

الجمهورية العربية السورية

المعهد العالي للعلوم التطبيقية والتكنولوجيا

اختصاص شبكات ونظم تشغيل

العام الدراسي 2024/2025

مشروع أُعِدَّ

لنيل درجة الإجازة في الهندسة المعلوماتية

(الشبكات ونظم التشغيل)

# بناء منصّة اجتماعية لمشاركة الصّور قابلة للتوسّع

إعداد

رنيم يوسف عصفورة

إشراف

م. محمد بشار دسوقي

# الفصل الأول

## التعريف بالمشروع

نقدّم في هذا الفصل تصوراً عاماً حول فكرة المشروع وأهدافه الأساسية، والمتطلبات

### 1.1 مقدمة

### 2.1 الهدف من المشروع

يهدف هذا المشروع إلى بناء منصة اجتماعية متكاملة تتميز بالمرونة وقابلية التوسع، بما يتيح تشغيل كل مكون أو وظيفة بشكل مستقل، وتحقيق إدارة فعالة للموارد، واستيعاب النمو التدريجي في عدد المستخدمين والبيانات. يركّز المشروع على تلبية متطلبات الأداء العالي، وتحمل الضغط التشغيلي، وضمان استمرارية الخدمة دون انقطاع.

ولتحقيق هذه الأهداف، يعتمد المشروع على نمط معماري مرّن يتيح فصل المهام وتوزيعها، مع دمج أدوات مراقبة وتحليل أداء تمكّن من تتبّع سلوك النظام في الزمن الحقيقي، واكتشاف نقاط الضعف أو الأعطال بسرعة.

ولذلك، تمّ اختيار معمارية الخدمات المصغّرة (Microservices) كنمط تصميم رئيسي، مع نشرها باستخدام Kubernetes لتوفير بيئة قابلة للتوسع الآلي، وتسهيل عمليات التطوير، الاختبار، والنشر المستمر، مما يجعل المنصة نموذجاً تطبيقياً يعكس متطلبات الأنظمة الحديثة.

### 3.1 المتطلبات

#### 1.3.1 المتطلبات الوظيفية (Functional Requirements)

يجب أن يحقق النظام الوظائف التالية:

1. السماح بتسجيل مستخدم جديد ضمن المنصة.
2. السماح بتسجيل الدخول إلى المنصة.
3. السماح للمستخدم بنشر صورة جديدة مع وصف نصي (Caption).
4. السماح للمستخدم بتعديل معلومات حسابه.
5. السماح للمستخدم بمتابعة مستخدمين آخرين (Follow) أو إلغاء المتابعة (Unfollow).

6. السماح للمستخدم بالبحث عن مستخدمين آخرين باستخدام الاسم الأول.
7. السماح للمستخدم بعرض الخلاصات (Timeline) التي تجمع منشورات المستخدمين الذين يتابعهم، مرتبة حسب الزمن.
8. السماح للمستخدم بعرض ملفه الشخصي.
9. السماح للمستخدم بعرض الملف الشخصي لمستخدم آخر.

## 2.3.1 المتطلبات غير الوظيفية

### 1. قابلية التوسع (Scalability):

يجب أن يكون النظام قادراً على دعم عدد كبير من المستخدمين والطلبات مع الملايين من الزيارات اليومية. بالنظر إلى حجم البيانات المتوقع، حيث يقوم كل مستخدم برفع صورة يومياً بمتوسط حجم 2 ميجابايت، يجب أن يكون النظام قادراً على معالجة بيانات بحجم يصل إلى 1 تيرابايت يومياً.

### 2. التوفر (Availability):

يجب أن تضمن المنصة مستوى عالٍ من الإتاحة للخدمات بنسبة تصل إلى 99.99% لضمان استمرارية عمل النظام طوال الوقت دون انقطاع، حتى في حالات الضغط أو فشل إحدى العقد.

### 3. الأداء (Performance):

يجب على النظام أن يستجيب لمختلف أنواع الطلبات خلال زمن لا يتجاوز 500 ميلي ثانية في الظروف التشغيلية النموذجية، حتى عند الوصول إلى ذروة الاستخدام.

أما في حالة استرجاع الخلاصة الزمنية (Timeline)، والتي تُعد من أكثر العمليات كثافة من حيث البيانات، فيجب على النظام أن يقوم بعرض المحتوى خلال زمن لا يتجاوز 1000 ميلي ثانية (1 ثانية) في 99% من السيناريوهات التشغيلية، بما يضمن سرعة تحميل المنشورات والصور وتفاعل المستخدم مع المنصة دون تأخير ملحوظ.

## الفصل الثاني

### الدراسة النظرية

نعرض في هذا الفصل الدراسات النظرية المستخدمة ضمن هذا العمل

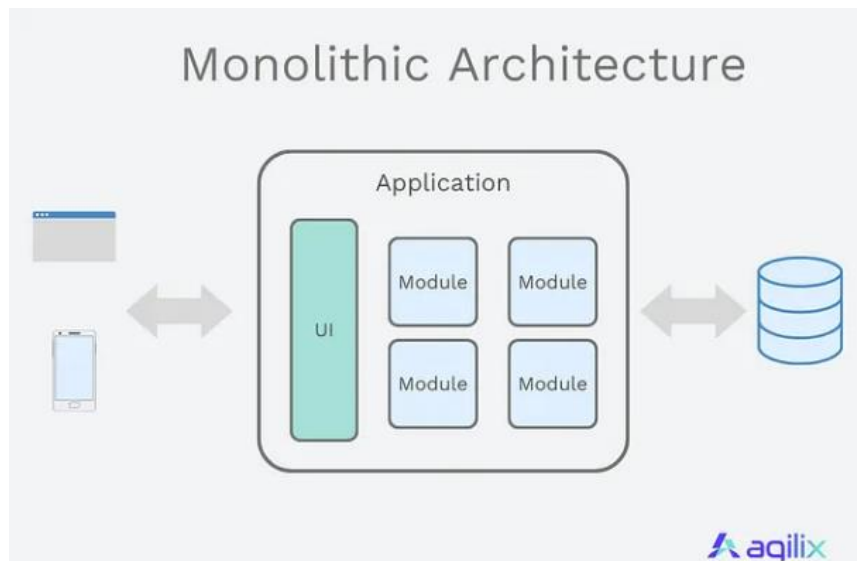
## 1.2 بنية تطبيقات الكتلة الواحدة monolithic architecture

### 1.1.2 تعريف

بنية تطبيقات الكتلة الواحدة (Monolithic Architecture) تمثل أسلوباً تقليدياً في تطوير البرمجيات، يقوم على بناء النظام البرمجي كوحدة متماسكة تضم كافة مكوناته الأساسية ضمن تطبيق واحد يتم نشره كوحدة واحدة. يشمل ذلك عادةً واجهة المستخدم (UI)، منطق الأعمال (Business Logic)، وطبقة الوصول إلى البيانات (Data Access Layer) وغيرها من المكونات، مما يؤدي إلى وجود درجة عالية من الترابط بينها. تتميز هذه البنية بسهولة التطوير والإدارة في المراحل الأولى من عمر المشروع، حيث يمكن لفرق العمل الصغيرة تطوير النظام واختباره ونشره بسرعة وبتعقيد أقل مقارنةً بالأنماط المعمارية الأخرى. إلا أن هذا الترابط الوثيق يُشكل تحدياً مع تزايد حجم النظام وتعقيد وظائفه، إذ يصبح إجراء أي تعديل أو تحديث في جزء معين من التطبيق بحاجة إلى إعادة بناء ونشر التطبيق بالكامل، كما قد يؤدي حدوث خلل في مكون واحد إلى التأثير على النظام بأكمله. إضافةً إلى ذلك، غالباً ما تُعاني هذه البنية من صعوبة التوسع الأفقي (Horizontal Scaling) نظراً لاعتمادها الكبير على النشر كوحدة متكاملة، وهو ما يحدّ من قدرتها على الاستجابة لارتفاع أعداد المستخدمين أو الطلبات. وعلى الرغم من هذه التحديات، لا تزال بنية الكتلة الواحدة خياراً عملياً في بعض الحالات، لاسيما في المشاريع الصغيرة أو عند إطلاق النسخة الأولية (Minimum Viable Product – MVP)، على أن يتم لاحقاً التفكير في إعادة هيكلة النظام باستخدام بنى أكثر مرونة مثل بنية الخدمات المصغرة (Microservices) لتحسين القابلية للتوسع والصيانة مستقبلاً [1].

(.Newman, 2015; Richards, 2015; Fowler, 2015; Richardson, n.d)

فيما يلي شكل يوضح بنية تطبيقات Monolithic :



الشكل 1: بنية تطبيق Monolithic

## 2.1.2 مميزات بنية تطبيقات الكتلة الواحدة (Monolithic Architecture)

### 1. سهولة التطوير الأولي

تصميم النظام كوحدة متكاملة يقلل الحاجة لتقسيم المكونات أو التخطيط المعقد، مما يُسهّل على فرق التطوير البدء بسرعة في إنشاء الوظائف الأساسية للتطبيق دون الحاجة لتعلم تقنيات متقدمة أو أنماط معمارية معقدة.

### 2. بساطة النشر (Deployment)

يتم نشر التطبيق في صورة ملف تنفيذي واحد أو حزمة (مثل JAR أو WAR في Java)، مما يجعل عملية النشر أبسط وأقل عرضة للأخطاء مقارنة ببنية معمارية تعتمد على خدمات متعددة منفصلة.

### 3. سهولة الاختبار كوحدة متكاملة

لأن جميع المكونات تعمل ضمن نفس العملية (Process)، يمكن اختبار النظام بالكامل بسهولة باستخدام بيئة واحدة، دون الحاجة لإعداد بيئات متعددة أو محاكاة اتصالات عبر الشبكة.

### 4. أداء عالٍ في الاتصالات الداخلية

الاستدعاءات بين مكونات النظام تتم داخلياً في الذاكرة (In-process calls)، بدلاً من التواصل عبر بروتوكولات الشبكة مثل HTTP أو gRPC، مما يقلل زمن الاستجابة ويحسن الأداء بشكل ملحوظ.

### 5. مناسب للمشاريع الصغيرة أو MVP

يُعد اختياراً عملياً عند إنشاء نموذج أولي (Prototype) أو منتج أولي قابل للإطلاق (MVP)، حيث لا يكون من المجدي استثمار وقت وجهد كبير في تصميم بنية معقدة قبل التأكد من ملاءمة الفكرة للسوق.

## 3.1.2 تحديات ومشاكل بنية تطبيقات الكتلة الواحدة (Monolithic Architecture)

### 1. صعوبة التوسع الأفقي (Horizontal Scaling)

يصعب تشغيل أجزاء مختلفة من النظام على خوادم متعددة بشكل مستقل لأن جميع المكونات مرتبطة ببعضها داخل التطبيق الواحد، مما يحدّ من قدرة النظام على استيعاب عدد أكبر من المستخدمين.

## 2. تعقيد التحديث والصيانة

أي تعديل أو إضافة ميزة جديدة في جزء من النظام غالباً يتطلب إعادة بناء ونشر التطبيق بأكمله، مما يزيد من وقت التطوير ويزيد احتمالية ظهور مشاكل جديدة.

## 3. الترابط الوثيق بين المكونات (Tight Coupling)

يعني أن كل مكون يعتمد بشكل مباشر على الآخرين، ما يجعل استبدال أو تحديث أي مكون أمراً معقداً، ويحدّ من إمكانية استخدام تقنيات أو لغات برمجة مختلفة.

## 4. انخفاض المرونة التقنية

بسبب طبيعتها الأحادية، يكون من الصعب اعتماد تقنيات مختلفة في مكونات محددة داخل النظام؛ على سبيل المثال، لا يمكن استخدام قاعدة بيانات مختلفة لمكون واحد فقط بسهولة.

## 5. تأثير الأعطال على النظام بالكامل

عطل أو خطأ في مكون واحد قد يؤدي إلى تعطل التطبيق بأكمله لأن جميع المكونات تعمل ضمن نفس العملية، مما يقلل من الاعتمادية (Reliability).

## 6. صعوبة فهم النظام مع زيادة الحجم

مع نمو حجم التطبيق وزيادة تعقيد الكود، يصبح من الصعب على المطورين الجدد فهم العلاقة بين المكونات أو تحديد تأثير أي تعديل على الأجزاء الأخرى.

# 2.2 بنية تطبيقات الأنظمة الموزعة Architecture Distributed

## Systems

### 1.2.2 تعريف

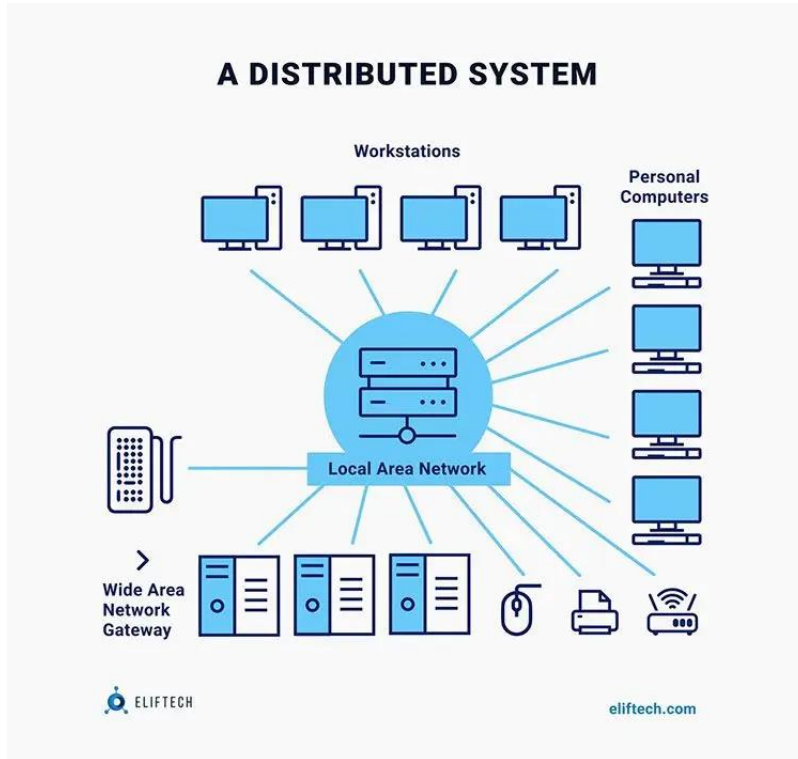
تُعد بنية تطبيقات الأنظمة الموزعة نموذجاً معمارياً يتم فيه تصميم النظام البرمجي بحيث تتوزع مكوناته ووحداته الوظيفية على عدة حواسيب أو خوادم مستقلة ومتصلة عبر شبكة اتصالات، وتتعاون هذه المكونات لإنجاز مهام مشتركة بشكل متكامل وشفاف للمستخدم النهائي.

ومن المهم التمييز بين المفهوم العام للأنظمة الموزعة وبين نمط الخدمات المصغرة (Microservices)، إذ إن التوزيع لا يعني بالضرورة اعتماد هذا النمط. فحتى التطبيقات المصممة كبنية واحدة (Monolithic) يمكن أن تُنشر وتُوزع على أكثر من خادم أو بيئة تشغيل، مما يجعلها تُصنّف ضمن الأنظمة الموزعة من حيث النشر، لا من حيث التصميم البنيوي.

تتميز هذه البنية بفصل الموارد والعمليات بين عدة مواقع جغرافية أو بيئات تشغيل، مما يعزز من قدرة النظام على التوسع، والموثوقية، والتوافر، كما تسمح بتوزيع الحمل وتقسيم الوظائف بما يتناسب مع متطلبات الأداء والقدرة. كما تعتمد الأنظمة الموزعة على آليات متقدمة لإدارة الاتصالات، التزامن، وتحمل الأخطاء، وذلك لضمان تنسيق فعال بين المكونات المتناثرة وإتاحة التكامل الوظيفي دون الإخلال بجودة الخدمة.

وتُستخدم هذه البنية بشكل واسع في التطبيقات التي تتطلب أداءً عالياً، مرونة في التوسع، وتحقيق استمرارية في العمل، مثل الحوسبة السحابية، قواعد البيانات الموزعة، والخدمات الإلكترونية. إن طبيعة التوزيع هذه تُحدث تحديات فريدة في مجال تصميم النظام، مثل تعقيد التنسيق، ضمان سلامة البيانات، وإدارة الاتصالات بين المكونات، والتي تستلزم تطبيق نماذج وبروتوكولات متخصصة لضمان عمل النظام بكفاءة وفعالية (Tanenbaum & van Steen, 2017; Coulouris et al., 2011; Buschmann et al., 2007).

فيما يلي شكل يوضح بنية تطبيقات الأنظمة الموزعة:



الشكل 2: بنية تطبيق نظام موزع

## 2.2.2 خصائص الأنظمة الموزعة

### ■ الشفافية (Transparency):

تعني إخفاء التعقيد الكامن في توزيع مكّونات النظام عن المستخدمين والمبرمجين، بحيث يبدو النظام وكأنه يعمل كوحدة موحدة. تنقسم الشفافية إلى عدّة أنواع مثل شفافية الوصول (Access Transparency)، شفافية الموقع (Location Transparency)، شفافية التكرار (Replication Transparency)، وشفافية الفشل (Failure Transparency)، حيث يهدف كل نوع إلى إخفاء جانب مختلف من تعقيد النظام.

### ■ التوسّعية (Scalability):

قدرة النظام على التعامل مع زيادة عدد المستخدمين أو حجم البيانات أو عدد العقد دون أن ينخفض الأداء بشكل ملحوظ. يتحقق ذلك عادةً عبر التوسّع الأفقي (إضافة أجهزة جديدة) أو التوسّع العمودي (زيادة قدرات الأجهزة الحالية).

### ■ الموثوقية (Reliability):

قدرة النظام على الاستمرار في تقديم خدماته حتى في حالة حدوث أعطال في بعض مكّوناته. تعتمد الموثوقية على آليات مثل التكرار، الاكتشاف التلقائي للأعطال، وإعادة التوجيه لتقليل تأثير الفشل.

### ■ تحمّل الأعطال (Fault Tolerance):

قدرة النظام على الاستمرار في العمل بشكل صحيح حتى عند حدوث أخطاء أو فشل في بعض الأجزاء. يتم ذلك باستخدام النسخ الموزعة، وخوارزميات الكشف عن الأخطاء، وآليات إعادة التشغيل أو التحويل التلقائي.

### ■ الاستقلالية (Heterogeneity):

دعم تشغيل النظام على مكّونات مختلفة ومتنوعة من حيث العتاد، أنظمة التشغيل، أو لغات البرمجة، مع ضمان التكامل بينها. هذه الخاصية تُمكن النظام من العمل في بيئات مختلطة دون الحاجة لتوحيد البنية التحتية.

### ■ التزامن (Concurrency):

تمكين عدة عمليات أو مستخدمين من الوصول إلى نفس الخدمة أو المورد في وقت متزامن، مع ضمان سلامة البيانات واتّساق النتائج. تتطلب هذه الخاصية استخدام آليات إدارة التزامن مثل الأقفال أو البروتوكولات الموزعة.

### ■ قابلية التوسّع الديناميكي (Dynamic Scalability):

تمكين النظام من إضافة أو إزالة عقد أو موارد أثناء التشغيل دون الحاجة لإيقاف النظام أو التأثير على المستخدمين.



## ■ الأمان (Security):

يشمل ضمان سرية البيانات (Confidentiality)، سلامة البيانات (Integrity)، وتوفر الخدمات (Availability). في الأنظمة الموزعة، يصبح الأمان أكثر تعقيداً نتيجة لانتشار المكونات وتبادل البيانات عبر الشبكات.

## ■ الإدارة المركزية أو اللامركزية (Centralized/Decentralized Management):

الأنظمة الموزعة قد تعتمد على إدارة مركزية لتبسيط الصيانة أو على إدارة لامركزية لزيادة المرونة وتحسين التحمل للأعطال.

## ■ قابلية التكيف (Adaptability):

قدرة النظام على التكيف مع التغيرات الديناميكية في الحمل أو البنية التحتية أو متطلبات العمل دون الحاجة إلى تغييرات جوهرية في التصميم.

## 3.2.2 فوائد ومميزات الأنظمة الموزعة

### ■ زيادة توافر الخدمة (Improved Service Availability):

تؤدي الأنظمة الموزعة إلى تقليل فترات التوقف غير المخطط لها بفضل وجود نسخ متعددة من الخدمات والبيانات، مما يوفر استمرارية التشغيل حتى عند حدوث فشل في بعض المكونات.

### ■ تسريع المعالجة وتحسين الأداء (Enhanced Performance):

تقسيم العمل وتوزيعه على عدة عقد يتيح تنفيذ العمليات بشكل متواز، مما يقلل زمن الاستجابة ويحسن الأداء الكلي للنظام.

### ■ التوسع وفق الحاجة (On-Demand Scalability):

يمكن إضافة موارد جديدة بسهولة لتلبية الزيادة في عدد المستخدمين أو حجم البيانات، دون الحاجة إلى تعديل جذري في النظام.

### ■ استغلال الموارد بشكل أفضل (Better Resource Utilization):

يمكن لمكونات النظام في مواقع مختلفة مشاركة الموارد المتاحة مثل التخزين ووحدات المعالجة والخدمات، مما يقلل الهدر ويحسن الكفاءة.

### ■ مرونة التحديث والتطوير (Ease of Evolution and Maintenance):

يسهل تعديل أو استبدال مكونات النظام دون التأثير الكبير على باقي الأجزاء، مما يمكن فرق التطوير من تحسين النظام بشكل تدريجي ودائم.

- التوزيع الجغرافي للخدمات (Geographical Flexibility):  
يتيح نشر أجزاء النظام في مواقع متعددة حول العالم لتقديم الخدمة بشكل أسرع للمستخدمين المحليين وتقليل زمن الاستجابة.
- زيادة الأمان من خلال التوزيع (Security Through Distribution):  
توزيع البيانات والخدمات عبر عدة عقد يقلل من المخاطر المرتبطة بوجود نقطة فشل واحدة أو اختراق وحيد قد يؤثر على النظام بالكامل.
- دعم التعاون بين الأنظمة المختلفة (Support for Heterogeneous Systems):  
يمكن للأنظمة الموزعة التكامل بسهولة مع مكونات مبنية بلغات مختلفة أو تعمل على أنظمة تشغيل مختلفة، مما يعزز إمكانية إعادة استخدام الأنظمة القائمة.
- خفض التكاليف التشغيلية (Cost Efficiency):  
يمكن استخدام خوادم منخفضة التكلفة موزعة بدلاً من شراء جهاز مركزي قوي ومكلف جداً، كما أن توزيع الاحمال يقلل الحاجة لصيانة معقدة.
- تحسين تجربة المستخدم النهائي (Better User Experience):  
بفضل التوزيع الجغرافي والتوافر المستمر، يحصل المستخدمون على خدمة أسرع وأكثر موثوقية، مما يرفع مستوى الرضا والثقة في النظام.
- تكرار البيانات (Data Replication):  
يسهم نسخ البيانات وتكرارها عبر عدة عقد في زيادة التوافر والموثوقية، بحيث يظل الوصول إلى البيانات ممكناً حتى عند حدوث فشل في بعض الخوادم.
- محلية البيانات (Data Locality):  
يتيح تخزين البيانات بالقرب من موقع استخدامها التقليل من زمن الوصول وتحسين الاداء للمستخدمين أو التطبيقات القريبة جغرافياً.

## 4.2.2 تحديات ومشكلات الأنظمة الموزعة

- إدارة التعقيد (Managing Complexity):

تصميم نظام موزع يتطلب تنسيقا دقيقا بين مكونات متعددة تعمل في بيئات مختلفة، مما يزيد من التعقيد مقارنة بالتطبيقات الاحادية.

■ التزامن (Synchronization):

ضمان ترتيب العمليات بشكل صحيح عبر عقد مختلفة يمثل تحديا كبيرا خاصة عند التعامل مع البيانات المشتركة وتفادي حالات التداخل بسبب تبادل العمليات في وسيط غير موثوق مثل الشبكة.

■ الكشف عن الأعطال (Fault Detection):

من الصعب تحديد ما اذا كانت العقدة غير مستجيبة بسبب عطل فعلي ام بسبب تأخر في الشبكة، مما يجعل معالجة الاعطال اكثر تعقيدا.

■ الاتساق (Consistency):

الحفاظ على نسخ البيانات متماثلة ومتسقة عبر النظام يمثل تحديا كبيرا خصوصا عند استخدام تكرار البيانات او التعامل مع تحديثات متزامنة.

■ الأمان (Security):

حماية البيانات والخدمات الموزعة تتطلب استراتيجيات امنية معقدة تشمل التحقق من الهوية والتفويض والتشفير، نظرا لتعدد نقاط الدخول المحتملة.

■ ادارة الموارد (Resource Management):

تخصيص موارد الحوسبة والتخزين والشبكة بشكل عادل وكفء بين العقد المختلفة يمثل تحديا في ظل تغير الطلب باستمرار.

■ التوسع (Scalability Challenges):

ضمان أن يظل النظام قادراً على تقديم أداء جيد عند زيادة عدد العقد أو حجم البيانات أو الطلب على التطبيق يتطلب تصميمًا خاصًا ودعمًا لبروتوكولات مناسبة.

■ تعدد المنصات (Heterogeneity):

دمج مكونات مبنية بلغات مختلفة او تعمل على انظمة تشغيل مختلفة يزيد من صعوبة التطوير والاختبار والصيانة.

■ زمن الاستجابة المتغير (Variable Latency):

تفاوت أوقات الاستجابة بين العقد بسبب عوامل الشبكة قد يؤدي الى بطء الخدمة أو تجربة مستخدم غير مستقرة.

■ ادارة التكوين (Configuration Management):

تحديث وتغيير مكونات النظام دون التأثير على الخدمة يتطلب عمليات نشر وصيانة دقيقة ومعقدة.

■ التكاليف (Costs):

بناء وتشغيل نظام موزع يحتاج الى استثمار في البنية التحتية والبرمجيات وأدوات الإدارة والمراقبة، مما يزيد التكاليف مقارنة بالانظمة البسيطة.

■ زمن الاتصال (Communication Latency):

زمن الانتقال بين العقد المختلفة يمكن ان يكون مرتفعاً خصوصاً عند التوزيع الجغرافي الواسع، مما يؤثر سلباً على سرعة المعالجة.

■ استكشاف الاخطاء واصلاحها (Debugging and Troubleshooting):

تحديد مصدر المشكلة في نظام موزع معقد يعد أمراً صعباً بسبب تعدد المكونات وتداخلها وصعوبة تتبع مسارات التنفيذ.

■ مراقبة الانظمة (System Monitoring):

تتطلب الانظمة الموزعة بناء أدوات قوية للمراقبة والتحليل لاكتشاف المشاكل والأعطال مبكراً وضمان الاستقرار المستمر.

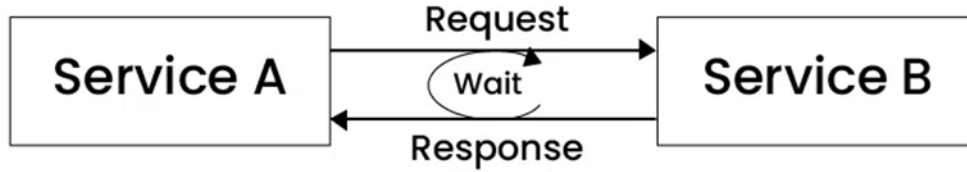
## 5.2.2 أنواع الاتصال في الأنظمة الموزعة

في الأنظمة الموزعة تعتمد مكونات النظام المختلفة – مثل الخوادم والعملاء والعقد الوسيطة – على الاتصال فيما بينها لتبادل الرسائل والبيانات والاحداث بمهدف تنسيق العمليات وتحقيق اهداف النظام بشكل موثوق وفعال. يمكن أن يتم هذا الاتصال عبر بروتوكولات ومستويات مختلفة، بدءاً من نقل الرسائل البسيطة وحتى استدعاء الاجراءات عن بُعد. تنقسم اساليب الاتصال بشكل عام الى عدة أنواع رئيسية، يختلف كل منها في خصائصه وطرق استخدامه ومدى ملائمته للتطبيقات المختلفة. فيما يلي توضيح لأهم هذه الأنواع:

1. الاتصال المتزامن (Synchronous Communication):

يعتمد هذا النوع على انتظار الجهة المرسل (client) حتى تستلم استجابة من الجهة المستقبلة (server) قبل متابعة التنفيذ. يستخدم عادة في استدعاء الاجراءات عن بُعد (RPC)، حيث يُسهل كتابة الكود بشكل متسلسل لكنه قد يؤدي إلى زيادة زمن الانتظار في حال تأخر الطرف الآخر.

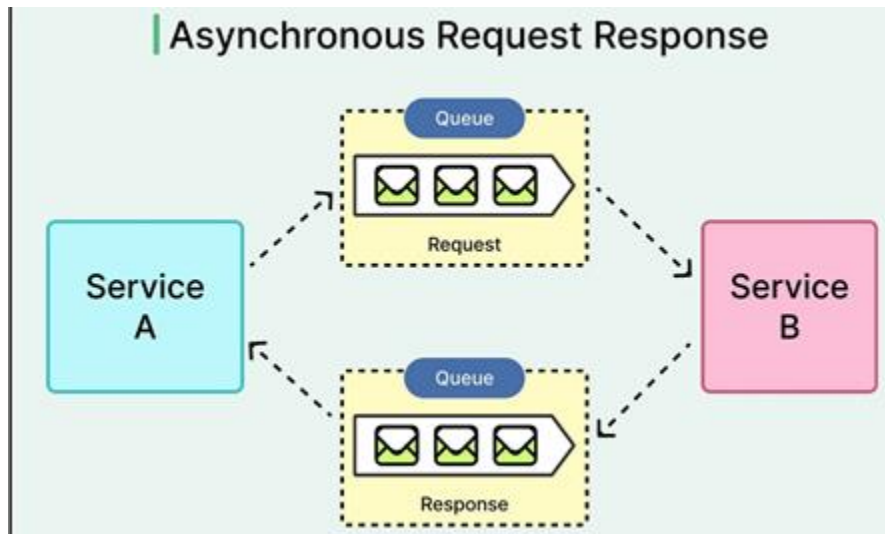
## Synchronous Communication



الشكل 3: الاتصال المتزامن بين خدمتين

2. الاتصال غير المتزامن (Asynchronous Communication):

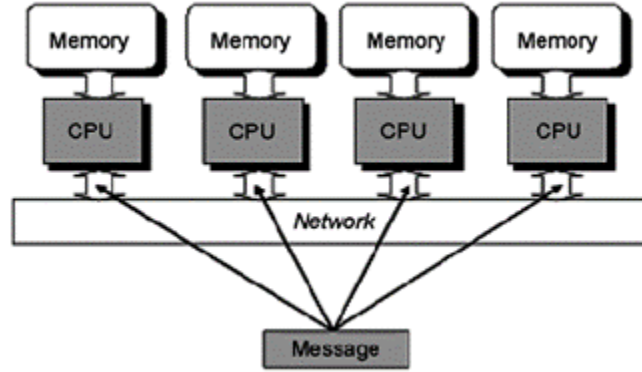
في هذا الأسلوب لا تنتظر الجهة المرسل استجابة فورية من الجهة المستقبلة، بل يمكنها متابعة المعالجة وإرسال المزيد من الرسائل. يعتمد هذا النمط غالباً على طوابير الرسائل (Message Queues) والأنظمة القائمة على الأحداث، مما يسمح بمرونة أكبر وتقليل زمن الحظر.



الشكل 4: نموذج الاتصال غير المتزامن بين الخدمات

3. نقل الرسائل (Message Passing):

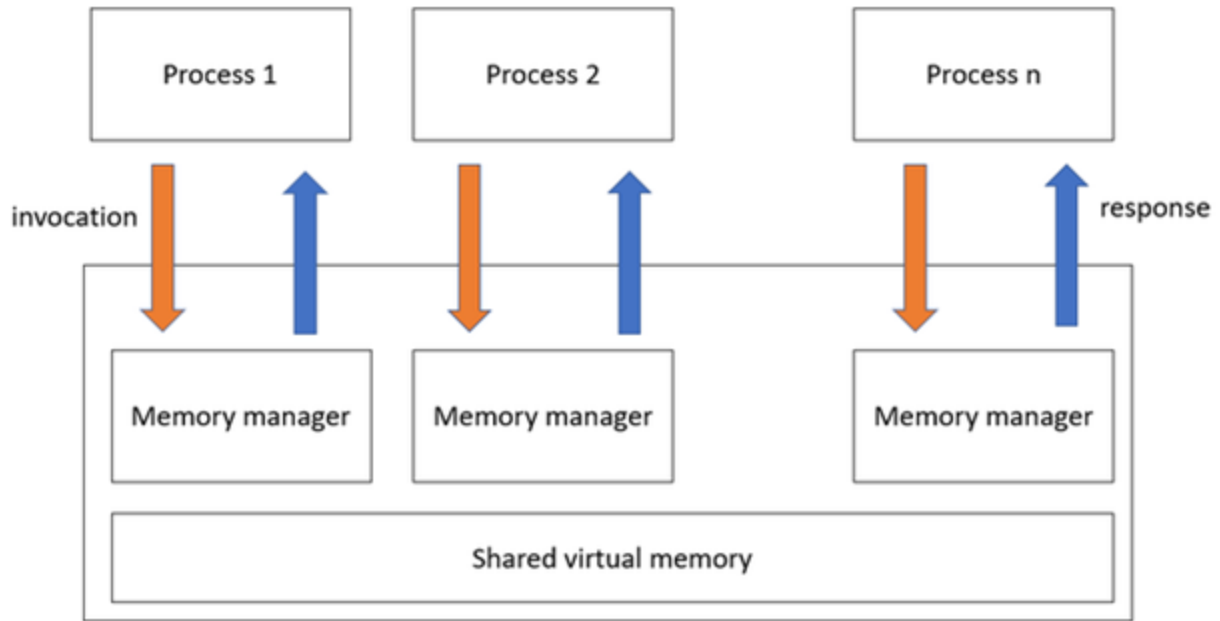
يمثل هذا النمط الأساس للاتصال في الأنظمة الموزعة، حيث يتم إرسال واستقبال رسائل تحتوي على البيانات أو الأوامر بين العقد المختلفة. يمكن ان يكون النقل مباشراً أو عبر وسائط مثل الوسطاء (Brokers)، ويُستخدم بشكل واسع في بنية الأنظمة القائمة على الخدمات (Service-Oriented Architectures).



الشكل 5: نموذج نقل الرسائل في الأنظمة الموزعة

#### 4. مشاركة الذاكرة الموزعة (Distributed Shared Memory):

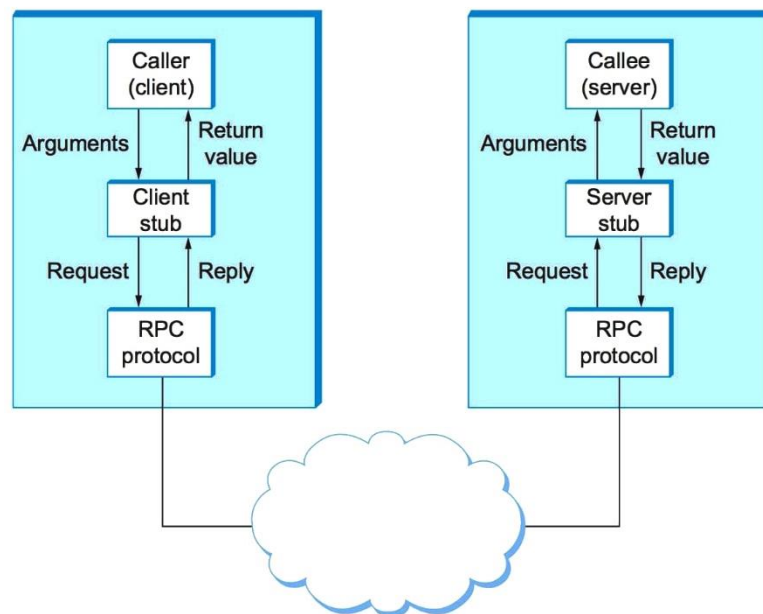
يتيح هذا الأسلوب للعقد المختلفة الوصول الى فضاء ذاكرة مشترك يُحاكي سلوك الذاكرة الموحدة، بالرغم من كونها موزعة فعلياً. يساعد هذا في تسهيل البرمجة لكنه يحتاج إلى تقنيات خاصة للحفاظ على الاتساق بين النسخ المختلفة من البيانات.



الشكل 6: نموذج مشاركة الذاكرة الموزعة بين العمليات

5. استدعاء الاجراءات عن بُعد (Remote Procedure Call – RPC):

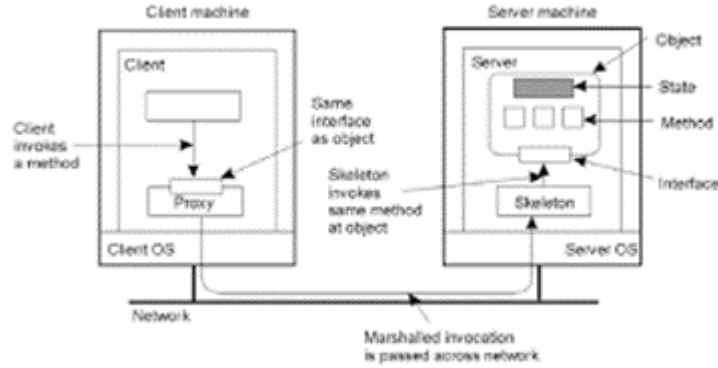
يُستخدم هذا النمط لتمكين الجهة العميلة من استدعاء دوال أو إجراءات موجودة على عقدة أخرى كما لو كانت محلية، حيث تُنقل البيانات المطلوبة عبر الشبكة وتُعاد النتائج بعد المعالجة.



الشكل 7: آلية استدعاء الإجراءات عن بُعد (RPC)

6. استدعاء الكائنات عن بُعد (Remote Object Invocation):

يُشبه استدعاء الإجراءات عن بُعد لكنه موجه للكائنات (Objects) في البرمجة غرضية التوجه، حيث يمكن للعميل استدعاء اساليب كائن بعيد والتعامل مع خصائصه، كما في تقنيات CORBA وJava RMI.



الشكل 8: آلية RMI بين العميل والخادم

7. تدفق البيانات (Data Streaming):

يُستخدم هذا النمط لتبادل كميات كبيرة من البيانات بشكل مستمر، مثل الفيديو أو الصوت أو بيانات الحساسات، ويتطلب بروتوكولات تدعم التدفق بكفاءة مثل WebRTC أو بروتوكولات البث.

### Data Stream Processing



الشكل 9: تدفق البيانات ومعالجتها لحظياً

تُظهر هذه الأنواع كيف يُمكن تصميم الأنظمة الموزعة بمرونة لتلبية متطلبات الاداء والتوسع والموثوقية، مع الموازنة بين سهولة التطوير وكفاءة التنفيذ.



## 6.2.2 الأنماط المعمارية في الأنظمة الموزعة ( Distributed System Architecture ) (Patterns)

مع تزايد تعقيد التطبيقات البرمجية الحديثة، وخاصة تلك التي تعمل في بيئات موزعة، أصبح من الضروري اعتماد أنماط معمارية (Architectural Patterns) تساعد في تنظيم بنية النظام وتحديد كيفية تفاعل مكوناته. تُعد هذه الأنماط حلولاً تصميمية مثبتة لمشكلات شائعة في هندسة البرمجيات، وهي لا تُحدد تفاصيل التنفيذ، بل ترسم الإطار العام لكيفية تنظيم وحدات النظام وتدفق البيانات والمهام فيما بينها.

ضمن سياق الأنظمة الموزعة (Distributed Systems)، تظهر الحاجة لهذه الأنماط بشكل أكثر إلحاحاً، نظراً للتحديات الفريدة المرتبطة بتوزيع المعالجة، وتحقيق التزامن، وضمان التوافر والموثوقية. تختلف هذه الأنماط من حيث أهدافها، بنيتها، ومستوى الفصل بين المكونات، كما أن بعضها يُستخدم كبنية أساسية لأنظمة ضخمة مثل تطبيقات السوشال ميديا، الحوسبة السحابية، وخدمات الويب.

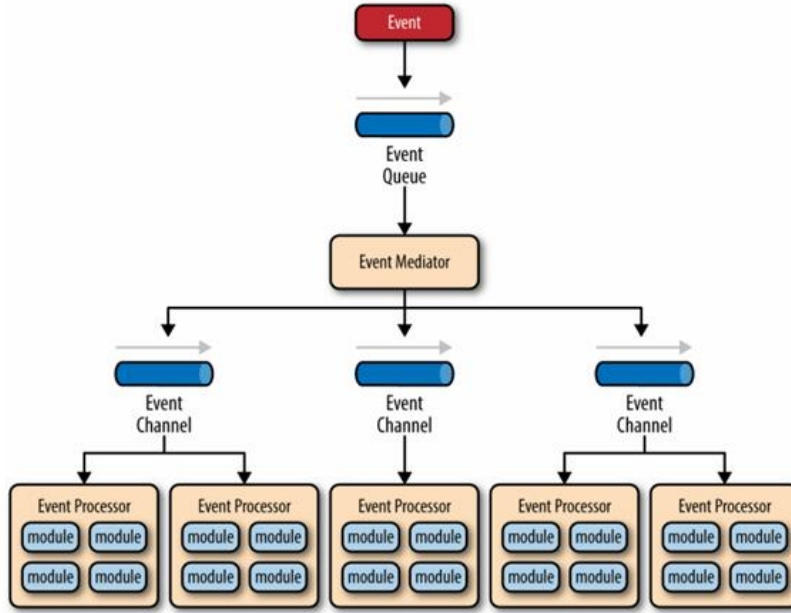
في هذا السياق، يمكن تصنيف أشهر الأنماط المعمارية المستخدمة في الأنظمة الموزعة كما يلي:

من أبرز هذه الأنماط:

### 1.6.2.2 النمط القائم على الأحداث (Event-Driven Architecture)

يعتمد هذا النمط على التفاعل غير المتزامن بين المكونات من خلال الأحداث (Events)، حيث تقوم مكونات بإنتاج أحداث، وأخرى تستهلكها. يُستخدم بكثرة في الأنظمة التي تتطلب مرونة وسرعة استجابة، مثل تطبيقات التجارة الإلكترونية أو أنظمة المراقبة. من أبرز مزاياه المرونة العالية، القابلية للتوسع، وإزالة الترابط المباشر بين المكونات، إلا أنه يتطلب نظاماً متقدماً لإدارة الأحداث، ويُصعب

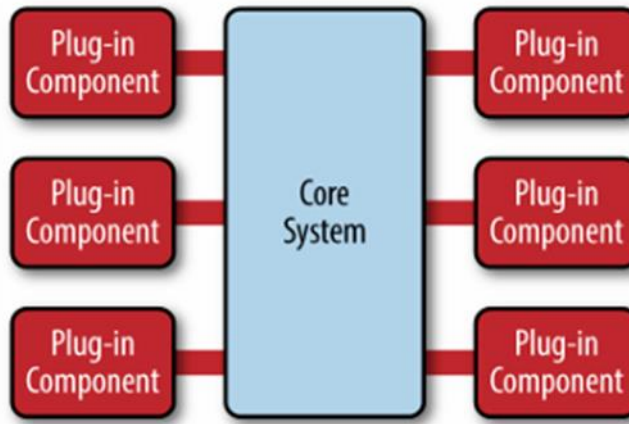
تتبع الأخطاء وفهم التدفق العام للتنفيذ [2] .



الشكل 10: نموذج البنية المعمارية القائمة على الأحداث

## 2.6.2.2 نمط النواة المصغرة (Microkernel Architecture)

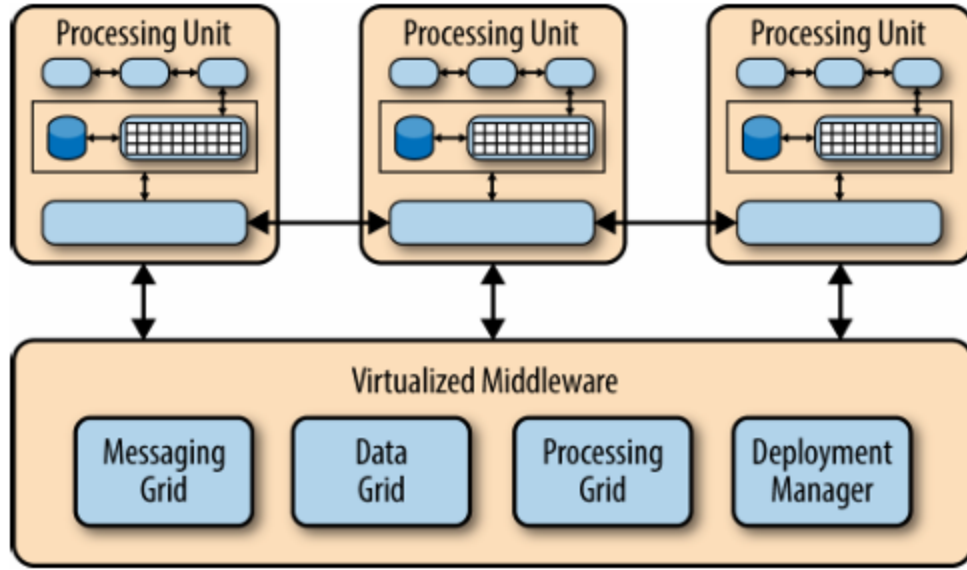
يُستخدم هذا النمط في التطبيقات التي تتطلب نواة أساسية صغيرة قابلة للتوسعة عبر مكونات إضافية (Plug-ins). تعتمد النواة على تشغيل الخدمات الأساسية فقط، بينما تُضاف الوظائف الخاصة حسب الحاجة. يوفّر هذا النمط مرونة في التحديث والصيانة، ويُستخدم كثيراً في أدوات سطح المكتب أو بيئات المعالجة المخصصة. من عيوبه تعقيد إدارة الإضافات وإمكانية تعارضها، كما أنه أقل ملاءمة للأنظمة الضخمة أو شديدة التوزيع [2].



الشكل 11: بنية نمط النواة المصغرة

### 3.6.2.2 النمط الفضائي (Space-Based Architecture)

صُمم هذا النمط خصيصاً للأنظمة التي تتعامل مع أحمال ضخمة وغير متوقعة، مثل أنظمة البنوك أو التجارة الإلكترونية ذات الحمل المرتفع. يقوم على إزالة الاعتماد على قاعدة بيانات مركزية، وتوزيع البيانات والمعالجة في "مساحات" مستقلة تُدار عبر شبكة. يمنح هذا النمط أداءً عالياً وتحملاً ممتازاً للأخطاء، لكنه معقد في التصميم ويحتاج إلى إدارة متقدمة للذاكرة المؤقتة والتكرار [2].



الشكل 12: بنية النمط الفضائي لمعالجة البيانات الموزعة

### 4.6.2.2 نمط الخدمات المصغرة (Microservices Architecture)

يُعرف نمط الخدمات المصغرة (Microservices Architecture) بأنه أسلوب معماري يُقسّم النظام البرمجي إلى مجموعة من الخدمات الصغيرة المستقلة (Independently Deployable Services)، بحيث تؤدي كل خدمة وظيفة محددة وتعمل بشكل منفصل عن بقية المكونات. تتواصل هذه الخدمات فيما بينها عبر بروتوكولات خفيفة الوزن مثل HTTP/REST أو عبر أنظمة رسائل مثل Kafka، ما يُسهّم في تحقيق مرونة عالية في التوسّع (Scalability) وإمكانية نشر الخدمات بشكل مستقل (Independent Deployment) [2].

تتميّز هذه البنية بقدرتها على توزيع عبء العمل على فرق تطوير مختلفة، وتسمح باختيار تقنيات وأطر عمل متنوعة لكل خدمة على حدة. كما تُبسّط عملية صيانة النظام وتحديثه دون الحاجة لإعادة نشر النظام بأكمله. ومع ذلك، فإنها تتطلب بنية تحتية متقدمة لإدارة الاتصال، المراقبة، التزامن، والأمان بين الخدمات.

بعد استعراض الأنماط المعمارية المختلفة، يمكننا تلخيص الخصائص الرئيسية لكل نمط في الجدول التالي الذي يُبرز أهم الفروقات ومجالات الاستخدام الأمثل لكل نمط.

النمط المعماري	درجة الترابط	قابلية التوسع	سهولة الصيانة	الأداء	مناسب لـ
(Coupling)	(Scalability)				
Monolithic	عالي جداً (Tight)	منخفضة	منخفضة عند النمو	ممتاز داخلياً	التطبيقات الصغيرة، MVP، المشاريع الأولية
Event-Driven	منخفض جداً (Loose)	عالية جداً	عالية	عالي باستخدام (async)	أنظمة الإشعارات، المراقبة، أنظمة الاستجابة الفورية
Microkernel	منخفض	متوسطة (مع الإضافات)	عالية	جيد داخلياً	أدوات سطح المكتب، نظم المعالجة الإضافية، الأنظمة القابلة للتوسعة
Space-Based	منخفض جداً (Decentralized)	عالية جداً	متوسطة	عالي جداً (بلا مركزية)	أنظمة البنوك، التجارة الإلكترونية تحت ضغط كبير
Microservices	منخفض (Loose)	عالية (أفقية وعمودية)	عالية	متغير (يعتمد على الشبكة)	أنظمة معقدة، مشاريع الفرق الكبيرة، الخدمات السحابية

## 3.2 الخدمات المصغرة Microservices

### 1.3.2 تعريف

تُعدّ بنية الخدمات المصغرة (Microservices Architecture) نموذجاً حديثاً لتصميم الأنظمة البرمجية المعقدة، يقوم على تجزئة النظام إلى مجموعة من الخدمات الصغيرة المستقلة، بحيث تؤدي كل خدمة مهمة محددة في منطق الأعمال، وتُدار وتُنشر بشكل مستقل. يُعرّف Sam Newman هذه البنية بأنها:

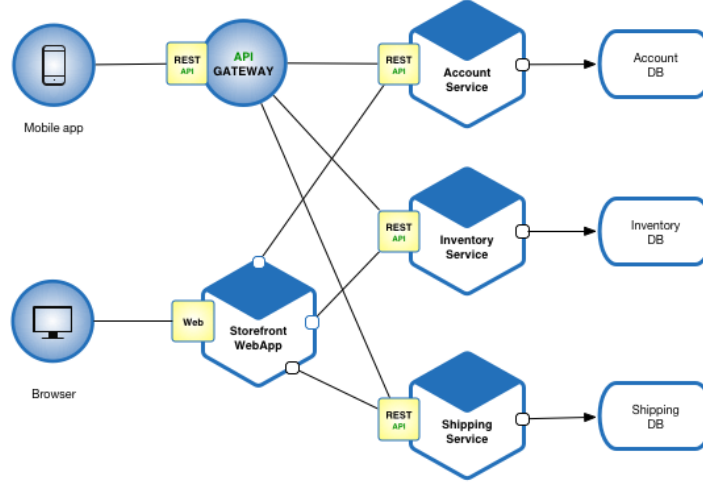
"طريقة لتطوير تطبيق برمجي واحد على شكل مجموعة من الخدمات الصغيرة، حيث تُنشر كل خدمة بشكل مستقل، وتُدير بياناتها الخاصة، وتتواصل مع غيرها من الخدمات من خلال واجهات خفيفة الوزن، غالباً باستخدام HTTP أو آليات مراسلة (Messaging)" [1] .

في هذا النموذج، يتم تصميم كل خدمة بحيث تكون:

- مستقلة من حيث النشر (Independently Deployable)
- صغيرة وذات مسؤولية واحدة (Single Responsibility)
- تدير بياناتها الخاصة دون مشاركة مباشرة (Database per Service)
- تتواصل مع الخدمات الأخرى باستخدام بروتوكولات خفيفة مثل REST أو رسائل غير متزامنة (Asynchronous Messaging).

تختلف هذه البنية مع النمط التقليدي المعروف بـ Monolithic Architecture، حيث يتم تجميع جميع المكونات في وحدة واحدة مترابطة، مما يُقيّد القابلية للتوسع والتحديث المستقل.

وقد جاءت الخدمات المصغرة كاستجابة طبيعية لتحديات النمو في الأنظمة المعقدة، لا سيما في تطبيقات الإنترنت واسعة النطاق، حيث يُصبح من غير العملي نشر التطبيق كاملاً عند كل تعديل، أو الاعتماد على قاعدة بيانات واحدة، أو تنسيق العمل ضمن فريق كبير في شيفرة واحدة.



الشكل 13: بنية الخدمات المصغرة

## 2.3.2 مكونات بنية الخدمات المصغرة (Core Components of Microservices Architecture)

تقوم بنية الخدمات المصغرة على مجموعة مترابطة من المكونات الأساسية، التي تُنظّم تفاعل الخدمات وتُعزّز استقلاليتها وقابليتها للتوسع. لا تقتصر هذه البنية على تقسيم منطق الأعمال إلى خدمات صغيرة فحسب، بل تشمل أيضاً العناصر الداعمة التي تُمكن النظام من العمل بكفاءة في بيئة موزعة. فيما يلي أبرز المكونات التي تُشكّل العمود الفقري لبنية الخدمات المصغرة:

### 1.2.3.2 الخدمات المستقلة (Independent Services)

كل خدمة مصغرة تُعدّ وحدة قائمة بذاتها، مسؤولة عن وظيفة محددة ضمن النظام، مثل إدارة المستخدمين، أو معالجة الدفع، أو إدارة الصور. تُكتب عادةً باستخدام تقنيات أو لغات مختلفة حسب الحاجة، ما يُعزّز المرونة التقنية (Technology Heterogeneity).

يُمكن لكل خدمة أن تُدار من قبل فريق منفصل، وتُختبر وتُنشر بشكل مستقل، مما يُسرّع دورة التطوير ويُقلّل الاعتماد بين الفرق.

### 2.2.3.2 واجهة برمجة التطبيقات (API Communication Layer)

تتواصل الخدمات فيما بينها من خلال واجهات خفيفة الوزن، غالباً باستخدام RESTful APIs أو بروتوكولات المراسلة مثل gRPC أو Kafka.

ويُمكن أن يكون الاتصال إما متزامناً (Synchronous) عبر HTTP، أو غير متزامن (Asynchronous) باستخدام Message Brokers، ما يُساهم في تحقيق الفصل الكامل بين المكونات وتحسين الاستجابة تحت الضغط.

### 3.2.3.2 بوابة الدخول (API Gateway)

تُشكّل البوابة الطبقة الأمامية التي تُوجّه الطلبات إلى الخدمات المناسبة، وتُخفي تفاصيل توزيعها عن المستخدم أو العميل. تُوفّر هذه البوابة ميزات مهمّة مثل:

- التجميع (Aggregation) لعدة ردود في استجابة واحدة.
- المصادقة والتحكم في الوصول (Authentication & Authorization).
- التحويل بين الإصدارات (Versioning).
- التحكم بمعدل التدفق (Rate Limiting).

يُعتبر وجود API Gateway عاملاً حاسماً في نجاح بنية الخدمات المصغّرة، لما يُوفّره من تنظيم مركزي ومرونة في التحكم.

### 4.2.3.2 إدارة الخدمة والاكتشاف (Service Discovery)

في بيئة متغيرة وديناميكية، قد تُنشّر الخدمات أو تُعاد جدولتها في أماكن مختلفة باستمرار. لذا، يُستخدم نظام اكتشاف الخدمة (Service Discovery) لِيُتيح للخدمات العثور على بعضها البعض دون الحاجة لعناوين ثابتة. ويمكن أن يكون هذا الاكتشاف:

- مركزياً (Centralized): باستخدام أدوات مثل Consul أو Eureka.
- أو مدفوعاً بالعميل (Client-Side): حيث تتولى كل خدمة تحديد موقع الخدمات الأخرى.

### 5.2.3.2 قواعد البيانات المستقلة (Database per Service)

لتلزم كل خدمة في بنية Microservices بامتلاك قاعدة بيانات خاصة بها، دون مشاركة مباشرة مع خدمات أخرى، ما يُعزز مبدأ استقلالية البيانات (Data Ownership).

يُمكن أن تختلف قواعد البيانات حسب الحاجة مما يُتيح تحسين الأداء حسب طبيعة كل خدمة.

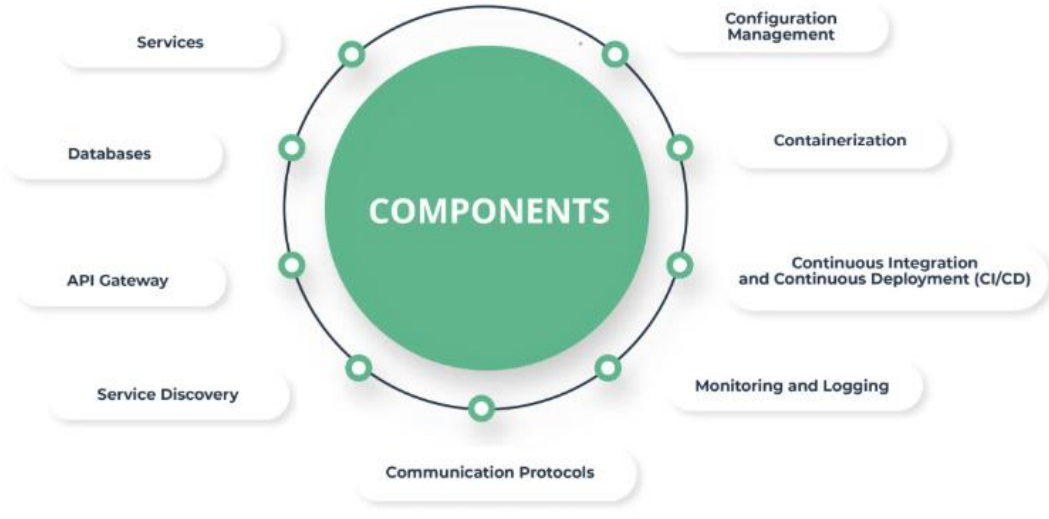
### 6.2.3.2 المراقبة والتسجيل (Monitoring & Logging)

نظراً لتعدد الخدمات وانتشارها، تصبح المراقبة المركزية ضرورة حتمية. تُستخدم أدوات مثل Prometheus, Grafana, و ELK Stack (Elasticsearch, Logstash, Kibana) لرصد الأداء، وجمع السجلات، وتنبيه الفرق الفنية عند وقوع مشاكل.

### 7.2.3.2 التكامل والتسليم المستمر (CI/CD Pipelines)

بما أن كل خدمة تُنشّر بشكل مستقل، يجب أن تُجهّز آليات أتمتة لبنائها واختبارها وتحديثها باستمرار. تُستخدم أدوات مثل:

- GitHub Actions, Jenkins, GitLab CI للبناء والاختبار.
- Docker و Kubernetes لإدارة الحاويات والنشر التلقائي.



الشكل 14: المكونات الأساسية لنظام الخدمات المصغرة

### 3.3.2 مزايا الخدمات المصغرة (Advantages of Microservices Architecture)

تُوفّر بنية الخدمات المصغرة (Microservices Architecture) عدداً من المزايا التقنية والتنظيمية التي جعلتها خياراً رئيساً في تصميم الأنظمة البرمجية الحديثة واسعة النطاق. وقد جاءت هذه المزايا كاستجابة مباشرة للتحديات المرتبطة بالأنظمة الأحادية (Monolithic Systems)، حيث تسعى الخدمات المصغرة إلى تحسين المرونة، وتسهيل النشر، وتعزيز قابلية التطوير والتوسع الأفقي.

فيما يلي أهم هذه المزايا:

#### 1. الاستقلالية في النشر والتطوير (Independent Deployment and Development)

تُبنى كل خدمة مصغرة كوحدة قائمة بذاتها، يمكن تطويرها، واختبارها، ونشرها بشكل منفصل عن بقية النظام. وهذا يسمح للفرق البرمجية بالعمل بشكل متوازي على مكونات مختلفة، دون الحاجة إلى تنسيق مستمر أو إعادة بناء النظام بالكامل.

كما يُساهم هذا الأسلوب في تقليل وقت التوصيل إلى السوق (Time to Market)، وتحقيق مرونة عالية في إدارة التحديثات.



## 2. قابلية التوسع المرنة (Scalability)

تُتيح الخدمات المصمّنة إمكانية التوسع الأفقي (Horizontal Scaling) لكل خدمة على حدة، بما يتناسب مع حجم الضغط عليها. على سبيل المثال، يمكن توسيع خدمة تحميل الصور وحدها دون الحاجة إلى توسيع النظام بأكمله، مما يؤدي إلى استخدام أكثر كفاءة للموارد، وخفض في التكلفة [3].

## 3. المرونة التقنية (Technological Flexibility)

نظراً لاستقلالية الخدمات، يُمكن لكل فريق أن يختار لغة البرمجة (Programming Language)، أو نظام إدارة قواعد البيانات (Database System)، أو البنية التحتية (Infrastructure) التي تناسب طبيعته. فمثلاً، قد تُكتب خدمة المصادقة بلغة Java، بينما تُطوّر خدمة البحث باستخدام Node.js، وهذا يُعرف بمبدأ Polyglot Programming [4].

## 4. تحسين الاستقرار والموثوقية (Improved Resilience and Fault Isolation)

في حال حدوث خلل في إحدى الخدمات، فإن الأثر لا ينتقل تلقائياً إلى بقية النظام كما هو الحال في الأنظمة الأحادية. تسمح بنية Microservices بعزل الأخطاء (Fault Isolation)، مما يُحسّن استمرارية الخدمة (Availability)، ويُقلّل من مخاطر الأعطال الكارثية [5].

## 5. تسهيل التوسع التنظيمي (Organizational Scalability)

تُساعد هذه البنية في توزيع العمل بين الفرق البرمجية (Team Autonomy)، حيث يتولى كل فريق مسؤولية كاملة عن دورة حياة خدمة محددة، مما يُمكنه من اتخاذ قرارات تقنية مستقلة، ويُسرّع عمليات التطوير، ويُعزّز الكفاءة الداخلية [4].

## 6. دعم قابلية التحديث المستمر (Continuous Delivery and Deployment)

بفضل استقلالية الخدمات، يُمكن تطبيق التسليم المستمر (Continuous Delivery) والنشر التلقائي (Continuous Deployment) بسهولة، مما يُساعد على إدخال التعديلات بشكل دوري وسريع دون التأثير على بقية مكونات النظام. وهذا يتماشى مع منهجيات التطوير الحديثة مثل Agile و DevOps [6].

## 7. تحسين أمان النظام (Security Segmentation)

إذ يُمكن عزل كل خدمة خلف بوابات تحكم مختلفة، وتحديد صلاحيات الوصول بدقة، واستخدام آليات مصادقة مختلفة لكل مكون. يُساهم ذلك في تقليل نقاط الاختراق المحتملة، ويُتيح بناء أنظمة أكثر أماناً.

## 4.3.2 التحديات المرتبطة بالخدمات المصغرة (Challenges of Microservices Architecture)

رغم ما تُقدّمه بنية الخدمات المصغرة (Microservices Architecture) من مزايا في التوسّع، والاستقلالية، والمرونة، إلا أنها ليست خالية من التحديات. بل إن تطبيق هذا النمط المعماري يتطلب وعياً عميقاً بالتصميم، والبنية التحتية، وآليات التشغيل، خصوصاً عند الانتقال من نظام أحادي (Monolith) إلى نظام موزّع.

فيما يلي أبرز التحديات التي تواجه فرق التطوير عند اعتماد هذا النموذج:

### 1. التعقيد التوزيعي (Distributed System Complexity)

أهم ما يُميّز بنية Microservices هو أنها نظام موزّع بطبيعته (Distributed System)، مما يعني أن التواصل بين الخدمات لم يعد يتم داخل الذاكرة (In-Process)، بل عبر الشبكات، وهذا يُضيف مشاكل جديدة تتعلق بتأخير الاستجابة (Latency)، وفقدان الرسائل، والفشل الجزئي، وصعوبة التزامن.

### 2. إدارة البيانات الموزعة (Data Consistency and Partitioning)

كل خدمة تحتفظ بقاعدة بيانات خاصة بها، مما يُعزّز استقلاليتها، لكنه يُعقّد عمليات الاستعلام المشترك (Cross-Service Queries)، ويُصعّب ضمان الاتساق (Consistency) عند إجراء تحديثات مترابطة. في كثير من الحالات، يكون الاتساق النهائي (Eventual Consistency) هو الحل، ما يتطلب إعادة تصميم منطق الأعمال.

### 3. إدارة التكوين (Configuration and Service Discovery)

في نظام يحتوي على عشرات أو مئات الخدمات، تزايد الحاجة إلى آليات فعالة لإدارة التكوين (Configuration Management)، واكتشاف المواقع الديناميكية للخدمات (عناوين شبكية متغيرة) (Service Discovery). بدون ذلك، قد تُصبح الصيانة معقّدة جداً، وتفشل الخدمات في العثور على بعضها البعض.

### 4. إدارة الفشل (Fault Tolerance and Resilience)

عند توزيع النظام على عدد كبير من الخدمات، يُصبح الفشل أمراً شائعاً يجب توقعه والتعامل معه. يجب تصميم الخدمات بحيث تكون قادرة على التعافي (Resilient)، باستخدام تقنيات مثل:

▪ إعادة المحاولة (Retry)

▪ قاطع الدارة (Circuit Breaker)

▪ الرد الاحتياطي (Fallback Response)

### 5. المراقبة وتتبع الطلبات (Monitoring & Observability)

يُعد تتبع مسار طلبٍ واحدٍ يمر عبر عدة خدمات تحدياً صعباً. تحتاج الفرق إلى أدوات متقدمة لرصد الأداء، وتحليل السجلات، وتتبع العمليات (Distributed Tracing)

## 6. صعوبة الاختبار (Testing Microservices)

بسبب الاعتمادية المتبادلة بين الخدمات، يصبح اختبار النظام ككل أكثر تعقيداً من اختبار تطبيق أحادي. يجب التفكير في أنواع متعددة من الاختبارات:

- اختبار كل خدمة بشكل منفصل (Unit & Integration Tests)
- اختبار التفاعل بين الخدمات (Contract Testing)
- اختبارات المنظومة ككل (End-to-End Testing)

رغم كل الإمكانيات التي توفرها الخدمات المصغرة، فإن تبنيها لا يُمثّل تبسيطاً للبنية، بل استبدالاً لتعقيدٍ داخلي بتعقيدٍ موزّع. ولذا، فإن قرار استخدامها يجب أن يكون مبنياً على دراسة دقيقة للمتطلبات التقنية والتنظيمية، وعلى وعي تام بالأدوات والممارسات التي تُساهم في التخفيف من هذه التحديات.

## 5.3.2 الأنماط المعمارية الأساسية في تصميم أنظمة الخدمات المصغرة Microservices

عند اعتماد نمط الخدمات المصغرة في تصميم الأنظمة البرمجية، تظهر مجموعة من التحديات التي تتطلب حلولاً معمارية خاصة، منها ما يتعلق بإدارة الاتصال بين الخدمات، ومنها ما يتصل بكيفية توزيع البيانات وضمان الاتساق وتحمل الأعطال. من هذا المنطلق، أصبح من الضروري الاعتماد على مجموعة من الأنماط المعمارية التي أثبتت فعاليتها في بيئات الأنظمة الموزعة.

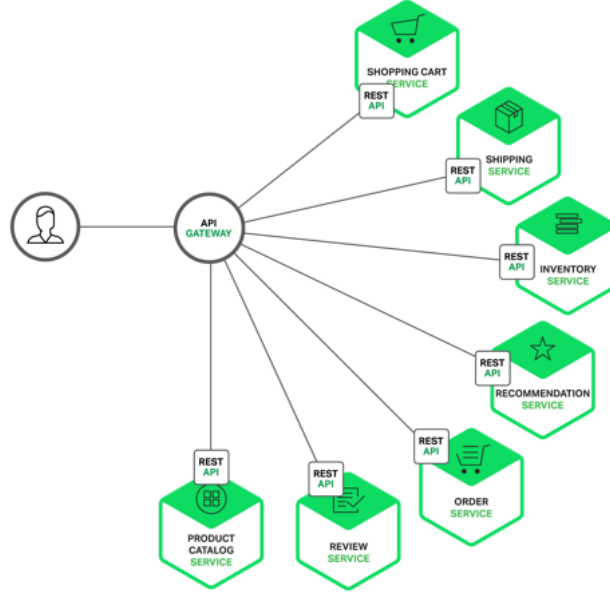
هناك أنماطاً معمارية أساسية تُستخدم بشكل متكرر في تصميم أنظمة الخدمات المصغرة، وتُساهم في تعزيز خصائص مثل قابلية للتوسع، وفصل الاهتمامات، والمرونة في التحديث والنشر. تقسم هذه الأنماط إلى:

## 1.5.3.2 نمط بوابة واجهة التطبيقات API Gateway Pattern

يُستخدم نمط بوابة واجهة التطبيقات (API Gateway) لتوفير نقطة دخول موحدة لجميع الطلبات الموجهة إلى النظام، بحيث تعمل البوابة كوسيط بين العميل ومجموعة الخدمات المصغرة. تتولى هذه البوابة مسؤوليات متعددة تشمل توجيه الطلبات إلى الخدمة المناسبة، ومعالجة المصادقة، وتطبيق سياسات الأمان، بالإضافة إلى تجميع الردود من عدة خدمات في استجابة واحدة [7].

يسهم هذا النمط في تقليل تعقيد الاتصال بين العميل والخدمات الداخلية، كما يُسهّل مراقبة الطلبات وتطبيق خصائص مشتركة مثل تسجيل الدخول والتفويض على مستوى مركزي. يُعدّ حلاً فعالاً لتجنب اتصال مباشر مع عدد كبير من الخدمات.

حيث يُستخدم هذا النمط في الأنظمة التي تحتوي على عدد كبير من الخدمات المصغرة، ويُعدّ ضرورياً عندما يكون لدى كل خدمة واجهة مستقلة وتحتاج إلى تنسيق مركزي للطلبات، أو عندما تتطلب بعض الخدمات الوصول من أجهزة مختلفة (مثل الويب أو الموبايل).



الشكل 15: بنية نمط بوابة واجهة التطبيقات (API Gateway Pattern)

### 2.5.3.2 نمط قاطع الدارة Circuit Breaker Pattern

يُستخدم نمط قاطع الدارة (Circuit Breaker) لحماية النظام من الفشل المتسلسل الناتج عن الاعتماد على خدمات خارجية قد تكون غير متاحة أو متأخرة في الاستجابة. يُشبه هذا النمط في مبدئه عمل القاطع الكهربائي، حيث يُراقب الاتصال مع خدمة معينة، وإذا تم رصد عدد معين من حالات الفشل المتتالية، يتم "فتح الدارة" ومنع المزيد من المحاولات مؤقتاً، مما يتيح للنظام الاستمرار في العمل دون انتظار استجابات متأخرة أو فاشلة.

يساعد هذا النمط في تحسين استقرار النظام من خلال عزل الأجزاء المتعطلة مؤقتاً، كما يُقلّل من استهلاك الموارد غير الضروري في حال استمرار الفشل، ويُتيح آلية للمحاولة لاحقاً (retry) بطريقة محسوبة.

حيث يُستخدم عند وجود اتصال مع خدمات غير موثوقة بشكل دائم، أو في بيئات تعتمد على طرف ثالث يمكن أن يتعرض لانقطاعات مؤقتة. كما يُعتبر ضرورياً في الأنظمة عالية التوفر التي يجب أن تتجنب الانهيار الكامل بسبب تبعية واحدة [8].

يعتمد نمط قاطع الدارة على آلية محددة لضبط سلوك النظام عند التفاعل مع الخدمات المتعثرة، ويقوم بذلك من خلال ثلاث حالات أساسية يمرّ بها أثناء التنفيذ:

### 1. الحالة المغلقة (Closed)

في هذه الحالة، يكون الاتصال بين الخدمة والعميل طبيعياً، وتُمرّر جميع الطلبات دون حظر. تُعدّ الحالة المغلقة هي الحالة الافتراضية للنظام، وتُستخدم طالما لم يتم رصد أية أعطال أو تأخيرات ملحوظة في الخدمة المستهدفة.

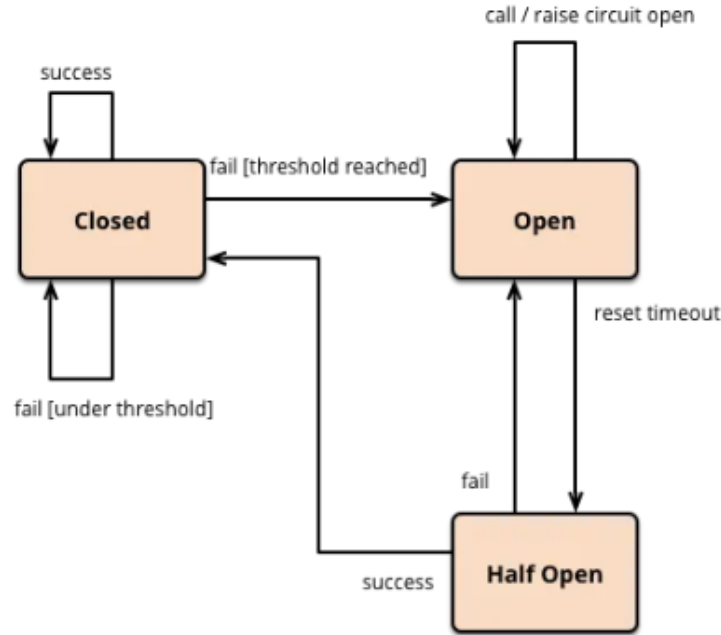
### 2. الحالة المفتوحة (Open)

عندما يتجاوز عدد حالات الفشل المتتالية حداً معيناً يتم تحديده مسبقاً (threshold)، يتم "فتح" قاطع الدارة، ما يعني أن جميع الطلبات اللاحقة تُمنع من الوصول إلى الخدمة المتسببة بالمشكلة، وتُعاد على الفور برسائل خطأ دون تنفيذ فعلي. تُستخدم هذه الحالة لحماية الموارد ومنع الضغط الزائد على الخدمة المتعثّرة.

### 3. الحالة نصف المفتوحة (Half-Open)

بعد مرور فترة زمنية محددة (cool-down period)، يدخل النظام في حالة "نصف مفتوحة" لاختبار ما إذا كانت الخدمة قد تعافت. في هذه المرحلة، يُسمح لعدد محدود من الطلبات بالمرور، ويتم تقييم النتائج. إذا استجابت الخدمة بشكل طبيعي، يُعاد القاطع إلى الحالة المغلقة. أما إذا استمر الفشل، فيُعاد فتح القاطع.

تُعتبر هذه الحالات الثلاث جزءاً أساسياً من بنية هذا النمط، حيث تتيح تحكماً ديناميكياً في سلوك النظام أثناء التعامل مع الأعطال المؤقتة أو التدهور في الخدمات الخارجية، مع الحفاظ على استقراره العام.



الشكل 16: مخطط حالة نمط *Circuit Breaker*

### 3.5.3.2 قاعدة بيانات لكل خدمة Database per Service Pattern

يعتمد هذا النمط على مبدأ أن كل خدمة مصغرة يجب أن تكون مستقلة تماماً عن غيرها، ليس فقط على مستوى التنفيذ، بل كذلك على مستوى قاعدة البيانات. في Database per Service Pattern، يتم تخصيص قاعدة بيانات منفصلة لكل خدمة ضمن النظام، بحيث تكون كل خدمة مسؤولة عن إدارة بياناتها الخاصة، دون مشاركة أو وصول مباشر من خدمات أخرى [9].

يساعد هذا النمط في تقليل الترابط بين الخدمات (Loose Coupling)، ويمنح كل خدمة حرية اختيار نوع قاعدة البيانات الأنسب لحاجاتها (مثل SQL أو NoSQL). كما يُسهّل من عمليات الصيانة والنشر، ويُعزز من المرونة عند توسيع نطاق النظام أو ترقية إحدى الخدمات دون التأثير على البقية [10].

باعتبار هذا النمط مثالاً في أنظمة Microservices التي تتطلب استقلالية كاملة لكل مكون، أو عندما تختلف طبيعة البيانات بين الخدمات. كما يُستخدم عند الرغبة في تطبيق نمط CQRS أو أنماط Event Sourcing، إذ تحتاج كل خدمة إلى التحكم التام في دورة حياة بياناتها [9].

من أبرز التحديات التي قد تظهر عند استخدام هذا النمط هو الحفاظ على الاتساق عبر الخدمات المختلفة، خصوصاً عندما تتطلب بعض العمليات بيانات مترابطة من أكثر من خدمة (عند إجراء مناقلة transaction). في هذه الحالة، يمكن اللجوء إلى أنماط إضافية مثل Saga Pattern أو Event-Driven Architecture لتنسيق التحديثات وضمان الاتساق النهائي.



الشكل 17: فصل قواعد البيانات بين الخدمات في بنية Microservices

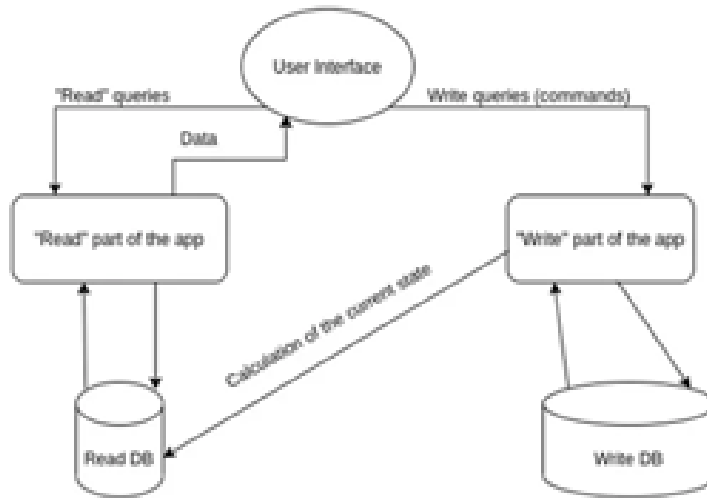
### 4.5.3.2 نمط فصل مسؤوليات القراءة والكتابة CQRS (Command Query Responsibility Segregation) Pattern

يُعدّ نمط فصل مسؤوليات القراءة والكتابة (Command Query Responsibility Segregation) من الأنماط الشائعة في تصميم الأنظمة الموزعة التي تتطلب أداءً عالياً وتحكماً دقيقاً في العمليات. يُبنى هذا النمط على مبدأ فصل الأوامر (Commands) التي تُحدث تغييرات على حالة النظام عن الاستعلامات (Queries) التي تُستخدم فقط لجلب البيانات [11].

يساعد هذا النمط في تحسين أداء النظام، من خلال استخدام نموذج بيانات مخصص لكل نوع من العمليات. على سبيل المثال، يمكن تصميم قاعدة بيانات للقراءة تكون مُحسّنة للاستعلامات المعقدة أو تقنيات التخزين المؤقت (Caching)، بينما تستخدم قاعدة بيانات أخرى للكتابة تكون مهيأة لحالات التعديل. كما يُسهّل هذا النمط تنفيذ أنماط إضافية مثل Event Sourcing و Materialized View لدعم تتبع التغييرات وتحقيق استجابة سريعة للقراءات.

حيث يُستخدم نمط CQRS في الأنظمة التي تحتوي على حجم كبير من الاستعلامات مقارنة بعمليات التعديل، أو عندما تكون الاستعلامات معقدة وتتطلب بيانات مجمعة من مصادر متعددة. كما يُفضّل استخدامه في التطبيقات التي تحتاج إلى مرونة في التوسع الأفقي لخدمات القراءة والكتابة كلٌّ على حدة.

يتطلب تطبيق هذا النمط مجهوداً إضافياً في التصميم والتنفيذ، خاصة في تنسيق الحالة بين واجهات القراءة والكتابة، بالإضافة إلى تحديات الاتساق النهائي في البيئات غير المتزامنة. ولهذا السبب، غالباً ما يُستخدم بالتوازي مع أنماط داعمة مثل Event-Driven Architecture أو Message Queueing لضمان نقل التعديلات بشكل فعال إلى واجهات القراءة.



الشكل 18: فصل طبقتي القراءة والكتابة في نمط CQRS

### 5.5.3.2 نمط التخزين المسبق للبيانات المحسوبة Materialized View Pattern

يُستخدم نمط العرض المادي (Materialized View) لتسريع عمليات القراءة والاستعلام، من خلال إنشاء نسخة مخزنة مسبقاً من نتيجة استعلام معقد، بدلاً من إعادة حسابه عند كل طلب. يُطبق هذا النمط عادةً في البيئات التي تكون فيها عمليات القراءة متكررة، وتُشكل عبئاً كبيراً على قواعد البيانات الأصلية، خاصة في أنظمة الخدمات المصغرة التي تعتمد على CQRS.

يساعد هذا النمط في تحسين الأداء وتقليل زمن الاستجابة، من خلال تقديم البيانات للمستخدم مباشرة من مخزن جاهز بدلاً من المرور بسلسلة استعلامات معقدة. كما يُتيح مرونة في تمثيل البيانات بطرق مختلفة لتلبية احتياجات الواجهات أو التقارير أو واجهات القراءة العامة [12].

حيث يُستخدم هذا النمط عندما تكون طبيعة البيانات معقدة أو مترابطة، وتُستخدم بكثرة في واجهات المستخدم، خاصة في أنظمة مثل أنظمة إدارة الطلبات أو منصات عرض المحتوى. كما يُستخدم في حالات يكون فيها من المناسب التوضيحية بـ "الاتساق الفوري" مقابل سرعة القراءة، اعتماداً على تحديث دوري أو بالاعتماد على نمط Event-Driven.

من أبرز التحديات المرتبطة بهذا النمط هو الحفاظ على تزامن العرض المادي مع البيانات المصدر، خاصةً عند حدوث تغييرات متكررة. لذلك، يُصح بدمج هذا النمط مع بنية تعتمد على الأحداث (Event-Driven) لضمان تحديث العرض عند حدوث تغييرات في البيانات.

### 6.5.3.2 النمط القائم على الأحداث Event-Driven Architecture Pattern

يعتمد هذا النمط على مبدأ التفاعل غير المتزامن بين الخدمات، حيث تتواصل مكونات النظام من خلال إطلاق واستقبال الأحداث (Events) بدلاً من الاستدعاءات المباشرة. في هذا النمط، تقوم إحدى الخدمات بنشر حدث معين عند حدوث تغيير في حالتها، ثم يتم التقاط هذا الحدث من قبل خدمات أخرى تحتاج إلى التفاعل معه، دون وجود ارتباط مباشر بينها [13].

يسهم هذا النمط في تقليل الترابط بين الخدمات (Loose Coupling)، مما يُعزز من قابلية التوسع وسهولة التحديث. كما يتيح مرونة عالية في التعامل مع المهام المتوازية و المهام غير المتزامنة (Asynchronous Tasks)، ويدعم نماذج معقدة مثل Event Sourcing و CQRS من خلال تمكين تدفق البيانات بطريقة قابلة للتعبق والقياس.

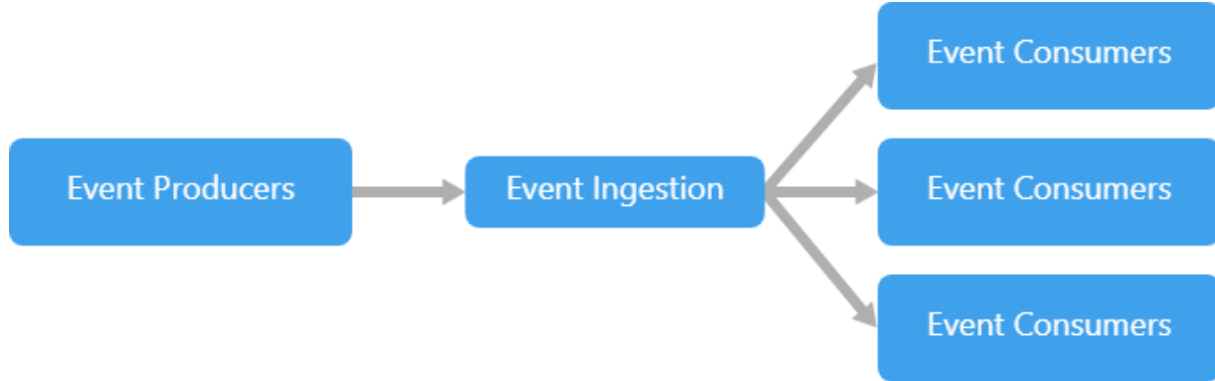
يُستخدم نمط EDA في الأنظمة التي تحتاج إلى استجابة فورية للتغيرات، مثل أنظمة الإشعارات، والتسجيل، وسلاسل التوريد، أو في حالات تتطلب التفاعل بين عدد كبير من الخدمات دون إنشاء روابط صلبة بينها. كما يُعدّ مناسباً عندما يكون من الضروري تسجيل وتحليل كل الأحداث التي تمر بالنظام.

يتكوّن هذا النمط غالباً من ثلاث مكونات رئيسية:

▪ المنتج: (Producer) يرسل الحدث.



- الوسيط (Event Bus/Broker) ينقل الحدث مثل Kafka أو RabbitMQ
- المستهلك (Consumer) يتلقى الحدث ويتخذ إجراء معيناً.



الشكل 19: البنية العامة للنمط القائم على الأحداث

### 7.5.3.2 نمط تخزين الأحداث Event Sourcing Pattern

يعتمد نمط تخزين الأحداث (Event Sourcing) على فكرة الاحتفاظ بسجل كامل للأحداث التي غيّرت حالة الكيان (Entity)، بدلاً من تخزين الحالة النهائية فقط في قاعدة البيانات. يُمثل كل حدث عملية تغيير محددة في النظام، مثل "تم إنشاء المستخدم" أو "تم تعديل العنوان"، ويتم تسجيل هذه الأحداث بشكل تسلسلي في سجل زمني يُعرف باسم سجل الأحداث (Event Log) [14].

يساعد هذا النمط في تحقيق عدة أهداف مهمة، منها إمكانية إعادة بناء الحالة الحالية لأي كيان عن طريق إعادة تطبيق تسلسل الأحداث، وتحقيق قابلية التدقيق (Auditability)، حيث يمكن معرفة كل ما حصل على الكيان منذ لحظة إنشائه. كما يُسهّل التكامل مع الأنظمة الأخرى من خلال نشر الأحداث تلقائياً بمجرد وقوعها، مما ينسجم بشكل مثالي مع البنية القائمة على الأحداث (Event-Driven Architecture) [14].

حيث يُستخدم هذا النمط في الأنظمة التي تتطلب سجلاً تاريخياً دقيقاً لكل تغيير، أو في الحالات التي تكون فيها الحاجة إلى استعادة الحالة بعد الفشل أو التراجع ضرورية. كما يُعتمد عليه غالباً بالتزامن مع نمط CQRS، حيث يتم تنفيذ الأوامر (Commands) بتسجيل الأحداث، في حين تُستخدم واجهات عرض (Read Models) منفصلة يتم تحديثها تلقائياً عند حدوث تلك الأحداث.

### 8.5.3.2 نمط الانتشار والتجميع Fan-out / Fan-in Pattern

يُعد نمط Fan-out / Fan-in من الأنماط الشائعة في تصميم الأنظمة الموزعة التي تتطلب تنفيذ عمليات متعددة بالتوازي، ثم تجميع نتائجها في مرحلة لاحقة. يُشير مصطلح Fan-out إلى توزيع مهمة واحدة إلى عدة مهام فرعية تُنفَّذ بالتوازي عبر عدة خدمات أو عقد، بينما يُشير Fan-in إلى مرحلة تجميع تلك النتائج بمجرد اكتمال المعالجة [15].

يُتيح هذا النمط تحسين الأداء العام للنظام من خلال المعالجة المتوازية، ويُساهم في تقسيم الحمل وتحقيق التوسع الأفقي عند التعامل مع عدد كبير من المهام المستقلة. كما يُساعد في تقليل زمن الاستجابة الكلي للعمليات المعقدة التي يمكن تفكيكها إلى مهام أصغر قابلة للتنفيذ المتزامن.

حيث يُستخدم هذا النمط في الحالات التي تتطلب توزيع الطلبات أو المهام على عدد كبير من المستهلكين أو العمليات الفرعية، مثل إرسال الأحداث إلى عدة خدمات، أو توزيع البيانات على عقد متعددة للمعالجة، ثم تجميع النتيجة النهائية. يُعتبر النمط مناسباً عند وجود بنية قائمة على الأحداث أو عند الحاجة إلى تنفيذ مهام معزولة يمكن دمج نتائجها لاحقاً.

## 4.2 مقارنة بين بنية التطبيقات الأحادية والخدمات المصغرة

### (Monolithic vs. Microservices Architecture)

عند تصميم الأنظمة البرمجية، يظهر أمام المهندسين خياران أساسيان في البنية المعمارية: إما استخدام بنية التطبيقات الأحادية (Monolithic)، أو اللجوء إلى بنية الخدمات المصغرة (Microservices). لكل من هذين النمطين مزاياه وتحدياته، ويُناسب كل منهما نوعاً معيناً من المشاريع حسب الحجم، متطلبات التطوير، وفريق العمل.

فيما يلي مقارنة شاملة تغطي أبرز الجوانب الجوهرية:

من حيث	التطبيقات الأحادية (Monolithic)	الخدمات المصغرة (Microservices)
فرق العمل	يعمل فريق واحد على التطبيق بالكامل ضمن دورة حياة موحدة	كل خدمة تُدار من قِبل فريق مستقل مسؤول عن دورة حياتها الكاملة
التطوير والتحديث	يتم تطوير النظام كوحدة واحدة مما يزيد من تعقيد التحديثات	إمكانية تطوير ونشر كل خدمة بشكل مستقل دون التأثير على بقية النظام
التوزيع Deployment	يُوزع التطبيق كوحدة واحدة، وأي تعديل يتطلب إعادة نشر شامل	يمكن نشر الخدمات بشكل مستقل (Independent Deployment)
اختبار وتكوين النظام	اختبار النظام يتم كوحدة واحدة، وتكوين بيئة العمل أبسط نسبياً	كل خدمة تتطلب إعداد واختبار وتكوين منفصل (Configuration)

أسهل في الفهم والإدارة عند المشاريع الصغيرة	تتطلب أدوات لإدارة الترابط، ولكنها مرنة أكثر في الأنظمة الكبيرة	بساطة الفهم والإدارة
مقيّدة بلغة وتقنية واحدة غالباً	يمكن لكل خدمة أن تستخدم التقنية أو لغة البرمجة الأنسب لها	المرونة التقنية
توسيع النظام يتم بشكل كامل، حتى لو كان المطلوب فقط خدمة واحدة	يمكن توسيع كل خدمة على حدة حسب الحاجة	قابلية التوسع (Scalability)
أسرع في البداية للمشاريع الصغيرة	أسرع على المدى الطويل في المشاريع الكبيرة عند توزيع المهام بين الفرق	سرعة التطوير
فشل مكون قد يؤثر على النظام ككل	كل خدمة مستقلة، مما يحدّ من تأثير الفشل	الاعتمادية (Reliability)
أبسط في الإدارة، لا تحتاج لتقنيات حديثة	تحتاج إلى أدوات إدارة متقدمة) مثل Docker, Kubernetes, API Gateway)	بيئة التشغيل والبنية التحتية

## الفصل الثالث

### الدراسة التحليلية

بوضح هذا الفصل تحليل النظام ودراسة المتطلبات

### 1.3 مخطط حالات الاستخدام

نعرض في هذا الشكل حالات الاستخدام المختلفة ضمن النظام



الشكل 20: حالات الاستخدام

## 2.3 مخططات تنالي النظام sequence diagram

## الفصل الرابع

### تصميم النظام

نعرض في هذا الفصل القرارات التصميمية التي تم الاعتماد عليها لبناء النظام وشرح للنظام المقترح

#### 1.4 مقدمة

في مرحلة تصميم النظام تبرز القدرة على تحويل الأهداف النظرية إلى قرارات هندسية واضحة تحدد بنية المكونات وطريقة تفاعلها وتضمن عمل المتطلبات غير الوظيفية. وبما أن منصة مشاركة الصور تستهدف عدداً كبيراً من المستخدمين مع معدلات رفع وتصفح مرتفعة، فقد كان من الضروري اعتماد معمارية موزعة قائمة على خدمات مصغرة (Microservices) مدفوعة بالأحداث، تتوسطها بوابة واجهات (API Gateway) تفصل العميل عن تفاصيل البنية الداخلية. حقق هذا الاتجاه قابلية توسع أفقية وعزلاً للأعطال بالإضافة إلى إمكانية نشر مستمر لا يوفرها النموذج التقليدي أحادي الخدمة.

تم بناء التصميم بالاعتماد على منهج علمي إذ تم إجراء مقارنات دقيقة لاختيار التقنيات والأنماط الأنسب بما يضمن معالجة فعالة للبيانات، وتسريع توليد الخلاصات، وتوفير بحث فوري، مع الاستفادة من مزايا الفصل بين مسارات القراءة والكتابة وتخزين سجل الأحداث لإعادة بناء الحالة عند الحاجة. كما يستند التصميم إلى بيئة حاويات قابلة للتدرج الآلي وتوزيع الأحمال بشكل ديناميكي.

#### 2.4 مكونات النظام

يتكون النظام في صورته العامة من عدة مكونات تعمل معاً لتقديم منصة اجتماعية لمشاركة الصور. في الواجهة توجد طبقة العرض وهي الطبقة التي تتفاعل مع المستخدمين (الواجهة الأمامية). يلي ذلك بوابة واجهات واحدة API Gateway تتولى استقبال الطلبات وتوجيهها إلى باقة الخدمات المصغرة الخلفية، وهي خدمات مستقلة تؤدي وظائف التسجيل والمصادقة وإدارة الصور والخلاصات وغير ذلك. تدعم هذه الخدمات تمرير رسائل تؤمن تبادل الأحداث، وخيارات تخزين متنوعة للبيانات المهيكلة وغير المهيكلة. وتتكون البنية التحتية من بيئة حاويات مُدارة على Kubernetes توفر التوسع والتعافي التلقائي، بينما تغطي طبقة المراقبة أدوات تجميع السجلات والقياسات ولوحات العرض لضمان الرصد المستمر.

#### 3.4 خدمات النظام

يتكون النظام من مجموعة من الخدمات المصغرة، يختص كل منها بمسؤولية محددة لضمان الفصل الواضح للمهام وسهولة التوسع والصيانة:

#### 1. خدمة المستخدم user-service

تتولى هذه الخدمة جميع المهام المتعلقة بإدارة حسابات المستخدمين، بما في ذلك إنشاء الحسابات الجديدة، تعديل البيانات الشخصية، وتوليد روابط تحميل الصور الشخصية.

#### 2. خدمة المصادقة auth-service

تتولى عملية المصادقة وإصدار رموز JWT، وتقديم خدمة تسجيل الدخول login، مع التحقق من صحة كلمة المرور.

#### 3. خدمة البحث search-service

تتيح البحث عن المستخدمين بالاسم أو جزء منه، كما تقدم اقتراحات المتابعة عبر فهرس مخصص قابل للتوسع.

#### 4. خدمة المتابعة follow-service

تدير هذه الخدمة علاقات المتابعة بين المستخدمين، إذ توفر عمليات المتابعة وإلغاء المتابعة، بالإضافة إلى حساب عدد المتابعين والمتابعين.

#### 5. خدمة نشر الصور post-service

تختص هذه الخدمة بإدارة المحتوى الذي ينشره المستخدمون على المنصة، أي تسمح للمستخدمين بمشاركة صورة (نشرها على المنصة) كما تتيح إضافة توصيف caption للصورة قبل نشرها.

#### 6. خدمة الحائط feed-service

تتيح عرض الحائط الشخصي كخلاصة تحتوي على أحدث الصور من الحسابات المتابعة، مرتبة زمنياً بشكل ديناميكي.

## 4.4 التحديات والحلول المقترحة

أثناء عملية تصميم النظام، برزت مجموعة من التحديات المعمارية التي كان لا بد من التعامل معها بدقة لضمان بنية قابلة للتوسع، ومستقرة، وتلبي المتطلبات الوظيفية وغير الوظيفية على حد سواء. هذه التحديات لم تكن محصورة في الجانب التقني أو البرمجي، بل شملت أيضاً قرارات استراتيجية أثّرت على كيفية توزيع المسؤوليات بين الخدمات، إدارة البيانات، وتنظيم تدفق الأحداث. في ما يلي أبرز هذه التحديات والحلول المعمارية المعتمدة لمعالجتها.

#### 1.4.4 تحدي الفصل بين المسؤوليات مع الحفاظ على التكامل بين الخدمات

في الأنظمة التقليدية المبنية بأسلوب أحادي (Monolithic)، يكون من السهل ربط المكونات داخلياً، لكن ذلك يؤدي إلى تداخل في المسؤوليات وصعوبة في التطوير والصيانة. في المقابل، اعتمد هذا المشروع على معمارية الخدمات المصغرة (Microservices)، ما استدعى إعادة توزيع المسؤوليات بعناية. كان التحدي الأساسي هو الحفاظ على فصل واضح بين الخدمات، دون الإخلال بالتكامل فيما بينها.

ولمعالجة ذلك، تم اعتماد مبدأ فصل المسؤوليات Separation of Concerns بشكل صارم، بحيث تختص كل خدمة بوظيفة واحدة فقط (مثل المصادقة، إدارة الحساب، النشر، المتابعة، إلخ)، مع استخدام API Gateway لتنسيق الوصول من الواجهة الأمامية، وتطبيق نمط Event-Driven Architecture باستخدام Kafka للتواصل غير المتزامن بين الخدمات. هذا النمط مكن كل خدمة من التطور بشكل مستقل دون الاعتماد على استدعاءات مباشرة، مما حقق درجة من العزل modularity وقابلية التوسع.

#### 2.4.4 تحدي تصميم نظام خلاصات فعال يدعم المستخدمين ذوي المتابعين الكثير

من أبرز التحديات في المنصات الاجتماعية هو التعامل مع المستخدمين الذين يمتلكون آلاف أو ملايين المتابعين، خاصة عند نشر محتوى يجب توزيعه على نطاق واسع. كان من غير الممكن اعتماد منطق موحد لتوزيع المنشورات على جميع أنواع المستخدمين.

لذلك، تم اتخاذ قرار استراتيجي بتصنيف المستخدمين إلى فئتين: عاديين ومؤثرين، مع تطبيق منطقين مختلفين:

- Fan-out-on-write للمستخدمين العاديين: حيث يتم إرسال المنشور مباشرة إلى جميع متابعيه عند النشر.
- Fan-in-on-read للمؤثرين: حيث لا يتم توزيع المنشور عند النشر، وإنما يتم دمج لاحقاً مع الخلاصة عند القراءة.

هذا التصميم الهجين ساهم في تخفيض الضغط على قاعدة البيانات، وتحقيق أداء ثابت حتى مع نمو عدد المتابعين، دون التأثير على تجربة المستخدم.

#### 3.4.4 تحدي اختيار آلية تخزين مناسبة للصور دون التأثير على أداء النظام

أحد التحديات الجوهرية تمثل في كيفية التعامل مع الصور المرفقة (في الحسابات والمنشورات)، خاصة أن تخزين الوسائط مباشرة داخل قواعد البيانات يؤدي إلى تضخم الحجم وانخفاض الأداء.

تم تجاوز هذا التحدي من خلال استخدام نظام تخزين كائني Object Storage، بحيث تُرفع الصور مباشرة من الواجهة الأمامية باستخدام روابط موقعة مسبقاً Presigned URLs، ويتم فقط تخزين الرابط داخل قواعد البيانات. هذا القرار ساهم في:

- الحفاظ على خفة قواعد البيانات.

- تأمين الصور عبر روابط مؤقتة.
- تمكين فصل الوسائط عن المنطق البرمجي.

#### 4.4.4 تحدي تنظيم المصادقة بطريقة مركزية وآمنة في بيئة موزعة

في نظام موزع يتضمن عدة خدمات، كان من الضروري وضع استراتيجية موحدة لإدارة المصادقة وتفويض الوصول. الاعتماد على المصادقة المنفردة داخل كل خدمة كان سيؤدي إلى تكرار الكود وصعوبة الإدارة. ولذلك، تم اعتماد JWT كنظام مصادقة موحد. هذا التصميم حافظ على مركزية المصادقة، وعزز الأمان من خلال تقليل فرص التلاعب بالهوية أو تجاوز الصلاحيات.

#### 5.4.4 تحدي دعم البحث الفوري دون تحميل إضافي على بقية الخدمات

مع تطور المنصة، برزت الحاجة إلى توفير إمكانية البحث السريع عن المستخدمين بالاسم، بالإضافة إلى تقديم اقتراحات متابعة. ولأن هذا النوع من البحث يتطلب معالجة مرنة وفعالة للنصوص، لم يكن من المناسب الاعتماد على قاعدة بيانات تقليدية سواء علائقية أو غير علائقية لأنها لا توفر دعماً كافياً للبحث النصي.

بناءً على ذلك، تم تصميم خدمة بحث مستقلة تستخدم قاعدة بيانات متخصصة بعمليات الفهرسة والاستعلام النصي الحر، وتحدث بشكل لحظي عند إنشاء أو تعديل بيانات المستخدمين، من خلال نظام تبادل الرسائل. هذا القرار مكن النظام من تنفيذ عمليات البحث بسرعة ودقة دون التأثير على أداء الخدمات الأساسية الخاصة بإدارة الحسابات، ووفر بنية مرنة تسهل التوسع في ميزات البحث لاحقاً.

#### 6.4.4 تحدي تقليل حجم الصور لتحسين الأداء وتجربة المستخدم

من أبرز التحديات التي واجهتنا أثناء تصميم النظام، هو التعامل مع الصور التي يرفعها المستخدمون سواء في ملفاتهم الشخصية أو ضمن منشوراتهم. الصور في الغالب تأتي بحجم كبير نسبياً، خاصة عند التقاطها بكاميرات الهواتف الحديثة، ما يؤدي إلى زمن طويل في عملية الرفع، وزيادة استهلاك بيانات المستخدم، بالإضافة إلى تحميل زائد على خوادم التخزين. وكان من الضروري البحث عن حل يحافظ على جودة الصورة من جهة، ويقلل حجمها واستهلاك الموارد من جهة أخرى، دون تعقيد إضافي على الخدمات الخلفية.

انطلاقاً من ذلك، تم اعتماد آلية لضغط الصور على جهة العميل (Client-Side) قبل رفعها إلى الخادم، وذلك بهدف تقليل حجم الصورة وتسريع عملية الرفع، دون التأثير الملحوظ على الجودة. هذا الأسلوب ينقذ داخل الواجهة الأمامية باستخدام أدوات مدمجة في المتصفح، ويساهم في تحسين الأداء الكلي للنظام وتخفيف الضغط على خادم التخزين.



اختيار ضغط الصور على الواجهة الأمامية يعكس التوجه نحو التحسين على الحافة (Edge Optimization)، ويؤكد على أهمية توزيع المسؤوليات بين العميل والخادم لتقليل الزمن الإجمالي للاستجابة وتحسين تجربة المستخدم، مع الالتزام بمبدأ التصميم البسيط والمركز لكل خدمة من خدمات النظام.

هذا النهج يقدم عدة فوائد تصميمية مهمة:

- تقليل حجم الصور المرسلة بنسبة كبيرة (تصل إلى 70% في بعض الحالات)، مما يخفف استهلاك الشبكة.
- تسريع عملية الرفع بشكل واضح، خاصة على الأجهزة ذات الاتصال الضعيف.
- إزالة الحاجة إلى ضغط الصور في الخادم أو تخزينها بنسخ متعددة.
- الحفاظ على استقلالية الخدمات المصغرة، إذ بقيت مسؤولة فقط عن توليد الرابط وتخزينه، دون التدخل في محتوى الصورة.

#### 7.4.4 تحدي عرض الصور بكفاءة عالية رغم تزايد عددها وعدد المستخدمين

مع تزايد عدد الصور داخل النظام وارتفاع عدد المستخدمين، ظهرت مشكلة جديدة تتعلق بكفاءة عرض الصور في الواجهة الأمامية. إذ أن تحميل كل صورة من الخادم مباشرة، خاصة الصور الشائعة (مثل صور المؤثرين أو الصور المكررة عبر العديد من الزيارات)، يؤدي إلى ضغط كبير على خادم التخزين وزمن تحميل أطول للمستخدم النهائي. هذه المشكلة لم تكن تقنية فقط بل تصميمية، إذ كان لا بد من فصل قناة عرض الصور عن خادم التخزين وتحسين الأداء دون تغيير في منطق رفع الصور أو صلاحيات الوصول.

لذلك، تم اعتماد استخدام شبكة توصيل المحتوى CDN Content Delivery Network داخلية أمام خادم تخزين الصور (نظام تخزين كائني Object Storage). بهذا الشكل، فإن CDN:

- يُقدّم الصور من الذاكرة مباشرة إذا كانت مخزنة في الكاش.
- يقوم بجلب الصورة من الخادم وتخزينها مؤقتاً إذا لم تكن موجودة.

#### 5.4 الأنماط المعمارية والتصميمية المستخدمة

اعتمد تصميم المنصة على مجموعة من الأنماط المعمارية والتقنية التي أثبتت فعاليتها في بناء أنظمة موزعة، وعالية الأداء، وقابلة للتوسع. فيما يلي استعراض لأهم هذه الأنماط كما طُبقت داخل المشروع:

#### 1.5.4 نمط الخدمات المصغرة (Microservices Architecture)

تم بناء النظام وفق بنية الخدمات المصغرة، حيث تم فصل الوظائف الأساسية إلى خدمات مستقلة user-service، auth-service، post-service، follow-service، feed-service، search-service، وغيرها.

كل خدمة مسؤولة عن نطاق وظيفي محدد وئدار وتُنشر بشكل مستقل، مما أتاح سهولة التطوير، واختبار كل واحدة بمعزل عن الأخرى، مع دعم التوسع الأفقي والاستجابة للأعطال بشكل مرن.

## 2.5.4 نمط فصل مسؤوليات القراءة والكتابة (CQRS)

تم اعتماد نمط CQRS في هذا المشروع لتحقيق فصل واضح بين عمليات الكتابة (مثل إنشاء أو تعديل البيانات) وعمليات القراءة (مثل البحث والاسترجاع).

وقد تم ذلك من خلال توزيع المهام بين خدمتين مستقلتين:

- **user-service** تتولى مسؤولية إدارة البيانات الأساسية للمستخدمين، بما في ذلك استقبال الطلبات التي تتضمن عمليات إنشاء أو تعديل، والتخزين في قاعدة بيانات.
- **search-service** خُصصت بالكامل لعمليات القراءة، وتحديدًا البحث عن المستخدمين، حيث تستقبل الأحداث من وسيط الرسائل، وتقوم بفهرسة البيانات داخل قاعدة البيانات المخصصة.

كما تم اعتماد هذا النمط أيضاً عند فصل خدمتي المنشور وعرض الخلاصة

## 3.5.4 النمط القائم على الأحداث (Event-Driven Architecture)

بدلاً من الاعتماد على استدعاءات مباشرة بين الخدمات، تم استخدام وسيط رسائل كوسيط لنقل الرسائل بين الخدمات المصغرة. عند تسجيل مستخدم جديد مثلاً، ترسل **user-service** حدثاً إلى وسيط الرسائل، وتقوم **search-service** باستهلاكه لفهرسة المستخدم، وأيضاً يتم بنفس الطريقة بين **post-service** و **feed-service** لتحديث الخلاصات المستقبلية. هذا النمط حقق فصلاً زمنياً ومنطقياً بين الخدمات، وساهم في تقليل الترابط بينها (**Loose Coupling**)، مع تحسين قابلية التوسع والأداء.

## 4.5.4 نمط بوابة الدخول الموحدة (API Gateway)

تم تصميم النظام وفق نمط **API Gateway** بحيث تمر جميع الطلبات من الواجهة الأمامية عبر بوابة مركزية، تُعد بمثابة نقطة الدخول الوحيدة (**Single Entry Point**) للتطبيق. يتولى الـ **Gateway** مسؤولية توجيه الطلبات (**Routing**) إلى الخدمة المصغرة المناسبة اعتماداً على مسار الطلب، مما يُخفي تفاصيل البنية الداخلية عن المستخدم النهائي ويمنع التواصل المباشر مع الخدمات. هذا النمط مكّن من التحكم الكامل بجميع نقاط الدخول إلى النظام، وسهّل تطبيق سياسات الأمان، وتسجيل الطلبات، وتخفيف العبء على الخدمات المصغرة، التي لم تعد بحاجة إلى معالجة المصادقة محلياً.

## 5.5.4 نمط قاطع الدائرة (Circuit Breaker)

ضمن طبقة API Gateway، تم تطبيق نمط قاطع الدائرة بشكل خاص عند التعامل مع الصور، حيث تمر طلبات الصور أولاً عبر خادم CDN داخلي في حال تعطل الـ CDN لأي سبب، يقوم الـ Gateway تلقائياً بتجاوزه والتوجه إلى خادم التخزين مباشرة. بهذا الشكل، يضمن النظام استمرارية الخدمة حتى في حال تعطل أحد مكونات البنية الوسيطة، مما يرفع من مستوى المرونة والاعتمادية.

## 6.5.4 نمط التحسين عند الحافة (Edge Optimization)

بهدف تقليل استهلاك الشبكة وتحسين تجربة المستخدم، تم اعتماد ضغط الصور على الواجهة الأمامية قبل رفعها، مما ساهم في تقليل حجم الملفات المنقولة، وتسريع عملية الرفع، خاصة على الشبكات البطيئة.

هذا النمط يعكس مفهوم التحسين على الأطراف، حيث يتم تنفيذ العمليات الثقيلة نسبياً بالقرب من المستخدم، لتقليل العبء عن الخوادم المركزية وتحقيق استجابة أسرع.

## 7.5.4 نمط التوجيه الذكي والتخزين المؤقت عبر CDN (Smart Caching via CDN)

لتسريع تحميل الصور وتحسين أداء عرض الوسائط، تم وضع خادم يعمل كـ CDN داخلي أمام خادم تخزين الصور. عند طلب صورة، يقوم CDN بإرجاعها من الذاكرة المؤقتة إذا كانت مخزنة، أو يجلبها من خادم تخزين الصور وتخزينها للاستخدام المستقبلي.

هذا التوجيه الذكي ساهم في تقليل عدد الطلبات إلى خادم التخزين، وتسريع عملية التحميل، وهيئة البنية التحتية لدعم توسع أفقي.

## 8.5.4 نمط العرض المادي المحسّن (Materialized View Pattern)

بهدف تحسين أداء استرجاع الخلاصات (Feeds) ضمن المنصة، تم اعتماد نمط العرض المادي المحسّن داخل خدمة feed-service، حيث يتم الاحتفاظ بنسخة جاهزة ومُحدّثة من الخلاصة الخاصة بكل مستخدم داخل قاعدة بيانات مؤقتة عند إنشاء منشور جديد.

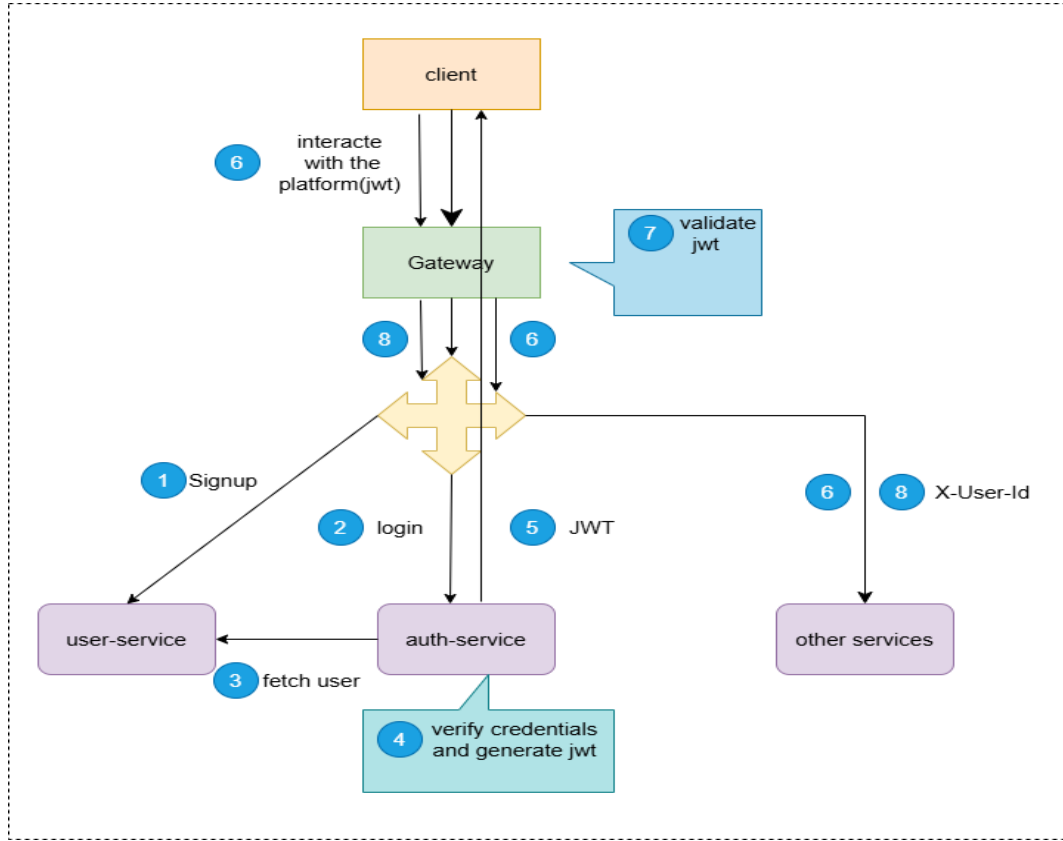
## 6.4 استراتيجية تصميم المصادقة (Authentication Strategy)

في الأنظمة الموزعة المبنية على الخدمات المصغرة، تُعد إدارة المصادقة والتفويض من أكثر التحديات حساسية، إذ يجب تأمين جميع نقاط الدخول وضمان أن كل خدمة تتلقى فقط الطلبات الصالحة والمخولة. ولتجنّب تكرار منطق المصادقة داخل كل خدمة، ولتحقيق مركزية القرار الأمني، تم تصميم بنية تعتمد على رموز الوصول (JWT) التي تصدر من خدمة المصادقة (auth-service)، وتُعالج داخل API Gateway، بحيث تتم المصادقة والتفويض قبل وصول الطلب إلى الخدمات الخلفية.

تم عملية المصادقة في النظام الموزع المعتمد على الخدمات المصغرة وفق تسلسل منطقي يضمن الأمان، ويُستط إدارة الهوية، وذلك كما يلي:

1. يبدأ المستخدم العملية بإرسال طلب لإنشاء حساب جديد من خلال واجهة التطبيق، حيث يُوجّه الطلب إلى خدمة المستخدم، والتي تتولّى التحقق من صحة البيانات وتخزين الحساب في قاعدة البيانات.
2. بعد إنشاء الحساب، يقوم المستخدم بتسجيل الدخول عن طريق إرسال بيانات الاعتماد متضمّن البريد الإلكتروني وكلمة المرور إلى خدمة المصادقة.
3. تقوم خدمة المصادقة بالتواصل مع خدمة المستخدم لاسترجاع معلومات الحساب والتحقق من صحتها.
4. في حال كانت المعلومات صحيحة، تقوم خدمة المصادقة بإصدار رمز دخول JWT يُمثّل هوية المستخدم ويُستخدم في الطلبات اللاحقة.
5. يُرسل هذا الرمز إلى واجهة التطبيق، حيث يتم تخزينه محلياً لاستخدامه في كل طلب جديد يُوجّه إلى النظام.
6. عند إرسال أي طلب لاحق، تقوم واجهة التطبيق بإرفاق الرمز مع الطلب لإثبات الهوية.
7. تقوم بوابة الدخول باستقبال الطلب، وتتحقّق من صحة الرمز، ثم تستخرج منه معلومات الهوية المطلوبة.
8. بعد التحقق، تُمرّر الهوية إلى الخدمة الخلفية المعنية، ليتم تنفيذ الطلب وفق الصلاحيات المحددة.
9. لا تقوم الخدمات الخلفية بالتحقق من الهوية أو الصلاحيات بشكل مباشر، بل تعتمد على المعلومات المستخرجة مسبقاً في بوابة الدخول.

يُساهم هذا التصميم في تحقيق مركزية المصادقة، وتوفير حماية إضافية من محاولات الوصول غير المشروع.



الشكل 21: استراتيجية المصادقة

## 7.4 تصميم قواعد البيانات لكل خدمة (Database Design per Service)

تم اعتماد مبدأ قاعدة بيانات مستقلة لكل خدمة ضمن بنية المنصة، بهدف تحقيق العزل الكامل بين الخدمات وضمان استقلالية التطوير والتوسع لكل مكون على حدة. كما تم تبني نهج تعدد محركات التخزين (Polyglot Persistence)، بحيث يتم اختيار نوع قاعدة البيانات الأنسب لطبيعة البيانات وطريقة الوصول إليها في كل خدمة.

ينطلق هذا التوجه من مبدأ أن كل نوع من البيانات يتطلب نمط تخزين خاص يلبي متطلبات الأداء والموثوقية وقابلية التوسع.

### 1.7.4 خدمة المستخدمين (User Service)

تتولى هذه الخدمة جميع المهام المتعلقة بإدارة حسابات المستخدمين، بما في ذلك إنشاء الحسابات الجديدة، تعديل البيانات الشخصية.

نظراً لطبيعة بيانات المستخدم المتغيرة والمتنوعة والتي قد تختلف من مستخدم لآخر من حيث عدد الحقول أو أنواعها. فقد تم اعتماد قاعدة بيانات من نوع NoSQL (غير علائقية)، وتحديدًا من النوع الذي يدعم تخزين الوثائق (Document-Oriented). تم اختيار هذا النوع لأنه يوفر مرونة كبيرة في تمثيل وتحديث البيانات. تساعد هذه البنية على استيعاب تعدد الحقول وتغيرها بمرور الوقت، مثل (الاهتمامات، الموقع الجغرافي، bio، ...). دون الحاجة إلى إعادة تصميم هيكل البيانات كما هو الحال في قواعد البيانات العلائقية. وبما أن المنصة تستهدف أعداداً كبيرة من المستخدمين مع معدل إنشاء وتحديث بيانات مرتفع، فإن هذا النوع يمكن من تحقيق متطلبات المرونة الهيكلية وقابلية التوسع الأفقي من خلال تطبيق كل من تجزئة البيانات (Sharding) لتوزيع الأحمال، والنسخ المتماثل (Replication) لضمان التوافر والاعتمادية، إضافة إلى دعم عمليات الكتابة المتزامنة بكفاءة.

أما بالنسبة للصور الشخصية، فقد تم اعتماد نظام تخزين كائني (Object Storage) بدلاً من تخزين الوسائط مباشرة داخل قاعدة البيانات. يُستخدم هذا النظام لتخزين الملفات الكبيرة مثل الصور بطريقة فعالة ومستقلة عن البيانات النصية، ويتم فقط الاحتفاظ برابط مرجعي (URL) داخل قاعدة البيانات يشير إلى موقع الصورة. حقق هذا النهج فوائد متعددة، أبرزها تقليل العبء على قاعدة البيانات، وتحسين كفاءة التخزين والأداء بفضل تصميم نظام التخزين الكائني للتعامل مع الوسائط، مع توفير آلية وصول مؤقتة وآمنة تضمن حماية الخصوصية.

## 2.7.4 خدمة البحث (Search Service)

تتيح هذه الخدمة البحث عن المستخدمين بالاسم أو جزء منه، بالإضافة إلى عرض اقتراحات متابعة.

نظراً لأن هذا النوع من الاستعلامات يتطلب معالجة نصية واستجابة سريعة، فقد تم اعتماد قاعدة بيانات NoSQL مخصصة للبحث والفهرسة، تعتمد على تقنيات مثل الفهرسة المقلوبة، تحليل النصوص، وتخزين العبارات إلى وحدات قابلة للبحث. وترتيب النتائج حسب الصلة، مع الحفاظ على أداء سريع وتوسع أفقي تلقائي حتى مع نمو البيانات.

## 3.7.4 خدمة المتابعة (Follow Service)

تتولى هذه الخدمة علاقات المتابعة بين المستخدمين، وتوفر عمليات المتابعة وإلغاء المتابعة، بالإضافة إلى حساب عدد المتابعين والمتابعين. تم اختيار قاعدة بيانات من نوع NoSQL (غير علائقية)، لأن طبيعة البيانات بسيطة وغير متشابكة، فلا تحتاج علاقات متعددة الجداول.

كما أن هذا النوع يسمح بإضافة حقول مستقبلية (مثل mute أو block أو mute-duration) دون التأثير على الهيكل العام، ما يدعم استراتيجية التصميم المستقبلي القابل للتوسع. إضافة إلى ذلك، توفر دعماً جيداً للقراءة والكتابة المتكررة، وهي عمليات تحدث بشكل مستمر عند التفاعل مع العلاقات الاجتماعية.

## 4.7.4 خدمة المنشورات (Post Service)

تختص هذه الخدمة بإدارة المحتوى الذي ينشره المستخدمون على المنصة، أي تسمح للمستخدمين بمشاركة صورة (نشرها على المنصة) بحيث يستطيع إضافة توصيف caption للصورة قبل نشرها.

نظراً لأن بنية بيانات المنشورات منظمة وثابتة نسبياً، وتحتوي على عناصر مترابطة مثل معرف المستخدم، التوصيف، الرابط، والتاريخ، فقد تم اعتماد قاعدة بيانات علائقية (Relational) لهذا الغرض.

هذا النوع من قواعد البيانات يوفر دعماً متقدماً للمعاملات (Transactions)، ويضمن التكامل المرجعي بين الكيانات المرتبطة، بالإضافة إلى إمكانية تنفيذ استعلامات دقيقة وتحليلية بكفاءة عالية. كما يساعد على الحفاظ على سجل منشورات منظم يسهل ربطه بخدمات أخرى مثل الخلاصات أو المتابعة.

أما الصور المرفقة، فيتم تحميلها مباشرة إلى نظام تخزين كائني (Object Storage) من خلال روابط تحميل مؤقتة، مع تخزين الرابط فقط داخل قاعدة البيانات بنفس الطريقة المعتمدة ضمن خدمة المستخدمين.

## 5.7.4 خدمة الخلاصات (Feed Service)

تعد خدمة الخلاصات أحد أهم مكونات المنصة، حيث تُقدّم للمستخدم خلاصة مخصصة تحتوي على أحدث المنشورات من الحسابات التي يتابعها، مرتبة زمنياً بشكل ديناميكي. لضمان الأداء العالي والقدرة على التوسع مع زيادة عدد المستخدمين. تم تصميم هذه الخدمة وفق المبدأين النشر عند الكتابة (Fan-out-on-Write) والدمج عند القراءة (Fan-in-on-Read)، مع تمييز نوع المستخدم (عادي أو مؤثر) وتخصيص طريقة المعالجة لكل فئة.

يُصنّف المستخدمون إلى:

- مستخدم عادي: عدد متابعيه محدود، وتوزيع منشوراته لا يشكل عبئاً كبيراً على النظام.
- مستخدم مؤثر: يمتلك عدداً كبيراً من المتابعين، مما يتطلب معالجة أكثر كفاءة لمنشوراته.

تم اعتماد قاعدة بيانات من نوع NoSQL تعتمد على التخزين في الذاكرة (In-Memory Key-Value Store) لإدارة بيانات الخلاصات، نظراً لما توفره من زمن وصول سريع مقارنة بقواعد البيانات التقليدية المعتمدة على التخزين القرصي. هذا النوع من القواعد مناسب للأنظمة التي تعتمد على قراءات وكتابات كثيفة ومتزامنة، كما هو الحال في خدمة الخلاصات التي تحتاج إلى إدراج منشورات جديدة واسترجاع مئات العناصر المرتبة زمنياً في كل لحظة دخول.

وقد تم اختيار هذا النمط من قواعد البيانات تحديداً لأنه يدعم بنى بيانات متخصصة مثل القوائم المرتبة زمنياً، مما يتيح تخزين معرفات المنشورات مع الطابع الزمني وترتيبها بكفاءة دون الحاجة إلى عمليات معالجة ثقيلة أو استعلامات معقدة. هذه الخصائص تُمكن من توليد خلاصة مخصصة لكل مستخدم بشكل لحظي وسلس، حتى تحت الضغط المرتفع.

على الرغم من تنوع أنواع قواعد البيانات المستخدمة عبر الخدمات المصغرة، فإن جميع الصور والوسائط الكبيرة في المنصة سواء كانت صور شخصية أو وسائط مرفقة بالمنشورات، تتم معالجتها من خلال نظام تخزين كائني (Object Storage) مستقل، منفصل عن قواعد البيانات النصية.

تم اعتماد هذا الخيار لأن قواعد البيانات سواء كانت علائقية أو غير علائقية، ليست مصممة لتخزين ملفات وسائط كبيرة الحجم، حيث يؤدي ذلك إلى تضخم غير ضروري في حجم البيانات، انخفاض في الأداء، وصعوبات في التوسع وإدارة النسخ الاحتياطية. بالمقابل، يتيح نظام التخزين الكائني تخزين الملفات بشكل منفصل، مع إمكانية الوصول إليها عبر روابط مؤقتة وآمنة تدعم التحكم في الصلاحيات.

تُخزن هذه الروابط فقط داخل قواعد البيانات المرتبطة بالخدمات المختلفة، بينما تُحفظ الملفات فعلياً داخل نظام التخزين الكائني، مما يضمن انفصلاً بين البيانات النصية والوسائط، ويُحسن من كفاءة التخزين والأداء الكلي للنظام.

## 8.4 تصميم تدفق البيانات والتواصل بين الخدمات

بعد تحديد البنية المعمارية للنظام وتوزيع الوظائف على شكل خدمات مستقلة (Microservices)، قمنا بتصميم تدفق البيانات وتحديد آلية التواصل بين هذه الخدمات بطريقة منهجية تضمن:

- التناسق بين البيانات (Data Consistency).
- الأداء العالي (High Performance).
- المرونة في التوسع (Scalability & Extensibility).

تمّ اعتماد أكثر من نمط للتواصل بين الخدمات بناءً على طبيعة كل حالة استخدام، مع الموازنة بين التواصل المتزامن (Synchronous Communication) عبر REST، والتواصل غير المتزامن (Asynchronous Communication) عبر نظام رسائل.

إلا أن هذه البنية، على الرغم من تماشيها مع المتطلبات النظرية، لم تحقق الأداء المطلوب عند اختبار بعض حالات الاستخدام الحرجة مثل استرجاع الخلاصة (Feed Retrieval). أظهرت نتائج اختبارات الأداء تأخيرات ملحوظة في الاستجابة (سنقوم بعرضها ضمن فصل الاختبارات)، مما استدعى إعادة النظر في بعض القرارات المعمارية.

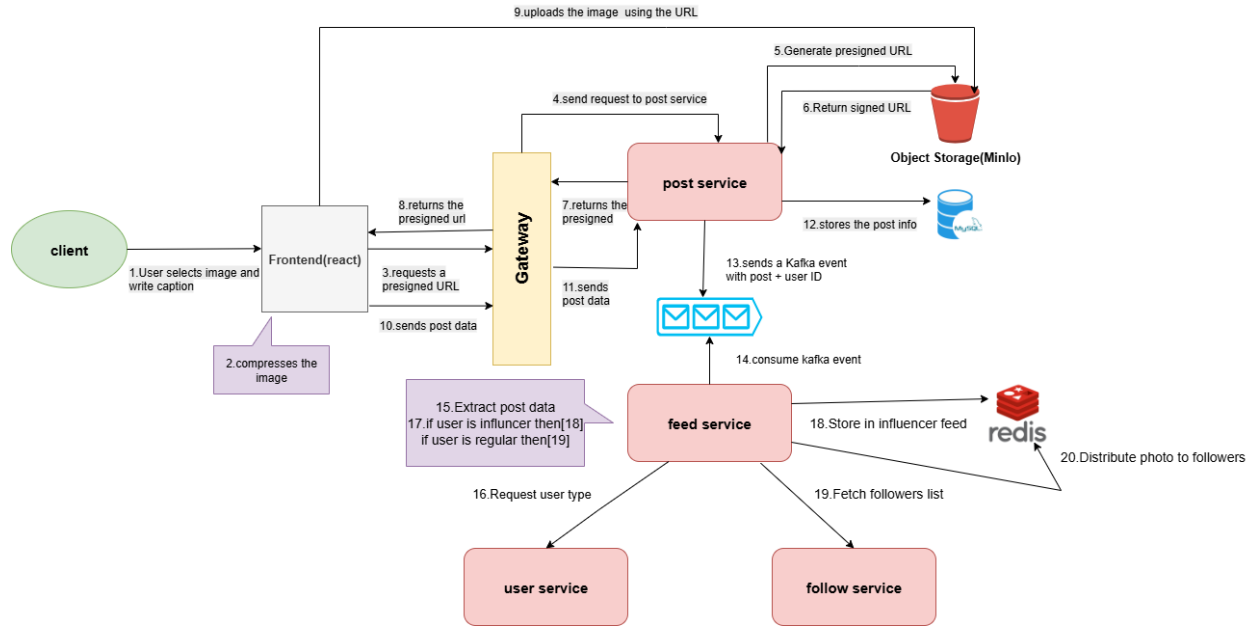
بناءً على ذلك، تم اعتماد بنية محسّنة تركز على مبادئ أكثر فعالية في توزيع الحمل والتخزين المؤقت، كما سيتم شرحها في الفقرات التالية.



## 1.8.4 تدفق البيانات في حالة إنشاء منشور جديد

تعتمد عملية إنشاء منشور جديد على تنسيق متكامل بين مكونات النظام، يتضمن تفاعلاً بين الواجهة الأمامية وخدمة خلفية، بما يضمن تجربة سلسة للمستخدم وكفاءة عالية في المعالجة. تتم العملية من خلال عدة عمليات تواصل متزامنة وغير متزامنة، يمكن تلخيص سير العملية كما يلي:

- 1 يبدأ المستخدم إدخال بيانات المنشور (الصورة، الوصف) من خلال واجهة الاستخدام.
- 2 تقوم الواجهة الأمامية بضغط الصورة.
- 3 ترسل الواجهة طلباً إلى الخدمة الخلفية (post-service) للحصول على رابط رفع مؤقت (Presigned URL) يتضمن الطلب معلومات عن نوع الملف واسمه المطلوب، ويتم إرساله عبر البوابة (API Gateway).
- 4 تقوم الخدمة الخلفية بتوليد رابط مؤقت للرفع باستخدام نظام التخزين الذي نعتمد، بحيث يكون صالحاً لفترة زمنية محددة، ثم تُعيد هذا الرابط إلى الواجهة الأمامية.
- 5 تستخدم الواجهة الرابط المؤقت لرفع الصورة مباشرة إلى نظام التخزين دون المرور بالخادم. هذا يقلل من الضغط على البنية التحتية ويزيد من كفاءة عملية الرفع.
- 6 بعد نجاح رفع الصورة، تُدمج الواجهة رابط الصورة الناتج مع باقي بيانات المنشور (مثل الوصف وتاريخ النشر)، ثم ترسلها في طلب إنشاء منشور جديد إلى post-service عبر Gateway.
- 7 داخل post-service:
  - 7.1 يتم استخراج معرف المستخدم المرسل من ال Gateway.
  - 7.2 يتم حفظ معلومات المنشور في قاعدة البيانات.
  - 7.3 يتم إرسال حدث عبر نظام الرسائل يحتوي على معلومات المنشور بالإضافة إلى معرف المستخدم.
- 8 تستهلك خدمة feed-service هذا الحدث تلقائياً.
- 9 داخل feed-service:
  - 9.1 تستخرج الخدمة بيانات المنشور من الحدث المستلم.
  - 9.2 تطلب الخدمة من user-service نوع المستخدم الذي نشر المنشور.
  - 9.3 في حال كان المستخدم من نوع "مؤثر"، يتم تخزين منشوره في قاعدة بيانات المخصصة له فقط.
  - 9.4 في حال كان المستخدم من النوع "العادي"، تطلب الخدمة من follow-service قائمة متابعيه.
  - 9.5 يتم توزيع المنشور إلى كل متابع عبر تخزينه في قاعدة البيانات ضمن القائمة الخاصة بالمتابع.



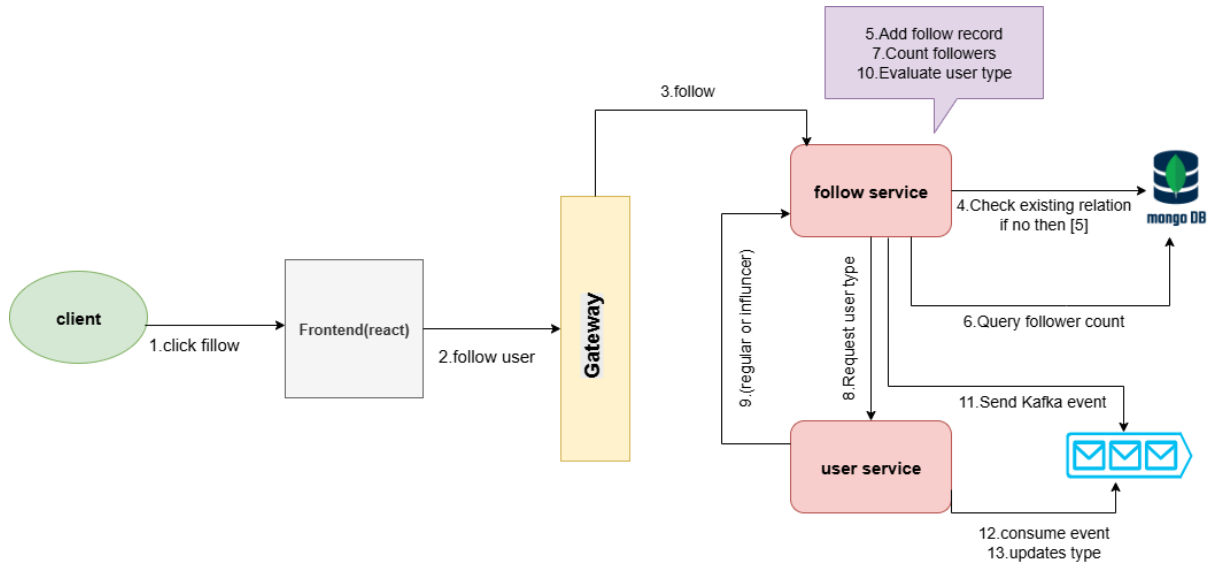
الشكل 22: تدفق البيانات ضمن عملية إنشاء منشور

## 2.8.4 تدفق البيانات في حالة استخدام متابعة مستخدم

عند قيام المستخدم بمتابعة مستخدم آخر ، يتم تنشيط سلسلة من عمليات التواصل المتزامنة وغير المتزامنة بين الخدمات الخلفية ، يتم تنفيذ العملية على النحو التالي:

- 1 يبدأ المستخدم طلب المتابعة عبر الواجهة الأمامية، ويتم تمرير الطلب إلى follow-service عبر البوابة المركزية API Gateway.
- 2 تقوم follow-service بالتحقق مما إذا كانت العلاقة موجودة مسبقاً ضمن قاعدة بيانات. إذا كانت العلاقة موجودة مسبقاً، يتم تجاهل العملية؛ أما إذا كانت غير موجودة، يتم إنشاء كائن متابعة جديد وتخزينه.
- 3 بعد إضافة العلاقة، تحسب follow-service عدد متابعي المستخدم المتابع عبر استعلام مخصص على نفس القاعدة.
- 4 تقوم follow-service بإرسال طلب متزامن إلى user-service لطلب نوع المستخدم الحالي (مستخدم عادي regular أو مؤثر influencer).
- 5 استناداً إلى عدد المتابعين والنوع الحالي للمستخدم، تُحدّد الحاجة إلى تغيير التصنيف:
- 5.1 إذا تجاوز عدد المتابعين حداً معيناً (مثل 1000)، وكان المستخدم مصنفاً كمستخدم عادي، يتم ترفيقه إلى مؤثر.
- 6 في حال حدوث أي تغيير، تقوم الخدمة بإرسال حدث غير متزامن عبر نظام الرسائل، يحتوي على معرف المستخدم ونوعه الجديد.

7 تستهلك خدمة user-service هذا الحدث وتغير نوع المستخدم.



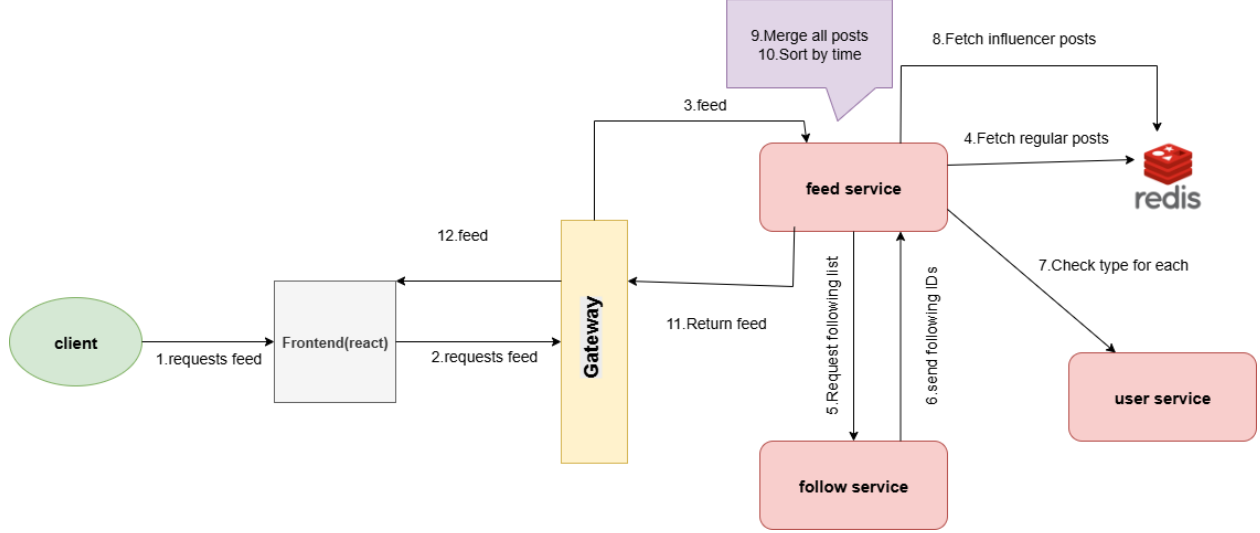
الشكل 23: تدفق البيانات ضمن حالة متابعة مستخدم

### 3.8.4 تدفق البيانات في حالة استخدام عرض الحائط (الخلاصة)

عند قيام المستخدم بفتح صفحة الخلاصة في الواجهة الأمامية، يبدأ النظام سلسلة من عمليات التواصل (المتزامنة وغير المتزامنة) بين الخدمات المختلفة، تهدف إلى عرض منشورات مخصصة له بناءً على من يتابعهم. يتم تنفيذ هذه العملية بالشكل التالي:

- 1 يرسل المستخدم طلباً لعرض الخلاصة من خلال واجهة الاستخدام، فيُرسل طلب إلى feed-service عبر البوابة المركزية API Gateway.
- 2 يتم التعرف على هوية المستخدم من خلال معلومات الجلسة المضمنة في الطلب.
- 3 تبدأ الخدمة بجلب المنشورات المخصصة لهذا المستخدم من قاعدة البيانات، والذي يحتوي على منشورات المستخدمين العاديين الذين يتابعهم.
- 4 بعد تحميل المنشورات العادية، تُرسل الخدمة طلباً إلى follow-service للحصول على قائمة المستخدمين الذين يتابعهم هذا المستخدم.
- 5 تتلقى feed-service القائمة التي تحتوي على معرفات المستخدمين المتابعين.
- 6 تمرّ الخدمة على كل مستخدم متابع من القائمة، وتُرسل طلباً إلى user-service لمعرفة نوعه (مستخدم عادي أو مؤثر).
- 7 إذا كان المستخدم المتابع من النوع "مؤثر"، تبحث feed-service في قاعدة البيانات باستخدام المفتاح لجلب منشوراته.
- 8 تدمج feed-service جميع المنشورات التي تم تحميلها (العادية والمؤثرين) في قائمة واحدة.

- 9 تقوم الخدمة بترتيب هذه المنشورات تنازلياً حسب الطابع الزمني timestamp.
- 10 تعيد feed-service المنشورات الجاهزة إلى الواجهة الأمامية لعرضها للمستخدم.



الشكل 24: تدفق البيانات ضمن حالة عرض الحائط الشخصي

#### 4.8.4 التصميم المحسن لخدمتي نشر الصور والخلاصة وتحسين تدفق البيانات بعد القيام بالاختبارات

يعد القيام بالمزيد من الاختبارات على خدمة الخلاصة من أجل عرض الخلاصة (الحائط) للمستخدم لاحظنا أن هذه العملية تأخذ وقت طويل جداً للاستجابة (ستعرض النتائج ضمن فصل الاختبارات) وذلك بسبب عمليات التواصل المتزامنة مع الخدمات الأخرى مثل خدمة المستخدم (user service) وخدمة المتابعة (follow service) حيث كنا نحتاج من أجل كل طلب أن نقوم بالتواصل مع الخدمتين من أجل معرفة المتابعين و أنواعهم (مستخدم عادي أو مؤثر) وذلك لأجل معرفة كيفية التعامل وجلب المنشورات، وهذا بدأ يؤثر مع ازدياد عدد المتابعين. وبناء على هذا تم إعادة النظر في التصميم حيث قمنا بالتحسينات التي سنشرحها في الفقرات التالية :

#### 1.4.8.4 تحسين تدفق البيانات في حالة إنشاء منشور جديد

رغم كفاءة البنية السابقة في تسيير عملية إنشاء المنشورات، إلا أن نتائج اختبارات الأداء، ولا سيما على مستوى خدمة الخلاصة (Feed Service)، أظهرت ارتفاعاً في زمن الاستجابة عند استرجاع المنشورات، لذلك تم اعتماد طريقة مختلفة للتعامل مع عملية إنشاء المنشور ونشره ضمن نظام الرسائل ، حيث قمنا بنقل منطق تصنيف المستخدم إلى post-service.

في التصميم السابق، كانت خدمة feed-service هي من تتواصل مع user-service لتحديد نوع المستخدم (عادي أو مؤثر) بعد استقبال الحدث. ولكن هذا تسبب في تأخير معالجة الحدث ضمن feed-service، نظراً لاعتمادها على طلب متزامن إضافي.

ولذلك، قمنا بتعديل post-service بحيث يقوم بالاستعلام عن نوع المستخدم مباشرة من خدمة المستخدم (user service) قبل إرسال حدث إنشاء منشور إلى نظام الرسائل ، ويضيف هذا الحقل نوع المستخدم (userType) إلى الحدث المرسل عبر نظام الرسائل. هذا يقلل من الاعتمادية داخل feed-service ويزيد من سرعة المعالجة.

أي تبقى نفس الخطوات المذكورة ضمن الفقرة السابقة (تدفق البيانات في حالة إنشاء منشور جديد) حتى الخطوة 8

#### 9 داخل feed-service:

9.1 تستخرج الخدمة بيانات المنشور من الحدث المستلم ونوع المستخدم .

9.2 في حال كان المستخدم من النوع "العادي"

9.2.1 تطلب الخدمة من follow-service قائمة متابعيه.

9.2.2 يتم توزيع المنشور إلى كل متابع عبر تخزينه في قاعدة البيانات ضمن القائمة الخاصة بالمتابع.

9.3 في حال كان المستخدم من النوع "مؤثر"

9.3.1 يتم تخزين منشوره في قاعدة بيانات المخصصة له فقط.

9.3.2 تطلب الخدمة من follow service قائمة المتابعين ، و تُسجل قائمة متابعي هذا المؤثر ضمن قاعدة البيانات باستخدام مفاتيح منظمة.

9.3.3 ومن ثم يتم تسجيل المستخدم كمؤثر ضمن مجموعة خاصة (all\_influencers) تُستخدم لاحقاً في عمليات الصيانة الدورية.

### 2.4.8.4 التصميم المحسن لآلية استرجاع الخلاصة (Feed)

يعتمد التصميم الجديد على مبدأ التخزين المسبق للمنشورات (Precomputed Feeds)، حيث يتم توزيع المنشورات لحظة إنشائها على القوائم المناسبة في النظام، بدلاً من بنائها لحظة طلب المستخدم للخلاصة.

تبقى جميع الخطوات كما هي في فقرة (تدفق البيانات في حالة استخدام عرض الحائط(الخلاصة)) حتى الخطوة 3

4 بعد تحميل المنشورات العادية، يتم فحص ما إذا كان المستخدم متابعاً لأي من المؤثرين المسجلين.

5 إن وُجد، تُحمّل منشورات هؤلاء المؤثرين من قوائمهم الخاصة، وتُدمج مع المنشورات الأخرى.

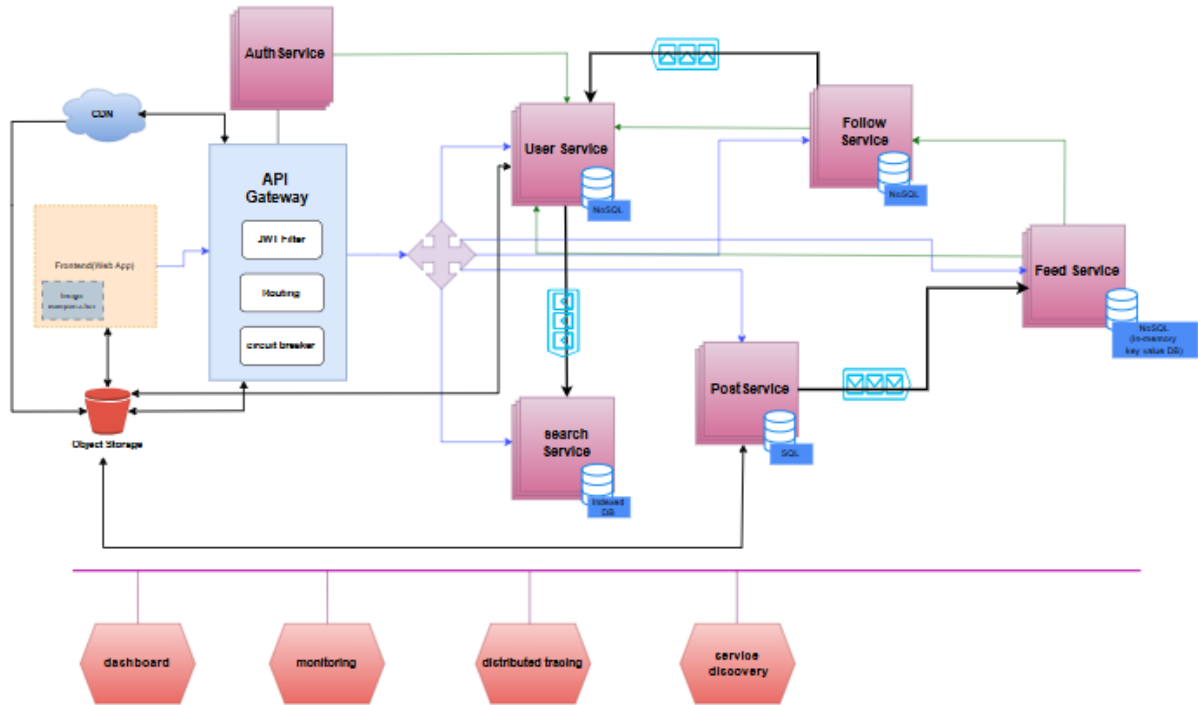
6 تُدمج جميع المنشورات في قائمة واحدة ويتم ترتيبها زمنياً بشكل تنازلي لضمان حداثة العرض.

7 تُطبق تقنية التقسيم إلى صفحات (pagination) بعد ترتيب المنشورات، مما يتيح التحكم في عدد المنشورات المعروضة لكل طلب.

❖ تتم إجراء صيانة دورية لقوائم المتابعين لتفادي حدوث تباين بين الواقع وقائمة المتابعين المخزنة في قاعدة البيانات، حيث كل مدة زمنية تقوم خدمة ضمن خدمة الخلاصة بالمرور على جميع المستخدمين الموجودين في قائمة المؤثرين all\_influencers ، تُعيد استعلام متابعي كل مؤثر من follow-service ،ومن ثم تُحدّث القائمة ضمن قاعدة البيانات بالقائمة الجديدة من المتابعين. هذه الآلية تضمن أن المنشورات المعروضة من المؤثرين تصل إلى المتابعين الفعليين فقط، وتُعالج أي تغييرات في علاقات المتابعة بصورة تلقائية وبأقل كلفة زمنية.

## 9.4 هيكلية وتصميم النظام

عند تصميم منصة اجتماعية لمشاركة الصور، لم يكن بالإمكان الاكتفاء بتلبية المتطلبات الوظيفية فقط، بل كان للمتطلبات غير الوظيفية مثل الأداء، القابلية للتوسع، التوفر العالي، وزمن الاستجابة دور محوري في تشكيل القرار المعماري وتحديد الهيكل العام للنظام. وقد أظهرت دراسة هذه المتطلبات ضرورة تبني بنية موزعة قائمة على نمط الخدمات المصغرة (Microservices Architecture)، تُدار كل منها بوحدة مستقلة مسؤولة عن جزء محدد من منطق العمل، مما يعزز قابلية التطوير والنشر والصيانة بشكل منفصل لكل خدمة. تحقيق المرونة في التفاعل بين المكونات، تم اعتماد عدة أنواع للتواصل بين الخدمات التواصل المتزامن (Synchronous) عبر بروتوكول HTTP والتواصل غير المتزامن (Asynchronous) باستخدام نظام رسائل قائم على الأحداث، بما يسمح بفصل المهام وتحقيق استجابة أفضل تحت الضغط المرتفع. كما تم توحيد إدارة المصادقة باستخدام JWT والتحقق منه داخل الـ Gateway، ما ساعد على تبسيط الخدمات الخلفية وزيادة أمان النظام.



## الشكل 25: تصميم النظام

يمنح هذا التصميم القائم على الخدمات المصغرة بنية تحتية مرنة تسمح بتنفيذ العديد من العمليات الهندسية المتقدمة دون التأثير على بقية النظام. فعلى سبيل المثال، يمكن إجراء نسخ متعددة (Replicas) لأي خدمة خلفية بهدف تعزيز التوافر وتحقيق موازنة الأحمال، حيث يُدار هذا التكرار ضمن بيئة Kubernetes. كما يُمكن إضافة خدمة جديدة سواء لمعالجة ميزة جديدة أو لتحسين مكون قائم دون الحاجة لإيقاف النظام أو إعادة بناء بقية المكونات، وذلك بفضل الفصل بين المهام والتكامل غير المتزامن بين الخدمات عبر الاتصالات غير المتزامنة واستخدام نمط الاتصال القائم على الأحداث.

أما على مستوى قواعد البيانات، فقد تم اختيار تقنيات تدعم التوسّع الأفقي (Horizontal Scaling)، مما يتيح التعامل مع الحجم المتزايد للبيانات دون عنق زجاجة. وأيضاً يتم استخدام مكُونات مراقبة الأداء وتتبع الطلبات، التي تشمل أنظمة جمع القياسات الزمنية (Metrics Collection)، ولوحات العرض التحليلية (Monitoring Dashboards)، وآليات تتبّع موزعة للطلبات عبر الخدمات (Distributed Tracing)، يُمكن للنظام مراقبة الحالة التشغيلية بدقة، وتحليل الاختناقات وتحديد المشاكل في وقت مبكر، مما يعزز من القدرة على الصيانة الاستباقية وتحقيق موثوقية عالية في البيئة الإنتاجية.

## الفصل الخامس

### التنفيذ والتطوير (Implementation & Development)

نستعرض في هذا الفصل الجوانب التطبيقية للمشروع، بما في ذلك الأدوات والتقنيات البرمجية المستخدمة، وآلية بناء الخدمات وتكاملها

## 1.5 مقدمة

بعد وضع تصوّر المعماري العام وتحديد مكُونات النظام وآلية التفاعل بينها، قمنا بالانتقال إلى مرحلة التنفيذ العملي التي تُترجم هذا التصميم إلى نظام فعلي متكامل وقابل للنشر والمراقبة والاختبار. في هذا الفصل، نعرض الهيكلية البرمجية المعتمدة، والأدوات والتقنيات المستخدمة أثناء التطوير، إلى جانب أهم القرارات التي اتُخذت أثناء كتابة الكود البرمجي وتنظيم الخدمات. يُشكّل هذا الفصل مرآة عملية لفهم كيفية تفعيل المبادئ التصميمية على أرض الواقع، بطريقة تُراعي قابلية التطوير، الأداء، والوضوح في كتابة الكود البرمجي وتوزيع المهام البرمجية.

## 2.5 الأدوات والتقنيات المستخدمة

يعتمد تنفيذ هذا المشروع على مجموعة من التقنيات الحديثة والأدوات المتكاملة التي تُمكن من بناء نظام موزّع وقابل للتوسّع، يعتمد في بنيته على الخدمات المصغّرة. وقد جرى اختيار هذه الأدوات وفق معايير تشمل الكفاءة، وسهولة الاستخدام، والدعم المجتمعي، وسلاسة التكامل. وفيما يلي عرض تفصيلي لكل أداة ودورها في تنفيذ المشروع.

### Spring Boot و Spring Framework 1.2.5

يعدّ Spring Framework أحد أشهر الأطر المستخدمة في تطوير تطبيقات جافا الاحترافية، وخاصة في بيئات المؤسسات الكبيرة. وقد تم اختيار Spring Boot و Spring كأساس لتطوير الخدمات الخلفية (Back-end) في هذا المشروع، وذلك لما يقدمانه من ميزات قوية ودعم واسع، وسهولة في بناء أنظمة موزعة تعتمد مبدأ الخدمات المصغرة (Microservices).

#### 1.1.2.5 ما هو Spring؟

يُعد Spring Framework إطاراً شاملاً لتطوير تطبيقات جافا الحديثة، ويُستخدم على نطاق واسع في بناء أنظمة ويب قابلة للتوسع والصيانة، خصوصاً في بيئات الأعمال المتوسطة والكبيرة.

تم تصميم Spring لحل التحديات التي كانت تواجه مطوّري Java EE التقليدية، مثل تعقيد التهيئة وضعف المرونة وتشابك الكود. ويتيح Spring بنية برمجية تعتمد على الحقن التلقائي للاعتمادات (Dependency Injection) والبرمجة الموجهة للجوانب (AOP)، ما يمنح المطورين تحكماً أدق في تدفق البرنامج وتقليل التكرار في الكود.

❖ ما الذي يمكن إنجازه من خلال Spring؟

- بناء تطبيقات موزعة تعتمد على الخدمات المصغرة (Microservices).
- إدارة الاتصالات بين الخدمات بسهولة وكفاءة.
- تحقيق أمان عالي ومرونة في المصادقة والتفويض.
- تطوير REST APIs قابلة للتوسع والصيانة.
- التكامل مع أنظمة التخزين المختلفة مثل MongoDB، MySQL، وغيرها.
- دعم العمل في البيئات السحابية (Cloud-Native).

❖ أهم الميزات التقنية لـ Spring:

1. برمجة غير مترابطة (Non-blocking):



يمكن لـ Spring تطوير تطبيقات تعتمد على نموذج non-blocking باستخدام Spring WebFlux، مما يتيح أداءً أعلى وفعالية في التعامل مع آلاف الطلبات المتزامنة.

2. تكامل سهل مع الحوسبة السحابية (Cloud Integration):

يدعم Spring أدوات مثل Spring Cloud لتسهيل التفاعل مع بيئات مثل Kubernetes، مما يسمح بإنشاء خدمات مرنة قابلة للتوسع الأفقي (Horizontal Scaling).

3. تطوير واجهات برمجية تفاعلية (Responsive APIs):

يوفر Spring MVC إمكانيات متقدمة لبناء واجهات REST مرنة وسريعة، مع دعم مدمج لتنسيقات JSON/XML وخيارات الفلترة والترتيب والتقسيم إلى صفحات.

4. سهولة اختبار الخدمات (Testability):

يعزز Spring إمكانية اختبار كل وحدة برمجية على حدة، من خلال دعم JUnit وMockito، مما يُسهل اكتشاف الأخطاء مبكراً.

5. المرونة (Flexibility):

يتيح Spring إمكانية تهيئة التطبيقات باستخدام ملفات Properties أو YAML، مع القدرة على تخصيص أي سلوك تقريباً دون تعديل جوهري في الكود.

6. دعم للمهام الفورية والمتأخرة (Real-time & Batch Tasks):

يدعم Spring تنفيذ المهام المتزامنة باستخدام @Async، بالإضافة إلى جدولة المهام الدورية عبر @Scheduled، مما يتيح تنفيذ كل من المهام اللحظية والمهام المجدولة بسهولة.

7. سهولة الصيانة والتوسع:

يمكن فصل منطق الأعمال عن طبقات الوصول للبيانات والواجهات بسهولة، مما يجعل التطبيق قابلاً للتوسع على المدى الطويل.

8. مجتمع ودعم واسع:

يتميز Spring بمجتمع مطورين ضخم ومكتبات كثيرة جاهزة (Starter Dependencies)، مما يوفر حلولاً جاهزة ويقلل زمن التطوير.

## 2.1.2.5 ما هو Spring Boot؟

يُعد Spring Boot امتداداً حديثاً لإطار Spring التقليدي، تم تصميمه خصيصاً لتبسيط وتوحيد عملية تطوير تطبيقات Java، لا سيما التطبيقات القائمة على بنية الخدمات المصغرة (Microservices Architecture). وقد أنشئ هذا الإطار لمعالجة التعقيد المتزايد في إعداد مشاريع Spring التقليدية، وذلك من خلال تقديم منهجية "الإعداد التلقائي" (Auto Configuration) و"الاتفاق بدل التهيئة" (Convention over Configuration)، مما يتيح للمطورين التركيز على منطق الأعمال بدلاً من الانشغال بالإعدادات والتهيئة اليدوية.

### ❖ المفهوم العام لـ Spring Boot

يوفر Spring Boot بيئة تنفيذ جاهزة تدمج المكونات الأساسية التي تحتاجها أي خدمة ويب، مثل خوادم التطبيقات المضمنة (Embedded Servers)، ملفات الإعداد، وإدارة التبعيات. وهذا يعني أن كل خدمة يتم بناؤها باستخدام Spring Boot تُصبح وحدة مستقلة وقابلة للتنفيذ (Executable Unit) يمكن تشغيلها مباشرة باستخدام أمر واحد مثل `java -jar service.jar` دون الحاجة إلى خادم تطبيق خارجي.

### ❖ أبرز ميزات Spring Boot

#### 1. تهيئة تلقائية (Auto Configuration):

يقوم Spring Boot بتحليل بيئة المشروع تلقائياً وتفعيل الإعدادات اللازمة بناءً على التبعيات الموجودة في ملف `pom.xml` أو `build.gradle`، مما يوفر وقتاً وجهداً كبيرين في إعداد المشروع.

#### 2. خادم مدمج (Embedded Server):

يدعم Spring Boot تشغيل التطبيقات من خلال خوادم مثل Tomcat، Jetty أو Undertow، دون الحاجة لتثبيت خادم خارجي، مما يبسط عملية النشر والنقل بين البيئات.

#### 3. مبادرات جاهزة (Starters):

يقدم Spring Boot ما يُعرف بـ Starter Dependencies، وهي مجموعات جاهزة من المكتبات تلائم مختلف الاستخدامات (مثل: `spring-boot-starter-web`، `spring-boot-starter-security`) مما يقلل من الحاجة لتحديد التبعيات يدوياً.

#### 4. المراقبة والتحليل (Actuator):

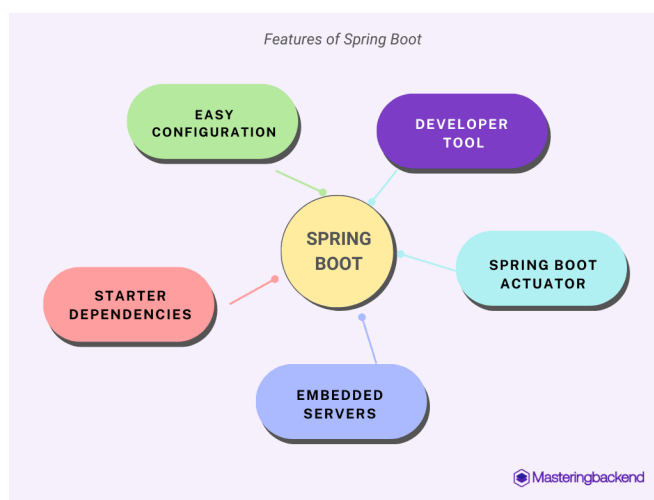
يتضمن Spring Boot وحدة مراقبة مدمجة تُعرف بـ Spring Boot Actuator، تتيح الاطلاع على بيانات تشغيلية مهمة مثل مؤشرات الأداء ونقاط النهاية الفعالة واستهلاك الموارد.

5. سهولة التهيئة والتخصيص:

باستخدام ملفات application.properties أو application.yml، يمكن التحكم بسلوك التطبيق وتحديد إعدادات قواعد البيانات، المنافذ، المصادقة، بطريقة مرنة وموحدة.

6. جاهزية للإنتاج (Production Readiness):

تم تصميم Spring Boot ليكون ملائماً للإنتاج بشكل افتراضي، مع دعم للأمان، السجلات، والمعايير الصناعية الحديثة، مما يقلل الحاجة إلى تعديلات عميقة عند الانتقال من التطوير إلى التشغيل.



الشكل 26: مميزات Spring Boot

❖ دور Spring Boot في بناء الأنظمة الموزعة

تتوافق بنية Spring Boot مع متطلبات تطوير الأنظمة المعتمدة على الخدمات المصغرة، حيث تمكن كل خدمة من أن تُبنى وتُدار بشكل مستقل. هذا يساهم في تحقيق:

- مرونة التطوير والنشر (Agility): إذ يمكن لكل فريق العمل على خدمة منفصلة دون التأثير على بقية النظام.
- قابلية التوسع الأفقي (Horizontal Scalability): من خلال نشر نسخ متعددة من نفس الخدمة بسهولة.
- عزل الأعطال (Fault Isolation): حيث لا تؤثر مشاكل خدمة معينة على باقي النظام.

❖ سبب اختيار Spring Boot في هذا المشروع

بُنيت جميع الخدمات الخلفية في هذا المشروع باستخدام Spring Boot، وذلك للأسباب التالية:

- سهولة وسرعة التطوير: تم إنشاء الخدمات في وقت قياسي باستخدام Spring Initializer.
- جاهزية للنشر: بُني كل خدمة كملف JAR تنفيذي، ثم تُضمّن ضمن صورة Docker، مما يسهل نشرها داخل بيئة Kubernetes.
- دعم متكامل للمكونات الأخرى: تمّ تكامل Spring Boot بسهولة مع قواعد البيانات (MongoDB, MySQL)، نظام الرسائل (Kafka)، خوادم الملفات (MinIO)، ومحرك البحث (Elasticsearch).
- إدارة مركزية للإعدادات: يتم تخصيص إعدادات كل خدمة من خلال ملفات application.properties، مما يسهل ضبط سلوك الخدمات في بيئات مختلفة.
- مرونة في بناء REST APIs والمصادقة: باستخدام Spring MVC لتصميم الواجهات البرمجية، و Spring Security مع JWT لتنفيذ المصادقة والتفويض بطريقة آمنة.

### Spring Security 3.1.2.5 – إطار المصادقة والتفويض في Spring

يُعد Spring Security أحد أهم مكونات منظومة Spring، وهو إطار متكامل لحماية تطبيقات الويب وتطبيقات REST API من خلال توفير آليات متقدمة للمصادقة (Authentication) والتفويض (Authorization). يتمتع هذا الإطار بدرجة عالية من القابلية للتخصيص، ويُستخدم على نطاق واسع في المشاريع التي تتطلب حماية عالية للمصادر ومرونة في ضبط الصلاحيات.

❖ المفهوم العام لـ Spring Security

تم تصميم Spring Security ليكون طبقة مستقلة توضع فوق تطبيقات Spring، بحيث تُؤمن الوصول إلى الموارد، وتمنع أي محاولة وصول غير مصرح بها. يتكامل الإطار بسهولة مع بقية مكونات Spring مثل Spring Boot و Spring MVC، كما يدعم تقنيات المصادقة القياسية مثل:

- المصادقة القائمة على النموذج (Form-based authentication)
- المصادقة عبر HTTP Basic

- المصادقة عبر رموز JWT (المستخدمة في هذا المشروع)
- OAuth 2.0 و SSO في التطبيقات المتقدمة.

❖ المهام الأساسية التي يوفرها Spring Security

1. مصادقة المستخدمين (Authentication):

يتحقق من هوية المستخدم باستخدام بيانات الاعتماد (مثل البريد الإلكتروني وكلمة المرور)، ويدعم آليات تخزين متعددة مثل قواعد البيانات، LDAP، أو رموز JWT.

2. تصفية الطلبات عبر سلسلة فلاتر (Security Filters):

يُعتمد على سلسلة من الفلاتر (Filter Chain) تقوم بتحليل كل طلب HTTP قبل تسليمه إلى التطبيق، مما يسمح بالتعامل مع رموز المصادقة Tokens وملفات تعريف الارتباط Cookies والترويسات headers، ومختلف آليات التحقق من الهوية.

3. الحماية من الثغرات الشائعة:

يتضمن حماية تلقائية من هجمات XSS، CSRF، Session Fixation، وغيرها، مما يجعله إطاراً موثقاً في بيئات الإنتاج.

4. تكامل مع JWT:

في هذا المشروع، تم استخدام Spring Security مع JWT لتأمين الوصول إلى الخدمات. حيث يتم إصدار رمز JWT عند تسجيل الدخول، ويتم التحقق منه في كل طلب باستخدام فلتر مخصص في بوابة Gateway.

❖ كيف تم استخدامه في المشروع؟

في بنية هذا المشروع التي تعتمد على Spring Boot و Gateway، تم تطبيق Spring Security على عدة مستويات:

■ داخل auth-service:

تم تنفيذ آلية توليد رموز JWT بعد نجاح تسجيل الدخول، وتضمين معلومات المستخدم في رمز المصادقة (مثل المعرف، والبريد الإلكتروني).

■ في بوابة Gateway:

تم إنشاء فلتر مخصص (GlobalFilter) يقوم بتحليل رمز المصادقة، التحقق من صحته، واستخلاص معرف المستخدم ودوره، ثم إدراج هذه المعلومات ضمن رؤوس الطلب (X-User-Id) قبل تمريرها إلى الخدمات الأخرى.

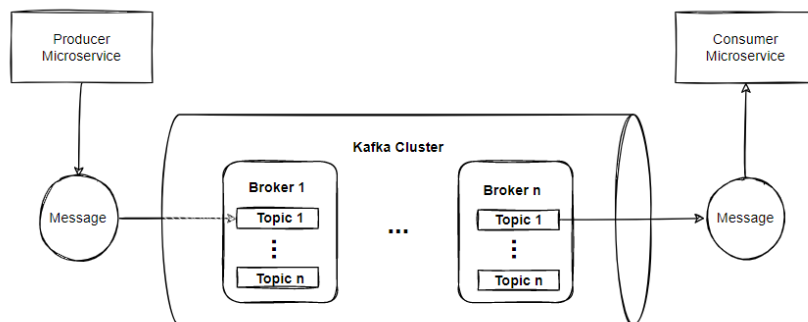
## 2.2.5 Apache Kafka – نظام الرسائل في الأنظمة الموزعة

يُعد Apache Kafka أحد أقوى نظم معالجة تدفق البيانات (Stream Processing Systems) وأكثرها استخداماً في البيئات الموزعة، وهو مشروع مفتوح المصدر طُوّر في البداية من قبل شركة LinkedIn، ثم أصبح جزءاً من Apache Software Foundation.

يعمل Kafka كمنصة لنقل البيانات بين الخدمات أو التطبيقات بطريقة غير متزامنة (Asynchronous)، ويُبنى على نموذج النشر/الاشتراك (Publish/Subscribe)، مما يُمكنه من تحقيق موثوقية عالية وأداء فعّال في معالجة كميات ضخمة من البيانات في الوقت الحقيقي.

كما ورد في وثائق Apache [16]، يتميز Kafka بالخصائص التالية:

- التحمل العالي للأخطاء (Fault Tolerance): بفضل إعادة التكرار (Replication) وتقسيم البيانات (Partitioning).
- الإرسال السريع (High Throughput): ما يجعله مناسباً لمعالجة ملايين الرسائل في الثانية.
- قابلية التوسع الأفقي (Horizontal Scalability): من خلال إضافة Brokers بسهولة.
- مرونة التوصيل (Loose Coupling): مما يسمح بفصل الخدمات دون الاعتماد المباشر بينها.



الشكل 27: بنية Kafka

❖ استخدام Kafka في مشروع منصة اجتماعية لمشاركة الصور

تم اعتماد Apache Kafka في هذا المشروع كقناة وسيطة موثوقة لنقل الأحداث بين الخدمات المصغرة، وقد تم تكوينه ليعمل كبنية مركزية لتبادل الرسائل، حيث ترتبط به مختلف الخدمات الخلفية من خلال إعدادات موحدة، تتيح لكل خدمة إرسال واستقبال الأحداث دون الحاجة إلى اتصال مباشر فيما بينها.

تم استخدام Kafka لتنفيذ نمط التكامل غير المتزامن (Asynchronous Integration) ، حيث تقوم بعض الخدمات بنشر أحداث (مثل إنشاء منشور أو تعديل بيانات مستخدم)، بينما تقوم خدمات أخرى بالاستماع لتلك الأحداث وتنفيذ إجراءات مرتبطة بها تلقائياً.

هذا التوزيع أدى إلى تقليل الترابط بين الوحدات (Low Coupling) ، وتحقيق بنية مرنة قابلة للتوسع، تُمكن من إضافة مستهلكين جدد أو استبدال الخدمات بسهولة دون التأثير على النظام ككل.

يسر تكامل Kafka مع إطار العمل Spring Boot عملية الربط بين النظام ومنصة الرسائل، حيث وفّرت مكتبة Spring Kafka واجهات سهلة للتعامل مع الأحداث، مما ساعد على توجيه تدفق البيانات بين الخدمات بكفاءة وموثوقية، مع الحفاظ على هيكلية النظام المنفصلة والمنظمة.

### 3.2.5 قاعدة البيانات غير العلائقية – MongoDB

تُعد MongoDB قاعدة بيانات وثائق (Document-Oriented Database) غير علائقية (NoSQL) ، تُستخدم على نطاق واسع في تطبيقات تعتمد على البنى الموزعة والسحابية، نظراً لقدرتها على تمثيل الكيانات المعقدة دون الحاجة إلى مخطط بيانات ثابت، ودعمها الأصلي للتوسع الأفقي والتكرار التلقائي.

وفقاً للتوثيق الرسمي لـ MongoDB [17] ، تستند القاعدة إلى نموذج BSON (Binary JSON) مما يُمكن المطورين من تمثيل الكيانات بمرونة تامة، سواء كانت بسيطة أو متداخلة.

يُعدّ كل مستند (Document) في MongoDB وحدة مستقلة من البيانات، تُخزّن داخل مجموعات (Collections) وتُحدد باستخدام معرف فريد id\_. تتميز MongoDB بأنها schema-less، ما يسمح بإضافة أو إزالة الحقول بحرية دون التأثير على بقية الوثائق، وهي خاصية مثالية في أنظمة تتطور بسرعة مثل منصات التواصل الاجتماعي، حيث قد تطرأ تغييرات متكررة على خصائص المستخدمين أو هيكل المنشورات.

#### 1.3.2.5 آليات توزيع البيانات وضمان التوافر في MongoDB

❖ النسخ المتماثل (Replication)

النسخ المتماثل هو مفهوم أساسي في قواعد البيانات الموزعة ويهدف إلى تحسين توفر البيانات (Availability) واستمرارية الخدمة في حال تعرّض إحدى العقد للفشل.

في MongoDB، يتم تكوين ما يُعرف بـ Replica Set يتكون من:

- عقدة رئيسية (Primary) تستقبل جميع عمليات الكتابة والتحديث.

- عقد ثانوية (Secondaries) تقوم بنسخ البيانات من العقدة الأساسية.
- عملية انتخاب (Election) تلقائية تضمن تعيين عقدة بديلة في حال توقف العقدة الأساسية.

هذا المفهوم يضمن مقاومة الأعطال ويسمح بإجراء نسخ احتياطي وقراءة البيانات من أكثر من مصدر.

## ❖ التقسيم (Sharding)

عندما تنمو قاعدة البيانات بشكل يتجاوز قدرة الخادم الواحد على المعالجة والتخزين، يصبح الحاجة إلى تقسيم البيانات إلى شرائح (Shards) أمراً ضرورياً.

ويُعرف الـ Sharding بأنه عملية تقسيم أفقي للبيانات إلى شرائح مستقلة، بحيث تُوزع كل شريحة على خادم منفصل.

ولإدارة هذا النظام، تستخدم MongoDB بنية تشمل:

- خوادم التكوين (Config Servers): تحتفظ ببيانات التكوين ومواقع الشرائح.
- الشرائح (Shards): وهي التي تحتوي على البيانات الفعلية، حيث تكون كل شريحة عبارة عن Replica Set.
- الموجه Mongos: وسيط يوجه الطلبات إلى الشريحة المناسبة استناداً إلى قيمة المفتاح المجرى.

## ❖ مفتاح الشريحة (Shard Key):

يُستخدم مفتاح الشريحة لتحديد كيفية توزيع المستندات بين مختلف الشرائح (Shards) في قاعدة البيانات. ويُعد اختيار هذا المفتاح بدقة أمراً بالغ الأهمية لتحقيق توازن في الحمل (Load Balancing) بين العقد.

تدعم MongoDB عدة أنواع من مفاتيح الشريحة، من أبرزها:

- التقسيم القائم على النطاق (Ranged Sharding): حيث تُقسّم المستندات وفق تسلسل القيم، مما قد يؤدي إلى تركّز البيانات في شريحة واحدة (data hotspot).
- التقسيم القائم على التجزئة (Hashed Sharding): حيث يُطبّق خوارزمية تجزئة على قيمة المفتاح، مما يُسهم في توزيع البيانات بشكل أكثر توازناً.
- التقسيم القائم على النطاقات المنطقية (Zone Sharding): حيث تُوزّع البيانات بناءً على نطاقات محددة مسبقاً (مثل الموقع الجغرافي للمستخدمين).

في هذا المشروع، تم اختيار Hashed Sharding باستخدام المفتاح \_id لضمان توزيع متساوٍ للبيانات بين العقد، وهو خيار موصى به في الوثائق الرسمية لمشاريع ذات أحجام متوسطة إلى كبيرة.

## ❖ استخدام MongoDB في هذا المشروع



تم استخدام MongoDB في مشروعنا ضمن خدمتين أساسيتين:

■ خدمة المستخدم user-service:

تُخزن معلومات المستخدمين بما يشمل الاسم، البريد الإلكتروني، الصورة، كلمة المرور، نوع الحساب (regular أو influencer)، والتوصيف الشخصي (bio).

- تم استخدام MongoTemplate لتحديث الحقول بطريقة مرنة.
- استخدمت المستودعات MongoRepository في العمليات القياسية.
- تم تفعيل فهرسة فريدة (Unique Indexing) على البريد الإلكتروني لضمان التميز وسرعة التحقق أثناء تسجيل الحساب.

لتوفير أداء عالٍ وقابلية توسع أفقي حقيقية، قمنا بإعداد موّج MongoDB Sharded Cluster داخل بيئة التطوير والإنتاج يتضمن:

- خادم تكوين (Config Server)
  - شريحتين shards لضمان توزيع البيانات.
  - 3 نسخ متماثلة (Replication) لكل شريحة Shard لتوفير التكرار، وتعزيز الاستمرارية في حال فشل إحدى العقد.
  - Mongos Router لإدارة توجيه الاستعلامات تلقائياً إلى الشريحة Shard المناسبة.
- وقد تم إعداد التقسيم (Sharding) على مستوى مجموعة users، واختيار المفتاح id\_ كمفتاح تجزئة Hashed Shard Key. إن استخدام Hashed Sharding على مفتاح id\_ يحقق توزيعاً عشوائياً ومتوازناً للمستندات بين الشرائح shards، مما يمنع تركّز البيانات في عقدة واحدة (Data Skew)، وهو أمر بالغ الأهمية في خدمات مثل المستخدمين حيث لا يوجد نمط وصول متكرر لمفتاح معين.
- كما أنه مع وجود 3 نسخ متماثلة لكل shard، تضمن المنصة استمرار الخدمة حتى عند تعطل عقدة واحدة أو أكثر، بفضل آلية الانتخابات التلقائية لعقدة بديلة.

```

---
shardedDataDistribution
[
  {
    ns: 'user-db.users',
    shards: [
      {
        shardName: 'shard2ReplSet',
        numOrphanedDocs: 0,
        numOwnedDocuments: 1020,
        ownedSizeBytes: 319260,
        orphanedSizeBytes: 0
      },
      {
        shardName: 'shard1ReplSet',
        numOrphanedDocs: 0,
        numOwnedDocuments: 1011,
        ownedSizeBytes: 316443,
        orphanedSizeBytes: 0
      }
    ]
  },
]

```

الشكل 28: توزيع البيانات بين الشرائح في MongoDB باستخدام Hashed Shard Key.

■ خدمة المتابعة follow-service:

تُخزن العلاقات الثنائية بين المستخدمين (follower/following) ضمن مجموعة follows

- اعتمدت الخدمة على مستودعات MongoDB.
- تم تصميم الوثائق لتشمل توقيت المتابعة followedAt.
- تدعم الخدمة استرجاع المتابعين، المتابعين، والتحقق من العلاقة الثنائية.

## Redis 4.2.5

Redis هو نظام إدارة قواعد بيانات مفتوح المصدر، يعمل في الذاكرة (In-Memory Data Store) ويُستخدم بشكل رئيسي كمخزن مؤقت (Cache) أو كوسيط رسائل (Message Broker) بفضل سرعته الفائقة ودعمه لعدة هياكل بيانات مثل القوائم والمجموعات والقواميس. وبحسب وثائق Redis الرسمية، يتميز Redis بسرعة استجابة زمنية تقل غالباً عن ميلي ثانية واحدة، ما يجعله مثالياً للتطبيقات التي تتطلب أداءً عالياً مثل أنظمة التوصيات، الخلاصات (Feeds)، أو إدارة الجلسات (Sessions) [18].

ضمن هذا المشروع، تم دمج Redis في خدمة الخلاصات (Feed Service) لتخزين المنشورات الحديثة للمستخدمين بألية فورية وسريعة الاسترجاع، ما يضمن تجربة استخدام سلسة وسريعة بدون الحاجة للوصول إلى قاعدة بيانات تقليدية في كل مرة. تقوم الخدمة

بتمييز المستخدمين إلى نوعين: مؤثرين (Influencers) وعاديين (Regular Users). بالنسبة للمؤثرين، يتم تخزين منشوراتهم ضمن قائمة Redis مخصصة influencer:feed:{userId}، بحيث يمكن للمستخدمين المتابعين هؤلاء المؤثرين استعراض منشوراتهم مباشرة عبر Redis دون تحميل على قواعد البيانات.

أما بالنسبة للمستخدمين العاديين، فعند نشرهم لمنشور جديد، يتم إرسال هذا المنشور إلى Redis ضمن قائمة لكل متابع feed:{followerId}، مما يضمن وصولاً سريعاً ومباشراً للمنشورات في لحظة ظهورها دون تأخير. يتم الاحتفاظ بحد أقصى 50 منشوراً لكل مستخدم أو مؤثر، مما يحافظ على حجم الذاكرة المستخدمة ويضمن عرض أحدث المحتويات فقط.

من الناحية البرمجية، يتم التعامل مع Redis باستخدام StringRedisTemplate من Spring Boot، ويُحفظ كل منشور على شكل سلسلة JSON، ما يسهل تخزينه واسترجاعه بمرونة.

إن استخدام Redis في هذا السياق يعكس فهماً لمبادئ التصميم عالي الأداء (High Performance Architecture)، ويساهم بشكل جوهري في تحقيق متطلبات الاستجابة السريعة والتجربة التفاعلية للمستخدمين، خاصة ضمن نظام قائم على الأحداث (Event-Driven) ويحتوي على كمية كبيرة من المحتوى الديناميكي الذي يتغير لحظياً.

## 5.2.5 نظام تخزين كائني (Object Storage) – MinIO

MinIO هو نظام تخزين كائنات (Object Storage System) حديث ومفتوح المصدر، يتميز بأداء عالٍ وقابلية للتوسع، ويُعدّ خياراً مثالياً لتخزين البيانات غير المهيكلة مثل الصور، الفيديوها، الوثائق، وملفات النسخ الاحتياطي. تم تصميم MinIO وفق معمارية موزعة (Distributed Architecture) تدعم التكامل مع بيئات الحوسبة السحابية أو المحلية، ويعتمد على بروتوكول Amazon S3 API، مما يتيح للمطورين استخدام نفس أدوات وخبرات AWS S3 لكن داخل بيئة محلية مرنة وآمنة.

### ❖ بنية تخزين الكائنات: Object Keys و Buckets

يستخدم MinIO نموذجاً مبنياً على ال Buckets، حيث يُعدّ ال Bucket بمثابة الحاوية المنطقية الأعلى لتنظيم الكائنات ضمن النظام. يُنشئ كل مستخدم أو خدمة Bucket خاصاً به أو بها لتجميع الملفات المرتبطة ببعضها. وداخل هذا ال Bucket، تُخزن الكائنات باستخدام مفتاح فريد يُعرف بـ Object Key، وهو يشير إلى الملف داخل ال Bucket، ويمكن أن يتضمن بنية تراتبية ظاهرية باستخدام رموز مثل / لتقليد المجلدات التقليدية.

### ❖ استخدام الروابط الموقعة مسبقاً (Presigned URLs)

واحدة من أبرز خصائص MinIO - التي تم استغلالها بفعالية ضمن المشروع - هي الروابط الموقعة مسبقاً ( Presigned URLs)، والتي تسمح بالوصول المؤقت إلى كائن معين داخل Bucket دون الحاجة إلى تمرير مفاتيح الوصول السرية ( Access Keys) إلى المستخدم أو العميل.

يتم توليد هذه الروابط عبر مكتبات S3/MinIO API باستخدام مفاتيح الوصول الخاصة بالخادم، وتحتوي على توقيع رقمي وفترة صلاحية محددة. الرابط الناتج يمكن استخدامه لرفع ملف جديد مباشرة من الواجهة الأمامية إلى MinIO.

هذا النموذج يُعتبر آمناً ومنطقياً للأسباب التالية:

- عزل بيانات المصادقة: لا يتم كشف مفاتيح الوصول السرية للعميل.
- صلاحية مؤقتة: الروابط تنتهي صلاحيتها تلقائياً خلال دقائق.
- قابلية التقييد: يمكن تحديد نوع العملية (قراءة أو كتابة) بدقة.
- الفرق بين استخدام Presigned URL وعدمه

عند بناء أنظمة تعتمد على رفع ملفات كبيرة من قبل المستخدمين - مثل الصور والفيديوهات في المنصات الاجتماعية - تظهر الحاجة إلى آلية فعالة وآمنة لتخزين هذه البيانات. واحدة من أبرز هذه الآليات هي استخدام ما يُعرف بـ الرابط الموقع مسبقاً ( Presigned URL).

استخدام الروابط الموقعة يمنح العميل (المستخدم) صلاحية مؤقتة ومحددة للوصول إلى كائن معين داخل خدمة التخزين (مثل MinIO)، دون الحاجة لكشف بيانات المصادقة أو منح صلاحيات دائمة. تقوم الخدمة الخلفية (backend) بتوليد هذا الرابط باستخدام مفاتيح الوصول الخاصة، وتُضمّنه توقيعاً رقمياً يحدد نوع العملية (مثل الرفع أو التحميل)، وصلاحياتها الزمنية. بعد ذلك، يُرسل الرابط إلى الواجهة الأمامية التي تستخدمه مباشرة للتواصل مع خادم التخزين.

هذا النموذج يوفر عدة مزايا مهمة: أولاً، يعزز الأمان من خلال عزل مفاتيح الوصول عن المستخدمين، وثانياً، يخفف الضغط عن الخادم الخلفي، حيث لا يتم تمرير الملفات عبره، بل تُرسل مباشرة إلى MinIO، مما يقلل من استهلاك الموارد (RAM, CPU, Bandwidth). كما يتيح إمكانية التوسع الأفقي، حيث يمكن استقبال آلاف عمليات الرفع بالتوازي دون عنق زجاجة في المعالجة.

في المقابل، عدم استخدام Presigned URLs يعني أن المستخدم سيرسل الملف إلى الخادم أولاً، والذي بدوره يرفع الملف إلى MinIO. هذه الطريقة تُضعف الأداء، وتزيد من زمن الاستجابة، وتفرض حملاً مضاعفاً على الخوادم، خاصة في الأنظمة كثيفة الاستخدام.

❖ استخدام MinIO ضمن هذا المشروع

ضمن مشروع المنصة الاجتماعية لمشاركة الصور، كان من الضروري تضمين آلية موثوقة وفعالة لتخزين الوسائط المتعددة، وبشكل خاص الصور التي يرفعها المستخدمون ضمن حساباتهم الشخصية أو منشوراتهم. وبما أن هذه الملفات تُعتبر بيانات غير مهيكلة (Unstructured Data)، فإن استخدام نظام تخزين كائني (Object Storage) مثل MinIO يُعد خياراً مثالياً من الناحية التقنية والهندسية.

### آلية العمل:

عند قيام المستخدم برفع صورة – سواءً ضمن ملفه الشخصي أو كجزء من منشور – يتم أولاً توليد اسم فريد للكائن يُستخدم كمفتاح داخل MinIO، وعادةً ما يتضمن معرف المستخدم أو توقيت الرفع لتفادي التكرار. بعد ذلك، تقوم الخدمة الخلفية (مثل user-service أو post-service) بتوليد رابط موقع مسبقاً (Presigned URL) باستخدام مكتبة MinIO SDK، حيث يُحدد في هذا الرابط:

- اسم الـ Bucket الهدف.
- اسم الكائن المطلوب رفعه.
- نوع العملية (عادةً PUT).
- زمن الصلاحية (مثل 10 دقائق).

يُعاد هذا الرابط إلى واجهة المستخدم (React Frontend)، والتي تستخدمه لرفع الملف مباشرة إلى MinIO دون المرور بالخادم. وبعد رفع الصورة بنجاح، تقوم الخدمة بتوليد رابط ثابت ودائم (Static Object URL) يمثل الموقع النهائي للكائن داخل MinIO. يتم تخزين هذا الرابط في قاعدة البيانات المرتبطة بالمستخدم أو المنشور. هذا الرابط هو الذي يُستخدم لاحقاً لاسترجاع الصورة عند عرض الحساب الشخصي، الخلاصة، أو أي واجهة تعتمد على وسائط مرئية.

## 6.2.5 قاعدة البيانات العلائقية (MySQL – Relational Database)

MySQL هو نظام لإدارة قواعد البيانات العلائقية – (Relational Database Management System) RDBMS مبني على لغة (Structured Query Language) SQL، ويُعد من أكثر الأنظمة استخداماً عالمياً في التطبيقات المؤسسية والسحابية. يتميز MySQL بقدرته على تخزين البيانات المنظمة ضمن جداول مترابطة، تدعم الفهارس (indexes)، العلاقات (joins)، وسلامة البيانات (data integrity) من خلال القيود مثل المفاتيح الأساسية (Primary Keys) والخارجية (Foreign Keys).

نلجأ إلى MySQL عندما نحتاج إلى:

- تخزين بيانات هيكلية مترابطة (Structured Relational Data) ، مثل المنشورات، التعليقات، أو العمليات المالية.
- إجراء استعلامات معقدة على البيانات (تصفية، ترتيب، تجميع)
- الحفاظ على اتساق البيانات في بيئة متعددة المستخدمين (multi-user).
- دعم العمليات الذرية والمعاملات الموثوقة (Transactions with Rollback).
- التكامل مع Hibernate و JPA ضمن تطبيقات Spring Boot.

في مشروعنا، تم تخصيص MySQL لتخزين المنشورات التي ينشرها المستخدمون، وذلك ضمن خدمة مستقلة تدعى post-service. وتم اتخاذ هذا القرار لضمان:

- سرعة المعالجة والاسترجاع.
- قابلية التوسع الرأسي (scalability) باستخدام الفهارس.
- دعم الاستعلامات المتقدمة عبر Pageable لاسترجاع منشورات المستخدمين بطريقة مقسّمة (pagination).
- الحفاظ على تكامل البيانات مع Kafka لإرسال أحداث post.created لباقي الخدمات.

## 7.2.5 محرك البحث النصي-Elasticsearch

يُعد Elasticsearch نظاماً مفتوح المصدر مبنياً على Apache Lucene، ويُستخدم كمحرك بحث وفهرسة قوي يعتمد على البنية الموزعة (Distributed Architecture). يتميز بسرعته العالية، ومرونته في التعامل مع البيانات شبه المنظمة، مما يجعله مثالياً لتنفيذ عمليات البحث المتقدمة والتحليل اللحظي (Real-time analytics).

عوضاً عن استخدام قواعد البيانات العلائقية (Relational Databases) لعمليات البحث المعقدة، والتي تكون غالباً بطيئة وغير فعالة في نطاق واسع، يعتمد Elasticsearch على هيكل يُعرف بالمؤشرات (Indices) التي تُخزّن داخلياً على شكل وثائق (Documents) بصيغة JSON، ويتم تنظيمها وفق مخطط (Mapping) يشبه مفهوم الجداول ولكن بمرونة أعلى.

من الناحية التقنية، يعتمد Elasticsearch على مبدأ تحليل النصوص (Text Analysis) باستخدام Tokenizers و Analyzers لتجزئة النصوص وتخزينها بأسلوب يتيح مطابقة مصطلحات البحث مع الكلمات ذات الصلة، مع دعم خاصيات مثل الترتيب بالأهمية (Relevance Ranking)، البحث الغامض (Fuzzy Search)، وتصحيح الأخطاء (Spelling Correction).

كما يتميز Elasticsearch بقدرته على التوسّع الأفقي (Horizontal Scaling) من خلال توزيع البيانات على عدّة عقد (Nodes) ضمن تجمّع (Cluster)، مما يسمح بتحمّل الضغط العالي وتوفير زمن استجابة منخفض حتى مع ملايين السجلات.

في إطار هذا العمل، كان من الضروري توفير تجربة بحث فعالة وسريعة تُمكن المستخدم من العثور على مستخدمين آخرين بسهولة من خلال الاسم الأول. لتحقيق هذا الهدف، تم توظيف Elasticsearch كمحرك بحث مستقل مخصص لهذا النوع من العمليات.

تم فصل وظيفة البحث ضمن خدمة خاصة (Search Service) تتكامل مع بقية مكونات النظام عبر الأحداث، حيث يتم تلقي بيانات المستخدمين من خلال رسائل يتم بثها عند إنشاء حساب جديد أو تعديل المعلومات. بعد ذلك، تُفهرس هذه البيانات تلقائياً في قاعدة بيانات Elasticsearch، مما يسمح بإجراء عمليات بحث لحظية دقيقة وفعالة.

## NGINX 8.2.5 كطبقة توزيع محتوى (CDN Layer)

يعتبر NGINX من أشهر خوادم الويب مفتوحة المصدر وأكثرها كفاءة وموثوقية. وهو مصمم ليؤدي عدة أدوار محورية في البنية التحتية الرقمية الحديثة، من بينها [19] :

### 1. خادم ويب (Web Server)

يُستخدم لتقديم الملفات الثابتة مثل الصور، الفيديوها، ملفات HTML و CSS و JS مباشرةً للمستخدمين عبر البروتوكولات HTTP/HTTPS، مع كفاءة عالية في إدارة الاتصالات المتزامنة.

### 2. وكيل عكسي (Reverse Proxy)

يقوم NGINX باستقبال طلبات المستخدمين وتوجيهها إلى خوادم التطبيقات الخلفية (Backends)، ومن ثم يعيد الاستجابة إلى العميل. هذا يعزز من الأمان، ويوفر تحكماً مركزياً في حركة المرور.

### 3. بوابة توزيع المحتوى (CDN Gateway)

NGINX قادر على تخزين الملفات الثابتة مؤقتاً (caching) ضمن طبقاته الوسيطة، ما يُمكنه من تسريع عملية تحميل الملفات وتقليل الضغط على خوادم التخزين أو التطبيقات.

### 4. موازن حمل (Load Balancer)

يمكن استخدامه لتوزيع الطلبات بين عدة نسخ من الخدمة (instances) بهدف تحقيق التوسع الأفقي (horizontal scaling) وضمان التوافر العالي.

### 5. إدارة الكاش (Caching Proxy)

يتملك NGINX إمكانيات قوية للتخزين المؤقت (cache)، تُمكنه من تقديم الاستجابات بسرعة كبيرة إذا كانت محفوظة سابقاً، مما يقلل من زمن الاستجابة ويزيد من الأداء الكلي.

في سياق منصة اجتماعية تعتمد بشكل أساسي على الصور، سواء في ملفات المستخدمين أو منشوراتهم، كان من الضروري تصميم آلية فعالة لتحميل وعرض الصور تُراعي الأداء وسرعة الاستجابة. ومن هذا المنطلق، تم توظيف NGINX كعنصر وسيط (Intermediate Layer) بين Gateway و MinIO لتقديم الصور بكفاءة عالية، وتخفيف العبء عن خادم التخزين الأساسي.

عند طلب صورة (مثل صورة الحساب أو صورة منشور)، لا يتم التواصل مباشرة مع MinIO. بدلاً من ذلك، يقوم Gateway بتوجيه هذا الطلب إلى خادم NGINX، عبر عنوان مخصص.

إذا كانت الصورة موجودة بالفعل في الكاش المحلي لـ NGINX (أي تم طلبها مسبقاً):

- يتم تقديم الصورة مباشرة إلى Gateway ومنه إلى واجهة المستخدم.
- العملية تتم بسرعة كبيرة لأن الوصول إلى الكاش لا يتطلب زمن شبكة خارجي.

إذا كانت الصورة غير موجودة في الكاش:

- يقوم NGINX تلقائياً بطلبها من خادم MinIO عبر proxy\_pass.
- يتم تخزين الصورة تلقائياً في الكاش بعد تحميلها.
- يتم إرسال الصورة المطلوبة إلى Gateway وبالتالي إلى المستخدم.

إن اعتماد هذا النموذج حسن زمن الاستجابة ليصل إلى أقل من 200 مللي ثانية، وخفف الضغط على خوادم MinIO، وأتاح تطبيق سياسات معالجة الصور ديناميكياً دون التأثير على منطق النظام الأساسي.

## Docker 9.2.5

Docker هو نظام مفتوح المصدر يُستخدم لإنشاء، ونشر، وتشغيل التطبيقات داخل حاويات (Containers)، والتي تمثل بيئة تشغيل معزولة تحتوي على كل ما تحتاجه الخدمة: من كود، ومكتبات، وإعدادات تشغيل، مما يضمن عمل التطبيق بنفس الشكل في جميع البيئات (الاختبار، التطوير، الإنتاج) دون اعتماد على النظام المضيف.

بحسب الوثائق الرسمية لـ Docker [20] ، والدراسة الأكاديمية التي قدّمها Merkel عام 2014 [21]، فإن Docker يُعد طبقة تجريدية فوق نواة نظام التشغيل (OS-level Virtualization)، ما يجعله أخف بكثير من الأنظمة الافتراضية التقليدية (Virtual Machines)، ويُمكنه تشغيل مئات الحاويات على نفس الجهاز دون استنزاف الموارد.

❖ أبرز مفاهيم Docker التي تم اعتمادها:

1. Docker Image: تمثل وصفاً ثابتاً للخدمة، يشمل نظام التشغيل، التطبيق، وكل اعتماداته.



2. Docker Container: نسخة حية من ال Image، يتم تشغيلها بشكل معزول وتفاعلي.
3. Docker Compose: أداة لإدارة عدة حاويات مرتبطة معاً (مثلاً: قاعدة بيانات + تطبيق).
4. Docker Registry: مكان مركزي لتخزين الصور الجاهزة وتوزيعها (مثل Docker Hub أو private registry).

#### ❖ توظيف Docker في العمل

اعتمد مشروع المنصة الاجتماعية على Docker كركيزة أساسية لتغليف وتشغيل الخدمات المصغرة (Microservices) في بيئات معزولة وخفيفة. تم إنشاء صورة مستقلة لكل خدمة (مثل خدمة المستخدم، المصادقة، المنشورات...) تحتوي على كافة مكوناتها: الكود، إعدادات التشغيل، والاعتمادات البرمجية اللازمة.

كما تم تشغيل قواعد البيانات والأنظمة الداعمة مثل MongoDB، MySQL، Redis، Kafka، MinIO، Elasticsearch، Nginx داخل حاويات منفصلة، مما سهّل إدارة البيئة وتوحيدها عبر مختلف مراحل التطوير والإنتاج. خلال التطوير المحلي، تم استخدام docker-compose لتشغيل الحزمة الكاملة من الخدمات والبنية التحتية كوحدة واحدة متكاملة، مما أتاح تجربة النظام بشكل كامل على أي جهاز دون إعدادات معقدة.

#### ❖ مبررات اعتماد Docker في هذا المشروع

تتبع الحاجة إلى Docker من طبيعة النظام الموزّع الذي يتطلب توافقاً ومرونة عالية بين الخدمات، ويمكن تلخيص الفوائد الرئيسية كما يلي:

1. تناسق البيئات: يضمن Docker تشغيل نفس الكود بنفس البيئة على جميع المستويات (المطور - الاختبار - الإنتاج)، مما يقلل من أخطاء "يعمل على جهازي فقط".
2. العزل والمرونة: يسمح بتشغيل كل خدمة بشكل مستقل عن الأخرى، مما يبسط عمليات التوسعة، الصيانة، واستبدال الخدمات دون التأثير على بقية النظام.
3. التكامل مع Kubernetes: سهّل Docker نشر الحاويات داخل العنقود (cluster) وإدارتها.
4. قابلية إعادة الإنتاج: يمكن إعادة إنشاء أي بيئة بنفس الدقة والاعتماد على نفس الصور الموثقة مسبقاً، مما يُفيد في اختبار الأخطاء أو مراجعة حالات سابقة.

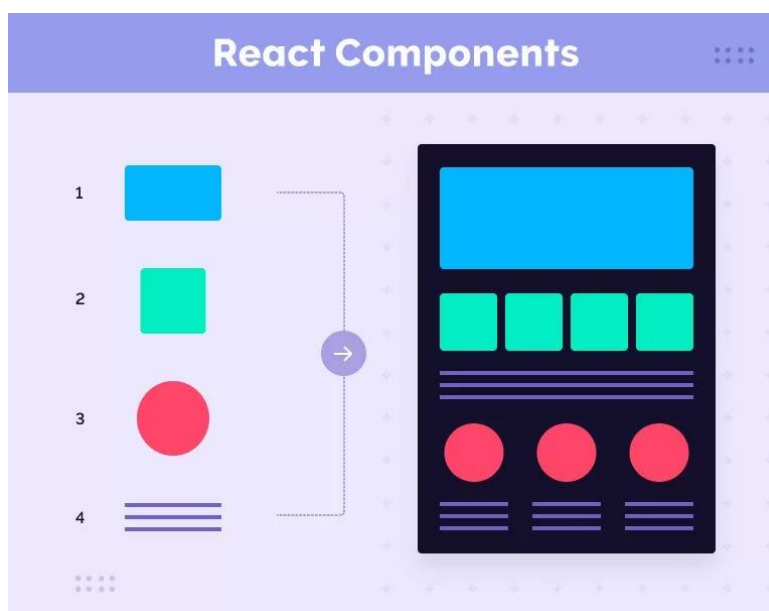
## React 10.2.5

React.js مكتبة مفتوحة المصدر لتطوير واجهات المستخدم (User Interfaces) باستخدام لغة JavaScript. تم تطويرها من قبل شركة Meta، وهي تركز على بناء واجهات مرنة، تفاعلية، وقابلة لإعادة الاستخدام بكفاءة عالية. تعتمد React على مبدأ

المكوّنات (Components) ، حيث يتم تقسيم واجهة المستخدم إلى وحدات مستقلة، كل منها يعالج منطق العرض الخاص به ويمكن دمجها مع باقي المكونات بسهولة.

تعتمد React على ما يُعرف بـ DOM الافتراضي (Virtual DOM) ، وهي نسخة خفيفة من DOM الحقيقي تُستخدم للمقارنة بين الحالة القديمة والجديدة للواجهة (Diffing Algorithm) ، مما يقلل من التحديثات غير الضرورية ويزيد من سرعة التفاعل وتجربة المستخدم. كما تستخدم React تدفق بيانات أحادي الاتجاه (Unidirectional Data Flow) ، حيث تنتقل البيانات من الأعلى للأسفل (من المكوّن الأب إلى الأبناء)، مما يُبسّط عملية تتبع الأخطاء وسلوك التطبيق.

تُتيح React أيضاً استخدام Hooks مثل useState, useEffect, و useContext لإدارة الحالة (State) داخل المكونات الوظيفية بدون الحاجة لاستخدام Classes، مما يجعل كتابة الكود أبسط وأكثر وضوحاً.



الشكل 29: مكوّنات (React Components) React

تم استخدام React لتطوير الواجهة الأمامية للمنصة الاجتماعية لمشاركة الصور، وتمّ تصميم بنية الكود بأسلوب منظم يراعي مبدأ الفصل بين العرض والمنطق، كما يلي:

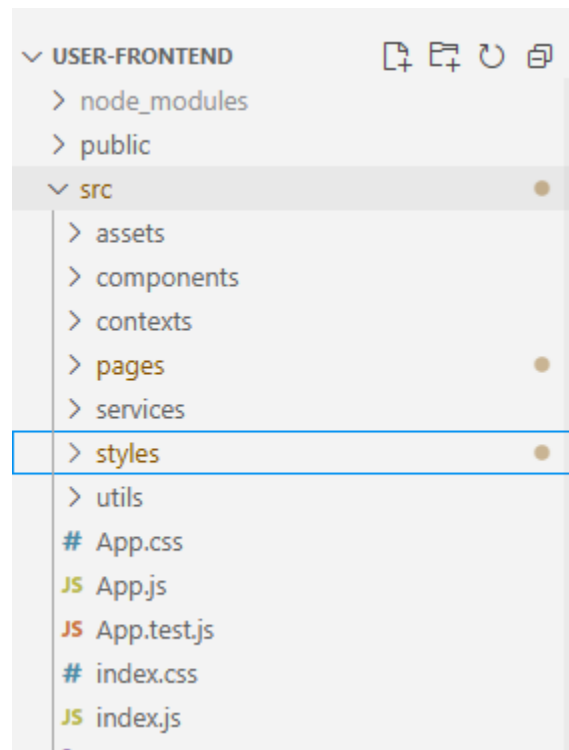
■ تنظيم الملفات والمجلدات:

1. components/: يحتوي على جميع المكونات القابلة لإعادة الاستخدام

2. pages/: يحتوي على الصفحات الكاملة مثل:

SignupPage.jsx, LoginPage.jsx

3. `/services`: تضم الخدمات التي تتواصل مع الـ `backend`.
4. `/contexts`: يضم ملف `UserContext.js` الذي يُستخدم لتخزين معلومات المستخدم الحالي بمجرد تسجيل الدخول، ويتيح الوصول إليه من جميع أجزاء التطبيق.
5. `/styles`: يحتوي على ملفات `CSS` منفصلة لكل مكون وصفحة للحفاظ على نظافة الهيكليّة.
6. `/utils`: يحتوي على أدوات مساعدة مثل:
  - `imageCompressor.js`: لضغط الصور قبل رفعها إلى `MinIO`.
  - `timeAgo.js`: لعرض الوقت بطريقة بشرية (مثل "قبل 3 دقائق").

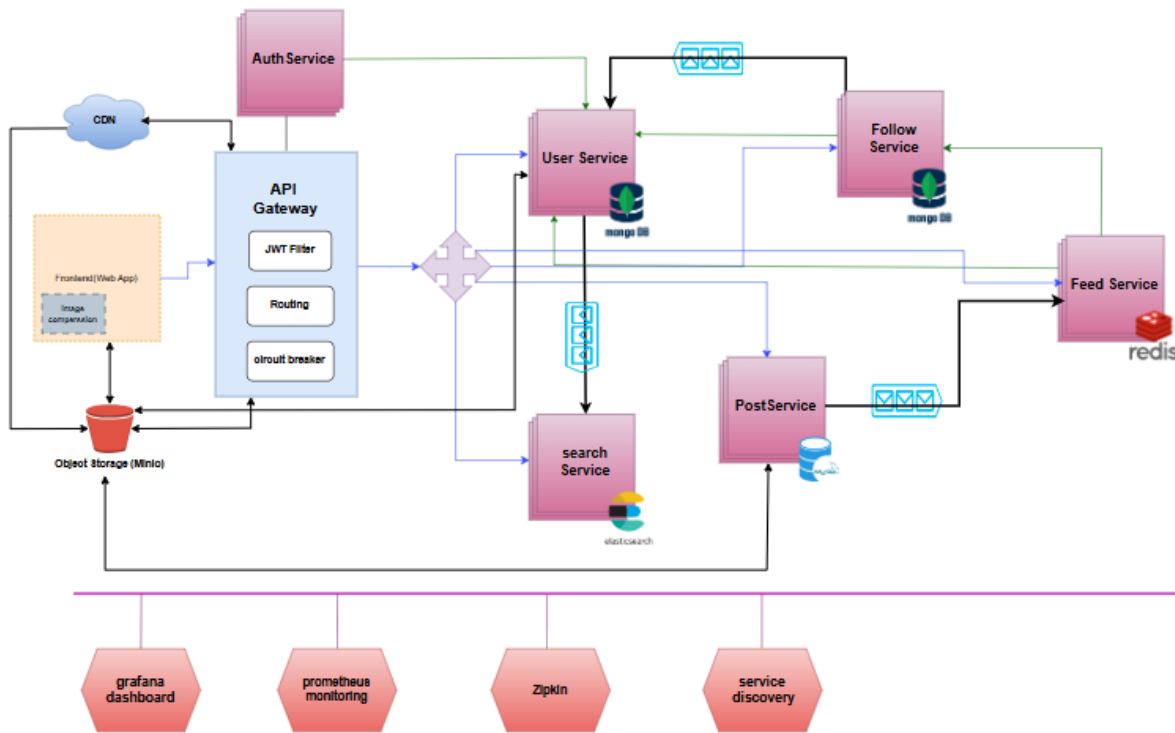


الشكل 30: بنية مجلدات مشروع *React*

## 3.5 البنية العامة للنظام

بعد تحليل المتطلبات واختيار الأنماط المعمارية والأدوات المناسبة بما في ذلك بنية الخدمات المصغرة (Microservices)، والاهتمام بالتوسّع الأفقي، المصادقة باستخدام JWT، التوجيه المركزي عبر API Gateway بحيث تكون نقطة الدخول الوحيدة، والفصل بين الواجهة الأمامية والخلفية. تمّ توظيف جميع المفاهيم للوصول إلى البنية النهائية .

يوضح الشكل (31) بنية النظام



الشكل 31: تصميم النظام

## 4.5 آلية بناء الخدمات وتكاملها ضمن النظام

### 1.4.5 إنشاء مشروع spring boot

بعد تحديد التصميم للنظام وتوزيع المسؤوليات على شكل خدمات مصغرة (Microservices)، تم الانتقال إلى مرحلة البناء العملي لتجهيز هذه الخدمات، وذلك من خلال اعتماد أدوات وتكنولوجيات حديثة مثل Spring Boot و Spring Cloud، والتي تتيح إنشاء خدمات مستقلة، قابلة للنشر، وسهلة التكامل، وتدعم التوسع الأفقي وإعادة التشغيل التلقائي في بيئة موزعة.

في هذا السياق، قمنا بتعريف كل خدمة كوحدة قائمة بذاتها تمتلك قاعدة بياناتها الخاصة، ومسؤولياتها المنعزلة، وتُقدّم للمستخدمين أو للخدمات الأخرى عبر واجهات REST واضحة. وقد تم اختيار هذا الأسلوب تماشياً مع متطلبات قابلية الصيانة، العزل، والتوسع، حيث يُسهّم في تقليل الترابط المباشر بين الخدمات (Loose Coupling)، ويُسهّل عمليات النشر المستقل لكل خدمة دون التأثير على باقي النظام.

تم بناء كل خدمة باستخدام مشروع Spring Boot REST API وذلك من خلال الموقع <https://start.spring.io>

كما هو موضح في الشكل، تم تهيئة الحقول ضمن Spring Initializr بما يتوافق مع متطلبات خدمة المستخدم (user-service)، حيث تم اختيار Maven و Java 17 لبيئة العمل، وتحديد اسم الحزمة والتنظيم المناسبين لتمييز كل خدمة عن الأخرى. تم أيضاً إدراج التبعية الأساسية لإنشاء REST API والتكامل مع MongoDB، مع دعم أدوات المراقبة والتحسين أثناء التطوير.

حيث هذا الموقع يقوم بتوليد ملف pom.xml يحوي على جميع الاعتماديات التي قمنا بإضافتها

<b>Project</b> <input type="radio"/> Gradle - Groovy <input type="radio"/> Gradle - Kotlin <input checked="" type="radio"/> Maven <b>Spring Boot</b> <input type="radio"/> 4.0.0 (SNAPSHOT) <input type="radio"/> 3.5.4 (SNAPSHOT) <input checked="" type="radio"/> 3.5.3 <input type="radio"/> 3.4.8 (SNAPSHOT) <input type="radio"/> 3.4.7 <b>Project Metadata</b> Group: com.example Artifact: user-service Name: user-service Description: Microservice responsible for managing user accounts Package name: com.example.user-service Packaging: <input checked="" type="radio"/> Jar Java: <input type="radio"/> 24 <input type="radio"/> 21 <input checked="" type="radio"/> 17	<b>Language</b> <input checked="" type="radio"/> Java <input type="radio"/> Kotlin <input type="radio"/> Groovy <b>Dependencies</b> <b>Spring Web</b> WEB Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container. <b>MongoDB Atlas Vector Database</b> AI Spring AI vector database support for MongoDB Atlas. Is a fully managed cloud database service that provides an easy way to deploy, operate, and scale a MongoDB database in the cloud. <b>Spring Boot DevTools</b> DEVELOPER TOOLS Provides fast application restarts, LiveReload, and configurations for enhanced development experience. <b>codecentric's Spring Boot Admin (Server)</b> OPS A community project to manage and monitor your Spring Boot applications. Provides a UI on top of the Spring Boot Actuator endpoints.
--	---

الشكل 32: إعداد مشروع Spring Boot باستخدام Spring Initializr

## 2.4.5 تنظيم هيكل المشروع لكل خدمة

بعد تحميل مشروع Spring Boot الناتج من Spring Initializr وفتحه داخل بيئة التطوير (IDE) مثل IntelliJ IDEA أو VS Code ، يظهر المشروع بهيكل أولي يتكوّن من مجلدات وملفات أساسية تُمثّل نقطة الانطلاق لتطوير الخدمة.

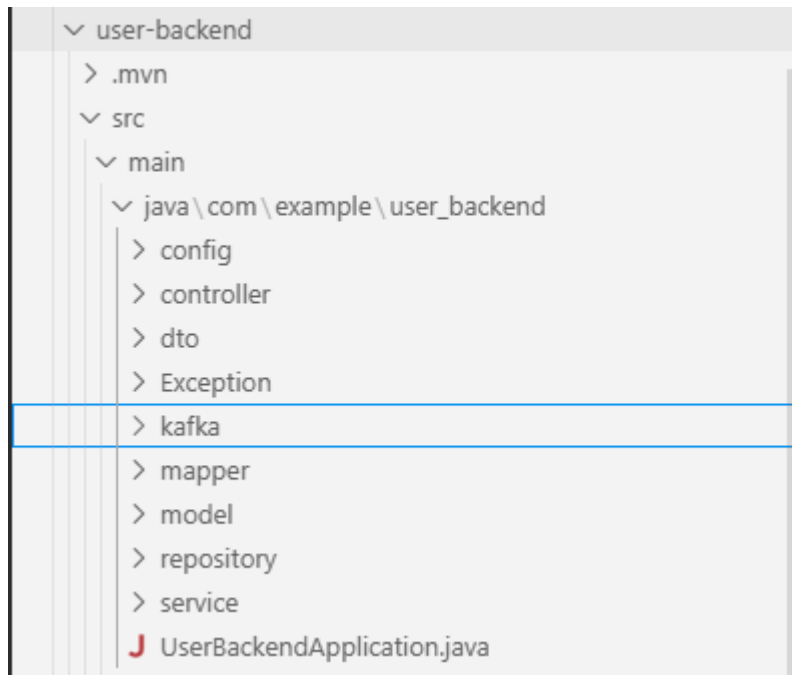
يتكوّن هيكل المشروع من جزأين رئيسيين:

1. المسار البرمجي الأساسي src/main/java

يحتوي على حزمة المشروع (مثل com.example.user-service) والتي يتم ضمنها بناء الهيكل المنطقي للخدمة.

قمنا بتنظيم هذا المسار داخلياً إلى مجلدات وفقاً لمبدأ الفصل بين الطبقات، كالتالي:

- controller: يحتوي على وحدات التحكم (Controllers) المسؤولة عن استقبال ومعالجة طلبات HTTP.
- service: يحتوي على منطق الأعمال (Business Logic) الذي يُعالج العمليات المطلوبة قبل التواصل مع البيانات.
- repository: يحتوي على واجهات الوصول إلى قواعد البيانات، ويستخدم MongoDB أو JpaRepository.
- model: يحتوي على الكيانات (Entities أو Documents) التي تُمثّل بنية البيانات.
- dto: يُستخدم لتغليف البيانات بين الطبقات، ويعزّز العزل والأمان.
- mapper: يحتوي على أدوات التحويل بين الكيانات و DTO باستخدام Mapper.
- config: يحتوي على تهيئة خاصة بالخدمة مثل تكوين CORS أو Beans خاصة.



الشكل 3.3: هيكل المشروع للخدمات

تمّ فصل منطق Kafka في مجلد خاص لأنه يُمثّل قناة اتصال خارجية غير متزامنة (Asynchronous Integration)، ويُسهّل عزل التواصل بين الخدمات.

## 2. المسار src/main/resources

يتضمن الملفات الثابتة وملفات الإعدادات، تمّ تنظيم ملفات الإعدادات بما يراعي التبديل السلس بين بيئات التطوير والتشغيل (Development & Production)، وذلك من خلال إنشاء ملفات إعدادات مستقلة لكل بيئة:

- application.properties: يُمثّل الملف الأساسي الذي يُقرأ أولاً ويحتوي عادةً على الإعدادات المشتركة بين جميع البيئات، مثل اسم الخدمة أو إعدادات الأمن العامة.
- application-dev.properties: يحتوي على إعدادات مخصصة لبيئة التطوير (development)
- application-prod.properties: يُستخدم في بيئة الإنتاج (production)، ويحتوي على إعدادات حقيقية موجهة نحو الخوادم الموجودة .

بعد تنظيم هيكل المشروع ضمن المسار src/main/java وفقاً لمبدأ الفصل بين الطبقات، تم بناء كل طبقة بحيث تؤدي دوراً محدداً ضمن سلسلة معالجة البيانات داخل الخدمة. وفيما يلي شرح لكل طبقة من هذه الطبقات مع أمثلة عملية مأخوذة من خدمة user-service التي تُدير حسابات المستخدمين ضمن المنصة:

## 1. طبقة التحكم (Controller)

تمثل هذه الطبقة نقطة التفاعل الأولى بين العميل (المستخدم أو خدمة أخرى) والخدمة. تقوم باستقبال الطلبات الواردة عبر HTTP، وتمررها إلى منطق الأعمال المناسب في طبقة service. كما تقوم بإرجاع الاستجابات على شكل JSON مهيكل.

في الشكل (34) نلاحظ أن الطبقة لا تحتوي على أي منطق خاص بالتحقق أو التخزين، وإنما توجّه البيانات فقط، مما يضمن الالتزام بمبدأ Single Responsibility.

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    private Tracer tracer;
    private static final Logger logger = LoggerFactory.getLogger(UserController.class);

    private final UserService userService;
    private final UserRepository userRepository;
    private final UserMapper userMapper;

    public UserController(UserService userService, UserRepository userRepository, UserMapper userMapper) { ...

    @PostMapping("/signup")
    public ResponseEntity<UserResponseDTO> createUser(@Valid @RequestBody UserRegisterRequestDTO dto) { ...

    @PutMapping("/me")
    public ResponseEntity<UserResponseDTO> editProfile(...)

    @GetMapping("/me")
    public ResponseEntity<UserResponseDTO> getMyProfile(@RequestHeader("X-User-Id") String userId) { ...

    @GetMapping("/{id}")
    public ResponseEntity<UserResponseDTO> getUserById(@PathVariable String id) { ...

    @GetMapping("/{id}/type")
    public ResponseEntity<UserTypeResponseDTO> getUserType(@PathVariable String id) { ...

    @GetMapping("/email/{email}")
    public ResponseEntity<UserResponseDTO> getUserByEmail(@PathVariable String email) { ...
```

الشكل 34: طبقة التحكم Controller

## 2. طبقة منطق الأعمال (Service)



تحتوي هذه الطبقة على القواعد المنطقية التي تُحدد كيف تتم معالجة الطلب. وهي تُعد القلب الديناميكي للخدمة، حيث تقوم بالتنسيق بين repository, mapper, dto وغيرها.

```
@Service
public class UserService {

    private final UserRepository userRepository;
    private final KafkaEventPublisher eventPublisher;
    private final PasswordEncoder passwordEncoder;
    // private final ModelMapper modelMapper;
    private final UserMapper userMapper;
    private static final Logger logger = LoggerFactory.getLogger(UserService.class);

    // @Autowired
    private MongoTemplate mongoTemplate;

    > public UserService(UserRepository userRepository, KafkaEventPublisher eventPublisher, ...
    > public User createUser(UserRegisterRequestDTO dto) { ...
    > public User updateUser(String id, UserUpdateRequestDTO dto) { ...

    public List<UserEventDTO> getRandomUsers(int limit) {
        return userRepository.findRandomUsers(limit);
    }

}
```

الشكل 35: طبقة منطق الأعمال *service*

### 3. طبقة الوصول إلى البيانات (Repository)

تعتمد هذه الطبقة على Spring Data لتوفير عمليات CRUD (إنشاء، قراءة، تعديل، حذف) بدون الحاجة لكتابة كود يدوي. باستخدام واجهات JPA أو Mongo، يمكننا تنفيذ استعلامات مخصصة، مع ضمان تكامل قاعدة البيانات واتباع مبدأ Repository Abstraction.

```

@Repository
public interface UserRepository extends MongoRepository<User, String> {
    Optional<User> findByEmail(String email);

    List<User> findByFirstnameContainingIgnoreCase(String firstname);

    @Aggregation("{ $sample: { size: ?0 } }")
    List<UserEventDTO> findRandomUsers(int size);
}

```

الشكل 36: طبقة الوصول إلى البيانات *Repository*

#### 4. طبقة الكيانات (Model)

تحتوي هذه الطبقة على الكائنات التي تُخزن فعلياً في قاعدة البيانات. وهي تعكس الهيكل الداخلي للبيانات.

```

@Document(collection = "users")
public class User {

    @Id
    private String id;

    private String firstname;

    private String lastname;

    @Indexed(unique = true)
    private String email;

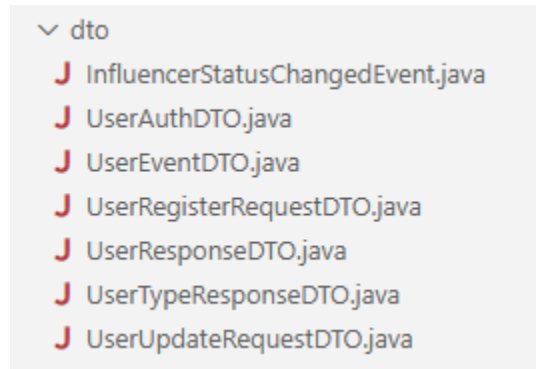
    private String password;
    private String imageUrl;
    private LocalDateTime createdAt;
    private String type = "regular";
    private String bio;
}

```

الشكل 37: طبقة الكيانات *Model*

## 5. طبقة DTO (Data Transfer Objects)

تُستخدم لعزل الكيانات الداخلية عن البيانات التي يتم إرسالها أو استقبالها عبر الشبكة. هذا يحسّن الأمان ويزيد من المرونة.



الشكل 38: طبقة DTO

لا يحتوي هذا الكائن على أي من الحقول الداخلية أو كلمات السر المشفرة، مما يُحافظ على خصوصية النموذج الداخلي.

## 6. طبقة المخرجات (Mapper)

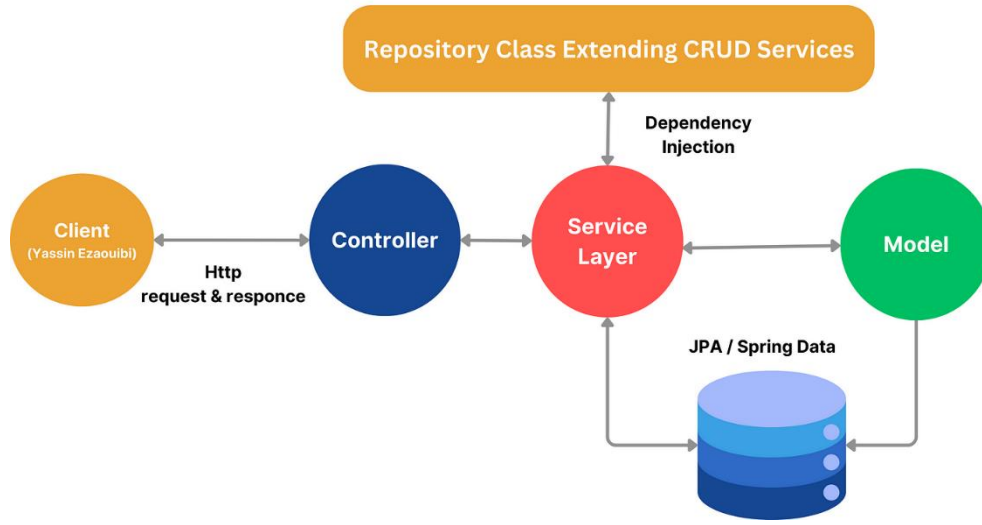
تقوم بتحويل الكيانات إلى DTO والعكس. يتم عادةً استخدام مكتبة ModelMapper.

```
@Configuration
public class MapperConfig {
    @Bean
    public ModelMapper modelMapper() {
        return new ModelMapper();
    }
}
```

الشكل 39: طبقة المخرجات (Mapper)

بهذا الأسلوب، يمكن بناء خدمات متماسكة هندسياً، منفصلة في المهام، سهلة الاختبار، ومرنة للتوسّع الأفقي. يتم تكرار هذا التنظيم في كل خدمة ضمن المنصة (follow-service, post-service, feed-service...) بما يتوافق مع منطق الخدمة التي ستقدمها.

يوضّح الشكل التالي التدفق العام داخل كل خدمة خلفية ضمن النظام، وفق بنية Spring Boot متعددة الطبقات.



الشكل 40: المعمارية الطبقة لتطبيق *Spring Boot* وتدفق البيانات بين الطبقات

## 5.5 خدمة البوابة Gateway

تم تطوير خدمة البوابة (Gateway Service) باستخدام Spring Cloud Gateway و Spring WebFlux، وذلك بهدف تأمين نقطة دخول مركزية موحدة لجميع الطلبات الواردة إلى المنصة. تعتمد البوابة على نمط API Gateway، حيث تعمل على توجيه الطلبات إلى الخدمات المصغرة المناسبة (مثل: user-service, auth-service, post-service...) وفقاً لمسارات محددة ضمن ملف application.yml.

تقوم البوابة بعدة مهام:

- التوجيه Routing: تقوم البوابة بتوجيه كل طلب إلى الخدمة المصغرة المناسبة اعتماداً على مسار الطلب.
- المصادقة والتحقق من الهوية (JWT Authentication): تُنفذ البوابة طبقة تحقق أمني باستخدام token(JWT)، بحيث يتم التحقق من صحة token واستنباط هوية المستخدم من خلاله، ثم يتم تمرير هذه الهوية بشكل شفاف إلى الخدمات الخلفية. هذا النموذج يُغني كل خدمة خلفية عن التحقق بنفسها من رمز المصادقة، ويحقق أماناً موزعاً.
- حماية الصور عبر Circuit Breaker: تمّ اعتماد آلية Circuit Breaker في خدمة البوابة لحماية النظام من تعطل مسار تحميل الصور عبر شبكة CDN. عند حدوث فشل متكرر في هذه الخدمة سواء بسبب بطء الاستجابة، أو تعذر الوصول المؤقت يقوم Circuit Breaker تلقائياً بقطع الاتصال مع خادم CDN، ويُعيد توجيه الطلب إلى مسار احتياطي fallback مُخصص يقوم من خلاله بتحميل الصور من نظام التخزين الكائني MinIO.

## تم ضبط الـ Circuit Breaker بحيث:

- إذا تجاوزت نسبة الفشل 50% من عدد معين من المحاولات، يتم فتح الدائرة.
  - بعد فترة زمنية محددة (مثلاً 10 ثوانٍ)، يتم اختبار عدد محدود من الطلبات للتحقق مما إذا كانت الخدمة قد تعافت.
  - إذا نجحت هذه الطلبات، يتم إغلاق الدائرة واستئناف المسار الأساسي (CDN).
- هذا السلوك يضمن أن المستخدم لا يلاحظ فشل الخدمة الأساسية، حيث تتم معالجة الطلب تلقائياً من مصدر احتياطي، مما يُعزز من الاستمرارية والمرونة ويمنع تعطل النظام ككل نتيجة لفشل نقطة واحدة فقط ضمن سلسلة الخدمات.
- بهدف التحقق من فعالية آلية الحماية من الأعطال (Circuit Breaker) المطبقة ضمن بوابة الدخول (API Gateway)، تم تنفيذ تجربة عملية تتضمن إيقاف خادم CDN مؤقتاً (وهو المصدر الأساسي لتحميل الصور). بعد ذلك، تم إعادة تحميل الصور ضمن الواجهة الأمامية لاختبار استجابة النظام في ظل غياب المصدر الرئيسي.
- وقد أظهرت النتائج نجاح العملية بشكل كامل، حيث تم تحميل الصور دون أي تأخير أو أخطاء مرئية للمستخدم، مما يؤكد أن الطلبات تم توجيهها تلقائياً إلى المسار البديل (Fallback Route)، والذي يعتمد على خادم داخلي (MinIO) ل جلب الصور المطلوبة.

نلاحظ من الشكل (41) يوجد حقل HIT :X-Cache الذي يدل على أن الصور تم تحميلها من CDN

X	Headers	Preview	Response	Initiator	Timing	Cookies
Access-Control-Allow-Origin	*					
Content-Length	57678					
Content-Type	image/webp					
Date	Wed, 30 Jul 2025 12:54:37 GMT					
Etag	"2477167483741b40efb1632f1078c6bf"					
Last-Modified	Mon, 28 Jul 2025 20:05:26 GMT					
Server	nginx/1.29.0					
Strict-Transport-Security	max-age=31536000; includeSubDomains					
Vary	Origin					
Vary	Access-Control-Request-Method					
Vary	Access-Control-Request-Headers					
Vary	Origin					
Vary	Accept-Encoding					
X-Amz-Id-2	dd9025bab4ad464b049177c95eb6ebf374d3b3fd1af9251148b658df7ac2e3e8					
X-Amz-Request-Id	1857089B1DB13309					
X-Cache	HIT					
X-Content-Type-Options	nosniff					
V DataLimit limit	1620					

الشكل 41: تحميل الصور مع وجود CDN

بينما فيما بعد قمنا بإيقاف CDN ولاحظنا أنه تم تحميل الصور بدون أي أخطاء تظهر للمستخدم حيث تم تفعيل circuitbreaker وتم تحميل الصورة من Minio

▼ Response Headers	<input type="checkbox"/> Raw
Accept-Ranges	bytes
Content-Length	57678
Content-Type	image/webp
Vary	Origin
Vary	Access-Control-Request-Method
Vary	Access-Control-Request-Headers

الشكل 42: تحميل الصورة بعد تعطيل CDN

## 6.5 تحليل أثر ضغط الصور على حجم البيانات وأداء النظام

تم التحقق من فعالية ضغط الصور من خلال تتبع الحجم الأصلي والحجم الناتج بعد الضغط باستخدام أدوات المتصفح في واجهة React. على سبيل المثال (انظر الأشكال)، تم ضغط صورة بحجم 4.4 ميغابايت إلى حوالي 996 كيلوبايت، وأخرى من 2.1 ميغابايت إلى 801 كيلوبايت، دون فقدان ملحوظ في الجودة.

كما تم حفظ الصور بصيغة webp وتحليل النتائج محلياً، مما أكد تقليل حجم البيانات المرسل بنسبة كبيرة. هذه التجربة دعمت فعالية القرار التقني المتخذ، وساهمت في تحسين سرعة الرفع وتجربة المستخدم، مع الحفاظ على بساطة المعمارية الخلفية.

Original image size: 2153.63 KB	<a href="#">UploadModal.jsx:16</a>
Converted to WebP. Size: 801.71 KB	<a href="#">imageCompressor.js:23</a>
Compressed image size: 801.71 KB	<a href="#">UploadModal.jsx:18</a>

الشكل 43: نتيجة ضغط صورة بحجم 4 ميغابايت

Original image size: 4441.63 KB	<a href="#">UploadModal.jsx:16</a>
Converted to WebP. Size: 996.69 KB	<a href="#">imageCompressor.js:23</a>
Compressed image size: 996.69 KB	<a href="#">UploadModal.jsx:18</a>

الشكل 44: نتيجة ضغط صورة بحجم 2 ميغابايت

<input type="checkbox"/>	d4a7ef1d-d399-4251-9cb2-c17836007eb3.webp	Today, 00:18	996.7 KiB
<input type="checkbox"/>	0b73e012-b4e4-428d-9184-d101e7da5826.webp	Today, 00:17	801.7 KiB

الشكل 45: حفظ الصور بعد الضغط ضمن الخادم

## 7.5 تسريع تحميل الصور وتقليل الضغط على MinIO باستخدام Nginx

تم تنفيذ خادم Nginx كذاكرة تخزين مؤقتة (CDN) أمام MinIO، مما أدى إلى نتائج عملية واضحة، أبرزها:

- تسريع زمن تحميل الصور بشكل ملحوظ، خاصة في الصفحات التي تحتوي على عدد كبير من الصور مثل الخلاصة والملف الشخصي.
- تقليل عدد الطلبات المباشرة إلى MinIO، مما حسن أداء الخادم في عمليات رفع الصور واستجابته العامة.
- تحسين تجربة المستخدم النهائية من خلال عرض الصور بسرعة أكبر وبدون تأخير.
- تحقيق بنية قابلة للتوسع تسمح بإضافة خوادم CDN أخرى أو دمج خدمات خارجية بسهولة لاحقاً.

توضح الصور التالية أثر هذا التحسين من خلال مقارنة زمن تحميل الصور قبل وبعد إدراج Nginx كوسيط توزيع.

Name	Headers	Preview	Response	Initiator	Timing
raselleLogo.dd405d7b5194fb8...			127.0.0.1/401e01a2b001102d5d77e4f4"		
d4805325-9ac2-4547-8c48-62...	Last-Modified		Wed, 16 Jul 2025 19:23:43 GMT		
0d3287bd-e3b2-4b09-96f9-83...	Server		nginx/1.29.0		
273b0378-6276-4ab1-9439-6...	Strict-Transport-Security		max-age=31536000; includeSubDomains		
58cda1b2-e20b-4f58-b95f-83...	Vary		Origin, Access-Control-Request-Method, Access-Control-Request-Headers, Origin, Accept-Encoding		
a3d23ac4-b234-40cd-a884-a3...	X-Amz-Id-2		dd9025bab4ad464b049177c95eb6ebf374d3b3fd1af9251148b658df7ac2e3e8		
c1e1e13a-bde7-4414-a44b-a4...	X-Amz-Request-Id		185368C9D71561D3		
1f36b456-3789-4ae6-95de-08...	X-Cache		MISS		
data:image/png;base...	X-Content-Type-Options		nosniff		
75de4469-349b-4574-b80a-3c...	X-Ratelimit-Limit		2925		
61ce3554-c2a8-4fcc-a225-8d9...	X-Ratelimit-Remaining		2925		
10ecc675-51f0-4720-a09e-8ca...	X-Xss-Protection		1; mode=block		
e5f7a965-2941-4dd9-abb9-bc...	Request Headers				
raselleLogo.dd405d7b5194fb8...					
d4805325-9ac2-4547-8c48-62...					
data:image/png;base...					
61ce3554-c2a8-4fcc-a225-8d9...					
75de4469-349b-4574-b80a-3c...					
10ecc675-51f0-4720-a09e-8ca...					
32 / 56 requests	8,556 kB / 8,571 kB				

Name	Headers	Preview	Response	Initiator	Timing
raselleLogo.dd405d7b5194fb8...					
d4805325-9ac2-4547-8c48-62...	Connection Start				DURATION
0d3287bd-e3b2-4b09-96f9-83...	Stalled				4.97 ms
273b0378-6276-4ab1-9439-6...	Proxy negotiation				0.80 ms
58cda1b2-e20b-4f58-b95f-83...	DNS Lookup				9 µs
a3d23ac4-b234-40cd-a884-a3...	Initial connection				37.03 ms
c1e1e13a-bde7-4414-a44b-a4...	Request/Response				DURATION
1f36b456-3789-4ae6-95de-08...	Request sent				0.16 ms
75de4469-349b-4574-b80a-3c...	Waiting for server response				64.35 ms
61ce3554-c2a8-4fcc-a225-8d9...	Content Download				157.59 ms
10ecc675-51f0-4720-a09e-8ca...					299.35 ms
e5f7a965-2941-4dd9-abb9-bc...	Explanation				
raselleLogo.dd405d7b5194fb8...					
d4805325-9ac2-4547-8c48-62...					
data:image/png;base...					
61ce3554-c2a8-4fcc-a225-8d9...	Server Timing				TIME
32 / 56 requests	8,556 kB / 8,571 kB				

الشكل 46: زمن تحميل الصور من minio

الشكل 47: عملية تحميل الصور من minio من قبل nginx



Name	Headers	Preview	Response	Initiator	Timing
data:image/png;base...			GMT		
61ce3554-c2a8-4fcc-a225-8d9...	Server		nginx/1.29.0		
75de4469-349b-4574-b80a-3c...	Strict-Transport-Security		max-age=31536000; includeSubDomains		
10ecc675-51f0-4720-a09e-8ca...	Vary		Origin, Access-Control-Request-Method, Access-Control-Request-Headers, Origin, Accept-Encoding		
e5f7a965-2941-4dd9-abb9-bc...	X-Amz-Id-2		dd9025bab4ad464b049177c95eb6bf374d3b3fd1af9251148b658df7ac2e3e8		
raselleLogo.dd405d7b5194fb8...	X-Amz-Request-Id		185268C0D71561D3		
d4805325-9ac2-4547-8c48-62...	X-Cache		HIT		
0d3287bd-e3b2-4b09-96f9-83...	X-Content-Type-Options		nosniff		
273b0378-6276-4ab1-9439-6...	X-Ratelimit-Limit		2925		
58cda1b2-e20b-4f58-b95f-83...	X-Ratelimit-Remaining		2925		
a3d23ac4-b234-40cd-a884-a3...	X-Xss-Protection		1; mode=block		
c1e1e13a-bde7-4414-a44b-a4...					
1f36b456-3789-4ae6-95de-08...					
75de4469-349b-4574-b80a-3c...					
e5f7a965-2941-4dd9-abb9-bc...					
10ecc675-51f0-4720-a09e-8ca...					
61ce3554-c2a8-4fcc-a225-8d9...					
32 / 56 requests	8,556 kB / 8,571 kB				

الشكل 49: تحميل الصور من cache ضمن cdn

Name	Headers	Preview	Response	Initiator	Timing
data:image/png;base...					
61ce3554-c2a8-4fcc-a225-8d9...					
75de4469-349b-4574-b80a-3c...					
10ecc675-51f0-4720-a09e-8ca...					
e5f7a965-2941-4dd9-abb9-bc...					
raselleLogo.dd405d7b5194fb8...					
d4805325-9ac2-4547-8c48-62...					
0d3287bd-e3b2-4b09-96f9-83...					
273b0378-6276-4ab1-9439-6...					
58cda1b2-e20b-4f58-b95f-83...					
a3d23ac4-b234-40cd-a884-a3...					
c1e1e13a-bde7-4414-a44b-a4...					
1f36b456-3789-4ae6-95de-08...					
75de4469-349b-4574-b80a-3c...					
e5f7a965-2941-4dd9-abb9-bc...					
10ecc675-51f0-4720-a09e-8ca...					
61ce3554-c2a8-4fcc-a225-8d9...					
32 / 56 requests	8,556 kB / 8,571 kB				

الشكل 48: زمن تحميل الصور من CDN

أثبت استخدام NGINX كخادم وسيط لتخزين الصور مؤقتاً (Cache Proxy) فعاليته في تحسين أداء النظام من حيث زمن الاستجابة وسرعة تحميل المحتوى. فعند إجراء اختبار تجريبي لمقارنة زمن تحميل إحدى الصور قبل وبعد تخزينها في الكاش، لوحظ أن الطلب الأول الذي يتم فيه جلب الصورة من خادم MinIO استغرق حوالي 299.35 ميلي ثانية، في حين أن الطلب الثاني لنفس الصورة – والذي تم التعامل معه من خلال كاش NGINX – استغرق فقط 87.47 ميلي ثانية.

يمثل هذا الفارق الزمني المكتسب، والمقدّر بأكثر من 211 ميلي ثانية لكل طلب، مؤشراً واضحاً على كفاءة التخزين المؤقت في تقليل زمن الانتظار، وتخفيف العبء عن خوادم التخزين الأساسية، لا سيما في التطبيقات التي تعتمد على تحميل عدد كبير من الملفات الثابتة مثل الصور. كما يعزز هذا الأسلوب من استقرار الخدمة ويقلل من التذبذب في أزمته الاستجابة، ما ينعكس إيجاباً على تجربة المستخدم وأداء النظام ككل.

يُظهر هذا القرار فهماً دقيقاً لتحديات نقل الوسائط في الأنظمة السحابية، كما أنه يُجسّد مبدأ تحسين الأداء عبر التوجيه الذكي للطلبات (Smart Proxying) مع الحفاظ على أمن الوصول باستخدام Presigned URLs، ودون المساس ببنية الخدمات المصغرة.



## 8.5 تنجيز واجهات المستخدم (سأقوم بالكتابة عنها حين الانتهاء من جميع

(التفاصيل)

1.8.5 بنية ال frontend(react)

2.8.5 مزايا الواجهات

3.8.5 استعراض الواجهات

## الفصل السادس

### النشر والمراقبة

نستعرض في هذا الفصل كيفية نشر النظام ضمن بيئة Kubernetes موزعة، من خلال شرح خطوات بناء الصور الحاوية، إعداد ملفات النشر والتكوين، وآلية تشغيل الخدمات داخل العنقود. كما نعرض أدوات المراقبة والتحليل التي تم استخدامها لضمان استقرار النظام ومراقبة أدائه في بيئة الإنتاج

## 1.6 المعمارية التشغيلية لـ Kubernetes

يُعدّ Kubernetes نظاماً مفتوح المصدر لإدارة حاويات التطبيقات (Containers Orchestration Platform)، وقد تم تطويرها في الأصل من قبل Google ثم أصبحت مشروعاً مفتوح المصدر تحت إشراف Cloud Native Computing Foundation (CNCF). يهدف Kubernetes إلى تبسيط عمليات النشر، التوسيع، مراقبة الصحة، واستعادة الأعطال للتطبيقات الحديثة، من خلال تقديم بنية تشغيلية قائمة على مبدأ العنقود (Clustered Architecture).

من الناحية المعمارية، يُدار نظام Kubernetes عبر طبقتين رئيسيتين:

- مستوى التحكم (Control Plane) التي تشكّل "العقل المركزي" للنظام.
- العُقد العاملة (Worker Nodes) التي تُشغّل الحاويات وتنقذ التعليمات فعلياً.

## 1.1.6 مستوى التحكم (Control Plane)

هو العنصر المحوري المسؤول عن إدارة وتشغيل العنقود cluster بأكمله في Kubernetes. فهو بمثابة العقل المركزي الذي يتخذ كافة القرارات الجوهرية، بدءاً من جدولة الحاويات، مروراً بتتبع حالتها الصحية، وصولاً إلى التنسيق بين جميع مكونات العنقود. تتحكم هذه الطبقة في الحالة العامة للنظام، وتُصدّر أوامر التنفيذ للعُقد العاملة (Worker Nodes).

يتكوّن مستوى التحكم من مجموعة مكونات مترابطة، يُنفَّذ كل منها على خوادم مخصصة (عادةً خوادم Master أو Control Nodes)، وتؤدي المهام التالية:

### 1. API Server

هو المكوّن الأساسي والأول الذي يتفاعل معه المستخدم أو أدوات الأتمتة مثل Helm، CI/CD pipelines، kubectl.

- يُوفّر واجهة RESTful موحّدة تتلقّى جميع الطلبات الخارجية.
- يتحقق من صحة الطلبات وصلاحياتها، ثم يُسجّلها في قاعدة البيانات etcd.
- يُعد نقطة الدخول الرسمية الوحيدة إلى العنقود، ويُمثّل البوابة الموحدة لجميع العمليات.

### 2. Etcd

هي قاعدة بيانات موزعة (Distributed Key-Value Store) تُخزّن فيها الحالة المرجعية الكاملة للعنقود، مثل:

- معلومات الـ Pods، Services، ConfigMaps، Secrets.
- سياسات التوزيع وحالة كل Node.

تُعد المصدر الوحيد للحقيقة (Single Source of Truth)، وكل تغيير في العنقود يجب أن يُسجّل فيها أولاً، تدعم ميزات متقدّمة مثل:

- النسخ المتماثل (Replication) لضمان التوافر العالي.
- الانتخابات (Leader Election) بين عقد التحكم.
- أخذ النسخ الاحتياطية واستعادة الحالة عند حصول أعطال.

### 3. Scheduler

مسؤول عن اتخاذ قرار أين يجب تشغيل كل Pod جديد تم طلبه.

- فحص الموارد المتوفرة على كل عقدة (CPU، RAM، Network).
- يحلّل القيود والسياسات مثل Pod Topology Spread، Taints، Node Affinity.

- ثم يحدد أنسب عقدة لتشغيل ال Pod.

لا يُشغّل الحاويات بنفسه، بل يُحدد المكان المناسب ثم يُسجل قراره في etcd لِيُنقَذ لاحقاً من kubelet على ال Node.

#### 4. Controller Manager

هو عبارة عن مجمّع لوحدات تحكم مستقلة (Controllers)، كل منها يتولّى مراقبة جانب معيّن من العنقود، ويطبّق مفهوم حلقة المراقبة الذاتية (Reconciliation Loop)، أي مقارنة الحالة الفعلية مع الحالة المطلوبة واتخاذ الإجراء اللازم تلقائياً.

من أشهر ال Controllers :

- Replication Controller / ReplicaSet Controller: يتأكد من وجود العدد المطلوب من نسخ ال Pods.

- Node Controller: يراقب صحة كل عقدة، ويُعلنها كـ "غير متاحة" إذا توقفت عن الاستجابة.

- Job/ CronJob Controller: يدير تنفيذ المهام الدورية أو لمرة واحدة.

هذه الحلقات تعمل باستمرار لضمان أن الحالة الحالية تتطابق مع الحالة المتوقعة.

### 2.1.6 العُقد العاملة (Worker Nodes)

كل عقدة تمثّل خادماً فعلياً أو افتراضياً مسؤولاً عن تشغيل التطبيقات. تحتوي كل عقدة على:

- Kubelet: وكيل يتحقق من الأوامر القادمة من ال API Server وينقذها محلياً، مثل تشغيل الحاويات، مراقبة حالتها، والتبليغ عن الأعطال.
- Kube-proxy: مكوّن مسؤول عن توجيه حزم الشبكة بين الخدمات و Pods باستخدام iptables أو ipvs، مع دعم ال ClusterIP و LoadBalancer و NodePort.
- Container Runtime: البرنامج الذي يدير تشغيل الحاويات فعلياً، مثل Docker أو containerd أو CRI-O، وهو ما يقوم بتنفيذ الصور الحاوية.

### 3.1.6 الكيانات التشغيلية (Workload Resources)

- Pod: أصغر وحدة قابلة للنشر، وقد تحتوي على حاوية واحدة أو أكثر تشترك في نفس الشبكة والتخزين. كل Pod يُنشأ ويُدار من قبل موارد أعلى.
- Deployment: كائن يُعرّف كيفية إدارة نسخ متعددة من Pod (replicas) مع دعم التحديث التدريجي (Rolling Updates) والتراجع الآلي عند الفشل (Rollback).

- ReplicaSet: يتأكد من أن عدد الـ Pods المطلوب دائماً في حالة عمل، ويُستخدم ضمن Deployment.
- StatefulSet: يُستخدم مع التطبيقات التي تحتاج إلى هوية ثابتة وتخزين مستمر (مثل قواعد البيانات).
- DaemonSet: يضمن تشغيل Pod واحد على كل عقدة، يُستخدم عادة لتجميع السجلات أو المراقبة.

## 4.1.6 الشبكات والخدمات والتوجيه

تُوفّر Kubernetes بنية شبكية داخلية مرنة تُتيح التواصل السلس بين الحاويات داخل العنقود، بالإضافة إلى إمكانية الوصول إلى الخدمات من خارج العنقود بطريقة آمنة وقابلة للتوسع. لتحقيق ذلك، تُستخدم مكونات متعددة لتنظيم حركة البيانات داخلياً وخارجياً:

### 1. Service

في Kubernetes، لا يُمكن الاعتماد مباشرة على عناوين الـ Pods لأنها غير ثابتة وتتغير عند إعادة التشغيل أو إعادة الجدولة. لذلك، تم تقديم مفهوم Service، وهو تجريد منطقي (Abstraction Layer) يُمثّل مجموعة من الـ Pods التي تُقدّم نفس الوظيفة (مثل: خدمة المتابعة أو قاعدة بيانات)، ويوفّر لها نقطة وصول موحدة وثابتة.

- تعمل الـ Service كـ Load Balancer داخلي يوزّع حركة المرور على جميع نسخ الـ Pods المرتبطة بها (باستخدام Label Selector).

- تُستخدم أنواع متعددة من Services حسب السيناريو المطلوب:
- ClusterIP: للوصول الداخلي فقط داخل العنقود.
- NodePort: يفتح منفذاً على كل عقدة للوصول من خارج العنقود.
- LoadBalancer: يُستخدم في البيئات السحابية لإنشاء عنوان IP خارجي تلقائياً.
- ExternalName: يربط الاسم بخدمة خارجية عبر DNS.

### 2. Ingress & Ingress Controller

عندما يتطلب النظام استقبال طلبات من الإنترنت (مثل فتح الموقع من متصفح المستخدم)، فإن توجيه هذه الطلبات إلى الخدمة المناسبة داخل العنقود يجب أن يتم بطريقة منظمة وآمنة.

- Ingress هو كائن يُعرّف قواعد التوجيه: أي مسار URL يُوجّه إلى أي خدمة، مع إمكانية استخدام الـ Hostnames، المسارات الجزئية (Path-based)، وحتى قواعد HTTPS.
- Ingress Controller هو التطبيق الفعلي الذي يقرأ كائنات Ingress وينفذ القواعد باستخدام خادم مثل Traefik، NGINX، أو HAProxy.

### 3. Network Policies

بشكل افتراضي، يُسمح لجميع الـ Pods بالتواصل مع بعضها داخل العنقود. في الأنظمة الحساسة، نحتاج إلى تقييد هذا السلوك لحماية الخدمات ومنع الوصول غير المشروع.

## 5.1.6 التخزين والإعدادات

### 1. PersistentVolume (PV) و PersistentVolumeClaim (PVC)

في Kubernetes، تُزال الحاويات والـ Pods تلقائياً عند الفشل أو إعادة النشر، مما يعني أن التخزين المحلي لا يمكن الاعتماد عليه للبيانات الدائمة. لذلك، تم إدخال مفهوم:

PersistentVolume (PV): هو تمثيل مجرد لمورد تخزين خارجي (مثل NFS، AWS EBS، أو قرص محلي)، تتم تهيئته بواسطة مسؤول العنقود.

PersistentVolumeClaim (PVC): هو طلب يُقدّمه الـ Pod للحصول على مساحة تخزين معينة وفق مواصفات محددة (الحجم، نوع الوصول...).

يُنشأ الربط بين PVC و PV تلقائياً، مما يسمح للحاوية بالقراءة والكتابة من التخزين الخارجي بغض النظر عن عدد مرّات إعادة تشغيل الـ Pod.

### 2. Secret و ConfigMap

من الأفضل دائماً فصل الإعدادات والبيانات الحساسة عن الكود. لذلك، يُقدّم Kubernetes آليتين فعاليتين:

- ConfigMap: تُستخدم لتمثيل إعدادات غير حساسة مثل أسماء الخدمات، العناوين، أو متغيّرات البيئة.
- Secret: تُستخدم لتمثيل معلومات حساسة مثل كلمات المرور، رموز المصادقة، مفاتيح API، مع تخزينها مشفرة داخل etcd.

كلا الكائنين يمكن تمريرهما إلى الحاويات عبر:

متغيّرات البيئة (ENV) – ملفات داخل الـ Container – Mounting Volume.

## 6.1.6 عزل الموارد والتنظيم باستخدام Namespaces

في Kubernetes، يُعد Namespace بمثابة "بيئة منطقية معزولة" داخل العنقود. وهو يوفر طبقة تجريدية تساعد على تنظيم الموارد، فصل البيئات، وتطبيق سياسات أمان وتخصيص موارد مختلفة حسب السياق (تطوير، إنتاج، اختبار...).

Namespace هو تجزئة افتراضية داخل العنقود تُستخدم لعزل المكونات عن بعضها البعض، كل Namespace يحتوي على:

- مجموعة من الموارد مثل Pods, Services, ConfigMaps ....

- مجال أسماء خاص به (Name Scope).
- سياسات وصول وصلاحيات مستقلة.
- حدود موارد مخصصة (Resource Quotas, Limits).

## 2.6 تنفيذ النشر الفعلي على Kubernetes: خطوات عملية داخل المشروع

قبل اعتماد Kubernetes، كانت الخدمات تُطوّر وتُشغّل محلياً باستخدام أدوات مثل Docker و Docker Compose، مما أتاح بيئة تجريبية بسيطة وسريعة. ومع ذلك، أظهرت هذه الطريقة محدودية عند الانتقال إلى بيئة تشغيل حقيقية تتطلب التوسع، المرونة، والتوافر العالي.

ولهذا تم الانتقال إلى بيئة Kubernetes، نظراً لما توفره من مزايا حيوية، أبرزها:

- التوسع الأفقي التلقائي (Horizontal Scaling): حيث يمكن إضافة المزيد من الحاويات تلقائياً حسب الحمل.
- التحمل العالي للأعطال (Fault Tolerance): عبر إعادة تشغيل الحاويات المتوقفة وتوزيعها على عدة عقد.
- التنظيم والإدارة عبر Namespaces: ما يتيح فصل البيئات والخدمات بشكل مرّن وآمن.
- إدارة الموارد بشكل دقيق: من خلال تخصيص حدود وطلبات CPU/RAM لكل خدمة.
- التحديثات الآمنة (Rolling Updates): دون التأثير على توافر النظام.
- المراقبة والتتبع (Observability): باستخدام أدوات مدمجة لمراقبة الأداء وتحليل المشاكل.

هذا الانتقال مكن النظام من الانتقال من نموذج تجريبي إلى منصة إنتاجية قابلة للتوسع والاستخدام الواقعي.

وبناء على ذلك، يُعرض في هذا القسم كيف تم تنفيذ عملية النشر داخل Kubernetes خطوة بخطوة:

### 1.2.6 بيئة النشر والبنية التحتية

في هذه المرحلة، تم استخدام cluster Kubernetes تمّ توفيره مسبقاً ضمن بيئة سحابية خاصة (Private Cloud) مقدّمة من المعهد العالي، حيث أُتيح لنا إمكانية الوصول إلى العنقود واستخدامه لنشر مكونات المنصة الاجتماعية بطريقة موزعة.

وقد مثّل هذا العنقود بنية تشغيل حقيقية تدعم نمط البنية التحتية كخدمة (IaaS – Infrastructure as a Service) وتوفّر الموارد اللازمة لتشغيل عدد كبير من الحاويات والخدمات المصغرة ضمن نظام متكامل.

تم تزويدنا بإمكانية الوصول إلى ثلاث آلات افتراضية تمثل بيئة التشغيل:

- عقدتان عاملتان (Worker Nodes)

نظام التشغيل: Ubuntu Server 22.04

الذاكرة (RAM): 16 GB

عدد المعالجات: 8

سعة التخزين: 50 GB

العدد: 2 عقدة

■ عقدة تحكم واحدة (Master Node)

نظام التشغيل: Ubuntu Server 22.04

الذاكرة (RAM): 16 GB

عدد المعالجات: 8

سعة التخزين: 50 GB

العدد: 1 عقدة

```
ubuntu@k1:~$ kubectl get nodes
NAME     STATUS   ROLES    AGE   VERSION
k1       Ready    control-plane  20d   v1.31.10
k2       Ready    <none>      20d   v1.31.10
k3       Ready    <none>      20d   v1.31.10
ubuntu@k1:~$
```

الشكل 50: العقد العاملة ضمن العنقود

كما تم توفير سجل صور حاوية (Private Docker Registry) على نفس الشبكة الداخلية، يُستخدم لتخزين الصور الحاوية الخاصة بكل خدمة. وقد مكّننا هذا من رفع الصور بعد بنائها محلياً واستخدامها مباشرة في ملفات النشر داخل العنقود دون الحاجة للرجوع إلى Docker Hub أو أي سجل خارجي.

## 2.2.6 تقسيم الخدمات ضمن Namespaces وبناء الصور الحاوية

بعد تجهيز منطق التطبيق وتطوير كل خدمة من الخدمات المصغرة على حدة (user-service, auth-service, post-service, feed-service, ...) باستخدام Spring Boot، تم الانتقال إلى مرحلة التحزيم والنشر داخل العنقود. تم اتباع الخطوات التالية لتقسيم النظام ونشره بطريقة منظمة:

لأجل تنظيم بيئة التشغيل وفصل الخدمات حسب نوعها ووظيفتها، تم اعتماد نمط تقسيم الخدمات باستخدام Namespaces داخل العنقود، على النحو التالي:

الهدف	Namespace
يحتوي على جميع الخدمات المصغرة الخاصة بمنصة الصور (backend)	rm-photo-app
يحتوي على قواعد البيانات والبنية التحتية (Mongo, MySQL, Redis...)	databases
يستخدم لتشغيل خدمة تتبع الطلبات بين الخدمات (Distributed Tracing)	zipkin

الجدول 1: تقسيم ال namespace

ساهم هذا التقسيم في تسهيل عمليات النشر والتوسع، كما أتاح الفصل بين المكونات التطبيقية ومكونات البنية التحتية، مما عزز من وضوح البنية، وسهولة مراقبتها وصيانتها.

### 3.2.6 بناء الصور الحاوية (Docker Images)

بمجرد الانتهاء من تطوير كل خدمة من الخدمات المصغرة، تم الانتقال إلى مرحلة تحويلها إلى صور حاوية (Docker Images) تمهيداً لنشرها داخل العنقود. في سياق الأنظمة المبنية على الخدمات المصغرة (Microservices)، يُعدّ بناء الصور الحاوية (Docker Images) خطوة محورية في تحويل الكود المصدري لكل خدمة إلى وحدة تشغيل مستقلة، قابلة للنشر، التكرار، والتوسع داخل بيئة Kubernetes.

تسمح الصورة الحاوية بتغليف التطبيق، وجميع اعتماديّاته (Dependencies)، ومتغيرات البيئة، ضمن وحدة محمولة يمكن تشغيلها في أي بيئة متوافقة مع Docker دون اختلاف في السلوك أو النتائج.

الهدف من بناء صورة حاوية لكل خدمة:

- عزل كل خدمة في بيئة مستقلة عن الأخرى.
  - تمكين نشر الخدمات بشكل منفصل ومستقل دون الاعتماد على نظام ملفات مشترك أو سيرفر واحد.
  - دعم النشر الآلي والتوسع الأفقي (Horizontal Scaling) باستخدام Kubernetes.
  - تحسين قابلية النقل (Portability) وتشغيل التطبيق بنفس الشكل على أي Node داخل العنقود.
- لتحقيق ماسبق، تم إنشاء ملف تعريف بناء (Dockerfile) داخل كل مشروع خدمة، يحدّد كيفية إنشاء الصورة.



بعد إعداد ملفات Dockerfile، تم بناء الصور الحاوية محلياً باستخدام Docker، ثم رفعها إلى سجل داخلي ( Private Registry)، ما أتاح استخدام هذه الصور لاحقاً في تعريفات النشر (YAML) داخل Kubernetes دون الاعتماد على سجلات خارجية.

## 4.2.6 نشر الخدمات باستخدام ملفات YAML

بعد بناء الصور الحاوية ورفعها إلى الـ Registry الداخلي، تم الانتقال إلى مرحلة نشر الخدمات داخل العنقود باستخدام ملفات تعريف الموارد في Kubernetes المكتوبة بصيغة YAML .

تُعد ملفات YAML بمثابة "الوصف المهيكل" للبنية المراد نشرها، وتحتوي على تعليمات واضحة لـ Kubernetes لإنشاء الحاويات، تنظيم الوصول، تخصيص الموارد، وضبط الشبكات والمراقبة.

تم اتباع هيكلية منظمّة حيث قمنا بوضع ملف خاص لكل نوع من أنواع ملفات .yaml.

الوظيفة	الملف
يحتوي على تعريف تشغيل الخدمة، عدد النسخ(replicas) ، صورة الحاوية، البيئة، الموارد المطلوبة(CPU/Memory) .	deployment.yaml
يُستخدم لإنشاء Service مرتبطة بالPods ، وتحديد نوع الوصول ( ClusterIPأوNodePort )	service.yaml
يحتوي على تعريفات مراقبة خاصة مثل التعريف بـ ServiceMonitor أو Metrics.	monitoring.yaml
يُستخدم لتفعيل التوسّع التلقائي لأعداد الـ Pods اعتماداً على استهلاك الموارد (مثل CPU,memory)، مع تحديد حد أدنى وأقصى للنسخ.	hpa.yaml

تم تصميم كل ملف YAML بحيث يمكن استخدامه بشكل مستقل عبر أداة kubectl، مما يُتيح مرونة في النشر، التحديث، أو الحذف دون التأثير على باقي مكونات النظام.

تظهر هذه الصور نموذجاً للملفات :

```
user-deployment...
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: user-service
5   namespace: rm-photo-app
6 spec:
7   replicas: 1
8   selector:
9     matchLabels:
10      app: user-service
11 template:
12   metadata:
13     labels:
14       app: user-service
15   spec:
16     containers:
17     - name: user-service
18       image: 172.29.3.41:5000/user-service:1.1
19       imagePullPolicy: IfNotPresent
20       ports:
21       - containerPort: 8080
22     env:
23     - name: SPRING_PROFILES_ACTIVE
24       value: "prod"
25     - name: JAVA_TOOL_OPTIONS
26       value: >-
27       -Dspring.zipkin.sender.type=webclient
28       -Djdk.httpclient.allowRestrictedHeaders=connection
29       -Dreactor.netty.http.http2=false
30     resources:
31       requests:
32         cpu: "250m"
33         memory: "256Mi"
34       limits:
35         cpu: "500m"
36         memory: "512Mi"
37
```

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: user-service
5   namespace: rm-photo-app
6   labels:
7     app: user-service
8 spec:
9   selector:
10    app: user-service
11   ports:
12   - name: http
13     protocol: TCP
14     port: 8080
15     targetPort: 8080
16   type: ClusterIP
```

الشكل 52: ملف service

الشكل 51: ملف تعريف النشر Deployment

```
1 apiVersion: monitoring.coreos.com/v1
2 kind: ServiceMonitor
3 metadata:
4   name: user-service-monitor
5   namespace: rm-photo-app
6   labels:
7     release: prometheus
8 spec:
9   selector:
10     matchLabels:
11       app: user-service
12   namespaceSelector:
13     matchNames:
14     - rm-photo-app
15   endpoints:
16   - port: http
17     path: /actuator/prometheus
18     interval: 15s
19
```

الشكل 53: ملف المراقبة Monitor

```

1 apiVersion: autoscaling/v2
2 kind: HorizontalPodAutoscaler
3 metadata:
4   name: search-service-hpa
5   namespace: rm-photo-app
6 spec:
7   scaleTargetRef:
8     apiVersion: apps/v1
9     kind: Deployment
10    name: search-service
11  minReplicas: 1
12  maxReplicas: 8
13  metrics:
14  behavior:
15    scaleDown:
16      stabilizationWindowSeconds: 300
17      policies:
18        - type: Percent
19          value: 90
20          periodSeconds: 15
21    scaleUp:
22      stabilizationWindowSeconds: 0
23      policies:
24        - type: Percent
25          value: 50
26          periodSeconds: 15
27  metrics:
28    - type: Resource
29      resource:
30        name: cpu
31        target:
32          type: Utilization
33          averageUtilization: 70
34

```

الشكل 54: ملف التوسع الأفقي HPA

## 3.6 نشر قواعد البيانات داخل العنقود:

لضمان أداء عالي وقدرة على التوسع الأفقي، تم نشر كل من قواعد البيانات الضرورية داخل العنقود على النحو التالي:

### 1.3.6 نشر MongoDB Sharded Cluster

تم اعتماد بنية موزعة (Sharded Cluster) لـ MongoDB بهدف تحسين أداء عمليات القراءة والكتابة مع تزايد حجم البيانات وعدد المستخدمين. يتكوّن العنقود من:

- **Shard1 و Shard2:** تم إنشاء كل منهما كـ **StatefulSet** مكوّن من 3 نسخ (replica set) تُدار باستخدام **StatefulSets**، مع ربط كل نسخة بـ **PersistentVolume** و **PersistentVolumeClaim** مخصصة تُشير إلى مجلدات تخزين فعلية
- **Config Server:** مكوّن أساسي لإدارة بيانات التكوين الخاصة بالتقسيم (sharding)، وقد تم نشره أيضاً باستخدام **StatefulSet** مع تخزين دائم.
- **Mongos Router:** يعمل كواجهة دخول (Query Router) تستقبل طلبات المستخدمين وتوزعها على الـ **Shards** بشكل شفاف، وتم نشره باستخدام **Service** و **Deployment**.

## 2.3.6 نشر Elasticsearch

تم نشر خدمة **Elasticsearch** كمكوّن أساسي لفهرسة بيانات المستخدمين ودعم عمليات البحث النصي داخل المنصة. وقد تم تنفيذ عملية النشر داخل **Namespace** مخصص لقواعد البيانات (databases)، وذلك باستخدام مجموعة من ملفات **YAML** التي تُعرّف كافة موارد **Kubernetes** اللازمة لتشغيل الخدمة.

كما تمّ إنشاء وحدة تخزين **PV+PVC** من أجل عمليات التخزين، بعد إعداد التخزين، تم نشر خدمة (Service) من نوع **ClusterIP**.

## 3.3.6 نشر kafka+ookeeper

تمّ في هذا المشروع نشر نظام المراسلة **Apache Kafka** وخدامه التنسيق **Zookeeper** داخل عنقود **Kubernetes** ضمن **Namespace** خاص بالبيانات، وذلك بهدف تمكين التواصل غير المتزامن بين الخدمات المصنّعة بطريقة موثوقة وقابلة للتوسع.

تمّ نشر كل من **kafka**، **zookeeper** كـ **pod** مستقل.

تمّ ضبط المتغيرات البيئية اللازمة لتحديد **Broker ID** وربطه بـ **Zookeeper**.

## 4.3.6 نشر My SQL

تم نشر قاعدة بيانات **MySQL** ضمن العنقود باستخدام مجموعة من ملفات **YAML** المخصصة التي تضمنت جميع الموارد اللازمة لضمان التشغيل المستقر وحفظ البيانات بطريقة دائمة. تم إنشاء **PersistentVolume (PV)** لتحديد مساحة تخزين فعلية على القرص المحلي داخل عقدة **Kubernetes**، بعد ذلك، تم تعريف **PersistentVolumeClaim (PVC)** لطلب هذه المساحة التخزينية من داخل الـ **Pod**، وضمان ارتباطها بالحاوية العاملة. ثم تم إعداد **Deployment** خاص بقاعدة البيانات، حيث تضمن المواصفات الضرورية لتشغيل صورة **MySQL** مع تعيين اسم المستخدم وكلمة المرور وقاعدة البيانات الافتراضية باستخدام متغيرات البيئة.

أخيراً، تم إعداد Service من نوع ClusterIP لتأمين الوصول إلى قاعدة البيانات من الخدمات الأخرى داخل العنقود، باستخدام اسم ثابت كـ DNS داخل الـ Cluster.

### 5.3.6 نشر redis

تم أولاً إنشاء وحدة التخزين الدائمة (Persistent Volume)، ثم قمنا بتعريف المطالبة بالتخزين (Persistent Volume Claim) لربط Redis مع وحدة التخزين بشكل ديناميكي. بعد ذلك، أنشأنا خدمة داخلية (ClusterIP Service) باستخدام الملف 1 لتتيح الوصول إلى Redis من قبل الخدمات الأخرى ضمن العنقود دون الحاجة لكشفها خارجياً. وأخيراً، قمنا بتهيئة النشر الفعلي (Deployment) في ملف redis-deployment.yaml لتشغيل حاوية Redis مع إعدادات الموارد المطلوبة.

### 6.3.6 نشر MinIO

تم إنشاء وحدة تخزين دائمة (PersistentVolume) تحدد المسار الفيزيائي لتخزين البيانات على عقدة Kubernetes. تلا ذلك إنشاء مطالبة بالحجم (PersistentVolumeClaim) لربط وحدة التخزين بالحاوية. بعد تجهيز التخزين، تم تعريف كائن الخدمة (Service) الذي يتيح الوصول إلى MinIO داخل العنقود باستخدام ClusterIP، بالإضافة إلى كائن خدمة آخر من نوع NodePort يتيح الوصول الخارجي إلى واجهة MinIO عبر منفذ مخصص.

بعدها تم نشر الحاوية عبر كائن Deployment يحتوي على مواصفات صورة MinIO والبيئة التشغيلية الخاصة بها، بما في ذلك متغيرات البيئة مثل بيانات الدخول، مسار التخزين، وعدد النسخ (replicas). كما تم ربط الـ Deployment بالـ PVC لضمان حفظ البيانات خارج الحاوية.

بعد إتمام النشر، تم الوصول إلى MinIO من داخل الكلاستر باستخدام حاوية مؤقتة من صورة minio-mc، وتم إعداد alias للتعامل مع الخادم عبر عنوان الخدمة الداخلية. من خلال هذه الواجهة، جرى إنشاء دلوين (Buckets) رئيسيين: user-images لحفظ صور المستخدمين، و post-images لحفظ صور المنشورات.

### 7.3.6 نشر Nginx

تم إعداد وحدة تخزين دائمة (PersistentVolume) مع مطالبة تخزين (PersistentVolumeClaim) لتأمين المساحة اللازمة لتخزين ملفات الإعدادات الخاصة بـ Nginx. بعد ذلك، تم استخدام ConfigMap لتحميل إعدادات الخادم (nginx.conf) بشكل ديناميكي داخل الحاوية. تم نشر الحاوية عبر كائن Deployment لضمان توفرها واستمراريتها، مع إمكانية إعادة تشغيلها تلقائياً في حال حدوث أي فشل.

## 4.6 المراقبة وتحليل الأداء (Monitoring & Observability)

تُعَدّ المراقبة وتحليل الأداء من الركائز الأساسية لضمان موثوقية واستقرار الأنظمة الموزعة الحديثة، خصوصاً في بيئات تعمل بنمط الخدمات المصغرة (Microservices Architecture). إذ تُتيح هذه الآليات إمكانية رصد الحالة التشغيلية للخدمات، تحليل سلوكها تحت الضغط، واكتشاف الأعطال والأداء غير الطبيعي بشكل لحظي.

في هذا المشروع، تم تصميم منظومة مراقبة شاملة تغطّي الجوانب التالية:

- تتبّع حالة الحاويات واستهلاك الموارد
- مراقبة استجابة الخدمات وكفاءة الاتصال بينها
- تحليل أثر الأحمال المتغيرة على استقرار المنصة.

### 1.4.6 الأدوات المستخدمة

#### Prometheus + Grafana 1.1.4.6

لمراقبة حالة النظام وقياس مؤشرات الأداء الحيوية بشكل دوري، تمّ اختيار نظام مراقبة متكامل مبني على أداتي Prometheus وGrafana ضمن الكلاستر.

تم اختيار Prometheus كمحرك لجمع المؤشرات (metrics- Collector)، حيث يقوم بمسح الخدمات المصغرة بانتظام، وجمع مؤشرات مثل استهلاك المعالج والذاكرة، وعدد النسخ النشطة، ومعدل الأخطاء.

وتمّ اختيار Grafana التي تعمل كأداة عرض رسومية (Visualization Dashboard)، حيث تعرض البيانات المجموعة في شكل لوحات رسومية واضحة، تتيح فهم الأداء العام ومراقبة سلوك النظام عبر الزمن.

تم إقران Prometheus بأداة Grafana لعرض المؤشرات ضمن لوحات رسومية تفاعلية، تُظهر:

- استخدام CPU و Memory لكل Pod
- عدد النسخ النشطة (Replicas) عبر HPA .
- مؤشرات استجابة الخدمات وتكرار الأخطاء.

### Zipkin 2.1.4.6

لمتابعة تدفق الطلبات عبر سلسلة الخدمات، تم نشر نظام تتبّع طلبات موزعة (Distributed Tracing) باستخدام Zipkin. يسمح ذلك بتحليل زمني دقيق (Trace-level Latency) للطلبات العابرة للخدمات، مما يتيح تحديد نقاط التأخير (bottlenecks) واختناقات الأداء.

## 2.4.6 نشر أدوات المراقبة داخل العنقود

تم نشر كل من Prometheus و Grafana و Zipkin داخل Kubernetes باستخدام ملفات YAML منظمة ضمن بيئة منفصلة (Namespace معزول)، دون التأثير على سير عمل الخدمات الأساسية.

### 1.2.4.6 نشر أداة تتبع الطلبات Zipkin

تم نشر أداة Zipkin داخل العنقود كخدمة مستقلة، من خلال إنشاء تعريف نشر (Deployment) مخصص يحتوي على الصورة الجاهزة ل Zipkin.

تم ربط الخدمة بشبكة العنقود عبر Service، يتيح استقبال وتتبع الطلبات القادمة من باقي الخدمات في النظام.

كما تم تهيئة الخدمات المصغرة الخلفية لإرسال بيانات التتبع إلى Zipkin بشكل تلقائي، مما أتاح مراقبة انتقال الطلبات عبر أكثر من خدمة، وتحليل أزمدة المعالجة.

```
# ——— Tracing and Zipkin ———
management.tracing.sampling.probability=1.0
management.zipkin.tracing.endpoint=http://zipkin.zipkin.svc.cluster.local:9411/api/v2/spans
management.zipkin.tracing.sender.type=webclient
logging.pattern.level=%5p [${spring.application.name:},%X{traceId:-},%X{spanId:-}]
# Add connection timeouts (critical!)
management.zipkin.tracing.connect-timeout=5s
management.zipkin.tracing.read-timeout=10s
```

```
ubuntu@k1:~$ kubectl get svc -n zipkin
NAME      TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
zipkin    NodePort    10.104.206.77    <none>            9411:30911/TCP    4d3h
ubuntu@k1:~$ kubectl get pod -n zipkin
NAME                                READY   STATUS    RESTARTS   AGE
zipkin-54ffcc5767-whkrb             1/1     Running   0           2d2h
ubuntu@k1:~$
```

## 2.2.4.6 نشر نظام المراقبة Prometheus و Grafana





قمنا بنشر نظام مراقبة مركزي باستخدام حزمة kube-prometheus-stack بالاعتماد على مستودع Helm الرسمي (prometheus-community)، وذلك من أجل جمع وتحليل القياسات (metrics) الصادرة عن خدمات التطبيق المختلفة ضمن بيئة Kubernetes، مثل معدل الاستهلاك، وأزمدة الاستجابة، وعدد الطلبات الواردة، وغيرها من المؤشرات التشغيلية.

كذلك Grafana التي تمثل الواجهة الرسومية الأساسية لمراقبة وتحليل أداء الخدمات.

حيث تمّ تهيئة الخدمات لإرسال المقاييس (Metrics) باستخدام مكتبة Micrometer، وتمّ ربطها بنظام Prometheus عبر نقاط النهاية من نوع actuator/prometheus/ في كل خدمة Spring Boot.

```
# ——— Monitoring ———  
# Enable actuator endpoints  
management.endpoints.web.exposure.include=health,info,prometheus  
management.endpoint.prometheus.enabled=true  
management.metrics.export.prometheus.enabled=true
```



serviceMonitor/rm-photo-app/feed-service-monitor/0	1 / 1 up  ▾
serviceMonitor/rm-photo-app/follow-service-monitor/0	1 / 1 up  ▾
serviceMonitor/rm-photo-app/post-service-monitor/0	1 / 1 up  ▾
serviceMonitor/rm-photo-app/search-service-monitor/0	1 / 1 up  ▾
serviceMonitor/rm-photo-app/user-service-monitor/0	1 / 1 up  ▾

## الفصل السابع

### الاختبارات ومناقشة النتائج

نقوم ضمن هذا الفصل بعرض نتائج الاختبارات وتلبيتها للمتطلبات

## 1.7 اختبارات الأداء

تُعد اختبارات الأداء (Performance Testing) من المراحل الحيوية في تقييم كفاءة المشروع، وخصوصاً في الأنظمة الاجتماعية التي تتطلب قدرة عالية على التعامل مع أعداد كبيرة من المستخدمين والطلبات في وقت متزامن. في مشروع منصة اجتماعية لمشاركة الصور،



لا تكفي اختبارات الوظائف (Functional Tests) وحدها لضمان جاهزية النظام، بل لا بد من اختبار مدى تحمّله للأحمال المتزايدة، واستجابته في ظل ظروف تشغيل مختلفة، واستقراره عند الوصول إلى حدود الاستخدام القصوى

تهدف هذه الاختبارات إلى قياس مجموعة من المؤشرات الأساسية مثل:

- زمن الاستجابة (Response Time): كم من الوقت يحتاج النظام لمعالجة الطلبات؟
- معدل الطلبات في الثانية (Throughput): كم عدد الطلبات التي يمكن للنظام التعامل معها في الثانية الواحدة؟
- استهلاك الموارد (CPU / Memory Usage): كيف يتغير استهلاك النظام للموارد تحت الضغط؟
- نسبة الأخطاء (Error Rate): ما نسبة الطلبات التي تفشل أثناء التحميل المرتفع؟

يتم تنفيذ هذه الاختبارات باستخدام أدوات مثل JMeter أو k6 لمحاكاة سيناريوهات استخدام فعلية، تشمل حالات الاستخدام المختلفة، بهدف تحديد نقاط الاختناق (Bottlenecks) وتحسين أداء الخدمات.

## 2.7 تحضير بيئة الاختبار

تتضمن عملية تحضير بيئة الاختبار نشر الخدمات في العنقود الذي تم إنشاؤه، وتجهيز الأدوات التي سيتم استخدامها للاختبار.

سيتم تنفيذ الاختبار على آلة محلية تعمل بنظام Windows، وتتمتع بالمواصفات التالية:

- ذاكرة رئيسية (RAM) بسعة 8 Gigabyte.
- معالج بعدد 8 معالجات منطقية (Logical Processors).

## 3.7 تجهيز خطة الاختبار

### 1.3.7 خطة الاختبار لخدمة الخلاصة (feed-service) :

تمثل خدمة الخلاصة (Feed Service) إحدى أهم الخدمات المحورية ضمن منصة مشاركة الصور، نظرًا لاعتماد تجربة المستخدم بشكل مباشر على سرعة تحميل هذه الخلاصة، فإن أداء هذه الخدمة يلعب دورًا في جودة المنصة.

قبل البدء في تصميم سيناريوهات الاختبار وتحديد القيم الرقمية المستهدفة، من المهم التأكيد على أن الغاية من هذه الاختبارات لا تقتصر على قياس مؤشرات الأداء فقط، بل تهدف أيضًا إلى تقييم قابلية النظام للتوسع (Scalability)، والتأكد من قدرته على التكيف مع التغيرات الديناميكية في الحمل، سواء في حالات التزايد التدريجي (Scale Up) أو التراجع (Scale Down). وتجدر الإشارة إلى أن الأداء الفعلي في بيئة الإنتاج سيتأثر بعدة عوامل، أبرزها خصائص البنية التحتية المخصصة للنشر وعدد المستخدمين الفعليين وتوزيع نشاطهم الزمني.

المتطلب الزمني الأساسي: يجب ألا يتجاوز زمن الاستجابة (Response Time) في 99% من الطلبات قيمة 1000ms .

الحمل المتوقع: من خلال تحليل استخدام المنصة وسيناريوهات التفاعل المتكررة، تم تقدير أن خدمة الخلاصة ستستقبل في المتوسط ما بين 150 إلى 200 طلبًا في الثانية.

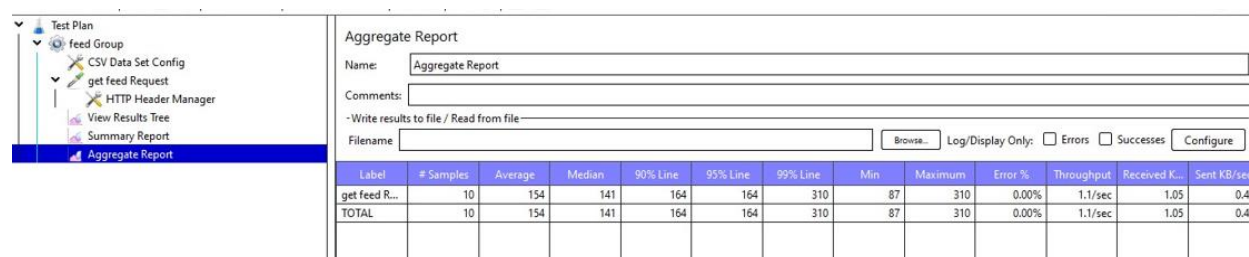
تم استخدام أداة k6 لتنفيذ اختبارات الأداء

### 1.1.3.7 ملخص خطة الاختبار

سنقوم بعدد من الاختبارات منها اختبار السلامة، اختبار الحمل، اختبار الجهد.

▪ اختبار السلامة smoke test :

نقوم باختبار السلامة للتأكد من عمل الخدمة بشكل صحيح من بعدها ننتقل الى اختبار الحمل.



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received K...	Sent KB/sec
get feed R...	10	154	141	164	164	310	87	310	0.00%	1.1/sec	1.05	0.41
TOTAL	10	154	141	164	164	310	87	310	0.00%	1.1/sec	1.05	0.41

الشكل 55: اختبار السلامة لخدمة الخلاصة



Sampler result	Request	Response data
get feed Request		Response Body Response headers
get feed Request		
get feed Request		
get feed Request		
get feed Request		
get feed Request		
get feed Request		
get feed Request		
get feed Request		
get feed Request		

الشكل 56: نتيجة اختبار السلامة لخدمة الخلاصة

نلاحظ ان النتيجة بدت صحيحة ننتقل الى اختبار الحمل

## ■ اختبار الحمل load test

قمنا في المرحلة الأولى بإجراء عملية اختبار بحمل ثابت على خدمة عرض الخلاصة

لاحظنا إنه عندما قمنا بعملية اختبار على عدد قليل من الطلبات حصلنا على نتائج سيئة

```
■ TOTAL RESULTS

checks_total.....: 3691    11.976771/s
checks_succeeded.....: 100.00% 3691 out of 3691
checks_failed.....: 0.00% 0 out of 3691

■ status is 200

HTTP
http_req_duration.....: avg=16.41s min=3.96s med=16.27s max=34.35s p(90)=21.68s p(95)=23.37s
  { expected_response:true }.....: avg=16.41s min=3.96s med=16.27s max=34.35s p(90)=21.68s p(95)=23.37s
http_req_failed.....: 0.00% 0 out of 3691
http_reqs.....: 3691    11.976771/s

EXECUTION
dropped_iterations.....: 14310  46.433917/s
iteration_duration.....: avg=16.42s min=3.96s med=16.27s max=34.35s p(90)=21.68s p(95)=23.37s
iterations.....: 3691    11.976771/s
vus.....: 15    min=15    max=200
vus_max.....: 200    min=100    max=200

NETWORK
data_received.....: 8.6 MB 28 kB/s
data_sent.....: 1.2 MB 4.0 kB/s
```

الشكل 57: تجربة 1 اختبار الحمل لخدمة الخلاصة

```

execution: local
  script: feed_test.js
  output: -

scenarios: (100.00%) 1 scenario, 60 max VUs, 6m30s max duration (incl. graceful stop):
  * spike_test: Up to 60 looping VUs for 6m0s over 3 stages (gracefulRampDown: 30s, gracefulStop: 30s)

TOTAL RESULTS

checks_total.....: 4864    13.509257/s
checks_succeeded.....: 100.00% 4864 out of 4864
checks_failed.....: 0.00% 0 out of 4864

@ status is 200

HTTP
http_req_duration.....: avg=4.08s min=121.83ms med=4.27s max=7.48s p(90)=5.39s p(95)=5.71s
  { expected_response:true }.....: avg=4.08s min=121.83ms med=4.27s max=7.48s p(90)=5.39s p(95)=5.71s
http_req_failed.....: 0.00% 0 out of 4864
http_reqs.....: 4864    13.509257/s

EXECUTION
iteration_duration.....: avg=4.08s min=121.83ms med=4.27s max=7.48s p(90)=5.39s p(95)=5.71s
iterations.....: 4864    13.509257/s
vus.....: 1    min=1    max=60
vus_max.....: 60    min=60    max=60

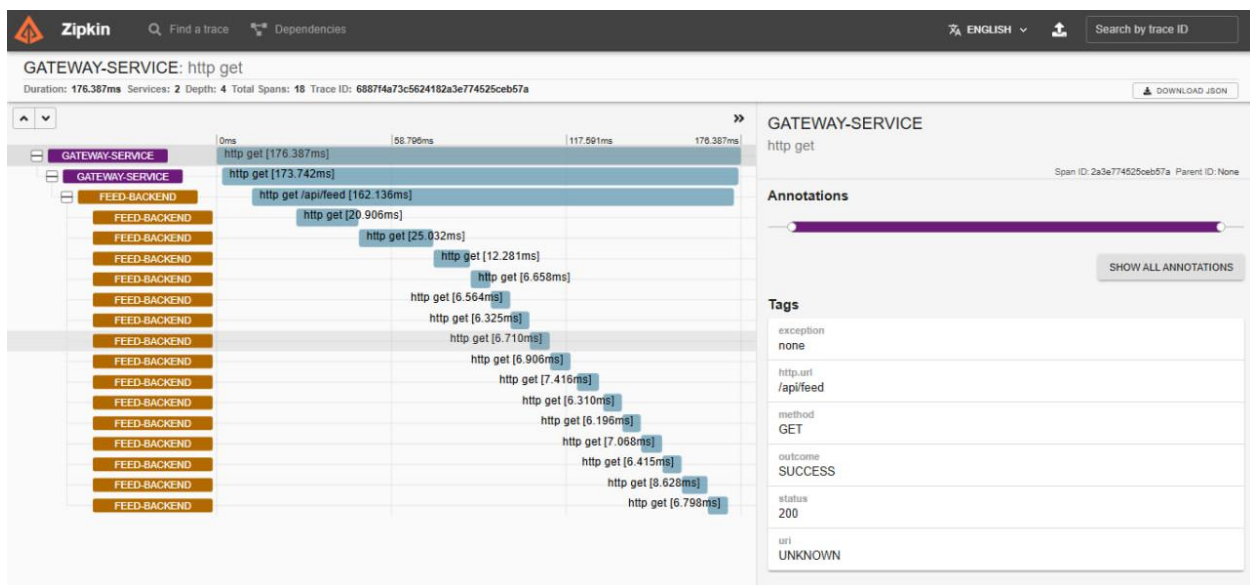
NETWORK
data_received.....: 7.6 MB 21 kB/s
data_sent.....: 1.7 MB 4.7 kB/s

```

الشكل 58: تجربة 2 اختبار الحمل لخدمة الخلاصة

قمنا بعملية التوسع الافقي والشاقولي ولم تتحسن النتائج، بعد البحث في الأسباب وملاحظة الطلبات تبين أن السبب في كثرة الطلبات المتزامنة التي تحتاجها الخدمة للقيام بالعمل حيث نحتاج من اجل كل طلب الى التواصل مع خدمة المتابعة لمعرفة المتابعين ونحتاج الى التواصل مع خدمة المستخدم لمعرفة نوع كل متابع لمعرفة كيفية التعامل معه وهذا يأخذ وقت طويل للمعالجة وعدد طلبات كبير.

نلاحظ في الشكل المدة المستغرقة لعملية عرض الخلاصة كان كبير حيث تحتاج الى عدد كبير من الطلبات وكل منها تستغرق وقت



الشكل 59: ملاحظة طلب عرض الخلاصة

وبعد ذلك قمنا بالتعديل على على تصميم خدمة المنشورات وخدمة الخلاصة (كما هو مذكور في الفصل الرابع )

وقمنا باعادة تجربة الاختبارات:

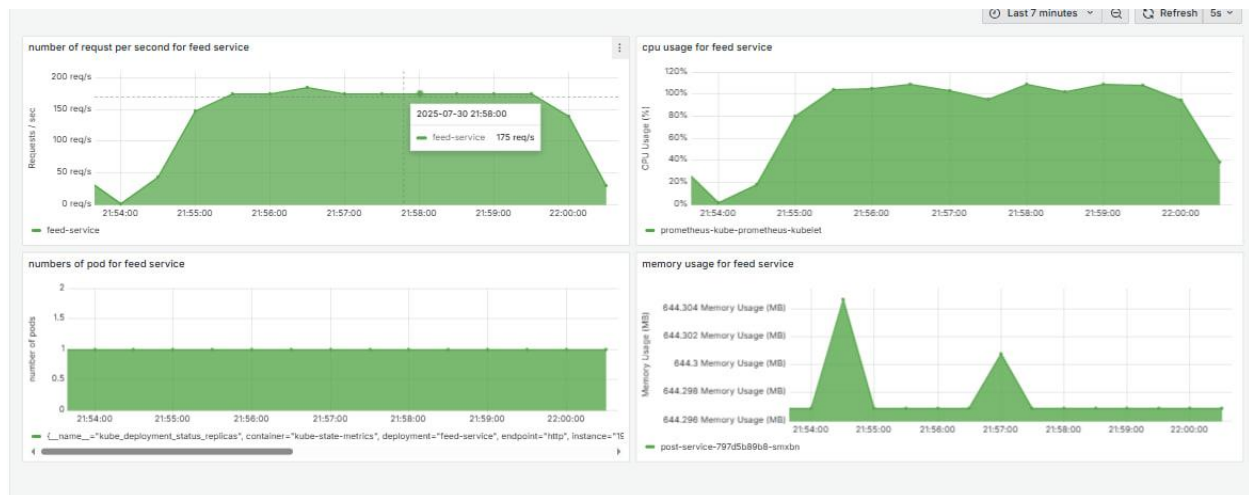
- تم تطبيق 150 طلباً في الثانية حيث كانت مدة الاختبار 5 دقائق ،نلاحظ نجاح التجربة حيث كان الزمن الوسطي المستغرق 31.12ms وهو زمن قليل جداً مقارنة بالمتطلب ويظهر أنه عند 95% من الطلبات كان الزمن المستغرق 86.49ms

TOTAL RESULTS	
checks_total	49799 138.330347/s
checks_succeeded	100.00% 49799 out of 49799
checks_failed	0.00% 0 out of 49799
@ status is 200	
CUSTOM	
request_duration	avg=31.129942 min=0 med=15.8792 max=1584.3429 p(90)=42.14138 p(95)=86.49131
slow_requests	139 0.386111/s
total_requests	49799 138.330347/s
HTTP	
http_req_duration	avg=31.12ms min=0s med=15.87ms max=1.58s p(90)=42.14ms p(95)=86.49ms
{ expected_response:true }	avg=31.12ms min=0s med=15.87ms max=1.58s p(90)=42.14ms p(95)=86.49ms
http_req_failed	0.00% 0 out of 49799
http_reqs	49799 138.330347/s
EXECUTION	
iteration_duration	avg=32.36ms min=0s med=16.69ms max=1.58s p(90)=44.69ms p(95)=88.9ms
iterations	49799 138.330347/s
vus	0 min=0 max=187
vus_max	2000 min=2000 max=2000
NETWORK	
data_received	121 MB 336 kB/s
data_sent	6.6 MB 18 kB/s

### الشكل 60: نتائج اختبار الحمل لخدمة الخلاصة

-تم تطبيق حمل 175 طلباً في الثانية حيث كانت مدة الاختبار 5 دقائق ،نلاحظ أيضاً نجاح العملية حيث كان الزمن الوسطي المستغرق 108.02ms وعند 95% من الطلبات كان الزمن المستغرق حوال 85ms

execution: local	
script: feed_test.js	
output: -	
scenarios: (100.00%) 1 scenario, 4000 max VUs, 6m30s max duration (incl. graceful stop):	
* spike_test: Up to 175.00 iterations/s for 6m0s over 3 stages (maxVUs: 2000-4000, gracefulStop: 30s)	
TOTAL RESULTS	
checks_total	58049 161.24684/s
checks_succeeded	100.00% 58049 out of 58049
checks_failed	0.00% 0 out of 58049
@ status is 200	
HTTP	
http_req_duration	avg=108.02ms min=323us med=14.01ms max=4.9s p(90)=33.28ms p(95)=85.33ms
{ expected_response:true }	avg=108.02ms min=323us med=14.01ms max=4.9s p(90)=33.28ms p(95)=85.33ms
http_req_failed	0.00% 0 out of 58049
http_reqs	58049 161.24684/s
EXECUTION	
iteration_duration	avg=108.81ms min=323us med=15.24ms max=4.9s p(90)=35.47ms p(95)=86.34ms
iterations	58049 161.24684/s
vus	0 min=0 max=818
vus_max	2000 min=2000 max=2000
NETWORK	
data_received	141 MB 392 kB/s
data_sent	7.7 MB 21 kB/s
running (6m00.0s), 0000/2000 VUs, 58049 complete and 0 interrupted iterations	
spike_test @ [=====] 0000/2000 VUs 6m0s 001.71 iters/s	



- تم تطبيق حمل 200 طلباً في الثانية حيث كانت مدة الاختبار 5 دقائق، نلاحظ أيضاً نجاح العملية استغرق وقت حوالي 853.85ms وهو ضمن الحد المقبول

```

execution: local
  script: feed_test.js
  output: -

scenarios: (100.00%) 1 scenario, 4000 max VUs, 6m30s max duration (incl. graceful stop):
  * spike_test: Up to 200.00 iterations/s for 6m0s over 3 stages (maxVUs: 2000-4000, gracefulStop: 30s)

TOTAL RESULTS
checks_total ..... 66299 184.156915/s
checks_succeeded ..... 100.00% 66299 out of 66299
checks_failed ..... 0.00% 0 out of 66299

status is 200

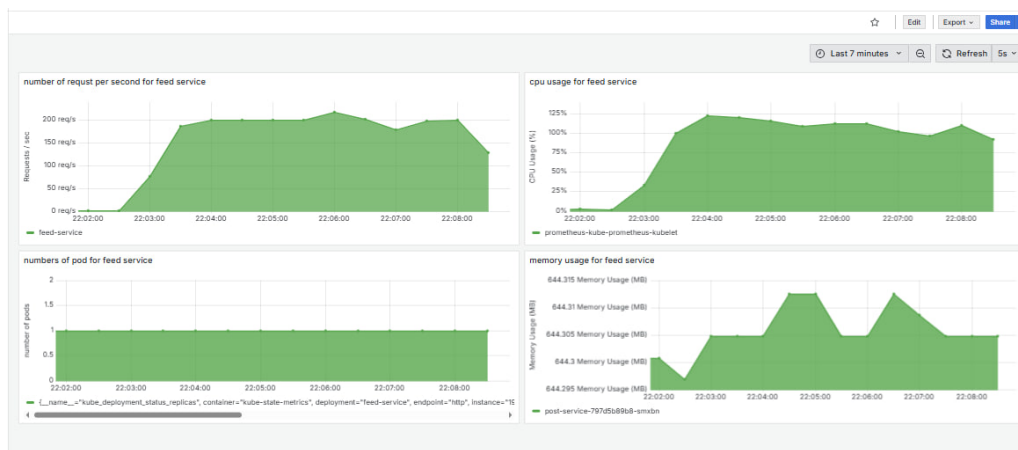
HTTP
http_req_duration ..... avg=853.85ms min=0s med=28.93ms max=49.69s p(90)=3.61s p(95)=4.97s
  { expected response:true }
http_req_failed ..... 0.00% 0 out of 66299
http_reqs ..... 66299 184.156915/s

EXECUTION
iteration_duration ..... avg=854.93ms min=0s med=30.84ms max=49.69s p(90)=3.61s p(95)=4.97s
iterations ..... 66299 184.156915/s
vus ..... 0 min=0 max=1518
vus_max ..... 2000 min=2000 max=2000

NETWORK
data_received ..... 161 MB 448 kB/s
data_sent ..... 8.8 MB 25 kB/s

running (6m00.0s), 0000/2000 VUs, 66299 complete and 0 interrupted iterations
spike_test [=====] 0000/2000 VUs 6m0s 001.83 iters/s

```



- تم تطبيق حمل 250 طلباً في الثانية لمدة 5 دقائق نلاحظ انه استغرق مدة طويلة 1906ms اي حوالي 2s وتم الحصول على عدد من الأخطاء

```
WARN[0320] Request Failed error="Get \"http://172.29.5.41:30085/api/feed?page=0&size=10\": read tcp 192.168.24.63:30278->172.29.5.41:30085: wsarecv: An existing connection was forcibly closed by the remote host."

TOTAL RESULTS
checks_total ..... 82559 229.329963/s
checks_succeeded ..... 99.99% 82558 out of 82559
checks_failed ..... 0.00% 1 out of 82559

status is 200
99% - 82558 / 1

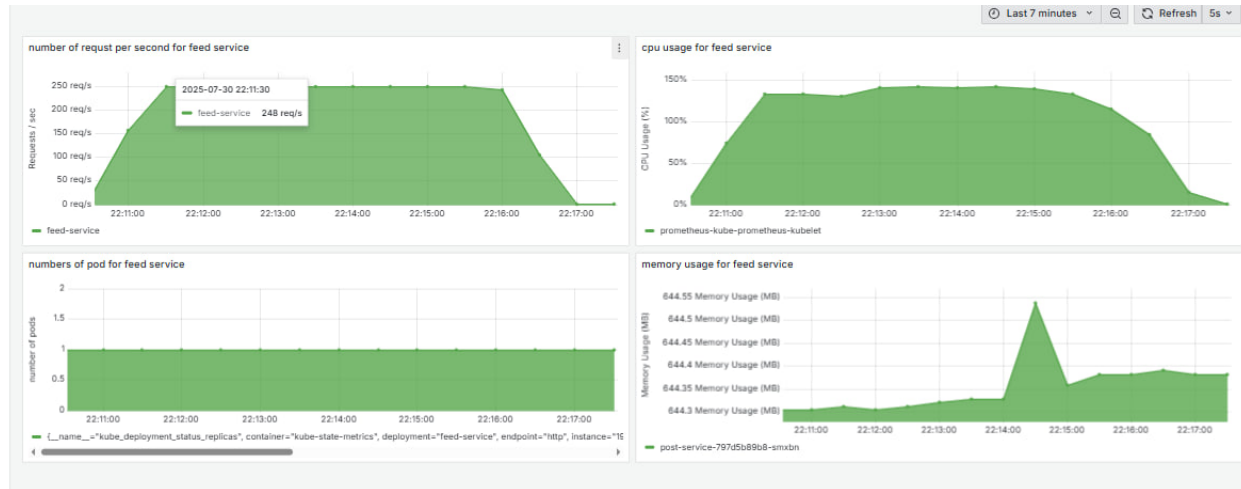
CUSTOM
request_duration ..... avg=1906.895389 min=0 med=60.3487 max=53532.8369 p(90)=6339.0435 p(95)=7600.41916
slow_requests ..... 1376 3.822212/s
total_requests ..... 82559 229.329963/s

HTTP
http_req_duration ..... avg=1.9s min=0s med=60.34ms max=53.53s p(90)=6.33s p(95)=7.6s
{ expected_response:true }
http_req_failed ..... 0.00% 1 out of 82559
http_reqs ..... 82559 229.329963/s

EXECUTION
dropped_iterations ..... 240 0.666665/s
iteration_duration ..... avg=1.02s min=0s med=65.88ms max=53.53s p(90)=6.39s p(95)=7.74s
iterations ..... 82559 229.329963/s
vus ..... 0 min=0 max=2221
vus_max ..... 2236 min=2000 max=2236

NETWORK
data_received ..... 201 MB 557 kB/s
data_sent ..... 11 MB 31 kB/s

running (6m00.0s), 0000/2236 VUs, 82559 complete and 0 interrupted iterations
spike_test [=====] 0000/2236 VUs 6m00s 002.04 iters/s
D:\Fifth year\My-Project\test(deploy)\k6>
```



استنتاج: نلاحظ ارتفاع في استخدام المعالج (CPU) حيث تجاوز نسبة 140%، ما يشير إلى استهلاك عالي للموارد المخصصة للحماية وحدوث حالات خنق في الأداء (CPU Throttling). أما استهلاك الذاكرة (Memory) فقد شهد ارتفاعاً تدريجياً استمر حتى وقوع قفزة مفاجئة تجاوزت 1.2 غيغابايت، مما قد يدل على تحميل ثقيل.

بالرغم من نجاح 99.99% من الطلبات، إلا أن الارتفاع الكبير في زمن الاستجابة يُعد مؤشر واضح على تدهور الأداء عند الحدود القصوى، مما يستدعي إعادة تقييم استراتيجية إدارة الموارد لهذه الخدمة تحديداً.

لهذا السبب ولمعالجة هذه المشكلة، يُوصى باعتماد استراتيجية التوسعة الرأسية التلقائية (Vertical Pod Autoscaler - VPA).

تتيح أداة VPA ضبط قيم requests وlimits تلقائياً لكل من CPU وMemory استناداً إلى بيانات المراقبة الزمنية لاستهلاك الخدمة. وعند تفعيلها، تقوم VPA بتحليل استهلاك الموارد وتحديث الحاوية وفق الاحتياج الفعلي، دون الحاجة لتدخل يدوي أو إعادة نشر يدوية للخدمة.

❖ الاختبار بعد تفعيل VPA

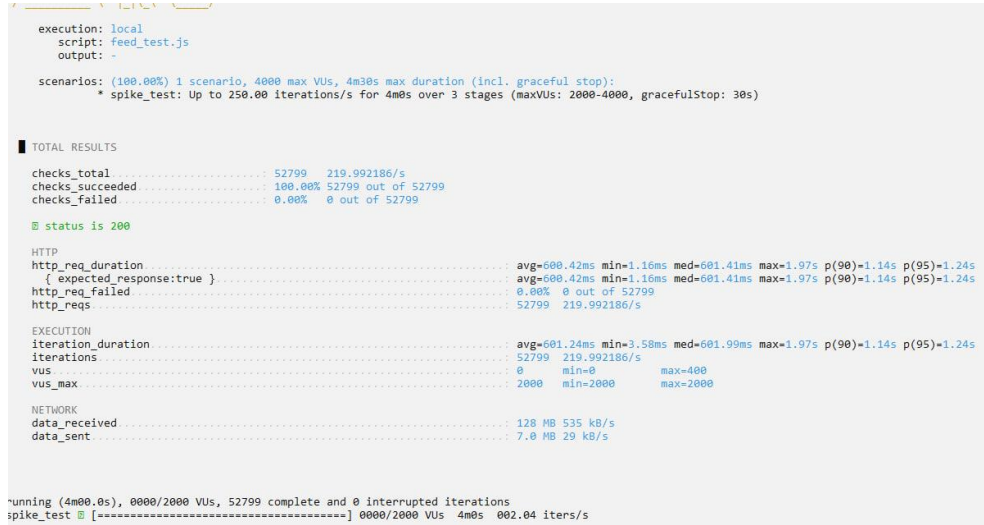
```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: feed-service-vpa
  namespace: rm-photo-app
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: feed-service
  updatePolicy:
    updateMode: "Auto"
  resourcePolicy:
    containerPolicies:
      - containerName: "feed-service"
        controlledValues: "RequestsOnly"
        minAllowed:
          cpu: "200m"
          memory: "256Mi"
        maxAllowed:
          cpu: "700m"
          memory: "1Gi"
---
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: feed-pdb
  namespace: rm-photo-app
spec:
  minAvailable: 1
  selector:
    matchLabels:
      app: feed-service
```

بوضح الشكل السابق كيفية وضع إعدادات التوسع العمودي للخدمة الخلاصة، حيث يتم تحديد حد أدنى وأعلى للموارد التي يجب تخصيصها للخدمة. بعد هذه الإعدادات يقوم تلقائياً بتخصيص الموارد المناسبة للخدمة.

- تم تطبيق حمل 250 طلباً في الثانية، أظهرت النتائج تحسناً ملحوظاً في مؤشرات الأداء الرئيسية؛ حيث انخفض متوسط زمن الاستجابة إلى حوالي 600 ميلي ثانية، كما لوحظ استقرار في استهلاك الموارد؛ حيث بقي استخدام المعالج ضمن نطاق



100% إلى 120% دون ظهور مؤشرات على اختناق في الأداء (CPU Throttling)، في حين حافظت الذاكرة على مستوى ثابت يتراوح بين 645 و650 ميغابايت، مع ذروة مؤقتة طبيعية ضمن منحنى التحميل. هذه النتائج تعكس الدور الفعال الذي أدته VPA في ضبط موارد الحاوية تلقائياً وفق الحاجة، دون الحاجة لتوسعة أفقية أو تدخل يدوي، مما ساهم في استقرار الخدمة وتحقيق الاستجابة المطلوبة تحت الضغط.



عند تطبيق آلية التوسعة الرأسية التلقائية (VPA) على خدمة الخلاصة، لاحظنا تحسناً فعلياً في استجابة الخدمة تحت الضغط، حيث قامت VPA بتعديل الموارد المخصصة لكل Pod بشكل ديناميكي بما يتناسب مع زيادة الحمل، مما انعكس بشكل إيجابي على الأداء الداخلي لكل نسخة من الخدمة. هذا السلوك أتاح لكل Pod أن يعمل بكفاءة أعلى دون الحاجة للتدخل اليدوي في ضبط الموارد.

ورغم أن هذا التحسن كان ملحوظاً، إلا أنه كشف عن حدود هذا النوع من التوسعة، خصوصاً في بيئات الحوسبة السحابية المصممة وفق مبادئ الأنظمة الموزعة. الاعتماد المفرط على رفع قدرة Pod واحد فقط قد يؤدي إلى تركيز المعالجة في نقطة واحدة،

مما يجعل النظام أكثر عرضة للانقطاع. ففي حال توقف العقدة التي تستضيف هذا Pod، يتم فقدان كمية كبيرة من الطاقة الحسابية والطلبات الجارية، الأمر الذي يُضعف من مرونة الخدمة ويؤخر عملية الاسترداد (recovery) ريثما يتم إعادة جدولة ال Pod على عقدة أخرى.

من هذا المنطلق، أصبح من الواضح أن تحسين الموارد الداخلية لا يكفي وحده لمواكبة التغيرات الحادة في عدد الطلبات، بل لا بد من وجود آلية تكاملية تضمن أيضاً زيادة عدد النسخ (Pods) عند الحاجة. ولذلك، تم الانتقال إلى اعتماد استراتيجية توسعة هجينة تجمع بين VPA و HPA، بحيث تتكامل الآليتان معاً: تقوم VPA بضبط الموارد الدقيقة لكل Pod، بينما تتولى HPA مهمة توسيع عدد التكرارات بشكل تلقائي عند ارتفاع مؤشرات الحمل.

### ❖ الاختبار بعد تفعيل HPA

بعد أن أظهرت اختبارات التوسعة الرأسية (VPA) تحسناً في إدارة الموارد داخل كل Pod على حدة، إلا أنها لم تكن كافية لضمان الاستجابة المثلى في حالات الحمل المرتفع والمتغير. لذلك، تم الانتقال إلى مرحلة اختبار جديدة تعتمد على تفعيل آلية التوسعة الأفقية التلقائية (Horizontal Pod Autoscaler - HPA)، بهدف دراسة قدرة النظام على التكيف مع ازدياد الضغط من خلال زيادة عدد النسخ (Pods) ديناميكياً دون تدخل يدوي.

جاء هذا الاختبار استجابة لحاجة أساسية في الأنظمة السحابية الموزعة: تحقيق التوافر المستمر ومقاومة الفشل (High Availability & Fault Tolerance). فعندما تزداد الأحمال، لا يكفي فقط إعطاء موارد إضافية لل Pod الواحد (كما تفعل VPA)، بل لا بد من توزيع الحمل على عدة Pods لضمان التوازن وتحقيق قابلية التوسع الحقيقي.

– تم تطبيق حمل في المرحلة الاولى 150 طلباً في الثانية، ملاحظ أن زمن الاستجابة ممتازة، أي ان النظام يعمل بشكل جيد عند الحمل الطبيعي وباستخدام HPA+VPA. ولم تظهر هناك حاجة للتوسعة الافقية حيث بقي ضمن الحد الطبيعي

TOTAL RESULTS	
checks_total	31949 133.11213/s
checks_succeeded	100.00% 31949 out of 31949
checks_failed	0.00% 0 out of 31949
status is 200	
HTTP	
http_req_duration	avg=36.83ms min=1.05ms med=20.13ms max=1.88s p(90)=73.56ms p(95)=115.51ms
{ expected_response:true }	avg=36.83ms min=1.05ms med=20.13ms max=1.88s p(90)=73.56ms p(95)=115.51ms
http_req_failed	0.00% 0 out of 31949
http_reqs	31949 133.11213/s
EXECUTION	
iteration_duration	avg=38.19ms min=1.19ms med=20.6ms max=1.88s p(90)=76.35ms p(95)=119.52ms
iterations	31949 133.11213/s
vus	0 min=0 max=110
vus_max	2000 min=2000 max=2000
NETWORK	
data_received	78 MB 324 kB/s
data_sent	4.2 MB 18 kB/s

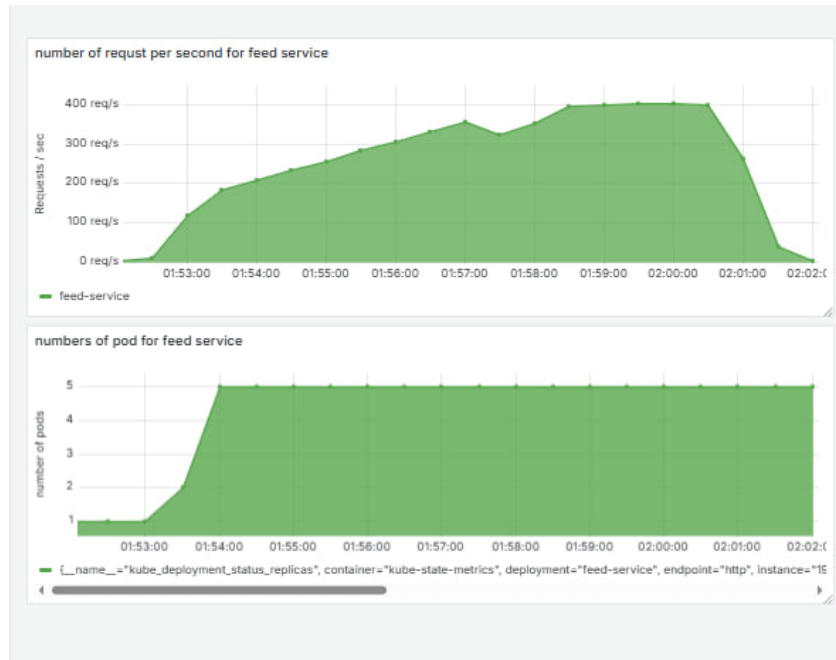
- تم تطبيق حمل 200 طلب بالثانية ف ظهرت الحاجة الى التوسعة الافقية وكان زمن الاستجابة مقبول ايضا

TOTAL RESULTS	
checks_total	42449 176.870269/s
checks_succeeded	100.00% 42449 out of 42449
checks_failed	0.00% 0 out of 42449
status is 200	
HTTP	
http_req_duration	avg=44.39ms min=0s med=17.16ms max=3.7s p(90)=68.69ms p(95)=132.35ms
{ expected_response:true }	avg=44.39ms min=0s med=17.16ms max=3.7s p(90)=68.69ms p(95)=132.35ms
http_req_failed	0.00% 0 out of 42449
http_reqs	42449 176.870269/s
EXECUTION	
iteration_duration	avg=47.22ms min=0s med=17.99ms max=3.7s p(90)=73.17ms p(95)=137.04ms
iterations	42449 176.870269/s
vus	0 min=0 max=209
vus_max	2000 min=2000 max=2000
NETWORK	
data_received	103 MB 430 kB/s
data_sent	5.6 MB 24 kB/s



- تم تنفيذ اختبار تحميل تدريجي على خدمة الخلاصة بعد تفعيل كل من VPA و HPA، حيث بدأ الحمل بـ 150 طلباً في الثانية وارتفع تدريجياً إلى 400 طلب/ثانية خلال 5 دقائق، تلاها فترة استقرار عند الذروة استمرت 3 دقائق. الهدف من هذا السيناريو كان تقييم قدرة النظام على التوسعة التلقائية ومعالجة الطلبات ضمن الزمن المقبول دون انهيار أو تأخير مفرط.

```
stages: [  
  { duration: '1m', target: 200 },  
  { duration: '1m', target: 250 },  
  { duration: '1m', target: 300 },  
  { duration: '1m', target: 350 },  
  { duration: '1m', target: 400 },  
  { duration: '3m', target: 400 },  
  { duration: '30s', target: 0 },  
],
```



نلاحظ في الشكل السابق بلغ عدد التكرارات (Pods) أثناء الاختبار حدّه الأعلى، حيث أظهرت المراقبة توسعة أفقية فعالة وصلت إلى أكثر من 4,000 وحدة تنفيذية (VUs)، بينما قامت VPA بضبط الموارد الداخلية للحفاظ على استقرار الأداء داخل كل Pod. على الرغم من نجاح جميع الطلبات، إلا أن زمن الاستجابة كان مرتفعاً نسبياً في المراحل الأخيرة من الاختبار، ما يدل على أن الخدمة تحتاج إلى توسيع الموارد القصوى المسموح بها ضمن الـ Pods.

```

TOTAL RESULTS
checks_total.....: 159380 312.489084/s
checks_succeeded.....: 100.00% 159380 out of 159380
checks_failed.....: 0.00% 0 out of 159380

 status is 200

CUSTOM
response_time.....: avg=4005.000212 min=0.5164 med=3687.49885 max=19257.2579 p(90)=8634.02 p(95)=9322.66951

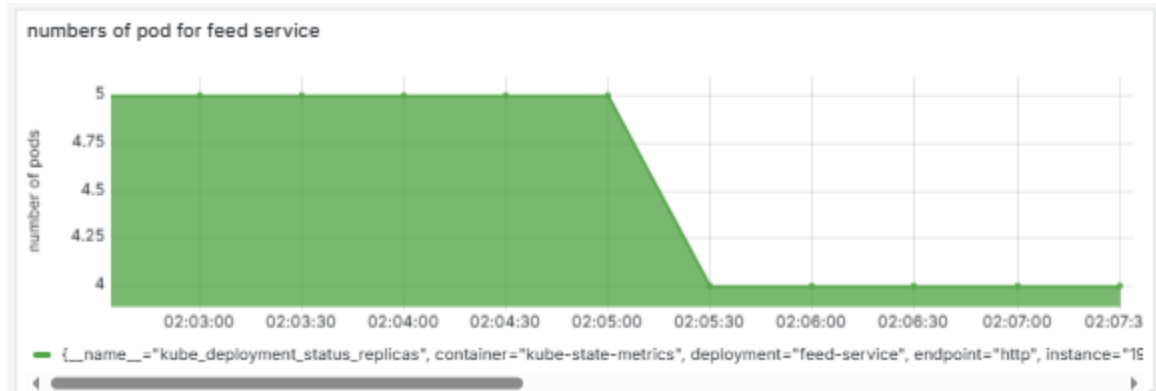
HTTP
http_req_duration.....: avg=4s min=516.4µs med=3.68s max=19.25s p(90)=8.63s p(95)=9.32s
{ expected_response:true }.....: avg=4s min=516.4µs med=3.68s max=19.25s p(90)=8.63s p(95)=9.32s
http_req_failed.....: 0.00% 0 out of 159380
http_reqs.....: 159380 312.489084/s

EXECUTION
dropped_iterations.....: 1119 2.193972/s
iteration_duration.....: avg=4.01s min=565µs med=3.69s max=21.39s p(90)=8.64s p(95)=9.35s
iterations.....: 159380 312.489084/s
vus.....: 0 min=0 max=4014
vus_max.....: 4044 min=3000 max=4044

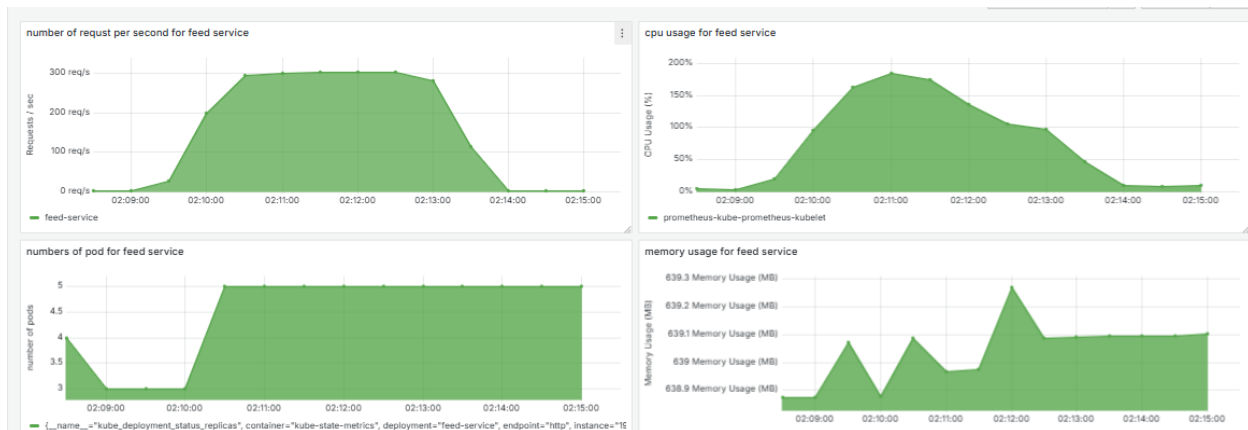
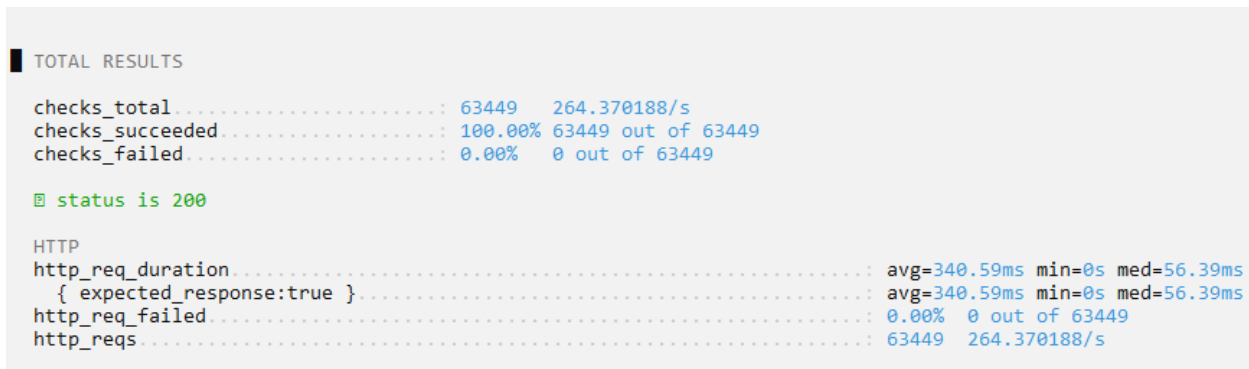
NETWORK
data_received.....: 387 MB 759 kB/s
data_sent.....: 21 MB 42 kB/s

```

نلاحظ انه عند انتهاء عملية الاختبار يتم عملية التراجع وتقليل عدد ال pods



- تم تطبيق حمل 300 طلب في الثانية وتم زيادة عدد ال pods وتم تفعيل vpa وحصلنا على زمن استجابة مقبول



## المراجع

- [1] S. Newman, "BUILDING MICROSERVICES: designing fine-grained systems," O'Reilly Media, 2018.
- [2] M. Richards, Software Architecture Patterns, O'Reilly Media, 2015.
- [3] Microsoft, "Microservices architecture style," 2023. [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices>. [Accessed 2025].
- [4] Microsoft, "Why a microservices architecture?," Microsoft Learn, 2023. [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices>. [Accessed 2025].
- [5] Atlassian, "5 Advantages of Microservices [+ Disadvantages]," 2024. [Online]. Available: <https://www.atlassian.com/microservices/cloud-computing/advantages-of-microservices>.
- [6] ValueCoders, "Microservices: Scalable and Flexible Software Development," 2023. [Online]. Available: <https://www.valuecoders.com/blog/software-engineering/microservices-scalability-flexibility-software-development/>.
- [7] F. Networks, "Building Microservices Using an API Gateway," 2020. [Online]. Available: <https://www.f5.com/company/blog/nginx/building-microservices-using-an-api-gateway>.
- [8] Microsoft, "Circuit Breaker pattern," [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker>.
- [9] A. K.M., " Microservices Architecture Patterns: Exploring the Essential," 2023. [Online]. Available: <https://medium.com/@alxkm/microservices-architecture-patterns-exploring-the-essential-27318b72c88f>.
- [10] C. O. Tech, "10 Microservices Design Patterns for Better Architecture," 2020. [Online]. Available: <https://medium.com/capital-one-tech/10-microservices-design-patterns-for-better-architecture-befa810ca44e>.
- [11] Microsoft, "CQRS pattern," [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs>.

- [12] Microsoft, "Materialized View pattern," [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/patterns/materialized-view>.
- [13] Microsoft, "Event-driven architecture style," [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven>.
- [14] Microsoft, "Event Sourcing pattern," [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing>.
- [15] Microsoft, "Queue-Based Load Leveling pattern," [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling>.
- [16] A. S. Foundation, " Apache Kafka Documentation," 2024. [Online]. Available: <https://kafka.apache.org/documentation/>.
- [17] M. Inc, "MongoDB Manual," 2024. [Online]. Available: <https://www.mongodb.com/docs/manual/>.
- [18] R. Ltd, "Redis Documentation," 2024. [Online]. Available: <https://redis.io/docs/>.
- [19] F. Inc., "NGINX Documentation," 2024. [Online]. Available: <https://docs.nginx.com/>.
- [20] D. Inc., "Docker Documentation," 2024. [Online]. Available: <https://docs.docker.com/>.
- [21] D. Merkel, "Docker: lightweight Linux containers for consistent development and deployment," *Linux Journal*, 2014.
- [22]