ChatGPT

التوافرية العالية والموثوقية

(1) المفاهيم النظرية الأساسية

التوافرية العالية (High Availability) : يقصد بها قدرة النظام أو التطبيق على العمل بشكل مستمر دون انقطاع ولفترة زمنية طويلة، حتى في حالة حدوث أعطال في بعض مكوناته [1] . عادةً ما يتم التعبير عن مستوى التوافرية كنسبة زمن التشغيل (Uptime) المستهدف، مثل تحقيق توافرية 99.999% (خمسة تسعات) مما يعني عدم تجاوز بضع دقائق توقف في السنة. يعتمد تحقيق التوافرية العالية على إزالة نقاط الفشل المفردة وتصميم بنية تحتية قادرة على التعامل مع الأحمال المختلفة والأعطال مع تقليل زمن التعطل للحد الأدنى [2] . على سبيل المثال، في نظام عالي التوافر يتم استخدام عنقود خوادم (Cluster) بحيث إذا فشل خادم في الكتلة، يتولى خادم آخر المهمة فورًا للحفاظ على استمرارية الخدمة دون انقطاع، مما يضمن عدم وجود نقطة فشل مفردة تؤدي لتوقف النظام [3] .

الموثوقية (Reliability) : يُشير هذا المفهوم إلى مدى قدرة النظام على أداء الوظيفة المطلوبة بشكل صحيح ومتسق دون فشل خلال فترة زمنية محددة [4] . بمعنى آخر، النظام الموثوق هو الذي يعمل بصورة صحيحة باستمرار ويُنتج المخرجات المتوقعة دون أخطاء أو أعطال متكررة. تختلف الموثوقية عن التوافرية في أن الموثوقية تُركز على ضمان صحة الأداء وندرة حالات الفشل، بينما التوافرية تُركز على بقاء النظام متاحًا وجاهزًا للعمل. على سبيل المثال، قد يكون النظام "متاحًا" (أي قيد التشغيل)، ولكنه غير "موثوق" إذا كان عرضة للأخطاء أو لا يحقق النتائج المطلوبة بشكل صحيح. لذا يمكن القول: التوافرية تُقاس بنسبة زمن التشغيل، أما الموثوقية فتُقاس باحتمالية أداء النظام لوظيفته بشكل صحيح في وقت معين [5] . لتحقيق موثوقية عالية، يهتم المهندسون بمراقبة معدلات الفشل (Failure Rate) وزيادة متوسط الزمن بين الأعطال (MTBF) عبر الصيانة الدورية وتحسين جودة المكونات والبرمجيات.

التكرارية (Redundancy) : يُقصد بها وجود مكونات بديلة أو إضافية تؤدي نفس وظيفة المكونات الأساسية في النظام. الهدف من التكرار هو تجنب الاعتماد على عنصر واحد فقط، وبالتالي إزالة "نقطة الفشل المفردة" (Single Point of Failure). في البنى عالية التوافر، يتم تصميم كل عنصر حرج في النظام بشكل مزدوج أو أكثر، بحيث إذا تعطل أحدها يقوم الآخر مقامه [6] . على سبيل المثال، يمكن توفير خادمين يعملان بالتوازي لنفس المهمة، أو اثنتين من وحدات التزويد بالطاقة (Power Supplies) في خادم واحد، لضمان أنه إذا تعطلت وحدة فإن الأخرى تستمر في العمل. التكرارية تعتبر عنصرًا أساسيًا لتحقيق Fault Tolerance (تحمّل الخطأ) ورافدًا لتحقيق كل من التوافرية العالية والموثوقية، لأنها تضمن استمرار الخدمة حتى عند فشل إحدى المكونات.

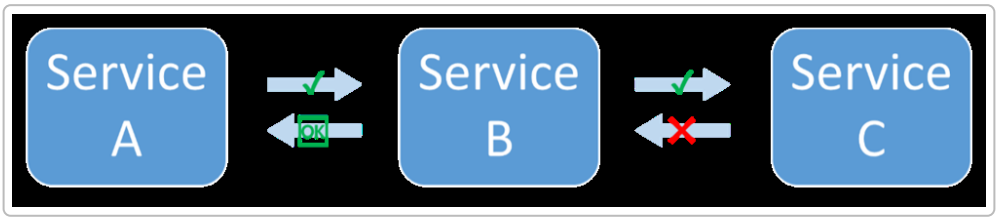
التحويل الاحتياطي (Failover): آلية **التحويل الاحتياطي** تعني الانتقال التلقائي إلى نظام بديل أو مكوّن احتياطي عند فشل المكوّن الرئيسي. فعندما يتوقف العنصر الأساسي عن العمل، تنتقل المهام والعمليات إلى العنصر الاحتياطي أو الخادم الثانوي دون تدخل يدوي وبشكل سريع [7] . غالبًا ما يُستخدم نظام **النشط-الاحتياطي (Active-Passive)** في هذه الحالات، حيث يكون لدينا خادم أساسي نشط يقوم بكل العمل الاعتيادي، وخادم آخر ثانوي خامد (Standby) جاهز ولكن لا يعمل إلا عند حصول مشكلة في الخادم الأساسي. عند اكتشاف فشل في الأساسي، يتم **تحويل** العمليات فورًا إلى الخادم الاحتياطي (أي يصبح هو النشط) لضمان استمرار الخدمة. هذه العملية يجب أن تحدث بسرعة بحيث لا يلاحظ المستخدمون انقطاعًا ملموسًا. يوصى دائمًا أن يكون نظام التحويل الاحتياطي خارج موقع النظام الأساسي (Off-premises) إن أمكن، لضمان استمرار الخدمة حتى في حال الكوارث المحلية. بعد عودة النظام الأساسي للعمل، قد يتم **إعادة التحويل (Failback)** للعودة إلى الوضع الأصلي خلال نافذة صيانة مجدولة.

التدهور التدريجي للخدمة (Graceful Degradation) : يُقصد به تصميم النظام بحيث إذا تعرض لبعض الأعطال أو الضغط الزائد، فإنه لا يتوقف كليًا عن العمل، بل يقوم **بتخفيض مستوى الخدمات** بشكل متدرج مع الحفاظ على

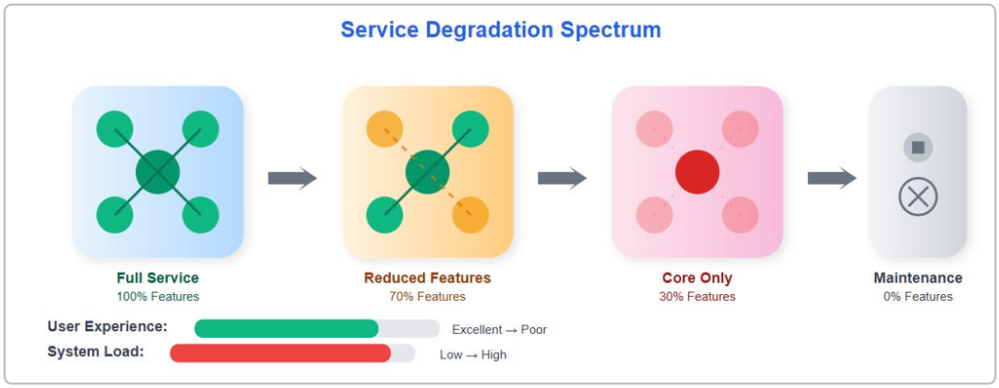
الوظائف الأساسية. أي أن النظام "يتدهور بشكل سلس" بدل أن ينهار بالكامل. على سبيل المثال: قد يقوم موقع تجارة إلكترونية تحت حمل زائد بإيقاف ميزات غير أساسية كعرض توصيات المنتجات أو التخصيص الشخصي للمحتوى، مع إبقاء وظيفة التسوّق وإتمام الطلبات متاحة [8]. كلما زاد الضغط أو استمرت المشاكل، يمكن تعطيل المزيد من الميزات الثانوية (مثل إخفاء قسم المراجعات والتقييمات) والاقتصار على الجوهر الأساسي للخدمة. الهدف هو **تقليل تأثير العطل على المستخدمين**؛ فبدلًا من حصول انقطاع كامل، يحصل المستخدم على تجربة مخففة لكنها ما زالت تعمل وتلبي الحد الأدنى من الاحتياجات. مثال آخر: في حال فشل إحدى الخدمات الخلفية (Microservice) التي تضيف وظيفة غير حرجة للتطبيق، يمكن للنظام أن يعرض بيانات مخزنة مؤقتًا (Cache) أو رسالة افتراضية بدلًا من تعطيل الصفحة بأكملها [9] [10]. بهذا الشكل، "ينحني" النظام تحت الضغط لكنه لا "ينكسر"، مما يمنح المطوّرين فرصة لإصلاح المشكلة بينما يبقى النظام قيد الخدمة (وإن بقدرات أقل).

(2) صور توضيحية للمفاهيم

رسم 1: نظام بنقطة فشل مفردة مقابل نظام ذو مكونات زائدة عن الحاجة. يظهر الجزء الأيسر مخططًا فيه نقطة فشل مفردة (خادم واحد يمثل عنق الزجاجة لجميع الاتصالات)، حيث تعني أي مشكلة في هذا الخادم توقّف النظام بأكمله. أما الجزء الأيمن فيبيّن إضافة التكرار (Redundancy) من خلال وجود خادم بديل يشارك في الخدمة. في الحالة اليمنى، إذا تعطل أحد الخوادم، يستمر الآخر في العمل مما يمنع توقف النظام [3] [6]. هذا المثال يوضح كيف أن تصميم النظام بتكرار المكونات يزيل نقاط الفشل الوحيدة ويزيد من الاعتمادية.



رسم 2: بنية تحويل احتياطي Active-Passive. يُوضح المخطط خادمين حيث أحدهما أساسي نشط يقوم بالعمل حاليًا، والآخر ثانوي في وضع الاستعداد. يقوم نظام المراقبة (Heartbeat) بمراقبة الخادم النشط باستمرار. عند اكتشاف فشل في الخادم الأساسي (X حمراء في الرسم)، يتم تحويل جميع الطلبات تلقائيًا إلى الخادم الاحتياطي الذي يصبح نشطًا ويتولى الخدمة (الأسهم الخضراء) [11] [12]. يضمن هذا السيناريو استمرار الخدمة دون انقطاع كبير، حيث يلاحظ المستخدم فقط تأخيرًا طفيفًا أثناء الانتقال. يستخدم هذا النمط كثيرًا مع قواعد البيانات أو التطبيقات الحرجة بحيث يكون هناك خادم قاعدة بيانات أساسي وآخر مُحدّث آنياً يكون جاهزًا لاستقبال الدور عند سقوط الأساسي.



رسم 3: التدهور التدريجي في الخدمات عند الضغط العالي. يبين المخطط مفهوم degrade gracefully حيث يمثل المحور الأفقي زيادة الحمل أو المشاكل، والمحور العمودي مقدار الوظائف المتاحة. في البداية تكون جميع الخدمات فعّالة (Full Service). عند زيادة الضغط يتم إيقاف بعض الخصائص غير الأساسية (Reduced Features) مثل توصيات

المنتجات أو ميزات مخصصة للمستخدم. إذا ازداد الضغط أكثر يتم الإبقاء فقط على النواة الأساسية للخدمة (Core Only) مثل الوظائف الأساسية لإتمام العمليات. في أسوأ الحالات قد يتحول النظام إلى وضع الصيانة (Maintenance) حيث تُعرض رسالة للمستخدم بأن بعض الخدمات غير متاحة بشكل مؤقت [13] . بهذا الشكل يتم تقليص الوظائف تدريجيًا للحفاظ على استمرارية النظام بدل انهياره بالكامل.

(3) أمثلة برمجية واقعية باستخدام Laravel

فيما يلي أمثلة عملية على تطبيق المفاهيم أعلاه في إطار عمل Laravel (بلغة PHP):

- **استخدام Redis لجعل الجلسات (Sessions) والتخزين المؤقت (Cache) غير مرتبطين بخادم محدد**: تعتمد تطبيقات الويب عادة على الجلسات لتخزين بيانات المستخدم عبر الطلبات المختلفة. إذا تم تخزين الجلسات في ملفات على قرص الخادم المحلي (الوضع الافتراضي)، فهذا يربط المستخدم بذلك الخادم. لمعالجة ذلك ولتحقيق **نشر stateless** دون اعتماد على خادم بعينه، يمكن استخدام مخزن مركزي مثل Redis للجلسات والكاش. في Laravel يتم ذلك بتعديل إعدادات **driver** لكل من الجلسات والتخزين المؤقت إلى Redis. مثلًا في ملف البيئة `.env` نعيّن:

```
SESSION_DRIVER=redis
CACHE_DRIVER=redis
```

وبالطبع يجب إعداد اتصال Redis في ملف الإعدادات `config/database.php` أو `config/cache.php` بما يناسب بيئتك. عند استخدام Redis كمخزن مركزي، أي خادم تطبيق يمكنه الوصول لبيانات الجلسة أو الكاش نفسها. هذا يضمن أنه إذا تم توجيه مستخدم إلى خادم آخر (مثلاً خلف Load Balancer موزع الأحمال)، سيظل قادرًا على متابعة العمل دون فقدان حالته session. **مثال توضيحي**: الكود التالي من إعدادات البيئة يبيّن تفعيل Redis للجلسات:

```
REDIS_HOST="..."        # إعدادات اتصال Redis
REDIS_PASSWORD="..."
...
SESSION_DRIVER="redis"  # تفعيل تخزين الجلسات على Redis
CACHE_DRIVER="redis"    # تفعيل تخزين الكاش على Redis
```

(المثال مقتبس من إعدادات Laravel باستخدام خدمة Upstash Redis [14]). بعد هذا التعديل، سيقوم Laravel بتخزين بيانات الجلسات وبيانات التخزين المؤقت في Redis. ونتيجة لذلك، يصبح التطبيق **عديم الحالة فعليًا (stateless)** من منظور الخوادم، مما يسمح بتشغيل نسخ متعددة من التطبيق خلف موزع أحمال دون القلق من ثبات الجلسات على خادم معين.

- **إعداد نسخ متماثل لقاعدة البيانات (Database Replication) بين خادم أساسي وخادم مستنسَخ**: لتحقيق موثوقية أعلى وتوفّر مستمر للبيانات، كثيرًا ما يتم إعداد قاعدة بيانات احتياطية كمستنسخة (Replica أو Secondary) تتلقى تحديثات بشكل لحظي من القاعدة الأساسية (Primary). في Laravel، يمكن تعريف ذلك بسهولة في ملف `config/database.php`. على سبيل المثال، إعداد اتصال MySQL بحيث يكون هناك خادم للقراءة (Replica) وخادم للكتابة (Primary):

```
'mysql' => [
    'read' => [
```

```
        'host' => env('DB_REPLICA_HOST', '192.168.1.2'), // خادم النسخ للقراءة
    ],
    'write' => [
        'host' => env('DB_HOST', '192.168.1.1'), // خادم الكتابة الأساسي
    ],
    'driver'    => 'mysql',
    'database'  => 'dbname',
    'username'  => 'dbuser',
    'password'  => 'dbpass',
    // بقية إعدادات الاتصال ...
],
```

في هذا المثال، أي استعلام قراءة (SELECT) سيُوجَّه تلقائيًا إلى المضيف 192.168.1.2 (النسخة)، بينما استعلامات الكتابة (INSERT/UPDATE) تذهب إلى 192.168.1.1 (الرئيسي) [15]. يقوم Laravel بهذا التوزيع تلقائيًا بمجرد تعريف مصفوفتي `read` و`write`. إذا تعطّل الخادم الأساسي، فإن Laravel سيحاول الاتصال تلقائيًا بالخادم المستنسخ (إن كان مهيّأً) ليحل محله، مما يقلل من فترة توقف التطبيق عن العمل [16]. بذلك يتحقق نوع من **الفشل الآمن**: قراءة البيانات تستمر من النسخة، وفي حال الحاجة يمكن ترقية (Promote) النسخة لتصبح أساسية للكتابة أيضًا.

- **مثال Job في Laravel يستخدم واجهة ShouldQueue مع آلية إعادة المحاولة (Retries) ومعالجة الفشل**: تعتمد Laravel على نظام طوابير مهام (Queue) للتعامل مع العمليات الخلفية بشكل غير متزامن لضمان استجابة سريعة في واجهة المستخدم. عندما نقوم بإنشاء **وظيفة مؤجلة** (Queued Job)، غالبًا ما نرغب في إعادة المحاولة تلقائيًا إذا فشلت المهمة، وكذلك التعامل مع الحالة التي تستنفد فيها كل المحاولات. في Laravel يمكن تحديد عدد المحاولات القصوى بتحديد خاصية `$tries` داخل صنف المهمة، كما يمكن تعريف دالة `failed` التي تستدعى بعد فشل المهمة نهائيًا (بعد استنفاد المحاولات). المثال التالي يظهر فئة Job بسيطة:

```
namespace App\Jobs;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;
use Throwable;

class ProcessPayment implements ShouldQueue
{
    use Dispatchable, Queueable, InteractsWithQueue, SerializesModels;

    public $tries = 5;            // عدد المحاولات قبل الفشل النهائي
    public $maxExceptions = 3;    // أقصى عدد استثناءات مسموحة قبل اعتبار المهمة فاشلة

    public function __construct(/* بيانات المهمة، مثلاً معرف العملية */) {
        // ...
    }
```

```
    public function handle() {
        منطق تنفيذ المهمة (مثلاً معالجة دفعة مالية) //
        // ...
        إذا حدث استثناء هنا ولم يُلتقط، سيقوم النظام بإعادة المحاولة تلقائيًا //
    }

    public function failed(?Throwable $exception = null) {
        هذا الأسلوب يستدعى عندما تفشل المهمة بعد كل المحاولات //
        يمكن هنا إرسال تنبيه لمشرف النظام أو اتخاذ إجراءات تعويضية //
        مثال: تسجيل الرسالة أو إشعار الفريق بالفشل //
        logger()->error("فشل نهائي في معالجة الدفع رقم X:", ['error' => $exception-
>getMessage()]);
    }
}
```

في هذا المثال، حددنا `$tries = 5` مما يعني أن Laravel سيعيد محاولة تنفيذ المهمة خمس مرات كحد أقصى قبل اعتبارها فاشلة [17] [18]. كما حددنا `$maxExceptions = 3` بحيث إذا حدثت 3 استثناءات غير معالجة خلال المحاولات، تتوقف المحاولات مبكرًا حتى لو لم نصل للحد الأقصى للمحاولات (هذا مفيد لتجنب تكرار الفشل السريع في حالات خطأ حرجة) [19] [20]. أخيرًا، أسلوب `failed` يقوم بمعالجة حالة الفشل النهائي (مثلاً تسجيل الخطأ أو إرسال بريد إشعار). إطار العمل Laravel يمرر كائن الاستثناء الأخير إلى هذا الأسلوب إذا كان الفشل ناتجًا عن استثناء أثناء التنفيذ [21] [22]، أما إذا كان الفشل بسبب استنفاد المحاولات بدون استثناء محدد، فسيكون `$exception` من نوع `Illuminate\Queue\MaxAttemptsExceededException`. باستخدام هذه الخواص والأسلوب `failed` نضمن أن التطبيق يتعامل بذكاء مع فشل المهام الخلفية: يحاول تلقائيًا عدة مرات، وإن لم ينجح بعد العدد المحدد من المحاولات فإنه لا يختفي بصمت بل يمكننا تتبع ومعالجة الموقف (كتسجيل المشكلة أو تنبيه فريق الدعم الفني).

- **استخدام Feature Flags لتفعيل/إيقاف ميزات في النظام حسب الحاجة**: خاصية الأعلام البرمجية أو مفاتيح الميزات (Feature Flags) تسمح بتشغيل أو إطفاء أجزاء معينة من التطبيق بشكل ديناميكي دون الحاجة لنشر جديد للكود. هذا يفيد في سياق التوافرية والموثوقية بعدة طرق، منها: تعطيل ميزات غير حرجة عند ارتفاع الحمل (كشكل من أشكال التدهور التدريجي)، أو تعطيل ميزة تسبب مشاكل لحين إصلاحها، أو طرح ميزة جديدة تدريجيًا لبعض المستخدمين. في Laravel يمكن استخدام حزمة **Pennant** الرسمية لإدارة Feature Flags. على سبيل المثال، نفرض أن لدينا ميزة ثقيلة نريد إيقافها مؤقتًا عند الضغط العالي اسمها `heavyReports`. نستطيع تعريف العلم وإعداد حالته (مثلاً عبر لوحة تحكم أو إعداد):

```
use Illuminate\Support\Facades\Feature;

مثلا أثناء الإقلاع يتم تعريف الحالة الافتراضية للعلم //
Feature::define('heavy-reports', false);
```

ثم في كود التطبيق نغلّف الجزء الخاص بهذه الميزة بشرط يفحص حالة العلم:

```
if (Feature::active('heavy-reports')) {
    تنفيذ ميزة التقارير الثقيلة //
    generateHeavyReports($data);
} else {
```

```
    // تخطي تنفيذ هذه الميزة لأن العلم مطفأ - ربما نقوم بخطوة بديلة أبسط
}
```

في المثال أعلاه، سيقوم التطبيق بتنفيذ التقارير الثقيلة فقط إذا كان العلم مفعّل (true). يمكن تغيير حالة العلم إلى false أثناء التشغيل بسهولة (مثلاً عن طريق أمر Artisan أو من خلال واجهة الإدارة)، وبذلك يتم إيقاف الميزة فورًا دون الحاجة لإعادة نشر الكود. توفر حزمة Pennant واجهات متعددة للفحص، مثل `Feature::active` أو `Feature::inactive` وحتى إمكانية ربط العلم بمستخدم معين أو نسبة من المستخدمين (للتشغيل التجريبي). في سياق الموثوقية، هذه الأعلام توفر **Kill Switch** سريع لأي جزء من النظام عند الاشتباه بأنه يسبب مشكلة؛ فبدلاً من توقف النظام بأكمله، نعطّل فقط الميزة المشكلة ونبقي بقية النظام عاملاً. (في الشفرة أعلاه، استخدمنا طريقة `Feature::active('heavy-reports')` للفحص الشرطي [23]).

(4) قصة واقعية: سيناريو نظام معاملات مالية

لنفترض أن لدينا نظامًا لتحويل الأموال إلكترونيًا (مشابه لتطبيقات البنوك أو المحافظ الرقمية). هذا النظام يتألف من واجهة مستخدم (تطبيق جوال/ويب) وخدمة خلفية للتعاملات المالية وقاعدة بيانات لحفظ أرصدة العملاء والمعاملات، بالإضافة إلى خدمات مساندة (مثل خدمة إرسال إشعارات أو خدمة التقارير).

وضع التصميم الاعتيادي : تم تصميم النظام بحيث يكون عالي التوافر وموثوق: - تم نشر التطبيق الخلفي على عدة خوادم تطبيق تعمل خلف موزع أحمال لضمان التوافرية حتى لو تعطل أحد الخوادم. - قاعدة البيانات تم إعدادها بأسلوب النسخ المتماثل Master/Replica. القاعدة الأساسية (Primary) تخدم عمليات الكتابة والتحديث، ولديها خادم Replica مُزامن يستقبل نسخًا حية من البيانات لاستخدامها في القراءة أو كنسخة احتياطية. - توجد طوابير للمهام (مثل عمليات إرسال البريد أو التسويات المؤجلة) حتى لا تتأثر واجهة المستخدم بزمن تنفيذ هذه المهام، مع آلية لإعادة المحاولة كما في المثال أعلاه لضمان إتمام المهام eventually حتى لو فشلت من المرة الأولى. - النظام مزوّد برمجيات مراقبة (Monitoring) تتحقق من صحة المكونات باستمرار، وبأعلام ميزات تمكننا من إيقاف أجزاء من الوظائف عند الحاجة.

ماذا يحدث عند سقوط قاعدة البيانات الأساسية؟ في منتصف يوم العمل وأثناء ضغط العمليات، تعطلت قاعدة البيانات الأساسية بشكل غير متوقع. في نظام تقليدي غير محصن، هذا يعني توقف كافة الخدمات التي تعتمد على قاعدة البيانات، وبالتالي توقف التطبيق كاملًا عن العمل (فشل شامل). لكن في نظامنا المصمم عالي التوافر، يتم تفعيل خطط الطوارئ التالية: 1. **التحويل التلقائي إلى القاعدة الاحتياطية** : بفضل إعداد **الفشل التلقائي (Failover)** ، سيكتشف النظام (عبر برامج المراقبة أو عن طريق محاولة الاتصال الفاشلة) أن القاعدة الأساسية لا تستجيب. فورًا سيقوم برنامج إدارة قاعدة البيانات أو تطبيقنا نفسه بترقية (Promote) قاعدة البيانات الـReplica لتصبح هي الأساسية الجديدة للتعامل مع كل من عمليات القراءة والكتابة. Laravel مثلًا سيحاول تلقائيًا استخدام اتصال القراءة (الذي كان Replica سابقًا) عندما يفشل اتصال الكتابة الأساسي [16] . هذا التحويل يحدث خلال ثوان معدودة. ربما قد يلحظ بعض المستخدمين تأخيرًا بسيطًا أو فشلًا في طلب واحد، لكن بسرعة يعود النظام لاستقبال الطلبات. 2. **تفعيل وضع "القراءة فقط" مؤقتًا** (اختياري حسب السيناريو): في بعض الأنظمة المالية الحساسة، قد يُقرر المسؤولون وضع النظام في حالة وظائف محدودة أثناء حدوث خلل كبير في قاعدة البيانات. مثلاً يمكن السماح للمستخدمين بعرض أرصدتهم وتحويلاتهم السابقة (عمليات قراءة) بينما يتم تعطيل مؤقتًا عمليات التحويل الجديدة أو السحب (عمليات كتابة) إلى أن يتم التأكد أن القاعدة الجديدة تعمل بشكل صحيح ومتزامن مع القديمة. هذا يشابه مفهوم **التدهور التدريجي** حيث نحافظ على الخدمات الأساسية (الاستعلام عن الرصيد) لكن نحد من الوظائف ذات الخطورة (إجراء معاملة مالية جديدة) إلى أن يستقر الوضع. 3. **إرسال تنبيهات وإشعارات** : نظام المراقبة سيرسل تنبيهًا لفريق الدعم بأن الخادم الأساسي للقاعدة تعطل وتم الانتقال إلى الخادم البديل. قد يقوم الفريق التقني باتخاذ إجراءات لإصلاح الخادم المتعطل وإعادته كسلسلة Replica من جديد (إعادة المزامنة) بعد حل المشكلة.

خلال هذا السيناريو، يظل المستخدمون قادرين على استخدام التطبيق لإجراء معظم العمليات . ربما لو حاول أحدهم إجراء تحويل مالي في اللحظات الأولى لسقوط القاعدة الأساسية، فشل الطلب، ولكن النظام يتدارك الأمر بسرعة: إما بإعادة

المحاولة تلقائيًا عبر الطابور queue، أو بإعلام المستخدم بمحاولة مرة أخرى بعد قليل. بفضل وجود قاعدة بيانات مستنسخة جاهزة، واستراتيجية التحويل السلس، فإن النظام يستعيد قدراته الكاملة خلال وقت قصير جدًا. المستخدم العادي ربما لم يلاحظ أي فرق سوى بطء طفيف.

"النظام شغّال" مقابل "النظام موثوق" : هذا السيناريو يساعدنا أيضًا في توضيح الفرق بين مجرد كون النظام شغالًا (Up and Running) وبين كونه موثوقًا (Reliable) بالفعل. قد يكون لديك نظام **يعمل** (أي خوادمه شغالة) لكنه ليس موثوقًا إذا كان عرضة للتعطل عند أي مشكلة صغيرة أو إذا كان لا يحتوي على ضمانات لاستمرار العمليات بشكل سليم. النظام الموصوف أعلاه موثوق لأنه حتى عند تعطل جزء أساسي (قاعدة البيانات)، لم يفشل في أداء مهمته الأساسية بشكل كامل بل وجد طرقًا للتعامل (قاعدة بديلة، تحويل مهام، الخ). النظام الموثوق يعني أن لدينا ثقة بأنه سيؤدي المطلوب منه تحت الظروف المختلفة باستمرار. أما مجرد كون النظام شغال حاليًا فلا يكفي إن لم يكن لديه القدرة على تحمل الأعطال المفاجئة. بشكل مختصر: الموثوقية تعني **الاستمرارية في الأداء الصحيح** مع مرور الوقت وفي مواجهة الظروف، بينما مجرد التوفر الحالي يعني أنه لا توجد أعطال في هذه اللحظة فقط.

في مثالنا، نظام التحويلات المالية عالي التوافر تمكّن من تجنب توقف الخدمة للمستخدمين، وهذا دليل على موثوقيته. أما لو كان النظام "just up" بدون هذه التحصينات، لانقطعت الخدمة تمامًا عن المستخدمين بمجرد سقوط قاعدة البيانات، وبذلك يخسر ثقة المستخدمين وتتعطل الأعمال حتى يعود للعمل.

(5) قائمة مصطلحات أساسية (إنجليزي – عربي)

- **High Availability (HA)** – التوافرية العالية: قدرة النظام على تقديم خدماته بشكل مستمر ودون انقطاع لفترات طويلة حتى أثناء الأعطال الجزئية.
- **Reliability** – الموثوقية: مدى اعتمادية النظام وقدرته على أداء وظائفه بشكل صحيح ومطرد دون فشل عبر الزمن.
- **Redundancy** – التكرار / الازدواجية: وجود عناصر مكررة في النظام تؤدي نفس الوظيفة لضمان استمرار العمل عند تعطل أي منها.
- **Single Point of Failure (SPOF)** – نقطة فشل مفردة: مكون مفرد لو تعطّل يؤدي إلى انهيار النظام ككل (يجب تجنب وجود مثل هذه النقاط عبر إضافة التكرار).
- **Failover** – التحويل الاحتياطي التلقائي: آلية التحويل إلى نظام بديل أو مكوّن احتياطي عند حدوث عطل في النظام الأساسي للحفاظ على استمرارية الخدمة.
- **Graceful Degradation** – التدهور التدريجي للخدمة: استراتيجية لتخفيض بعض الوظائف غير الضرورية عند حدوث مشكلة أو ضغط شديد، بهدف إبقاء الحد الأدنى من الخدمة عاملاً بدل التوقف الكامل.
- **Stateless** – عديم الحالة: وصف لتصميم تطبيق لا يحتفظ ببيانات الجلسة على الخادم نفسه، مما يسمح بتوزيع الحمل على عدة خوادم بسهولة (أي أن حالة المستخدم مخزنة خارجيًا مثلًا في قاعدة بيانات أو كاش مركزي).
- **Feature Flag** – علم الميزة: مفتاح برمجي يسمح بتفعيل أو تعطيل وظيفة معينة في التطبيق ديناميكيًا بدون الحاجة لتعديل الكود الأساسي أو إعادة نشره.

(6) مساحة للملاحظات والتقييم الذاتي

هذه المساحة مخصصة لتدوين ملاحظاتك الخاصة أو كتابة ملخصك وتقييمك الذاتي لفهم المادة:

- ملاحظة 1: .....
- ملاحظة 2: .....
- سؤال للتقييم الذاتي: .....

What Is High Availability? - Cisco 7 6 3 2 1
https://www.cisco.com/site/us/en/learn/topics/networking/what-is-high-availability.html

Reliability vs. Availability: Key Metrics for System Perform | Atlassian 5 4
https://www.atlassian.com/incident-management/kpis/reliability-vs-availability

Graceful Service Degradation Patterns - by sysdai 13 8
https://systemdr.substack.com/p/graceful-service-degradation-patterns

REL05-BP01 Implement graceful degradation to transform applicable hard dependencies into soft dependencies - AWS Well-Architected Framework (2022-03-31) 10 9
/https://docs.aws.amazon.com/wellarchitected/2022-03-31/framework
rel_mitigate_interaction_failure_graceful_degradation.html

Active-Passive vs. Active-Active Failover 12 11
/https://www.serverion.com/uncategorized/active-passive-vs-active-active-failover

Using Upstash Redis for Laravel Sessions | Upstash Blog 14
https://upstash.com/blog/using-upstash-redis-for-laravel-sessions

Mastering High Availability: A Dive into Database Replication in Laravel | by Osmarrod | Medium 16 15
https://medium.com/@osmarrod18/mastering-high-availability-a-dive-into-database-replication-in-laravel-4c5c91d16c4a

Queues - Laravel 12.x - The PHP Framework For Web Artisans 22 21 20 19 18 17
https://laravel.com/docs/12.x/queues

Laravel Pennant - Laravel 12.x - The PHP Framework For Web Artisans 23
https://laravel.com/docs/12.x/pennant