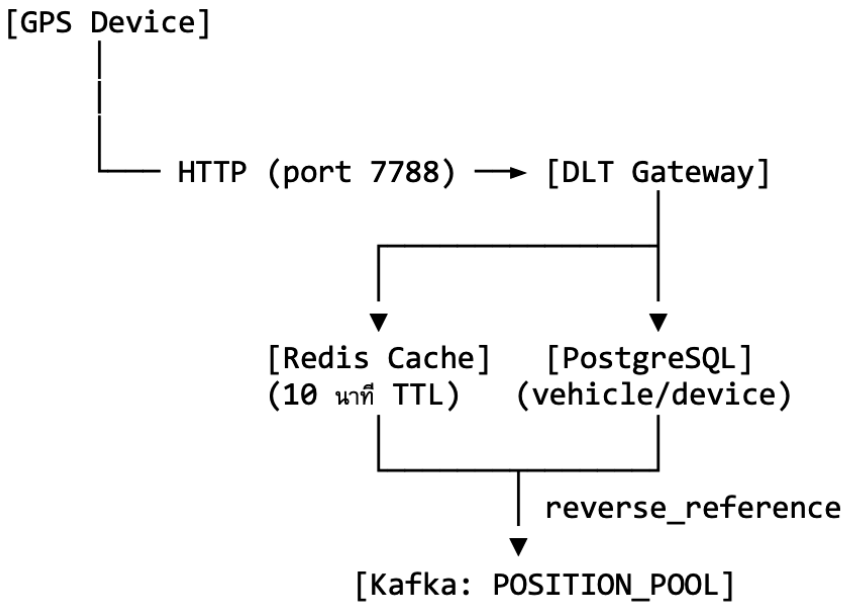
อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
Client ไม่ได้รับ notification	ไม่ได้ emit onRegister พร้อม userID และ organizationID	ตรวจสอบว่า client emit onRegister หลัง connect สำเร็จ

Build / Deploy

อาการ	สาเหตุที่เป็นไปได้	วิธีแก้
Docker build ล้มเหลว	package-lock.json ไม่ sync กับ package.json	รัน npm install แล้ว commit package-lock.json ใหม่

6.4. ระบบรับข้อมูล GPS (data pool)

DLT Gateway เป็นบริการ Gateway สำหรับรับข้อมูลตำแหน่ง GPS จากอุปกรณ์ติดตาม (Tracking Device) ผ่าน ช่องทาง HTTP จากนั้นระบบจะทำการรประมวลผลข้อมูลโดยค้นหาข้อมูลอ้างอิงยานพาหนะ แล้วส่งต่อข้อมูลที่ผ่านการตรวจสอบแล้วไปยัง Kafka เพื่อให้ระบบ downstream นำไปประมวลผลต่อ



รูปที่ 6-12 ภาพรวมระบบ Data pool GPS

ขั้นตอนการทำงานโดยย่อ:

1. รับข้อมูล RawPosition HTTP POST /
2. แปลงเป็น Position struct พร้อม uniqueid และ server_time
3. ค้นหา vehicle reference จาก Redis cache ก่อน หากไม่พบจึงค้นหาจาก PostgreSQL แล้ว cache ไว้
4. หากพบ reference ที่ถูกต้อง (ref_id, ref_type ไม่เป็น null) จึงส่งไปยัง Kafka topic POSITION_POOL

6.4.1. Tech Stack

รายการ	เวอร์ชัน	หน้าที่
Rust	1.85.0	ภาษาหลักของโปรเจกต์ (edition 2024)
Tokio	1.42.0	Async runtime (features: full, signal)
Axum	0.8.4	HTTP web framework
tower-http	0.6.1	HTTP middleware (CORS)
serde / serde_json	1.0	JSON serialization/deserialization
thiserror	2.0.12	Error type derivation
tracing / tracing-subscriber	0.1 / 0.3	Structured logging
chrono	0.4	การจัดการ datetime
uuid	1.16.0	สร้าง UUID v4 สำหรับ Kafka message key
bson	2.15.0	สร้าง Objectld สำหรับ uniqueid
mr_env_plus	0.1.0	โหลด environment variables
async-trait	0.1.88	รองรับ async fn ใน trait

6.4.2. โมดูลหลักของระบบ

Module	หน้าที่
main.rs	สร้าง shared AppState และเปิด MQTT server (port 1883) กับ HTTP server (port 7788) แบบ concurrent ด้วย tokio::try_join!
lib.rs	กำหนด AppState (database, cache, broker, publish) และ BrokerState (MQTT subscriptions map)
error.rs	รวม error ทุกประเภทเป็น enum ErrorType เดียว ใช้ thiserror
producer.rs	ห่อหุ้ม Kafka FutureProducer พร้อม method produce_with_topic() ใช้ UUID v4 เป็น message key
config/	อ่านค่า environment variable ผ่าน mr_env_plus พร้อม default values
database/	กำหนด trait DatabasePool และ implement ด้วย PostgreSQL เพื่อค้นหา vehicle referece
cache/	กำหนด trait CachePool และ implement ด้วย Redis connection pool (r2d2, max 15 connections)
protocol/http/	HTTP server ด้วย Axum รองรับ health check และรับ position data
schema/position.rs	แปลง RawPosition → Position พร้อมทำ reverse reference lookup
schema/reference.rs	เก็บผลการค้นหา vehicle reference (ref_id, ref_type, device_id)
utils/date_time.rs	parse datetime หลายรูปแบบ และ serialize ในรูปแบบ ISO 8601

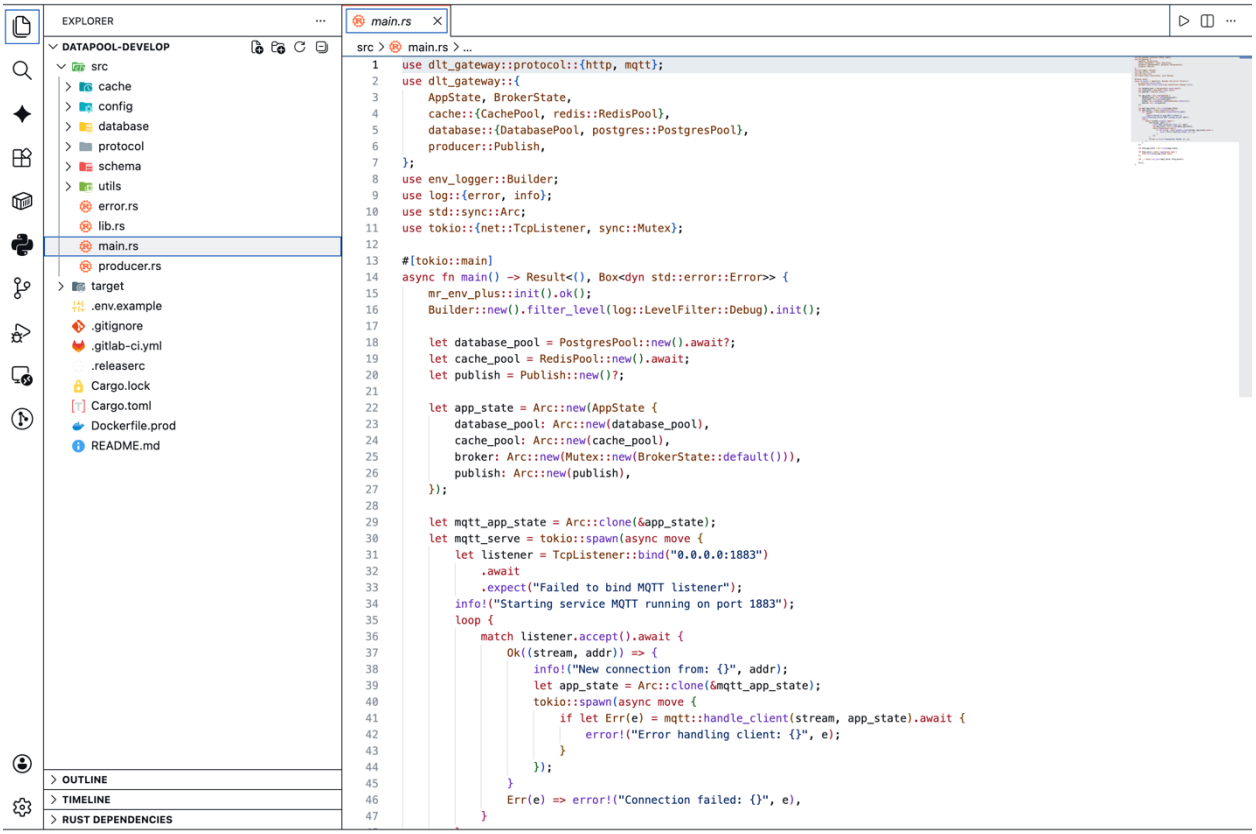
6.4.3. ข้อมูลขาเข้าของ GPS

Field	Type	คำอธิบาย
imei	String?	หมายเลข IMEI ของอุปกรณ์
device_time	DateTime<Utc>	เวลาจากอุปกรณ์
latitude	f64	ละติจูด
longitude	f64	ลองจิจูด
altitude	f64	ความสูง (เมตร)
speed	f64	ความเร็ว
course	f64	ทิศทางการเคลื่อนที่ (องศา)
address	String	ที่อยู่ภาษาไทย
address_en	String	ที่อยู่ภาษาอังกฤษ
accuracy	f32	ความแม่นยำของ GPS
magnetic_card	String?	หมายเลขบัตรแม่เหล็ก (driver ID)
driver_name	String?	ชื่อคนขับ
fuel	f32	ระดับน้ำมัน
odometer	f64	เลขไมล์สะสม
engine_hour	f64	ชั่วโมงการทำงานของเครื่องยนต์
ignition	bool	สถานะการจุดระเบิด
hdop	f32	ค่า HDOP (GPS precision)
sat	u16	จำนวนดาวเทียมที่รับสัญญาณ
gsm_signal_strength	u16	ความแรงสัญญาณ GSM
battery_voltage	f32	แรงดันแบตเตอรี่
chassis	String?	หมายเลขตัวถัง

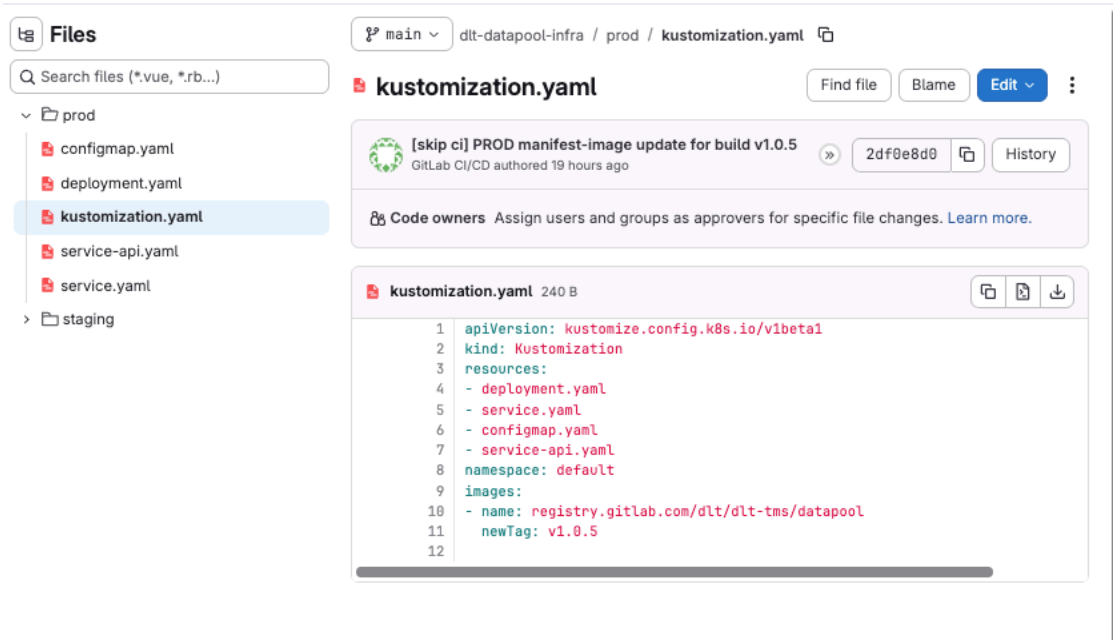
6.4.4. Environment Variable

ตัวแปร	คำอธิบาย
KAFKA_BOOTSTRAP_SERVERS	ที่อยู่ Kafka broker (รูปแบบ host:port)
REDIS_HOST	hostname ของ Redis server
REDIS_PORT	port ของ Redis server

REDIS_PASSWORD	รหัสผ่าน Redis
POSTGRES_HOST	hostname ของ PostgreSQL
POSTGRES_PORT	port ของ PostgreSQL
POSTGRES_DB	ชื่อฐานข้อมูล
POSTGRES_USER	username PostgreSQL
POSTGRES_PASSWORD	password PostgreSQL



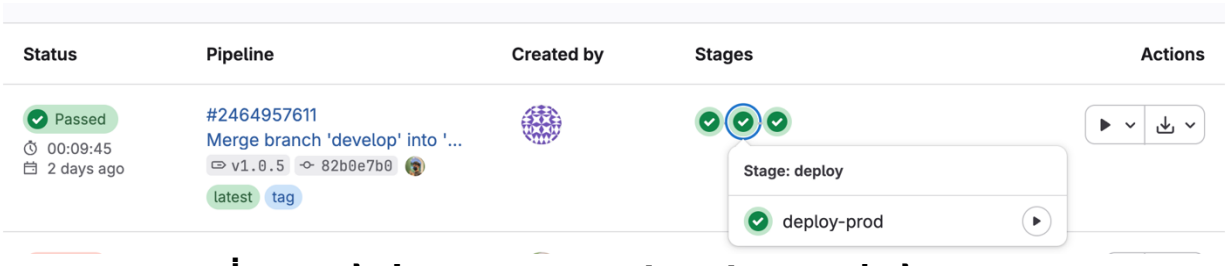
รูปที่ 6-13 ภาพแสดงโครงสร้างของ Repositories datapool



รูปที่ 6-14 Gitlab runner update image ที่ dlt-datapool-infra

6.4.5. ขั้นตอนการนำงานขึ้นระบบ

1. สร้าง git tag บน branch main
2. รอ build-prod รัน — Docker image จะถูก push ขึ้น registry อัตโนมัติ
3. เข้า GitLab UI กด "Play" ที่ job deploy-prod ดังรูปข้างล่าง
4. Pipeline จะ update kustomization.yaml ใน repo dlt-datapool-infra/prod แล้ว push → CD tool deploy ให้อัตโนมัติ



รูปที่ 6-15 หน้าต่างการกด Manual Deploy ระบบรับข้อมูล GPS

ภาพแสดงหน้าต่างการดำเนินการ Deploy ระบบรับข้อมูล GPS ผ่านกระบวนการ CI/CD Pipeline โดยเป็นขั้นตอนการสั่งงานแบบ Manual Deploy ภายหลังจากที่กระบวนการ Build และขั้นตอนก่อนหน้าเสร็จสมบูรณ์แล้ว ซึ่งในภาพแสดงสถานะของ Pipeline อยู่ในสถานะ “Passed” และแต่ละ Stage มีการดำเนินการสำเร็จตามลำดับ

ทั้งนี้ในขั้นตอน Stage: deploy ผู้ใช้งานสามารถสั่งดำเนินการ Deploy ไปยังสภาพแวดล้อมปลายทาง (เช่น production) ผ่านงาน deploy-prod ได้โดยตรง ซึ่งสะท้อนให้เห็นถึงกระบวนการควบคุมการนำระบบขึ้นใช้งานจริงอย่างเป็นขั้นตอน และสามารถตรวจสอบสถานะการทำงานย้อนหลังได้อย่างเป็นระบบ ช่วยให้การบริหารจัดการการปรับปรุงระบบมีความถูกต้อง โปร่งใส และมีประสิทธิภาพ

6.4.6. ปัญหาที่พบบ่อย

ระบบ start ไม่ขึ้น / panic ตอน startup

สาเหตุ	วิธีแก้
เชื่อมต่อ PostgreSQL ไม่ได้	ตรวจสอบ POSTGRES_HOST, POSTGRES_PORT, POSTGRES_USER, POSTGRES_PASSWORD และ network connectivity
เชื่อมต่อ Redis ไม่ได้	ตรวจสอบ REDIS_HOST, REDIS_PORT, REDIS_PASSWORD
Kafka bootstrap server ไม่ถูกต้อง	ตรวจสอบ KAFKA_BOOTSTRAP_SERVERS ว่า format เป็น host:port

6.4.7. ข้อมูล position ไม่ถูกส่งเข้า Kafka

อาการ: HTTP POST / ตอบ 400 Bad Request {"error": "Invalid reference data"}
สาเหตุ: ระบบหา vehicle reference ไม่พบ หมายความว่า IMEI หรือ chassis ที่ส่งมาไม่มีข้อมูลใน PostgreSQL
วิธีตรวจสอบ: 1. ตรวจสอบว่ามีข้อมูลใน table vehicle หรือ heavy_equipment ที่ตรงกับ imei หรือ chassis_no 2. ตรวจสอบว่า device ใน table device ผูกกับ vehicle แล้ว (field device_id) 3. ตรวจสอบว่า record ไม่ถูก soft delete (deleted_at is null)

```sql
-- ตรวจสอบด้วย query เดียวกับที่ระบบใช้
WITH vehicle AS (
    SELECT id AS ref_id, 'vehicle' AS ref_type, device_id, chassis_no
    FROM vehicle WHERE deleted_at IS NULL
    UNION ALL
    SELECT id, 'heavy_equipment', device_id, chassis_no
    FROM heavy_equipment WHERE deleted_at IS NULL
)
SELECT v.*
FROM vehicle v
LEFT JOIN device d ON v.device_id = d.id
WHERE d.imei = '<imei>' OR v.chassis_no = '<chassis>'
LIMIT 1;
```

6.4.8. Kafka message ไม่ถูก consume โดย downstream

สาเหตุ	วิธีแก้
Topic ไม่ตรง	downstream ต้อง subscribe topic `POSITION_POOL` (ตัวพิมพ์ใหญ่ทั้งหมด)
Kafka broker ต่างกัน	ตรวจสอบ `KAFKA_BOOTSTRAP_SERVERS` ให้ตรงกันทั้ง producer และ consumer
Message timeout	ตรวจสอบ network latency ไปยัง Kafka broker ค่า timeout ปัจจุบันคือ 5 วินาที

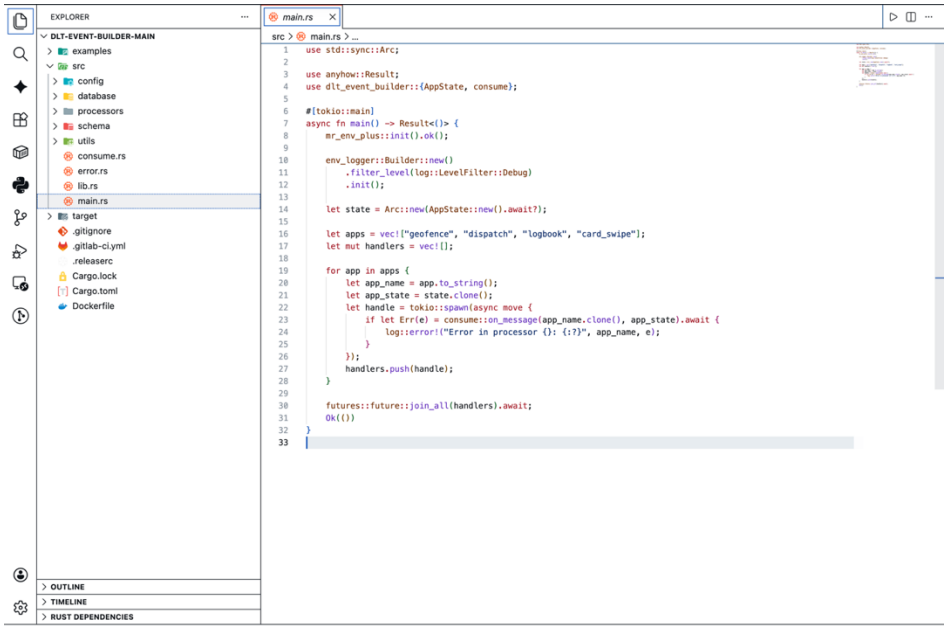
6.5. ระบบจัดการกิจกรรม (event builder)

ทางที่ปรึกษาได้ออกแบบ **ระบบจัดการกิจกรรม (Event Builder)** เพื่อใช้ในการกำหนด สร้าง และบริหารจัดการเหตุการณ์ (events) ที่เกิดขึ้นภายในระบบขนส่งสินค้า โดยระบบนี้มีลักษณะเป็นเครื่องมือที่ช่วยให้สามารถนิยามเงื่อนไข (conditions) และรูปแบบของเหตุการณ์ต่าง ๆ ได้อย่างยืดหยุ่น ตามความต้องการของผู้ใช้งานหรือหน่วยงาน

DLT Event Builder ระบบที่ออกแบบเพื่อประมวล Event ในรูปแบบ Real-time สำหรับสำหรับระบบติดตามยานพาหนะ (Fleet Management) โดย หน้าที่หลักจะประกอบไปด้วย รับข้อมูลตำแหน่ง GPS (Position) , วิเคราะห์และสร้าง Event แต่ละประเภท จากข้อมูลตำแหน่ง

ที่มีการส่งเข้ามา และ ถูกจัดเก็บบน ฐานข้อมูล ใน PostgreSQL และ MongoDB และ จัดการระบบคิวของข้อความผ่าน Redis
โดยจะใช้สำหรับฟีเจอร์

- Auto check-in Check-out ในงานขนส่ง
- ฟีเจอสมุดประจำรถ



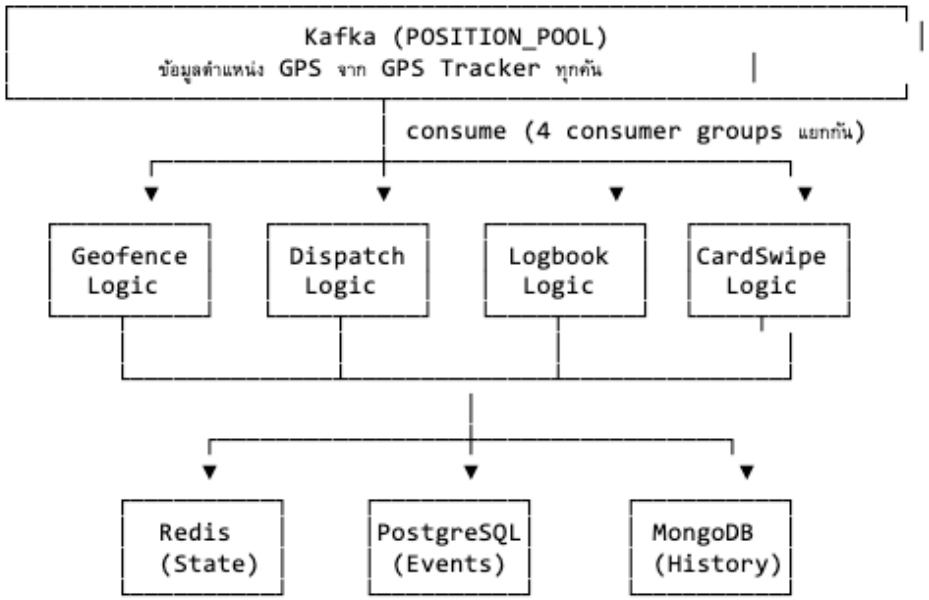
รูปที่ 6-16 ภาพ code ส่วน ระบบจัดการกิจกรรม

6.5.1. ประเภทของ Event

ประเภท Event	Sub-type	ความหมาย
GEOFENCE	CHECK_IN	ยานพาหนะเข้าพื้นที่ geofence
GEOFENCE	CHECK_OUT	ยานพาหนะออกจากพื้นที่ geofence
DISPATCH	—	ยานพาหนะมาถึงจุดส่งสินค้า
ENGINE	—	เครื่องยนต์ติด/ดับ
IDLING_30_MIN	—	จอดรถติดเครื่องนิ่งนานกว่า 30 นาที
CARD_SWIPE	—	พนักงานรูดบัตรเข้า/ออก

6.5.2. ขั้นตอนการทำงานโดยย่อ

1. GPS Tracker ส่งข้อมูลตำแหน่งเข้า Kafka
2. Processor แต่ละตัว consume message จาก Kafka แยก Consumer Group
3. ดึง State ล่าสุดของอุปกรณ์จาก Redis
4. เปรียบเทียบ State เก่าและใหม่ → สร้าง Event ถ้ามีการเปลี่ยนแปลง
5. บันทึก Event ลง PostgreSQL และ MongoDB 6. อัปเดต State ใหม่ลง Redis



รูปที่ 6-17 ภาพแสดงโครงสร้างของระบบ dlt-event-builder

ภาพแสดงกระบวนการส่งต่อและประมวลผลข้อมูลจาก Kafka (POSITION_POOL) ซึ่งทำหน้าที่เป็นศูนย์กลางรวบรวมข้อมูลตำแหน่งจากอุปกรณ์ GPS Tracker โดยมีการกระจายงานไปยังกลุ่มผู้บริโภคข้อมูล (Consumer Groups) 4 กลุ่ม ได้แก่ Geofence, Dispatch, Logbook และ CardSwipe เพื่อประมวลผลตามตรรกะทางธุรกิจที่แตกต่างกัน ก่อนจะนำผลลัพธ์ไปจัดเก็บลงในฐานข้อมูลปลายทางตามวัตถุประสงค์ ได้แก่ Redis (สถานะปัจจุบัน), PostgreSQL (เหตุการณ์) และ MongoDB (ประวัติการใช้งาน) เพื่อรองรับการทำงานที่มีประสิทธิภาพและขยายตัวได้สูง

6.5.3. Tech Stack

Layer	Technology	Version	Responsivity
Language	Rust	1.85 (Edition 2024)	ภาษาหลัก
Async Runtime	Tokio	1.47.1	จัดการ async/concurrent tasks
Message Queue	Apache Kafka (rdkafka)	0.36.2	รับ GPS position stream
PostgreSQL Driver	sqlx	0.8.6	บันทึก Event, ดึง Geofence/Reference
MongoDB Driver	mongodb	3.1.0	บันทึก Event history และ Position
Redis Client	redis + r2d2	0.26.1	จัดการ State, Connection Pool
Serialization	serde + serde_json	1.0	JSON parse/serialize
Date/Time	chrono	0.4	คำนวณเวลา, duration
Error Handling	thiserror + anyhow	2.0 / 1.0	จัดการ Error แบบ typed
Logging	log + env_logger	0.4 / 0.11	บันทึก log
Config	mr_env_plus	0.1.0	อ่าน environment variables
UUID	uuid	1.3	สร้าง Event ID
Regex	regex	1.11	Validate geofence format
Container	Docker (multi-stage)	—	Package และ Deploy
CI/CD	GitLab CI/CD	—	Build, Release, Deploy อัตโนมัติ
K8s Config	Kustomize	—	จัดการ Kubernetes manifest

6.5.4. โมดูลหลักของระบบ

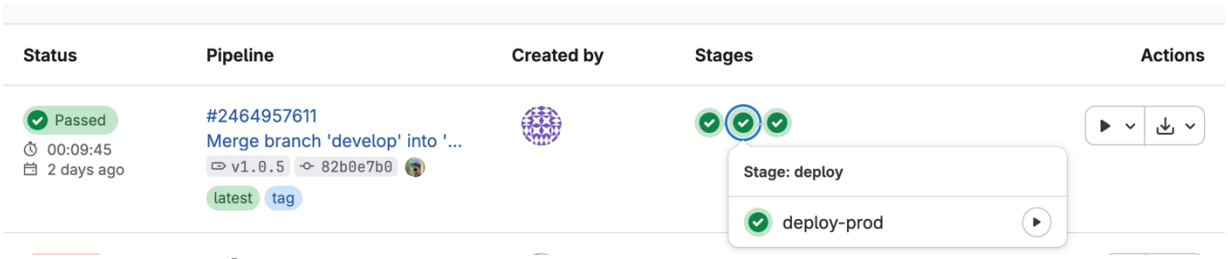
Module	หน้าที่
main.rs	Entry point ของระบบ โหลด env, ตั้ง logger, สร้าง AppState และรัน async tasks (geofence, dispatch, logbook, card_swipe) แบบ concurrent
lib.rs	เก็บ AppState (shared state) เช่น database connections, cache, producer และมี method สำหรับ insert/update ข้อมูล
consume.rs (consumer)	Kafka consumer รับ message จาก topic → ส่งเข้า processors → commit offset และ reload data ตาม interval
consume.rs (producer)	Kafka producer wrapper มี method produce_with_topic() และใช้ UUID v4 เป็น message key
config/	อ่าน environment variables ผ่าน mr_env_plus และสร้าง config struct สำหรับ services ต่าง ๆ
database/postgres.rs	จัดการ PostgreSQL เช่น insert event/position, select geofence/booking, update booking
database/mongo.rs	insert event/position และจัดการ ObjectId สำหรับเชื่อมข้อมูล
database/redis.rs	จัดการ state (geofence, logbook, dispatch, card) ผ่าน Redis + pipeline
cache/	นิยาม trait CachePool และ implement ด้วย Redis connection pool (r2d2)
processors/	Business logic หลักของระบบ (geofence, dispatch, logbook, card_swipe)
schema/	แปลงข้อมูล Raw → Structured (Position/Event) และจัดการ reference lookup
utils/	ฟังก์ชันช่วย เช่น datetime, geometry, magnetic card
error.rs	รวม error ทั้งหมดใน enum เดียว ใช้ thiserror

6.5.5. Environment Variable

ตัวแปร	คำอธิบาย
KAFKA_BOOTSTRAP	ที่อยู่ Kafka broker (รูปแบบ host:port รองรับหลายตัว คั่นด้วย ,)
KAFKA_CONSUMER_PREFIX	prefix สำหรับตั้งชื่อ consumer group
DB_HOST	hostname ของ PostgreSQL server
DB_PORT	port ของ PostgreSQL (ค่า default: 5432)
DB_NAME	ชื่อ database
DB_USER	username สำหรับเชื่อมต่อ
DB_PASS	password สำหรับเชื่อมต่อ
MONGO_URI	connection string ของ MongoDB (เช่น mongodb://user:pass@host:27017)
MONGO_DB	ชื่อ database
REDIS_HOST	hostname ของ Redis server
REDIS_PORT	port ของ Redis
REDIS_PASSWORD	password สำหรับ authenticate
REDIS_PREFIX	prefix สำหรับ namespace ของ key ใน Redis
RUST_LOG	ระดับ log (trace, debug, info, warn, error)

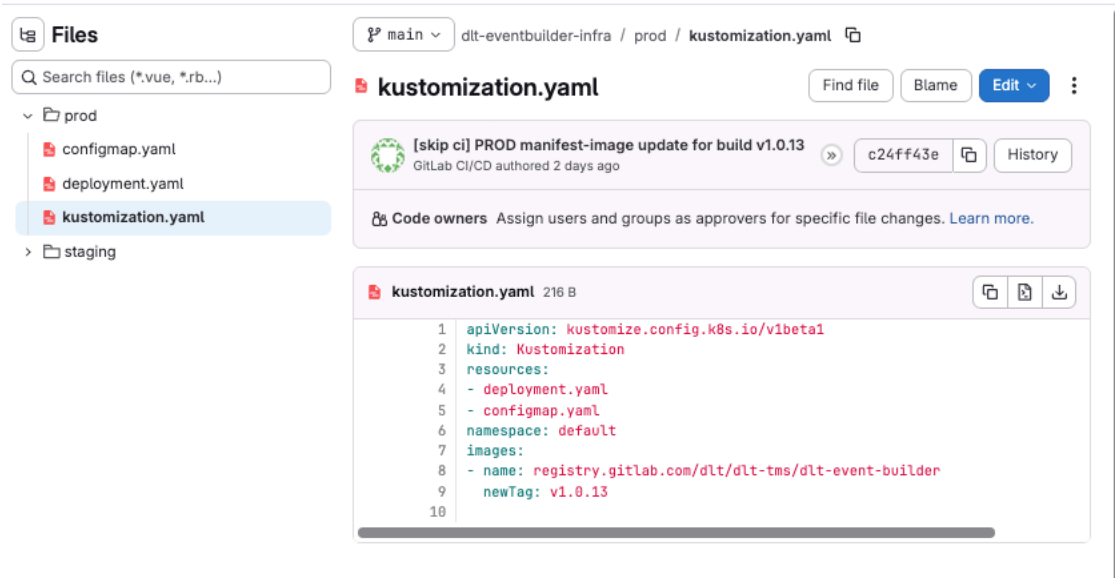
6.5.6. ขั้นตอนการนำงานขึ้นระบบ

1. สร้าง git tag บน branch main บน repository
2. รอ build-prod รัน — Docker image จะถูก push ขึ้น registry อัตโนมัติ
3. เข้า GitLab UI กด "Play" ที่ job deploy-prod ดังรูปข้างล่าง
4. Pipeline จะ update kustomization.yaml ใน repo dlt-event-builder-infra/prod แล้ว push → CD tool deploy ให้อัตโนมัติ



รูปที่ 6-18 หน้าต่างการกด Manual Deploy ระบบรับข้อมูล GPS

ภาพแสดงองค์ประกอบและกระบวนการทำงานของระบบ DLT-TMS ตามมาตรฐาน **DevOps** และ **GitOps** ซึ่งประกอบด้วยโครงสร้าง **Event-Driven Architecture** ที่ใช้ Kafka ในการกระจายข้อมูล GPS ไปยังกลุ่มผู้บริโภคข้อมูล (Consumer Groups) เพื่อประมวลผลและจัดเก็บในฐานข้อมูลตามวัตถุประสงค์ ควบคู่ไปกับกระบวนการ **CI/CD Pipeline** อัตโนมัติผ่าน GitLab ที่ควบคุมการ Deploy ผ่านระบบ ArgoCD โดยมีการใช้ไฟล์คอนฟิก **Kustomize** ในการบริหารจัดการเวอร์ชันซอฟต์แวร์อย่างแม่นยำ ซึ่งทำให้ระบบทั้งหมดมีความเสถียร (Reliability) สามารถตรวจสอบย้อนหลังได้ (Auditable) และพร้อมรองรับการขยายตัว (Scalability) ในระดับองค์กรอย่างมีประสิทธิภาพ



รูปที่ 6-19 Gitlab runner update image ที่ dlt-eventbuilder-infra

ภาพแสดงภาพรวมกระบวนการพัฒนาและบริหารจัดการระบบของโครงการ **DLT-TMS** ซึ่งนำมาตรฐาน **DevOps** และ **GitOps** มาประยุกต์ใช้อย่างเป็นระบบ โดยเริ่มจากกระบวนการ **CI/CD Pipeline** อัตโนมัติบน GitLab ที่ช่วยในการรวมโค้ดและส่งมอบซอฟต์แวร์ผ่านขั้นตอนการ Deploy ที่มีการบันทึกสถานะและข้อมูลการเปลี่ยนแปลง (Commit Hash) อย่างชัดเจนเพื่อความสามารถในการตรวจสอบย้อนกลับ (Traceability) ควบคู่ไปกับการใช้ **Kustomize** ในการบริหารจัดการค่าคอนฟิกูเรชันและเวอร์ชันของซอฟต์แวร์ (newTag) ผ่านไฟล์ kustomization.yaml เพื่อให้การอัปเดตระบบในแต่ละสภาพแวดล้อมมีความถูกต้อง แม่นยำ และเป็นอัตโนมัติ ซึ่งทั้งหมดนี้จะถูกตรวจสอบและจัดการผ่าน **ArgoCD** ในฐานะเครื่องมือควบคุมสถานะของระบบ (GitOps) ที่ช่วยยืนยันความสอดคล้องระหว่างสถานะใน Git กับสิ่งที่ทำงานจริงใน Kubernetes Cluster (Healthy & Synced) ทำให้ระบบโดยรวมมีความเสถียร (Reliability) สูงและพร้อมรองรับการขยายตัวในระดับองค์กรอย่างมีประสิทธิภาพ

6.5.7. ปัญหาที่พบบ่อย

Duplicate Events เมื่อ Redis Restart

อาการ: GEOFENCE:CHECK_IN ซ้ำหลายครั้งสำหรับยานพาหนะเดียวกัน
สาเหตุ: Redis State หายหลังจาก restart → ระบบไม่รู้ว่ายานพาหนะ CHECK_IN ไปแล้ว
วิธีแก้ชั่วคราว: - ตรวจสอบ Redis persistence (RDB/AOF) เปิดอยู่หรือไม่ - ดู log ว่า Redis disconnect เมื่อไหร่
วิธีแก้ถาวร: - เปิด Redis AOF persistence - ใช้ Redis Cluster สำหรับ HA

Kafka Rebalancing บ่อย

อาการ: Log มี "Rebalancing..." บ่อย, consumer lag ขึ้นลง
สาเหตุ: Pod restart บ่อย หรือ session timeout ต่ำเกินไป
ค่า timeout ปัจจุบัน: - Session timeout: 5 นาที (300,000 ms) - Max poll interval: 10 นาที (600,000 ms)
วิธีแก้: ตรวจสอบว่า process แต่ละ message ใช้เวลาไม่เกิน max poll interval - หากใช้เวลานาน ให้เพิ่ม max.poll.interval.ms

6.6. ระบบจัดการตามกำหนดเวลา (Cronjob)

ทางที่ปรึกษาได้ออกแบบ **ระบบจัดการตามกำหนดเวลา (Cronjob)** เพื่อใช้ในการควบคุมและดำเนินการประมวลผลงานต่าง ๆ ตามช่วงเวลาที่กำหนดไว้ล่วงหน้า โดยระบบนี้มีบทบาทสำคัญในการทำงานเบื้องหลัง (background processing) ที่ไม่จำเป็นต้องตอบสนองแบบทันที แต่ต้องทำอย่างสม่ำเสมอและมีความแม่นยำตามรอบเวลา

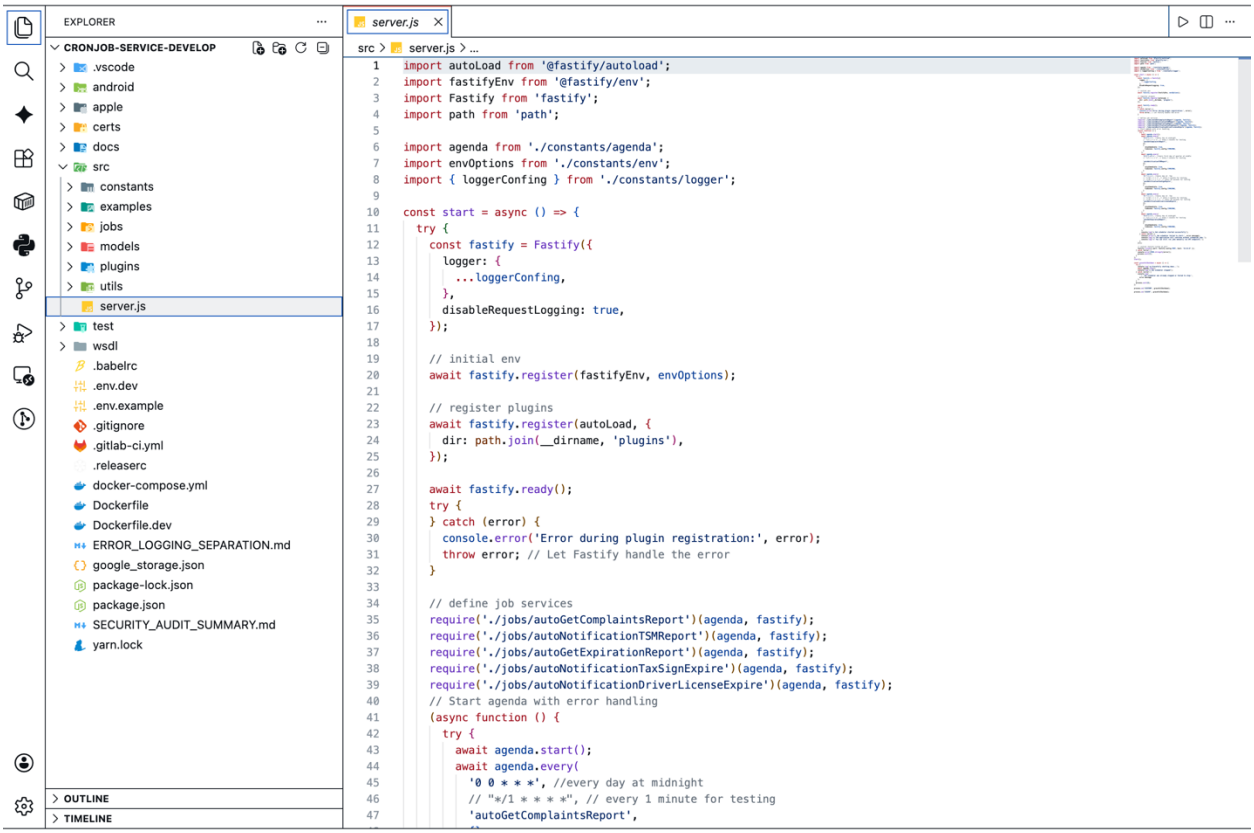
ระบบ Cronjob สามารถตั้งเวลาให้ทำงานแบบอัตโนมัติได้ เช่น รายนาที รายชั่วโมง รายวัน หรือช่วงเวลาที่กำหนดเอง โดยรองรับการทำงานที่หลากหลาย เช่น การสรุปข้อมูล (data aggregation), การล้างข้อมูลเก่า (data cleanup), การส่งรายงาน (scheduled reporting), หรือการประมวลผลข้อมูลจากระบบ GPS และ Event Builder เพื่อนำไปใช้ต่อในระบบอื่น

นอกจากนี้ ระบบยังรองรับการจัดการงานแบบ queue และสามารถทำงานร่วมกับสถาปัตยกรรมแบบ distributed system ได้ ทำให้สามารถกระจายภาระงาน (load distribution) และเพิ่มความสามารถในการรองรับงานจำนวนมาก (scalability) ได้อย่างมีประสิทธิภาพ รวมถึงมีระบบตรวจสอบสถานะการทำงาน (job monitoring) และการจัดการข้อผิดพลาด (retry / failure handling) เพื่อให้มั่นใจว่างานจะถูกดำเนินการครบถ้วน

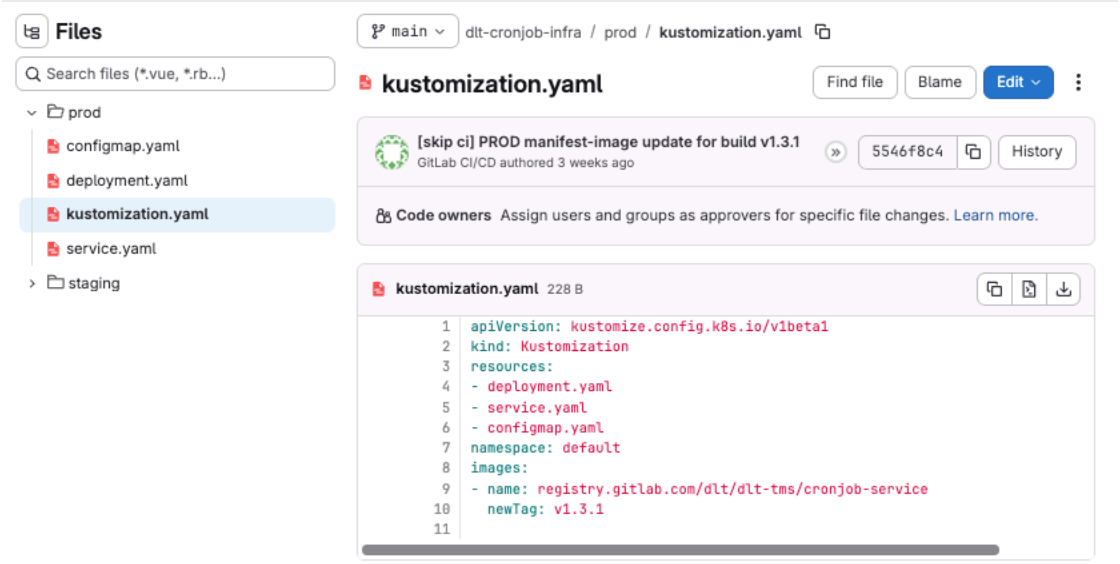
โดยระบบจัดการตามกำหนดเวลา (Cronjob) ในโครงการพัฒนาระบบเทคโนโลยีสารสนเทศบริหารจัดการขนส่งสินค้าทางถนน มีบทบาทสำคัญในการสนับสนุนการทำงานเบื้องหลังของระบบทั้งหมด ช่วยให้การประมวลผลข้อมูลเป็นไปอย่างต่อเนื่อง เป็นระบบ และมีประสิทธิภาพสูงสุด ลดภาระงานที่ต้องทำแบบ manual และเพิ่มความน่าเชื่อถือในการดำเนินงานโดยรวมของระบบ

Cronjob Service เป็น background service สำหรับระบบ DLT-TMS (กรมการขนส่งทางบก) มีหน้าที่หลักคือรัน scheduled jobs อัตโนมัติตามตารางเวลาที่กำหนด ได้แก่:

1. ดึงข้อมูลร้องเรียนกองตรวจการจาก DLT SOAP API และบันทึกลงฐานข้อมูล
2. ส่ง push notification แจ้งเตือนผ่าน Kafka สำหรับเอกสารที่ใกล้หมดอายุหรือหมดอายุแล้ว (ใบอนุญาตประกอบการ, ภาษีรถ, ใบขับขี่)
3. แจ้งเตือนรายงาน TSM รายไตรมาส
4. ลบข้อมูลผู้ใช้ที่รออยู่ใน deletion queue



รูปที่ 6-20 ภาพแสดงโครงสร้างของ Repositories cronjob-service



รูปที่ 6-21 Gitlab runner update image ที่ dlt-cronjob-infra

6.6.1. Tech Stack

Component	Technology	Version
Runtime	Node.js	16 (LTS)
Framework	Fastify	^4.8.1
Job Scheduler	Agenda	^4.3.0
Job Store (Agenda)	MongoDB	URI-based
Primary Database	PostgreSQL	ผ่าน pg ^8.8.0
ORM	Sequelize	^6.37.7
Message Broker	Kafka	kafka-node ^5.0.0
File Storage	Google Cloud Storage	^6.9.5
HTTP Client	fastify-axios	^1.2.8
SOAP Client	soap + xml2js	^1.5.0 / ^0.6.2
Date/Time	moment + moment-timezone	^2.29.4 / ^0.6.0
Logger	pino (built-in Fastify) + pino-pretty-compact	—
Transpiler	Babel (preset-env)	^7.x
Container	Docker	—
CI/CD	GitLab CI	—
Orchestration	Kubernetes + Kustomize	—

6.6.2. โครงสร้าง Module

ภาพรวมโครงสร้างไฟล์

```
src/
├── server.js              # Entry point — bootstrap Fastify, ลงทะเบียน jobs, เริ่ม Agenda
├── constants/
│   ├── agenda.js          # สร้าง Agenda instance เชื่อมต่อ MongoDB
│   ├── env.js             # JSON Schema validation ของ environment variables
│   ├── index.js           # Config object (database, kafka) อ่านจาก env
│   ├── event_type.js      # Enum ประเภท Kafka event ทั้งหมด
│   └── logger.js          # Config ของ pino logger
├── jobs/                  # Job definitions (logic ของแต่ละ scheduled task)
│   ├── autoGetComplaintsReport.js
│   ├── autoNotificationTSMReport.js
│   ├── autoGetExpirationReport.js
│   ├── autoNotificationTaxSignExpire.js
│   ├── autoNotificationDriverLicenseExpire.js
│   ├── autoNotificationLicenseExpire.js
│   └── autoDeleteUsers.js
├── models/                # Sequelize model definitions (PostgreSQL)
│   ├── ConnectOrganizations.js
│   ├── ConnectRegisterations.js
│   ├── ConnectUsers.js
│   ├── ConnectProfiles.js
│   ├── ConnectOrganizationsUsers.js
│   ├── ConnectRoles.js
│   ├── UserDeletionQueue.js
│   ├── Vehicle.js
│   ├── VehicleTaxSign.js
│   └── DltComplaints.js / DltProvinces.js
├── plugins/               # Fastify plugins (ถูก autoload ทั้งหมด)
│   ├── main.js            # ลงทะเบียน healthcheck, axios, moment, lodash
│   ├── sequelize.js       # เชื่อมต่อ PostgreSQL, โหลด models, decorate fastify
│   ├── kafka.js           # สร้าง Kafka producer, decorate fastify
│   ├── storage.js         # Google Cloud Storage client, decorate fastify
│   ├── utils.js           # Utility functions (createdAt/updatedAt, encodeArea ฯลฯ)
│   ├── createReport.js    # Helper สร้าง report และส่ง Kafka
│   ├── applicationConstants.js
│   ├── constants.js
│   └── derive.js
```

```
└── utils/
    └── errorLogger.js      # Centralized error logging (console + database)
```

6.6.3. หน้าที่แต่ละ Module

src/server.js — Entry Point

Bootstrap ทั้งระบบ: สร้าง Fastify instance → โหลด env → autoload plugins → ลงทะเบียน jobs กับ Agenda → เริ่ม Agenda → listen port
จัดการ graceful shutdown เมื่อได้รับ SIGTERM/SIGINT

src/constants/agenda.js — Agenda Scheduler

สร้าง singleton ของ Agenda โดยชี้ไปที่ MongoDB ที่กำหนดใน `MONGO_URI` ใช้ collection ที่กำหนดใน `MONGO_COLLECTION` สำหรับเก็บ job metadata

src/plugins/sequelize.js — Database Plugin

เชื่อมต่อ PostgreSQL ผ่าน Sequelize, โหลด model ทุกไฟล์ใน `/models` อัตโนมัติ, เรียก `associate()` ทุก model, แล้ว decorate `fastify.models` และ `fastify.sequelize`

src/plugins/kafka.js — Kafka Plugin

สร้าง `KafkaClient` และ `Producer` เชื่อม broker ที่กำหนดใน `KAFKA_HOST:KAFKA_PORT`
รองรับ `KAFKA_DISABLED=true` สำหรับการ run แบบ offline (log แทนการส่งจริง)
รองรับ `KAFKA_LOCAL_DNS_FALLBACK=true` สำหรับ local development (patch dns.lookup)
Decorate `fastify.kafka` ด้วย `makeMessage()` และ `publishKafkaMessage()`

src/plugins/storage.js — Google Cloud Storage Plugin

ใช้ service account key จาก `google_storage.json` สำหรับ upload/download ไฟล์บน GCS Decorate `fastify.storage`

src/utils/errorLogger.js — Error Logger

Utility กลางสำหรับ log error ทั้งไปที่ console และฐานข้อมูล (dlt_complaints table) พร้อม classify ประเภท error อัตโนมัติ

6.6.4. ความสัมพันธ์ระหว่าง Module

```
server.js
  └─ autoload plugins/
       ├─ main.js        (moment, lodash, healthcheck, axios)
       ├─ sequelize.js   (PostgreSQL + models)
       ├─ kafka.js       (Kafka producer)
       └─ storage.js     (GCS)
  └─ jobs/*.js
       ├─ อ่าน fastify.models (Sequelize)
       ├─ อ่าน fastify.sequelize.sequelize (raw query)
       └─ ส่ง fastify.kafka.publishKafkaMessage()
  └─ agenda.js
       └─ MongoDB (job store)
```

6.6.5. ขั้นตอนการนำงานขึ้นระบบ

1. Check-out repositories cronjob-service
2. Commit code โดยใช้ prefix เช่น feat: commit message
3. Push code ขึ้นไปยัง branch
4. Merge code ไปยังmain branch
5. Gitlab runner จะทำการสร้าง docker cronjob-service image และทำการอัปเดต version image
6. กด deploy เพื่อสั่งให้ gitlab runner อัพเดท repository dlt-cronjob-infra เพื่ออัพเดท image version
7. Argocd application จะทำการดึง image ใหม่มาและจะอัพเดท version ของ Kubernetes pod

6.6.6. ปัญหาที่พบบ่อย (และที่คาดว่าจะเจอ)

MongoDB ต่อไม่ได้ตอน boot

อาการ: Service start ได้ แต่ไม่มี job ถูก schedule
สาเหตุ: MONGO_URI ผิด หรือ MongoDB cluster ไม่พร้อม
แก้: ตรวจ MONGO_URI, ดู log [Agenda] MongoDB connection error

DLT SOAP API timeout

อาการ: Job autoGetComplaintsReport จบโดยไม่บันทึกข้อมูล
สาเหตุ: DLT server response ช้าเกิน 45 วินาที
แก้: ตรวจสอบ network connectivity, ดู log [กองตรวจการ] Request timeout

Duplicate record ใน dlt_complaints

อาการ: ข้อมูลซ้ำในตาราง
สาเหตุ: ระบบมีการ check issSeq + issOffLocCode แต่หากค่า default NODATA หรือ DATA ถูกสร้างหลายครั้ง unique index อาจขาดอยู่
แก้: เพิ่ม unique index บน column (iss_seq, iss_off_loc_code)

Kafka message ไม่ถึง consumer

อาการ: notification ไม่ไปถึง mobile app
สาเหตุ: Kafka broker ไม่พร้อม, topic ไม่มี, partition issue
แก้: ดู log Kafka message failed to send, ตรวจ KAFKA_HOST, KAFKA_PORT, KAFKA_NOTIFICATION_TOPIC

User deletion stuck ที่ EXPIRED

อาการ: user_deletion_queue มี row ค้างที่ status EXPIRED
สาเหตุ: Job autoDeleteUsers fail ก่อน update status เป็น DONE
แก้: ตรวจ log Error deleting users, ตรวจ database lock หรือ constraint ที่ blocking transaction

Certificate หมดอายุ

อาการ: Job `autoGetComplaintsReport` log `SSL/Certificate error` ทุกวัน
แก้: ต่ออายุ certificate จาก DLT, แปลงเป็น PEM ใหม่ (`npm run convert:cert`), redeploy