ChatGPT

جدول تعلّم المهارات المتقدمة لمهندس Backend Senior – خطة 60 يوم

في هذا الجدول، تم تقسيم المحتوى إلى *10* محاور رئيسية تغطي مهارات متقدمة لمهندسي *Backend* على مستوى *Senior* . يمتد الجدول على مدار شهرين *(60 يوم)*، حيث يُخصص لكل محور *6* أيام تقريبًا. كل يوم يحتوي على عنوان الموضوع، شرح مختصر بالعربية *(*مع مصطلحات إنجليزية أساسية مذكورة*)*، قائمة بالمصطلحات الأساسية بالإنجليزية، فكرة لتطبيق عملي أو تمرين، وخانة ملاحظات للتقييم أو التعليق.

المحاور الرئيسية وتوزيع الأيام:

- **المحور 1: تصميم الأنظمة على مستوى الإنتاج (System Design)** – الأيام *1–6*
- **المحور 2: الأنظمة الموزعة (Distributed Systems)** – الأيام *7–12*
- **المحور 3: هندسة البيانات في الخلفية (Data Engineering)** – الأيام *13–18*
- **المحور 4: المعمارية الموجهة بالأحداث (Messaging/Event-Driven Architecture)** – الأيام *19–24*
- **المحور 5: هندسة الصمود والمراقبة (Resilience & Observability)** – الأيام *25–30*
- **المحور 6: أمن الأنظمة الخلفية المتقدم (Advanced Backend Security)** – الأيام *31–36*
- **المحور 7: الأداء والتزامن (Performance & Concurrency)** – الأيام *37–42*
- **المحور 8: تصميم واجهات برمجية متقدمة (Advanced API Design)** – الأيام *43–48*
- **المحور 9: تجربة المطور وممارسات التسليم (DevEx & Delivery Practices)** – الأيام *49–54*
- **المحور 10: الوعي بالحوسبة السحابية والبنية التحتية (Cloud & Infrastructure Awareness)** – الأيام *55–60*

المحور 1: تصميم الأنظمة على مستوى الإنتاج (System Design – Production-Level)

*(*الأيام *1–6)*

وصف المحور: يتناول هذا المحور مبادئ تصميم الأنظمة البرمجية على نطاق واسع وبجودة إنتاجية. ستتعرف على كيفية بناء أنظمة قابلة للتوسع والاعتمادية، وفهم أنماط التصميم المعماري المختلفة، مع مراعاة متطلبات الإنتاج مثل التوفرية العالية والصيانة السهلة.

اليوم 1: مقدمة في تصميم الأنظمة الإنتاجية

شرح مختصر: مقدمة عامة لمفهوم تصميم الأنظمة على مستوى الإنتاج. في هذا اليوم نتعرّف على الفروق بين بناء نموذج أولي وبين تصميم نظام جاهز للإنتاج. التركيز سيكون على **المتطلبات اللاوظيفية** مثل القابلية للتوسع، والموثوقية، وقابلية الصيانة. سنتناول أهمية تصميم أنظمة تتحمل أحمال المستخدمين الكبيرة وتظل مستقرة وآمنة في بيئة الإنتاج. كذلك سنتطرق لإدارة **التعقيد** في الأنظمة الكبيرة عبر تقسيمها إلى مكونات ذات مسؤوليات واضحة.
المصطلحات الأساسية:
- Scalability (قابلية التوسع)
- Reliability (الموثوقية)

- Maintainability (قابلية الصيانة)
- Production Environment (بيئة الإنتاج)
- Non-functional Requirements (متطلبات لاوظيفية)

فكرة التطبيق العملي: قم بكتابة قائمة بمتطلبات نظام تخيُّلي (مثل موقع تجارة إلكترونية كبير) تشمل المتطلبات الوظيفية واللاوظيفية. حاول أن تحدد ما يجب مراعاته في التصميم لتحقيق هذه المتطلبات في بيئة إنتاجية.
ملاحظات: مساحة للملاحظات والتقييم الذاتي.

اليوم 2: قابلية التوسع وإدارة الحمل

شرح مختصر: التركيز على **قابلية التوسع (Scalability)** في تصميم الأنظمة. نشرح الفرق بين التوسع الرأسي (زيادة موارد الخادم نفسه) والتوسع الأفقي (إضافة مزيد من الخوادم)، ومفهوم **Balance Load (موزع الأحمال)** وتوزيع الضغط على عدة عقد. سنناقش أهمية بناء أنظمة **stateless** (عديمة الحالة) حيثما أمكن لتسهيل التوسع الأفقي. أيضاً نتناول استراتيجيات التقسيم (Partitioning) للبيانات والخدمات لتفادي الاختناقات. الهدف هو تصميم نظام يمكنه النمو بسلاسة عند زيادة عدد المستخدمين أو حجم البيانات.
المصطلحات الأساسية:
- Vertical vs. Horizontal Scaling (التوسع الرأسي مقابل الأفقي)
- Load Balancer (موازن الحمل)
- Stateless Services (خدمات عديمة الحالة)
- Partitioning (تقسيم/تجزئة البيانات)
- Scalability Testing (اختبار قابلية التوسع)

فكرة التطبيق العملي: ارسم مخططًا بسيطًا يوضح كيفية توزع مكونات نظامك (مثلاً طبقة التطبيق وقاعدة البيانات) في حالة وجود خادم واحد، ثم كيف ستبدو عند تطبيق التوسع الأفقي بإضافة خوادم متعددة خلف موازن حمل. حدد أين ستكون النقاط الحرجة عند زيادة الحمل.
ملاحظات:

اليوم 3: التوافرية العالية والموثوقية

شرح مختصر: نتعمّق في مفاهيم **High Availability (التوافرية العالية)** و**Reliability (الموثوقية)** في الأنظمة. نشرح كيف نتجنب نقاط الفشل الأحادية *Single Point of Failure* من خلال التكرار (Replication) وتعدد الخوادم أو قواعد البيانات. سنتعرف على أساليب تحقيق الاستمرارية التشغيلية حتى في حال تعطل أجزاء من النظام، مثل وجود خوادم احتياطية (Standby) أو توزيع عقد الخدمة عبر مراكز بيانات متعددة. نناقش أيضًا مفهوم **الفشل الآمن** وكيفية تصميم النظام بحيث يفشل بطريقة لا تؤثر كليًا على المستخدم (Graceful Degradation). الهدف هو تصميم نظام يستمر بالعمل دون انقطاع يُذكر، ويلبّي اتفاقيات مستوى الخدمة (SLA) الصارمة.
المصطلحات الأساسية:
- High Availability (التوافرية العالية)
- Redundancy (التكرار/ازدواجية المكونات)
- Failover (الانتقال عند الفشل)
- Single Point of Failure (نقطة فشل أحادية)
- Graceful Degradation (التدهور المُنظّم للخدمة)

فكرة التطبيق العملي: اختر خدمة حياتية (مثلاً خدمة بريد إلكتروني أو نظام معاملات بنكية) وفكّر في سيناريوهات فشل محتملة (تعطّل خادم، انقطاع شبكة، تعطل قاعدة البيانات). ضع خطة لكيفية معالجة كل سيناريو (مثل استخدام خادم احتياطي أو قاعدة بيانات مكررة) لضمان استمرار الخدمة.
ملاحظات:

اليوم 4: أنماط معمارية وتصميمية

شرح مختصر: استعراض لأبرز **أنماط معمارية (Architecture Patterns)** مستخدمة في بناء الأنظمة الخلفية. سنقارن بين **المعمارية الأحادية (Monolithic)** حيث يكون التطبيق وحدة واحدة، وبين **معمارية الخدمات المصغرة (Microservices)** حيث يتم تقسيم النظام إلى خدمات مستقلة. نناقش إيجابيات وسلبيات كل منهما؛ فالنظام الأحادي أبسط في البداية لكن قد يواجه صعوبة في التوسع مع كبر الحجم، بينما الخدمات المصغرة توفر مرونة وقابلية أعلى للتوسع لكنها تضيف تعقيدًا تشغيليًا [1] . كذلك سنتطرق إلى المعمارية ذات الطبقات (Layered Architecture) وفصل منطق العمل عن طبقة البيانات وطبقة العرض. الهدف هو فهم كيفية اختيار النمط المناسب بناءً على حجم الفريق والنظام وطبيعة المتطلبات [1] .

المصطلحات الأساسية:

- Monolithic Architecture (العمارة الأحادية)
- Microservices Architecture (عمارة الخدمات المصغرة)
- Layered Architecture (عمارة الطبقات)
- Service-Oriented Architecture (SOA) (عمارة معتمدة على الخدمات)
- Trade-offs (المقايضات في التصميم)

فكرة التطبيق العملي: قم بتحديد 3 مزايا و3 عيوب لكل من نهج النظام الأحادي ونهج الخدمات المصغرة. فكر في نظام معروف (مثل Netflix) كمثال: كيف استفاد من الخدمات المصغرة لتحقيق **المرونة وقابلية التوسع** مقابل التعقيدات التشغيلية المضافة [1] . حاول تلخيص متى تختار كل نهج.

ملاحظات:

اليوم 5: تصميم نظام – مثال تطبيقي

شرح مختصر: تطبيق عملي لما سبق تعلمه من خلال دراسة حالة **تصميم نظام واسع النطاق** . سنأخذ مثال **نظام شبكة اجتماعية** أو **منصة تجارة إلكترونية** ونحدد متطلبات التصميم الخاصة به: عدد المستخدمين الكبير، التحديثات اللحظية، التوصيات، إلخ. ثم نعمل على تصميم معماري مبدئي يشمل المكونات الرئيسية: بوابة API، خدمات تطبيقية، قاعدة بيانات للتعاملات (transactions)، قاعدة بيانات أخرى للقراءة (مثلاً للتحاليل أو البحث)، خادم CDN للوسائط، وخدمات خارجية. سنراعي في التصميم توفير التوسع الأفقي، والتكرار لضمان التوفرية، واستخدام التخزين المؤقت (Cache) لتخفيف الحمل على قاعدة البيانات. هذا التمرين يوضح كيفية التفكير بطريقة شمولية لتحقيق نظام **Production-Grade**.

المصطلحات الأساسية:

- Architecture Diagram (مخطط معماري)
- API Gateway (بوابة واجهات برمجية)
- Caching Layer (طبقة التخزين المؤقت)
- CDN (شبكة توزيع المحتوى)
- Read Replica (نسخة قراءة لقاعدة البيانات)

فكرة التطبيق العملي: ارسم مخططًا معماريًا لنظام **مدوّنة أو موقع أخباري واسع** تخيّلي. بيّن في الرسم كيف يمكن توزيع المكونات: خوادم الويب خلف موازن حمل، قاعدة بيانات أساسية ونسخ للقراءة، طبقة Cache (مثل Redis) بين التطبيق وقاعدة البيانات، ومكون مراقبة (Monitoring) لمتابعة أداء النظام. هذا الرسم سيكون بمثابة **High-Level Design** للنظام.

ملاحظات:

اليوم 6: اعتبارات نهائية في تصميم الأنظمة

شرح مختصر: خلاصة المحور الأول مع التركيز على اعتبارات إضافية مهمة في تصميم الأنظمة. سنتحدث عن **الاختبار في مرحلة التصميم** (Design Testing) مثل إجراء **جلسات مراجعة معمارية** والتفكير في حدود النظام (Limits) قبل وقوع المشاكل. نناقش مبدأ **YAGNI** ("لن تحتاجه") و**KISS** ("حافظها بسيطة") في التصميم لتجنب التعقيد الزائد. كما

نعرّج على أهمية **التوثيق المعماري** لكل القرارات المتخذة والتبريرات، وأهمية **المراجعات الدورية** للتصميم مع تطور المتطلبات. أيضًا سنشير إلى مفهوم تصميم الأنظمة لاستيعاب **المراقبة observability** من البداية (بزرع نقاط قياس metricsو logs) – تحضيرًا للمحاور القادمة. الهدف في النهاية أن يكتسب المتعلم منهجية تفكير شمولية تمكنه من تصميم أي نظام برمجي معقد بخطوات منهجية وواثقة.

المصطلحات الأساسية:

- YAGNI (اختصار لمبدأ "You Aren’t Gonna Need It")
- KISS Principle (مبدأ "Keep It Simple, Stupid")
- Scalability Limit (حدود التوسع)
- Architectural Decision Record (سجل القرار المعماري)
- Design Review (مراجعة التصميم)

فكرة التطبيق العملي: اختر نظامًا شهيرًا (مثلاً YouTube أو Twitter) وفكّر ما أبرز التحديات التصميمية فيه. حاول كتابة **فرضيات** حول كيف تم تصميمه لحل مشكلات التوسع والموثوقية (مثلاً: استخدام **Microservices** ، أو **CDN** للوسائط، أو **تجزئة المستخدمين حسب المنطقة**). هذه الفرضيات درب عقلك على **طرح الأسئلة التصميمية الصحيحة** حتى إن لم تكن الإجابات الدقيقة متوفرة لك.

ملاحظات:

المحور 2: الأنظمة الموزعة (Distributed Systems)

(الأيام 7–12)

وصف المحور: يغطي هذا المحور مبادئ تصميم وبناء الأنظمة الموزعة حيث تعمل مكونات النظام عبر شبكات وأجهزة متعددة. سنتعلم التحديات الخاصة بهذا المجال مثل اتصالات الشبكة غير الموثوقة، واتساق البيانات عبر عقد متعددة، ونظرية CAP Trade-off بين الاتساق والتوافرية. سنستكشف أيضًا طرق تحقيق التوافق بين العقد (Consensus) وإدارة البيانات الموزعة، مع دراسة أنماط تصميم معمارية خاصة بالأنظمة الموزعة.

اليوم 7: مقدمة في الأنظمة الموزعة والتحديات

شرح مختصر: تعريف الأنظمة الموزعة وسبب استخدامها. النظام الموزع هو نظام تعمل فيه عدة حواسيب معًا لتقديم خدمة واحدة. نستعرض ميزات الأنظمة الموزعة (مثل التوسع الجغرافي، زيادة التوفرية) مقابل التحديات التي تطرحها: **زمن استجابة الشبكة**، **الفشل الجزئي** (قد يتعطل جزء من النظام بينما بقية الأجزاء تعمل)، وصعوبة **ضمان اتساق البيانات** عبر المواقع المتعددة. سنتعرف على نموذج **عدم موثوقية الشبكة** (الشبكة قد تكون بطيئة أو مقطوعة في أي لحظة) وكيف يجب أن يصمم المهندس النظام متوقعًا تلك الأعطال. هذه المقدمة تؤسس لفهم باقي المواضيع التفصيلية في الأيام التالية.

المصطلحات الأساسية:

- Distributed System (نظام موزع)
- Network Latency (زمن انتقال الشبكة)
- Partial Failure (فشل جزئي)
- Fault Tolerance (تحمل الأخطاء)
- Data Consistency (اتساق البيانات)

فكرة التطبيق العملي: فكّر في تطبيق تستخدمه يوميًا (مثل تطبيق محادثة أو لعبة على الإنترنت) وحاول أن تحدد الأجزاء التي قد تكون موزعة فيه (مثلاً: خادم للمحادثة في منطقة جغرافية، خادم آخر لتخزين الوسائط، ... إلخ). دوّن سيناريو واحد على الأقل كيف يمكن أن يفشل جزء من النظام (مثل انقطاع خادم) ويبقى الجزء الآخر عاملاً، وكيف يُفترض بالنظام أن يتعامل مع ذلك.

ملاحظات:

اليوم 8: الاتصال والتواصل في الأنظمة الموزعة

شرح مختصر: مناقشة آليات **الاتصال بين الخدمات** الموزعة عبر الشبكة. سنتعرف على طرق التواصل: **نماذج Client-Server** التقليدية، مقابل **Peer-to-Peer** . نشرح البروتوكولات الشائعة مثل HTTP وRPC وgRPC، وكيفية اختيار بروتوكول يناسب الأداء المطلوب. كما نركز على مفهوم **زمن الاستجابة (Latency)** وتأثيره؛ على سبيل المثال، تأخير بضعة أجزاء من الثانية قد يتراكم عند استدعاء خدمات متعددة، مما يستلزم تصميم واجهات تقلل **الشبكات المتعددة hops** . سنتناول أيضًا فكرة **serialization (التسلسل)** لتحويل البيانات عبر الشبكة، ومتى نستخدم صيغة JSON أو بروتوكول أخف مثل بروتوبافر (Protobuf). كذلك سيتم التطرق لمشكلة **موثوقية الرسائل** وما إذا كانت الضمانات (at-most-once, at-least-once, exactly-once) متوفرة في بروتوكولات معينة.

المصطلحات الأساسية:

- RPC (استدعاء الإجراء البعيد) و REST
- Latency (زمن الاستجابة)
- Bandwidth (عرض النطاق)
- gRPC/Protobuf (بروتوكول اتصال وكودرة بيانات)
- Message Semantics (دلالات تسليم الرسالة: مرة على الأكثر/على الأقل/بالضبط)

فكرة التطبيق العملي: قم ببناء دالة بسيطة في أي لغة برمجة تقوم **باستدعاء خدمة وهمية** عبر بروتوكول HTTP (يمكنك استخدام طلب HTTP إلى موقع عام يقدم JSON). قِس الزمن المستغرق للاستجابة وسجّله. ثم تخيّل لو كان عليك تنفيذ نفس العملية 10 مرات متتالية للحصول على بيانات مختلفة – كيف سيؤثر ذلك على الأداء؟ هذا التمرين يوضح أثر **زمن الشبكة** على التطبيقات الموزعة.

ملاحظات:

اليوم 9: الاتساق والتوافرية – نظرية CAP

شرح مختصر: نظرية CAP الشهيرة في الأنظمة الموزعة. تنص هذه النظرية على أنه في أي نظام موزع لا يمكن ضمان أكثر من خاصيتين من ثلاث في نفس الوقت: **الاتساق (Consistency)**، **التوافرية (Availability)**، **تحمل انقسام الشبكة (Partition Tolerance)** [2] [3] . نشرح معنى كل خاصية بالتفصيل: الاتساق يعني أن جميع العقد ترى نفس البيانات في نفس الوقت، والتوافرية تعني أن النظام يستجيب دائمًا حتى في حالة تعطل بعض العقد، وتحمل الانقسام يعني الاستمرار بالعمل رغم انقطاع الاتصال بين أجزاء النظام [4] [5] . سنوضح لماذا لا يمكن تحقيق الثلاث معًا دومًا – فعند حصول انقسام للشبكة عليك أن تقرر التضحية إما بالاتساق أو بالتوافرية. سنربط ذلك بقواعد البيانات: مثلاً أنظمة **SQL** التقليدية تميل للاتساق والتوفر (CA) ولكنها لا تتحمل الانقسام، بينما **NoSQL** عديدة مصممة وفق مبدأ إما CP أو AP [3] .

المصطلحات الأساسية:

- CAP Theorem (نظرية CAP) [2]
- Consistency (الاتساق) [6]
- Availability (التوافرية) [7]
- Partition Tolerance (تحمل انقسام الشبكة) [5]
- Eventually Consistent (اتساق نهائي)

فكرة التطبيق العملي: راجع أحد أنظمة قواعد البيانات الشهيرة (مثل MongoDB أو Cassandra) واقرأ بإيجاز عن نهجها بالنسبة لـ CAP. مثلًا MongoDB يعتبر CP (يختار الاتساق على حساب بعض التوافرية)، بينما Cassandra يعتبر AP (يختار التوافرية مع اتساق نهائي) [8] . دوّن ملاحظاتك حول سبب هذا الاختيار وكيف يؤثر على استخدام النظام في التطبيق العملي.

ملاحظات:

اليوم 10: تكرار البيانات والتجزئة الموزعة

شرح مختصر: كيفية إدارة البيانات في الأنظمة الموزعة عبر **Replication (تكرار)** و**Sharding (تجزئة)** . نشرح أن تكرار البيانات عبر عقد متعددة يزيد التوفرية ويحمي من فقدان البيانات، لكنه قد يؤدي لمشاكل اتساق يجب التعامل معها (مثلاً عبر خوارزميات تزامن). نتعرف على نموذج **Master-Slave replication (رئيسي/ثانوي)** مقابل **Multi-Master**، ومزايا كل منهما. ثم نتطرق لمفهوم التجزئة: تقسيم البيانات عبر عدّة عُقد بحيث يحتوي كل خادم جزءًا من البيانات (مثال: تجزئة المستخدمين حسب الحروف أو الموقع الجغرافي). سنناقش **خوارزميات التجزئة** وكيفية ضمان توزيع متوازن (مثل Consistent Hashing – التجزئة المتسقة). كما نذكر تحديات مثل إعادة التجزئة عند إضافة عقد جديدة. الهدف هو فهم طرق توزيع البيانات لتحقيق أداء وتوسعية أفضل مع المحافظة على القدر المطلوب من الاتساق.

المصطلحات الأساسية:

- Data Replication (تكرار البيانات)
- Master-Slave / Primary-Secondary (رئيسي/ثانوي)
- Leader Election (انتخاب القائد)
- Sharding (تجزئة قاعدة البيانات)
- Consistent Hashing (تجزئة متسقة)

فكرة التطبيق العملي: تخيّل أنك تدير قاعدة بيانات لموقع عالمي، وقررت **تجزئة المستخدمين حسب المنطقة الجغرافية** (مثلاً أوروبا، آسيا، الأمريكيتين). ارسم مخططًا يوضح كيف يتم توجيه طلب مستخدم من أوروبا إلى مجموعة خوادم قواعد البيانات الخاصة بأوروبا. ثم فكر: إذا امتلأت مجموعة أوروبا وأردنا تقسيمها لمجموعتين أصغر، ما الخطوات اللازمة لنقل بعض المستخدمين إلى shard جديد دون فقدان البيانات أو توقف الخدمة؟

ملاحظات:

اليوم 11: التوافق الموزع و خوارزميات الإجماع

شرح مختصر: التعرف على مشكلة **التوافق (Consensus)** في الأنظمة الموزعة: كيف تتفق عدة عقد على قيمة أو قرار معين بالرغم من احتمال حدوث فشل أو تأخير. سنشرح بإيجاز خوارزميات مشهورة مثل **Paxos** و **Raft** لتحقيق الإجماع – هذه الخوارزميات تضمن أن مجموعة من الخوادم تتفق على حالة مشتركة (مثل سجل عمليات) حتى لو تعطل بعضها. نناقش سيناريو **انتخاب قائد (Leader Election)** في نظام مكرر، وكيف تعرف العقدة الجديدة أنها يجب أن تصبح القائد. كذلك سنتناول مصطلح **Distributed Transaction (المعاملة الموزعة)** ولماذا يكون معقدًا، ونشير إلى بروتوكول *Phase Commit-2* التقليدي ومشاكله في البيئات الموزعة. الهدف هنا فهم أن تحقيق الإجماع موثوق عليه هو أساس بناء أنظمة موزعة قوية (مثل نظام Google Chubby أو etcd).

المصطلحات الأساسية:

- Distributed Consensus (الإجماع الموزع)
- Paxos Algorithm (خوارزمية باكسوس)
- Raft Algorithm (خوارزمية رأفت)
- Leader Election (انتخاب القائد)
- Two-Phase Commit (االالتزام ذو المرحلتين)

فكرة التطبيق العملي: اقرأ عن نظام **etcd** أو **Apache Zookeeper** (أنظمة توافق موزع تُستخدم كعمود فقري لكثير من الخدمات) واكتب نقطة أو نقطتين حول كيف يستخدمان خوارزمية توافق (مثل Raft في etcd). على سبيل المثال: "يستخدم *etcd* خوارزمية *Raft* لضمان أن جميع العقد تحتفظ بنفس البيانات حتى في حالة فشل البعض، لذا يمكن استخدامه كـ *Configuration Store* موثوق." هذه الملاحظة تبين فهمك لأهمية التوافق.

ملاحظات:

اليوم 12: أنماط التصميم الموزع – تطبيق عملي

شرح مختصر: نختتم المحور بعرض بعض **أنماط التصميم المعماري الموزع** وحالة تطبيقية. سنتعرف على نمط **Client-Server الموسّع** حيث يوجد خادم مركزي يتعامل مع عملاء متعددين، مقابل نمط **Peer-to-Peer** حيث تتواصل العقد

مباشرة (مثل التورنت). نناقش أيضًا نمط **Microservices** كنظام موزع داخل مؤسسة، وكيفية تنظيم الخدمات (Discovery Service لاكتشاف الخدمات، API Gateway للتنسيق). نعرّج على مفهوم **Eventual Consistency (الاتساق النهائي)** كنمط مقبول في كثير من التطبيقات الموزعة (مثل أن ترى خدمة تحديث بعد بضع ثوانٍ من خدمة أخرى). أخيرًا، نطرح **حالة دراسية** صغيرة: تصميم نظام **دردشة عالمي** ، حيث المستخدمون موزعون حول العالم وخدمة الدردشة يجب أن تكون آنية. سنفكر في تقسيم المستخدمين حسب المنطقة، وكيفية مزامنة الرسائل عبر مراكز بيانات مختلفة مع السماح ببعض التأخير البسيط.

المصطلحات الأساسية:

- Peer-to-Peer Architecture (نظير-إلى-نظير)
- Service Discovery (اكتشاف الخدمات)
- Eventual Consistency (اتساق نهائي)
- Distributed Cache (ذاكرة تخزين موزعة)
- Global Distribution (توزيع عالمي للخدمة)

فكرة التطبيق العملي: صمم على ورقة بنية مبسطة لتطبيق **مشاركة ملفات لامركزي** (مثل بروتوكول Torrent). وضّح كيف يتواصل الأقران (Peers) فيما بينهم بدون خادم مركزي، وأين قد نحتاج على الأقل خادم تتبع (Tracker) لتنسيق الاتصال. هذا التمرين مختلف عن الخدمات المركزية التقليدية وسيجعلك تفكر في نوع مختلف من الأنظمة الموزعة.

ملاحظات:

المحور 3: هندسة البيانات في الـBackend (Data Engineering in Backend)

(الأيام 13–18)

وصف المحور: يغطي هذا المحور كيفية تعامل مهندس الـBackend مع البيانات بفعالية وكفاءة. سنستعرض أنواع قواعد البيانات (علاقية vs غير علاقية) ومتى نختار أياً منها، تصميم نماذج البيانات والجدولة، تحسين الاستعلامات والأداء، إدارة البيانات الضخمة (Big Data) وتكاملها، استخدام الذاكرة الوسيطة (Caching) لتحسين الأداء، وبناء خطوط نقل ومعالجة البيانات (ETL/Streaming) في سياق تطوير الـBackend.

اليوم 13: قواعد البيانات – SQL مقابل NoSQL

شرح مختصر: مقدمة لأهم الفروقات بين **قواعد البيانات العلاقية (SQL)** وقواعد البيانات **غير العلاقية (NoSQL)**. نستعرض خصائص قواعد بيانات SQL التقليدية: بنية جداول مترابطة، لغة استعلام قوية (SQL)، ودعم **الخصائص ACID** للمعاملات (الذرية، الاتساق، العزل، الديمومة) لضمان موثوقية عالية [9] . ثم نبيّن لماذا ظهرت NoSQL: الحاجة إلى المرونة في هيكل البيانات والتوسع الأفقي بسهولة [10] . نشرح الأنواع المختلفة لـ NoSQL (مخازن المفاتيح/القيم، الوثائق، الأعمدة، الرسوم البيانية) ومتى تناسب كل منها. كذلك نربط مع نظرية CAP: كثير من أنظمة NoSQL تقدم **اتساق نهائي** بدلاً من الاتساق الفوري لأجل التوسع [11] . هدف هذا اليوم هو أن تكون قادراً على اختيار نوع قاعدة البيانات المناسب حسب طبيعة البيانات ومتطلبات التطبيق (مثلاً: SQL للمعاملات المعقدة والعلاقات الواضحة، NoSQL للبيانات الضخمة غير المهيكلة أو عند الحاجة لتوسع أفقي كبير).

المصطلحات الأساسية:

- Relational Database (قاعدة بيانات علائقية)
- ACID Properties (خواص ACID للمعاملات) [9]
- NoSQL (غير علائقية، "ليست فقط SQL")
- Document Store / Key-Value Store (مخزن وثائق/مفاتيح-قيم)
- BASE (Basically Available, Soft state, Eventual consistency)

فكرة التطبيق العملي: قم بعمل بحث سريع عن اثنين من قواعد البيانات الشهيرة – واحدة علائقية (مثل

PostgreSQL) وواحدة NoSQL وثائقية (مثل **MongoDB**). اكتب مقارنة مختصرة (3 نقاط لكل منهما) تركّز على: أسلوب تخزين البيانات، قدرة التوسع، وضمان الاتساق. على سبيل المثال: "*MongoDB* يستخدم *JSON* لتخزين الوثائق ويسمح ببنية مرنة، يتميز بالتوسع الأفقي السهل *(CP* ضمن *CAP)*، لكنه يقدم اتساق نهائي افتراضيًا؛ بينما *PostgreSQL* يعتمد جداول ثابتة وبنية *SQL* صارمة ويدعم معاملات *ACID* كاملة *(*مما يجعله *CA* ضمن *CAP* لكنه غير موزع أفقيًا افتراضيًا)."

ملاحظات:

اليوم 14: تصميم قاعدة البيانات ونمذجة البيانات

شرح مختصر: التعرف على مبادئ **تصميم schema قاعدة البيانات** بشكل سليم. سنستعرض خطوات **نمذجة البيانات**: بدءًا من نموذج مفاهيمي (كيانات وعلاقات) إلى نموذج منطقي ثم تنفيذ فعلي. نناقش قواعد **التطبيع (Normalization)** في قواعد البيانات العلائقية لتقليل التكرار وضمان سلامة البيانات، ومتى قد نتخلى عن التطبيع لصالح الأداء (denormalization) في بعض الحالات. نشرح أهمية اختيار المفاتيح الرئيسية Primary Keys الصحيحة (ومفاتيح خارجية للعلاقات). كما سنتحدث عن نمذجة البيانات في NoSQL: مثل تصميم وثيقة في MongoDB لتشمل بيانات مرتبطة (embed vs reference) حسب احتياج الاستعلامات. سيتناول اليوم أيضًا **أدوات تخطيط ERD** وكيف تساعد في توثيق تصميم قاعدة البيانات. النتيجة المرجوة هي القدرة على تحويل متطلبات التطبيق إلى تصميم جداول/ وثائق فعال يلبي تلك المتطلبات.

المصطلحات الأساسية:

- Schema Design (تصميم المخطط البياني للبيانات)
- Entity-Relationship Diagram (ERD) (مخطط الكيانات والعلاقات)
- Normalization / Denormalization (تطبيع/إلغاء التطبيع)
- Primary Key, Foreign Key (مفتاح أساسي، مفتاح خارجي)
- Data Modeling (نمذجة البيانات)

فكرة التطبيق العملي: ارسم ERD بسيط لمشروع مكتبة كتب: جداول **الكتب** و**المؤلفين** و**المستعيرين** مثلاً. حدد العلاقات (كتاب إلى مؤلف علاقة عدة-إلى-عدة، عبر جدول رابط مثل *BookAuthors*). طبّق قواعد التطبيع: تأكد أن كل معلومة في المكان الصحيح (مثلاً معلومات المؤلف في جدول المؤلف وليس مكررًا في جدول الكتاب). هذا التمرين يساعد على تحويل وصف نظام بسيط إلى هيكل قاعدة بيانات منظم.

ملاحظات:

اليوم 15: تحسين الاستعلامات والفهارس

شرح مختصر: التركيز على **أداء قاعدة البيانات** وكيفية تحسين الاستعلامات (Queries). نتناول مفهوم **Index (الفهرس)** في قواعد البيانات العلائقية: كيف يُسرّع الوصول للبيانات عبر فهرسة الأعمدة المستخدمة في شروط البحث. نشرح أنواع الفهارس (B-Tree index، الفهرس المكون من عمود واحد أو مركب، الفهارس الفريدة). سنتحدث عن تكلفة الفهرسة من ناحية تباطؤ الكتابة مقابل تسريع القراءة. أيضًا سنناقش **تحليل خطط التنفيذ (Query Execution Plan)** لمعرفة كيف ينفذ محرك قاعدة البيانات الاستعلام وإيجاد النقاط المعيقة (Bottlenecks). سنستعرض أمثلة لاستعلامات SQL بطيئة وكيف نعيد كتابتها أو نضيف فهارس لتحسينها. في سياق NoSQL، نتحدث عن تصميم الاستعلام بشكل يتناسب مع طريقة تخزين البيانات (مثلاً استخدام المفاتيح الأساسية في DynamoDB للوصول السريع، لأن الاستعلامات المعقدة غير متاحة كما في SQL). الهدف أن يلمّ المهندس بتقنيات **ضبط أداء قاعدة البيانات** لضمان استجابة سريعة في التطبيقات الحرجة.

المصطلحات الأساسية:

- Database Index (فهرس قاعدة البيانات)
- Query Optimization (تحسين الاستعلام)
- Query Execution Plan (خطة تنفيذ الاستعلام)
- ACID Transactions (المعاملات)
- Bottleneck (عنق الزجاجة في الأداء)

فكرة التطبيق العملي: استخدم قاعدة بيانات علائقية (يمكن أن تكون SQLite بسيطة أو أي منصة لديك) وأنشئ

جدولًا كبيرًا نسبيًا (مثلاً 100 ألف سجل وهمي). نفّذ استعلام SELECT بشرط على عمود غير مفهرس وقِس الوقت. ثم أضف فهرسًا على هذا العمود ونفّذ نفس الاستعلام وقارن الزمن. سجل الملاحظات على الفارق في الأداء.
ملاحظات:

اليوم 16: إدارة البيانات الضخمة والمعالجة الدفعية

شرح مختصر: هذا اليوم مخصص للتعرّف على التعامل مع **البيانات الضخمة (Big Data)** في سياق الـBackend. عندما تتجاوز البيانات حجم وقدرة قواعد البيانات التقليدية، نحتاج لتقنيات خاصة. سنشرح مفهوم المعالجة الدفعية (Batch Processing) للتعامل مع مجموعات كبيرة من البيانات دفعة واحدة، باستخدام منصات مثل **Hadoop** (خاصة نظام ملفات HDFS وMapReduce) وكيف تقسم المهام عبر عدة عقد. نعرّف بإيجاز إطار عمل **Apache Spark** كأداة حديثة لمعالجة البيانات في الذاكرة بسرعة أعلى. كذلك نتناول فرق **OLTP vs OLAP** – أي المعاملات الفورية مقابل تحليل البيانات التاريخية – ولماذا قد نفصل قاعدة البيانات التشغيلية عن مستودع تحليلي. سنتطرق أيضًا إلى مفهوم **Data Warehouse** ومستودعات حديثة مثل **Snowflake** أو **Google BigQuery** وكيف يمكن لمهندس الـBackend التكامل معها لإجراء تحليلات ضخمة. الهدف هو رفع مستوى الوعي بما يحدث عندما يصبح حجم البيانات تحديًا بحد ذاته، والحلول المتاحة لذلك.
المصطلحات الأساسية:
- Big Data (بيانات ضخمة)
- Batch Processing (معالجة دفعية)
- Apache Hadoop / MapReduce (تقنيات هادوب)
- Apache Spark (سبارك)
- OLTP vs OLAP (نظام المعاملات الفورية مقابل نظام التحليلات)

فكرة التطبيق العملي: اقرأ عن مفهوم MapReduce – مثلاً خطواته الثلاث (Map -> Shuffle -> Reduce). حاول كتابة نسخة مبسطة منه (ولو تجريدية) بلغة برمجة لديك: على سبيل المثال، خذ قائمة ضخمة من الأرقام وأوجد تكرار كل رقم (Frequency count) عبر تقسيم القائمة لأجزاء (this simulates mapping) ثم جمع النتائج (reducing). حتى لو لم يكن حجم البيانات كبير فعليًا في تجربتك، تمرين التفكير بهذا الأسلوب يعطيك فكرة عن **النمط البرمجي الموازي** المستخدم في معالجة البيانات الضخمة.
ملاحظات:

اليوم 17: التخزين المؤقت (Caching) واستراتيجياته

شرح مختصر: نتعلم عن دور **التخزين المؤقت (Cache)** في تحسين أداء تطبيقات الـBackend. الكاش هو وسيلة للاحتفاظ بنتائج العمليات أو البيانات الأكثر استخدامًا في ذاكرة سريعة (RAM) لتقليل الوصول المتكرر للمصادر البطيئة (قاعدة البيانات أو خدمات خارجية). سنشرح أنواع الكاش: **Cache داخلي في التطبيق** (مثلاً تخزين النتائج في الذاكرة محليًا)، **Cache موزع** مثل استخدام نظام Redis أو Memcached لتشارك الكاش بين الخوادم. نناقش استراتيجيات تحديث الكاش: **Cache Aside** (قراءة من الكاش أو قاعدة البيانات وتحديثه)، **Write-through** و**Write-back**. سنتناول أيضًا مشكلة **صلاحية البيانات (Cache Invalidation)** وأنه "المشكلتان الصعبتان في علم الحاسوب: تسمية الأشياء وتبطيل الكاش." كما سنشير لأماكن مختلفة للكاش (كاش المتصفح، كاش API Gateway، إلخ) ولكن تركيزنا على الكاش في الطبقة الخلفية لتحسين استجابة قواعد البيانات. هدف اليوم هو فهم متى تستخدم الكاش بحكمة لتحقيق أسرع أداء ممكن دون التضحية بصحة البيانات.
المصطلحات الأساسية:
- Caching (التخزين المؤقت)
- Redis / Memcached (أمثلة نظم كاش)
- Cache Invalidation (إبطال الكاش)
- Cache Hit / Miss (ضربة كاش ناجحة/فشل)
- Time To Live – TTL (زمن الحياة للكاش)

فكرة التطبيق العملي: إذا كان لديك إمكانية تشغيل Redis (أو استخدم خدمة سحابية مجانية)، جرّب وضع بيانات

بسيطة فيه – مثلاً نتائج استعلام مكلف. اكتب سكريبت صغير بلغتك المفضلة يقوم بالآتي: يجلب البيانات المطلوبة إما من Redis إن وجدت وإلا من المصدر الأساسي (يمكن أن تكون قراءة من ملف كبير)، ويسجل الوقت المستغرق. قارن بين زمن الحصول على النتيجة في حال وجودها في الكاش مقابل عدم وجودها. سيساعدك ذلك على لمس فائدة الكاش عمليًا.

ملاحظات:

اليوم 18: تكامل البيانات وخطوط الـETL

شرح مختصر: نسلّط الضوء على عملية **تكامل البيانات** في التطبيقات الخلفية، خصوصًا عند الحاجة لجمع البيانات من مصادر متعددة أو نقلها من نظام لآخر. يُعرف ذلك اختصارًا بـ **ETL (Extract, Transform, Load)** أي استخراج البيانات ثم تحويلها ثم تحميلها إلى وجهتها. نشرح سيناريو شائع: لديك قاعدة بيانات تشغيلية وتريد نقل بياناتها بشكل دوري إلى مستودع تحليلي؛ هنا تبني بايب لاين ETL يقوم باستخراج البيانات (مثلاً يوميًا)، وتنظيفها وتحويل الهيكل إذا لزم (مثلاً تحويل ترميز الحقول أو دمج جداول)، ثم إدخالها في قاعدة أخرى. نعرّف بعض الأدوات مثل **Apache NiFi** أو **Airflow** أو خدمات سحابية (AWS Glue مثلًا) لإدارة جداول ETL. أيضًا نتحدث عن البديل الأحدث **ELT** (استخراج ثم تحميل ثم تحويل) خاصة مع قدرات مستودعات البيانات الحديثة. نربط المفهوم بمهندس الـBackend حين يكتب سكريبتات لنقل البيانات أو يتكامل مع فريق تحليل البيانات. كذلك سنعرّج على **معالجة البيانات اللحظية (Streaming)** باختصار – باستخدام أدوات مثل Kafka Streams أو Spark Streaming – وكيف يختلف عن ETL الدفعي التقليدي.

المصطلحات الأساسية:

- ETL Pipeline (خط استخراج-تحويل-تحميل)
- Data Warehouse (مستودع البيانات)
- Apache Airflow (منصة جدولة مهام)
- Stream Processing (معالجة متدفقة)
- CDC (Change Data Capture)

فكرة التطبيق العملي: صمّم عملية ETL بسيطة: لنفترض أن لديك ملف CSV يحوي معلومات منتجات وتريد استيرادها إلى قاعدة بيانات. التخطيط: (Extract) قراءة (Transform) ،CSV التأكد أن الأسعار أرقام والتحويل إلى العملة الموحدة مثلاً، (Load) إدخال النتائج في جدول بقاعدة SQLite محلية. جرّب تنفيذ ذلك برمجياً إن أمكن، أو على الأقل اكتب الخطوات المفصّلة التي ستقوم بها. هكذا ستفهم خطوات ETL بمثال ملموس.

ملاحظات:

المحور 4: المعمارية الموجهة بالأحداث (Messaging / Event-Driven Architecture)

(الأيام 19–24)

وصف المحور: يركز هذا المحور على بناء الأنظمة غير المتزامنة واستخدام الرسائل والأحداث في تصميم الأنظمة الخلفية. سنشرح فوائد المعمارية الموجهة بالأحداث في فصل الخدمات وتحسين التوسعية، ونتعرف على نظم الرسائل (مثل Message Queues و Pub/Sub)، تصميم مخططات الأحداث، ضمانات التسليم (at-least once وغيرها)، وكذلك أنماط متقدمة مثل Saga للتعامل مع معاملات موزعة، مع تطبيقات عملية.

اليوم 19: مقدمة في معمارية الأحداث ولماذا نستخدمها

شرح مختصر: نظرة عامة على **Event-Driven Architecture (المعمارية الموجهة بالأحداث)** ومزاياها. في هذا النموذج، تتواصل المكونات عبر **أحداث (Events)** بدلاً من الاستدعاءات المباشرة، مما يفك الارتباط الزمني بينها (Decoupling). نشرح كيف يسمح ذلك بتوسع الخدمات وتطويرها بشكل مستقل – فخدمة ما تنشر حدثًا، وخدمات أخرى تستهلكه عند حدوثه [12] . نذكر أمثلة: نظام تسجيل الطلبات في متجر إلكتروني يرسل حدث "Order Placed"، فتقوم

خدمة الشحن وخدمة الفواتير بالتعامل معه كلٌ على حدة. سنناقش **سيناريوهات مناسبتها** : عند الحاجة لمعالجة غير آنية، أو التكامل بين أنظمة مختلفة دون ربط وثيق. أيضًا نذكر التحديات: مثل صعوبة تتبع تدفق الأحداث (لأنها غير متزامنة) وما يتطلبه ذلك من مراقبة إضافية. الإجمالي: هذه المعمارية تزيد المرونة والموثوقية (failure of one service لا يوقف الأخرى) [13] ، لكنها تأتي مع تعقيدات في **تصحيح الأخطاء** وفهم التدفق.

المصطلحات الأساسية:

- Event (حدث)
- Producer / Consumer (منتج/مستهلك الحدث)
- Decoupling (فصل الخدمات) [12]
- Async vs Sync (لاتزامن مقابل تزامن)
- Event-Driven Architecture (المعمارية الموجهة بالأحداث) [12]

فكرة التطبيق العملي: ارسم مخطط تسلسلي Sequence Diagram بسيط يوضح الفرق بين سيناريو متزامن (Service A تستدعي B مباشرة ثم تنتظر الرد) وسيناريو موجه بالأحداث (Service A تنشر حدثًا إلى Broker وخدمة B تلتقطه وتعالجه مستقلًا). وضّح في الرسم أين يكون A حرًا وغير معلق في الحالة الثانية. هذا سيساعدك على تصور **التدفق غير المتزامن** بشكل عملي.

ملاحظات:

اليوم 20: نظم المراسلة – الوسطاء والصفوف

شرح مختصر: التركيز على البنية التحتية للرسائل: **وسطاء الرسائل (Message Brokers)** مثل RabbitMQ, Apache Kafka, ActiveMQ وغيرها. نشرح مفهوم **Message Queue (صف الرسائل)** حيث تُرسل الرسائل وتُخزن حتى يستلمها المستهلك. نتناول الفرق بين **نموذج الطابور Queue** (نقطة-لنقطة: مستهلك واحد لكل رسالة) ونموذج **الناشر/المشترك Pub/Sub** (بث: يمكن لعدة مستهلكين استقبال نفس الحدث عبر مواضيع Topics). سنتعرف على كيفية ضمان تسليم الرسائل: *Acknowledgement* ورسائل *DLQ* (طابور الرسائل الميتة) في حال فشل معالجتها. كما نناقش الخصائص المهمة: **التأكد من مرة واحدة** (Exactly-once) مقابل **على الأقل مرة** (At-least-once) في التسليم، وكيف يقدم Kafka مثلاً **at-least-once** افتراضيًا مع إمكانية تحقيق exactly-once بصعوبة. كذلك سنشير إلى هيكل الرسائل (قد تحوي بيانات JSON أو غيره) وأهمية تضمين معرفات فريدة ورقم إصدار الرسالة. بعد هذا اليوم، ستعرف أي أداة رسائل تناسب سيناريوهاتك وكيفية استخدامها لضمان التدفق السلس للأحداث في نظامك.

المصطلحات الأساسية:

- Message Broker (وسيط رسائل)
- Queue vs Topic (صف مقابل موضوع نشر)
- Publish/Subscribe (نشر/اشتراك)
- Acknowledgment (إقرار الاستلام)
- Dead Letter Queue – DLQ (طابور الرسائل الفاشلة)

فكرة التطبيق العملي: إذا أمكنك تشغيل **RabbitMQ** أو **Kafka** محليًا (أو استخدم خدمات سحابية تجريبية)، قم بإنشاء منتج ومستهلك بسيطين. المنتِج يرسل رسالة نصية "Hello" كل 5 ثوانٍ إلى موضوع/صف معين، والمستهلك يتلقى الرسالة ويطبعها. لاحظ كيف يتم التخزين المؤقت في الوسيط لو أوقفت المستهلك لفترة ثم أعدته. هذا التطبيق البسيط يعطيك فكرة عن **التدفق عبر الوسيط** . (يمكن استخدام مكتبات جاهزة بلغة Python/JavaScript للاتصال بالوسيط بسهولة).

ملاحظات:

اليوم 21: تصميم الأحداث وهيكلة الرسائل

شرح مختصر: كيفية تصميم **رسائل الأحداث** بطريقة فعّالة وقابلة للتوسع. سنتحدث عن محتوى الحدث: مثلًا حدث "OrderCreated" يجب أن يحمل البيانات الأساسية (رقم الطلب، الوقت، هوية العميل) وربما رابط للحصول على تفاصيل إضافية عند الحاجة، بدلاً من تضمين كل شيء. هذه الموازنة بين حجم الحدث وسهولة استخدامه مهمة. سنناقش **تنسيق الرسائل** (JSON, Avro, Protobuf) ومزايا كل منها – JSON مقروء لكن قد يكون أكبر حجمًا، Avro/Protobuf

ثنائي وأكثر كفاءة مع مخطط ثابت. أيضًا نعرّج على موضوع **تطور مخطط الرسائل (Schema Evolution)** وكيفية إضافة حقول جديدة للأحداث بشكل متوافق مع الإصدارات القديمة (Backward Compatibility). سنتطرق لمفهوم Event Schema Registry لضبط نسخ مخططات الأحداث عبر الخدمات. كذلك نشير لأسلوب **حدث-كارئ (Event Sourcing)** كفكرة متقدمة حيث يخزن النظام تسلسلات الأحداث بدلاً من الحالة الحالية. المحصلة: فهم أن تصميم الأحداث ليس عشوائيًا، بل يحتاج تخطيطًا يُسهل صيانة النظام مع مرور الوقت.

المصطلحات الأساسية:

- Event Schema (مخطط الحدث)
- JSON vs Avro/Protobuf (تنسيقات الرسائل)
- Backward Compatibility (التوافقية مع الإصدارات السابقة)
- Event Sourcing (منبع الأحداث)
- Idempotency (الخصائص المعرفية – منع التكرار)

فكرة التطبيق العملي: صمم هيكل رسالة JSON لحدث **UserRegistered** (عند تسجيل مستخدم جديد). حدّد الحقول المطلوبة (مثلاً userId, name, email, timestamp) وفكّر إذا ما احتجت إلى حقل *eventId* أو *eventType* . بعد ذلك، تخيّل أنك بعد فترة قررت إضافة حقل جديد (مثلاً referralCode). كيف تضيفه دون أن تتوقف الخدمات القديمة التي لا تتوقع وجوده؟ الإجابة: بجعل وجوده اختياريًا أو توفير قيمة افتراضية عند غيابه. هذا التمرين يجعلك تفكر في **تطور الرسالة عبر الزمن**.

ملاحظات:

اليوم 22: ضمانات التسليم والتعامل مع التكرارات

شرح مختصر: تحديات **ضمان تسليم الرسائل** في الأنظمة الموزعة. سنتناول الأنماط الثلاثة: **At-most-once** (قد تُفقد رسائل لكن لا تتكرر)، **At-least-once** (لا تُفقد رسائل لكن قد تتكرر)، **Exactly-once** (لا فقدان ولا تكرار، وهو الأصعب تحقيقًا) [14] . نوضح أن معظم النظم عمليًا توفر *at-least-once* ؛ أي قد يتلقى المستهلك نفس الرسالة مرتين في حالات معينة، لذا يجب تصميم المستهلك بحيث يكون **Idempotent** (أي معالجة التكرار بدون آثار جانبية). نذكر أمثلة لتحقيق Idempotency: استخدام معرّف فريد للرسالة وتخزين آخر معرّف تم معالجته لتجاهل التكرار، أو تصميم العملية نفسها لتكون قابلة للتكرار (مثلاً إرسال نفس البريد الإلكتروني مرتين قد لا يضر إن كان التصميم يسمح بذلك أو يتحاشاه). كما نشرح ترتيب الرسائل : كيف أن بعض الوسطاء (مثل Kafka) يضمن ترتيب الرسائل ضمن **Partition** واحد ولكن ليس عبر جميعها، مما يتطلب أخذ ذلك بالاعتبار في التصميم. الهدف هنا أن يكون المهندس واعيًا لتلك التفاصيل لضمان أن نظام الأحداث يعمل بشكل صحيح حتى في الظروف الحديّة (network glitches, retries).

المصطلحات الأساسية:

- At-least once delivery (ضمان مرة على الأقل) [14]
- At-most once delivery (مرة على الأكثر)
- Exactly once (مرة واحدة تمامًا)
- Idempotency (عدم التأثر بالتكرار)
- Message Ordering (ترتيب الرسائل)

فكرة التطبيق العملي: اكتب دالة بسيطة **غير idempotent** (مثلاً تضيف عنصرًا إلى قائمة كلما تم استدعاؤها). ثم تخيل أنها تتلقى نفس الرسالة مرتين بسبب إعادة المحاولة – النتيجة ستكون مزدوجة. عدّل الدالة لتصبح **Idempotent** – مثلاً بالتحقق من **معرّف** العنصر قبل إضافته لتجنب التكرار. هذا المثال البسيط يعكس لماذا نحتاج idempotency في معالجة الأحداث.

ملاحظات:

اليوم 23: نمط Saga والمعاملات الموزعة

شرح مختصر: يتناول هذا اليوم **نمط Saga** كحل لإدارة **معاملات موزعة** عبر خدمات متعددة دون وجود قفل مركزي. نشرح المشكلة: في نظام موزع، قد تحتاج إلى تنفيذ سلسلة عمليات في خدمات مختلفة كجزء من معاملة واحدة (مثال: حجز رحلة تشمل حجز طيران وفندق وسيارة). من الصعب استخدام معاملات تقليدية ACID عبر الحدود الخدمة. هنا يأتي

Saga: هي سلسلة من **المعاملات المحلية** في كل خدمة، مع **إجراءات معوّضة (Compensating Actions)** عند فشل أحدها لإلغاء ما تم سابقًا. نوضح نوعين: *Choreography Saga* حيث يتم التنسيق عبر تبادل الأحداث بشكل ضمني (كل خدمة تستمع وتقرر)، و*Orchestration Saga* حيث توجد خدمة منسّق مركزي ترسل أوامر لكل خطوة. سنقدّم مثالًا خطوة بخطوة: إذا فشل حجز الفندق بعد حجز الطيران، يتم إرسال حدث إلغاء الطيران للتراجع [15] . سنشير أيضًا لفارق Saga عن **Two-Phase Commit** (الذي يصعب تطبيقه في microservices نظراً لتشبثه والانتظار الطويل مما يؤثر على التوفرية). فهم Saga مهم لضمان الاتساق في الأنظمة الموزعة دون التضحية بالمرونة والتعافي من الأخطاء.

المصطلحات الأساسية:

- Distributed Transaction (معاملة موزعة)
- Saga Pattern (نمط الساجا) [16]
- Compensation (إجراء معوّض/تصحيحي)
- Orchestration vs Choreography (تنسيق مركزي مقابل تفاعلي)
- Two-Phase Commit (بروتوكول المرحلتين)

فكرة التطبيق العملي: ارسم **مخطط تدفق** لسيناريو "عملية طلب في متجر إلكتروني" يستخدم Saga: الخدمات المعنية مثلاً (خدمة الطلبات، خدمة الدفع، خدمة المخزون). ارسم الخطوات: إنشاء الطلب -> حجز المخزون -> سحب الدفع. ثم ضع سيناريو فشل (مثلاً فشل سحب الدفع بعد حجز المخزون) وأضف خطوات الإرجاع (Release inventory event). هذا التمرين سيوضح لك كيف يتم التعويض للحفاظ على اتساق نظام متعدد الخدمات.

ملاحظات:

اليوم 24: تطبيق معماري موجه بالأحداث – مشروع مصغر

شرح مختصر: ختام المحور مع مشروع تطبيقي مصغر يدمج مفاهيم المعمارية الموجهة بالأحداث. لنفترض تصميم **نظام إشعارات** في منصة تعليمية: عندما ينهي الطالب درسًا، يتم إرسال حدث "LessonCompleted"، فتقوم خدمة أخرى (Notification Service) بالتقاط الحدث وإرسال إشعار بريد إلكتروني للطالب، بينما خدمة ثالثة (Analytics) تحصي الإنجاز في إحصاءات تفاعل المستخدم. سنكتب بشكل نظري (أو كود بسيط إن أمكن) كيف سيتم بناء هذا. نحدد موضوع (Topic) باسم *LessonEvents* مثلًا، خدمة **Course Service** تنشر الحدث مع بيانات (studentId, lessonId, timestamp)، **Notification Service** مشترك وتقوم بإرسال رسالة التهنئة، **Analytics Service** تخزّن الحدث. نناقش كيف أن توقف أي خدمة لا يمنع الآخر: لو تعطلت الإشعارات، يظل الحدث محفوظًا ويمكن معالجته لاحقًا دون فقدان. أيضًا نشير لأهمية مراقبة الأخطاء: مثلاً إذا فشل إرسال البريد، قد نضعه في **طابور إعادة المحاولة** . الهدف هو رؤية صورة متكاملة لكيفية بناء نظام عملي بالأحداث يعزز فهمنا النظري.

المصطلحات الأساسية:

- Event Bus (حافلة الأحداث – مجازًا لمجموعة المواضيع في النظام)
- Notification Event (حدث إشعار)
- Retry Mechanism (آلية إعادة المحاولة)
- Event Logging (تسجيل الأحداث لمراقبة النظام)
- Loose Coupling (ترابط ضعيف بين الخدمات)

فكرة التطبيق العملي: حاول (إن كانت لديك بيئة تطوير) تنفيذ هذا السيناريو بشكل مصغر: اكتب برنامجًا ينشر رسالة **اكتمال درس** (يمكن أن تكون مجرد كتابة سطر في Console تمثل الإرسال)، ثم اكتب برنامجًا آخر يستقبل تلك الرسالة (مثلاً ينتظر إدخال من البرنامج الأول عبر نظام ملفات أو message broker بسيط) ويطبع "إشعار: مبروك إنهاء الدرس!". إن لم يكن التنفيذ الفعلي ممكنًا، اكتب **خطوات التنفيذ** والواجهات بين الخدمات بوضوح.

ملاحظات:

المحور 5: هندسة الصمود والمراقبة (Resilience Engineering & Observability)

(الأيام 25–30)

وصف المحور: يتناول هذا المحور بناء أنظمة **صامدة** وقابلة للرصد. سنستكشف تقنيات لجعل الأنظمة تتحمّل الأعطال (كدوائر الحماية وإعادة المحاولة والحد من المعدل) بحيث لا تؤدي الأعطال الجزئية إلى انهيار كامل، وكذلك مفاهيم **المراقبة الشاملة (Observability)** عبر جمع **القياسات (Metrics)** و**السجلات (Logs)** وتتبع الطلبات **Tracing** لفهم حالة النظام الداخلي من مخرجاته [17] . سنغطي أيضًا موضوع **هندسة الفوضى (Chaos Engineering)** لاختبار قدرة التحمل.

اليوم 25: مبادئ التصميم المرن (Resilient Design)

شرح مختصر: مقدمة لهندسة الصمود: كيف نصمم أنظمة **مرنة** تcontinue بالعمل تحت الضغط والأخطاء. نشرح مفهوم **الصمود (Resilience)** : قدرة النظام على التعافي والاحتفاظ بوظائفه حتى عند حدوث مشكلات. نناقش مبادئ أساسية مثل **التراجع الآمن (Fail-Safe)** – تصميم النظام بحيث عند الفشل يتوقف بطريقة آمنة أو ينتج مخرجات افتراضية بدلاً من الانهيار الكامل. كذلك مبدأ **Graceful Degradation** بحيث لو تعطل جزء غير أساسي، يستمر بقية النظام بتقديم خدمة (مثال: تعطل خدمة توصيات في موقع تجارة إلكترونية لا يجب أن يمنع إتمام عملية الشراء، بل يظهر الموقع بدون توصيات). سنتحدث عن بناء **زائدية (Redundancy)** ليس فقط في المعدات بل أيضًا في المسارات (path) داخل البرنامج؛ مثلاً وجود نسخة احتياطية من خدمة في منطقة أخرى. سنشير أيضًا لأهمية **عزل الأعطال** حتى لا تنتشر (مثلاً خزانة دوائر كهربائية تمنع انفصال دائرة من إسقاط الشبكة كلها). هذا اليوم يضع أسس التفكير بشكل "ماذا لو فشل X؟" في كل أجزاء التصميم.

المصطلحات الأساسية:

- Resilience (الصمود/المرونة)
- Graceful Degradation (تدهور تدريجي آمن)
- Redundancy (زيادة/ازدواجية)
- Fault Isolation (عزل العطل)
- Single Point of Failure (نقطة فشل أحادية)

فكرة التطبيق العملي: خذ ميزة في تطبيق معروف (مثل ميزة التعليقات في موقع تواصل اجتماعي). فكّر: ماذا يحدث لو توقفت خدمة التعليقات؟ كيف يمكنك تصميم الموقع ليظل يعمل (يظهر المحتوى الرئيسي بدون التعليقات بدل أن يظهر خطأ عام)؟ أكتب تصورك لكيفية التعاطي مع هذا الفشل وإعلام المستخدم بشكل مناسب بدون انهيار النظام. هذا التفكير يقربك من عقلية التصميم المرن.

ملاحظات:

اليوم 26: أنماط الصمود – إعادة المحاولة والتراجع الدائري

شرح مختصر: دراسة **أنماط تصميم** محددة لتحقيق الصمود في وجه الأعطال المؤقتة. أولاً **إعادة المحاولة (Retry)**: عندما تفشل عملية اتصال بخدمة ما بشكل عارض (مثلاً Timeout)، القيام بمحاولات إضافية بعد تأخير قصير قد يحل المشكلة. لكن يجب الحذر من إغراق النظام بمحاولات كثيفة؛ لذا نستخدم **منحنى تراجع (Exponential Backoff)** لزيادة الفاصل الزمني بين المحاولات تدريجيًا. ثانيًا **الدائرة الكهربائية (Circuit Breaker)** : نستعير المفهوم الكهربائي لحماية النظام من الأعطال المتكررة – عندما تتكرر فشل خدمة خارجية مرات عديدة، يفصل القاطع الدائرة فيوقف الطلبات لفترة معينة بدلاً من الاستمرار بمحاولات غير مجدية [18] . نشرح حالات القاطع: مغلق (تمرير الطلبات)، مفتوح (منعها وإعطاء رد فوري بالفشل لتجنب الضغط)، نصف مفتوح (اختبار الخدمة بتجربة محدودة بعد مدة). هذه الآلية تمنع **الفشل المتتابع** من نشر أثره. سنناقش أيضًا نمط **Timeout** – تحديد زمن أقصى للانتظار على استدعاء خدمة حتى لا تحجز موارد النظام بلا طائل. عند تطبيق هذه الأنماط ستحصل على نظام أكثر تحملاً وكفاءة في استخدام الموارد [18] .

المصطلحات الأساسية:

- Retry with Backoff (إعادة المحاولة التدريجية)
- Circuit Breaker (قاطع الدائرة) [18]
- Open / Closed / Half-Open (حالات القاطع) [19]
- Timeout (المهلة الزمنية)
- Throttling (الإبطاء/تقييد معدل الطلبات)

فكرة التطبيق العملي: حاول محاكاة **Circuit Breaker** بسيط في منطق برمجي: اكتب دالة تستدعي API خارجي (يمكن استعمال طلب HTTP لموقع غير موجود لإحداث فشل). اضف منطقًا أنه إذا فشل الاستدعاء 3 مرات متتالية، فإن الدالة مباشرة تُرجع خطأ لمدة 60 ثانية دون محاولة الاتصال الحقيقي (كأنها circuit مفتوح). بعد 60 ثانية جرّب الاتصال الحقيقي مرة أخرى (نصف مفتوح). هذه المحاكاة البسيطة – حتى لو pseudo-code – ستساعدك على فهم آلية القاطع وكيف يحمي من الفشل المستمر.

ملاحظات:

اليوم 27: العزل والحدود – Bulkhead و Rate Limiting

شرح مختصر: مناقشة تقنيات **العزل (Bulkhead)** وتحديد المعدلات (**Rate Limiting**) كوسائل لتحسين صمود الأنظمة. **نمط Bulkhead** (مسمى على اسم حواجز عزل غرف السفينة) يقضي بتقسيم موارد النظام إلى مقصورات معزولة بحيث لو امتلأت إحداها (أي استنفدت موردًا ما) لا تنتقل المشكلة لباقي النظام [20] [21] . مثال: تخصيص مجموعة من مؤشرات الترابط Thread Pool لكل خدمة خارجية؛ لو علقت تلك الخدمة لا تستهلك سوى خيوط قسمها ولا توقف بقية النظام. نشرح تطبيق ذلك في برمجيات الخادم. أما **Rate Limiting** فهو تحديد عدد الطلبات المسموح بها خلال فترة زمنية من أجل حماية النظام من الحمل الزائد أو إساءة الاستخدام. نذكر خوارزميات بسيطة مثل Token Bucket و Leaky Bucket لضبط المعدل. هذه الآليات مهمة أيضًا أمنيًا لمنع هجمات DoS. الجمع بين Rate Limitg Bulkhead يمنع **الانهيار المتسلسل** : فحتى لو أساء عميل ما أو حدث تدفق غير طبيعي، يبقى الأثر محدودًا ومعزولاً. سنشير أيضًا لنمط **Fallback** – أي تقديم استجابة افتراضية أو مخزنة مؤقتًا عندما لا يمكن تنفيذ الطلب الحقيقي، كجزء من التعامل الذكي مع الفشل.

المصطلحات الأساسية:

- Bulkhead Pattern (نمط الحاجز/العزل) [20] [21]
- Thread Pool Isolation (عزل عبر مجموعة خيوط)
- Rate Limiting (تحديد المعدل)
- Token Bucket / Leaky Bucket (خوارزميات تحديد السرعة)
- Fallback (الرجوع لاستجابة بديلة)

فكرة التطبيق العملي: ارسم تصورًا لكيفية تطبيق Bulkhead على مستوى قاعدة بيانات: مثلاً لديك خادم تطبيق يقوم بعمليات قراءة وأخرى كتابة. كيف تستطيع فصل الموارد بحيث لا تؤدي كثرة عمليات القراءة إلى منع عمليات الكتابة؟ ربما الحل بتخصيص **اتصالات منفصلة** أو حتى قاعدة بيانات ثانوية للقراءة. اكتب تصوّرك وخطوات ما سيحدث لو زاد حمل القراءة كثيرًا في هذا التصميم، وكيف يظل نظام الكتابة بمنأى.

ملاحظات:

اليوم 28: هندسة الفوضى واختبار التحمل

شرح مختصر: التعرف على **Chaos Engineering (هندسة الفوضى)** كمنهج لاختبار صمود الأنظمة عن طريق إدخال أعطال متعمدة. سنسرد قصة **Netflix Chaos Monkey** – أداة تقوم بشكل عشوائي بإيقافInstances في الإنتاج للتأكد أن الخدمات مصممة للتعامل مع فقدان أي عقدة [22] . نوضح أن الفكرة ليست إحداث فوضى حقيقية بل محاكاة ظروف الفشل في بيئة مسيطر عليها لاستخلاص دروس قبل أن تحدث في الواقع. سنناقش المبادئ الأساسية: ابدأ بخطوات صغيرة (على نطاق خدمة واحدة)، راقب النتائج، انشر "قرود" أخرى لتعطل أشياء مختلفة (قاعدة بيانات، شبكة، حاوية...) – وقد طبقت Netflix ذلك في *Simian Army* بأكملها (Chaos Gorilla, Chaos Kong لتعطيل منطقة كاملة) [23] [22] . سنتحدث عن أهمية وجود أدوات مراقبة متقدمة قبل تبني اختبارات الفوضى حتى تلاحظ آثارها بدقة. أيضًا نشجع ثقافة **اللعب الحربي (Game Days)** حيث يقوم الفريق بتمارين انقطاع خدمات لمعرفة جاهزيتهم. الهدف هو أن يكون

النظام **مضادًا للهشاشة** – أي يتحسن من خلال التجارب الفاشلة بدلاً من أن ينهار.

المصطلحات الأساسية:

- Chaos Engineering (هندسة الفوضى)
- Chaos Monkey (قرد الفوضى من Netflix) [23] [22]
- Failure Injection (حقن الأخطاء)
- Simian Army (جيش القردة – أدوات Netflix) [23]
- Game Day (تمرين محاكاة الأعطال)

فكرة التطبيق العملي: اختر خدمة في نظامك أو تطبيق صغير بنيته (حتى لو على جهازك)، وقم بخلق سيناريو عطل متعمد: مثلاً قطع الاتصال عن قاعدة البيانات لمدة 30 ثانية وانظر كيف يتصرف التطبيق (قد تحتاج لبيئة اختبارية لذلك). سجّل ما حدث: هل انهار فورًا؟ هل أعاد المحاولة؟ هل ظهرت رسائل أخطاء واضحة؟ فكر ما التغييرات التي ستجريها ليصمد بشكل أفضل (مثلاً إضافة Retry أو Timeout أقصر). هذه التجربة تشبه تطبيق مبدأ Chaos Engineering على نطاق صغير ومتحكم به.

ملاحظات:

اليوم 29: المراقبة والقياس – المؤشرات والتنبيهات

شرح مختصر: انتقال إلى الجانب الثاني من المحور: **Observability (قابلية المراقبة)** . نبدأ بالشق الأول منها وهو **المؤشرات (Metrics)** . نشرح أنظمة المراقبة التقليدية تجمع مقاييس رقمية عن النظام: معدل الطلبات في الثانية، زمن الاستجابة المتوسط، نسبة استخدام CPU والذاكرة، عدد الأخطاء... هذه المقاييس تسمى **الركائز الثلاث للملاحظة** إلى جانب السجلات والتتبع [17] . نوضح أداة مثل **Prometheus** وكيف تسحب المقاييس من التطبيقات بشكل دوري وتخزنها. أيضًا **Grafana** كأداة لعرض الرسوم البيانية للمقاييس. سنتحدث عن بناء **لوحات مراقبة (Dashboards)** تظهر الحالة الصحية الحالية. ثم **التنبيهات (Alerts)** : تحديد عتبات على هذه المقاييس (مثلاً إذا تجاوزت نسبة الخطأ 5% خلال 5 دقائق) وإطلاق تنبيه إلى الفريق (بريد، PagerDuty...). نشرح أهمية ضبط التنبيهات بعناية لتجنب الإرهاق بالتنبيهات الكاذبة. كذلك نفرّق بين **المقاييس التقنية** (CPU, Memory) و**مقاييس العمل** المهمة (مثل عدد عمليات التسجيل الجديدة، معدلات التحويل) فكلاهما مهم لتقييم صحة النظام. بنهاية اليوم، ستكون قد كوّنت صورة عن كيفية مراقبة نظام في الإنتاج بشكل استباقي.

المصطلحات الأساسية:

- Metrics (المؤشرات/المقاييس) [17]
- Monitoring (المراقبة)
- Prometheus / Grafana (برمجيات رصد ورسم)
- Dashboard (لوحة معلومات)
- Alerting (التنبيه)

فكرة التطبيق العملي: حتى لو لم يكن لديك نظام معقد، جرّب استخدام أدوات بسيطة متاحة: مثلاً مكتبة *Prometheus client* لأي لغة (مثل **prom-client** لجافاسكريبت) لعمل عدّادrequests. قم بتشغيل برنامج يولّد بعض المؤشرات وصمم **لوحة Grafana** (يمكن استخدام Grafana Cloud مجانًا) لعرض تلك المؤشرات. إذا تعذر التطبيق العملي، على الأقل خطط لشكل لوحة تحب أن تراها لخدمة ما، وحدد 5 مؤشرات أساسية وترسمها (مثل: معدلRequests، زمن استجابة P95، عدد الأخطاء 500 خلال آخر 1 ساعة، ...).

ملاحظات:

اليوم 30: السجلات وتتبع الطلبات (Logs & Tracing)

شرح مختصر: استكمال عناصر **Observability** بالشقين الآخرين: **السجلات (Logs)** و**التتبّع (Tracing)**. أولاً، **السجلات**: ملفات أو نظم تجمع رسائل نصية تسجل أحداث تشغيل النظام (طلبات واردة، أخطاء استثنائية، معلومات تصحيحية). نشرح أهمية هيكلة السجلات – من الأفضل أن تكون بشكل منظم (مثل JSON) لتسهيل تحليلها بأدوات مركزية (كمنصة ELK: Elasticsearch/Logstash/Kibana). نذكر ممارسات مثل تضمين معرّفات تتبع في كل سجل لربط الأحداث بطلب محدد. ثم **التتبّع الموزع (Distributed Tracing)** : نوضح مشكلة تتبع طلب عبر عدة خدمات، والحل عبر بروتوكولات مثل

OpenTelemetry أو **Jaeger** حيث يتم توليد معرف Trace ID يرافق الطلب عبر الخدمات المختلفة، مما يمكّن رسم خط رحلة الطلب من البداية للنهاية. هذا مفيد جدًا لتشخيص مواطن البطء: سترى أن الخدمة X استغرقت 100ms والخدمة Y 2s في ذات الطلب مثلا. نشرح أدوات مثل **Jaeger** و**Zipkin** التي تقدم واجهة لاستعراض هذه الـTraces. بنهاية اليوم، يكون واضحًا كيف تتكامل السجلات والقياسات والتتبع لتحقيق الرؤية الكاملة للنظام (ما يسمى **"Three Pillars of Observability: logs, metrics, traces"** [17]).

المصطلحات الأساسية:

- Structured Logs (سجلات منظمة)
- ELK Stack (Elastic Logstash Kibana)
- Distributed Trace (التتبع الموزع)
- OpenTelemetry (أوبن تيليمتري)
- Trace ID / Span ID (معرّف التتبع والأجزاء)

فكرة التطبيق العملي: إذا لديك تطبيق متعدد الخدمات (حتى لو خدمتين)، جرّب تطبيق تتبع بسيط: اجعل كل خدمة تطبع سطر Log يحوي `trace_id` موحد يرسل مع الطلب (يمكن محاكاة ذلك بتمرير رقم يدوي). أرسل طلبًا يمر بالخدمتين ولاحظ في ملفات السجل كيف يمكنك تتبع التسلسل عبر `trace_id` . لو توفر لك استخدام Jaeger أو Zipkin فذلك أروع – أضف مكتبة التتبع في كل خدمة واختبر رؤية المسار عبر واجهة Jaeger. هذا التدريب يوضح لك قيمة إضافة trace id في فهم سلوك النظام الموزع.

ملاحظات:

المحور 6: أمن الـBackend المتقدم (Advanced Backend Security)

(الأيام 31–36)

وصف المحور: يتناول هذا المحور جوانب متقدمة من أمن تطبيقات وخدمات الـBackend. سنغطي آليات التحقق والصلاحيات (Auth) بشكل متقدم (OAuth2, JWT, SSO)، حماية الواجهات البرمجية من الهجمات الشائعة (حقن SQL, XSS, CSRF)، التشفير وتأمين البيانات الحساسة، أمن الخدمات المصغرة (مثل تأمين الاتصالات بين الخدمات عبر mTLS)، دمج الأمن في خط التطوير (DevSecOps) وأتمتة الفحص الأمني، وأخيرًا المراقبة الأمنية والاستجابة للحوادث.

اليوم 31: المصادقة والتفويض – OAuth2 و JWT

شرح مختصر: الغوص في مجال **Authentication** (التحقق من هوية المستخدم) و**Authorization** (تفويض الصلاحيات). نبدأ بمراجعة سريعة لأساليب التحقق التقليدية (مثل الجلسات Tokens مع **Cookies**) ثم ننتقل إلى الأنماط الحديثة: **OAuth2** كإطار للتفويض يستخدم بكثرة للسماح لتطبيقات خارجية بالوصول لحساب المستخدم بشكل آمن (مثل تسجيل الدخول عبر Google). نشرح مفاهيم OAuth2: *Resource Owner, Client, Authorization Server, Access Token*. نذكر التدفق الشائع Authorization Code Flow. ثم نتطرق إلى **JSON Web Tokens (JWT)** وهي رموز ترميزية تُستخدم لنقل معلومات الهوية والصلاحيات بشكل آمن بين الأطراف؛ JWT موقّع رقمياً ويمكن التحقق منه دون طلب مستمر من الخادم. نشرح تركيب JWT (Header.Payload.Signature) وكيف نضمن عدم التلاعب به بالتوقيع. نشير لاستخدام JWT في **OAuth2** كـ Access Token أحيانًا (في OpenID Connect). نوضح أيضًا مزايا JWT (ذاتي الاكتفاء stateless) مقابل مخاطره (لو سُرّب فخطير). كذلك نذكر Single Sign-On (SSO) وكيفية بناء نظام موحّد للتحقق في مؤسسة كبيرة.

المصطلحات الأساسية:

- Authentication vs Authorization (المصادقة مقابل التفويض)
- OAuth2 (أووث2)
- Access Token / Refresh Token (رمز دخول/تجديد)
- JSON Web Token – JWT (توكين JWT)
- OpenID Connect (الطبقة التعريفية فوق OAuth2)

فكرة التطبيق العملي: جرّب استخدام خدمة OAuth عامة: مثلاً، تابع دليل "تسجيل دخول بواسطة GitHub API" لإنشاء

تطبيق OAuth بسيط. أو بشكل أبسط: استخدم مكتبة لإنشاء JWT (يتضمن claims مثل user_id role) وتوقيعه بمفتاح سري، ثم في جزء آخر من البرنامج تحقق من صحة التوقيع واستخرج البيانات. سيساعدك ذلك على فهم تدفق إنشاء وتحقق JWT وكيف يمكن استخدامه لتخزين حالة تسجيل الدخول.

ملاحظات:

اليوم 32: حماية الواجهات البرمجية والثغرات الشائعة

شرح مختصر: تسليط الضوء على **أمان واجهات الـAPI** ضد أشهر الهجمات. نبدأ بـ **حقن SQL** : كيف يمكن لاستعلام غير آمن أن يسمح للمهاجم بتمرير أوامر غير متوقعة (مثل `'; DROP TABLE --`). نشرح أهمية الاستعلامات المعلمة (Prepared Statements) أو ORM لتفادي ذلك. ثم **XSS (Cross-Site Scripting)** حتى لو هو في الواجهة الأمامية، لكن الـBackend يمكنه المساعدة بتنقية المخرجات وعدم تخزين سكربتات خطرة. نتحدث عن **CSRF (Cross-Site Request Forgery)** وكيفية حماية APIs (مثلاً رموز CSRF في التطبيقات ذات الـCookie، أو في APIs stateless بالاعتماد على نفس مفاهيم OAuth2 scopes). نمر على **HEADERS الأمن** مثل Content-Security-Policy, CORS اعتبارات CORS عند بناء API مفتوح. سنشير إلى مصطلح **OWASP Top 10** – أشهر 10 ثغرات تطبيقات ويب – والتأكيد أن المهندس الخلفي يجب أن يكون ملمًّا بها (مثل مشاكل deserialization، ضبط خاطئ للأمن، الخ). كذلك سنتحدث عن **تValidations** – أي التحقق من مدخلات المستخدم على الطبقة الخلفية دومًا وعدم افتراض أن الواجهة الأمامية قامت بذلك. الهدف: بناء خلفية حصينة تجعل الاختراق أصعب بكثير.

المصطلحات الأساسية:

- SQL Injection (حقن SQL)
- XSS (ثغرة البرمجة عبر المواقع)
- CSRF (تزوير الطلبات عبر المواقع)
- Input Validation (تحقق المدخلات)
- OWASP Top 10 (أهم 10 ثغرات حسب أوواسپ)

فكرة التطبيق العملي: اختر ثغرة من OWASP Top 10 (مثلاً SQLi أو XSS) وابحث عن مثال لها. قد تجد أمثلة جاهزة في بيئات تعليمية (مثل DVWA – Damn Vulnerable Web App). جرّب بشكل آمن تنفيذ هجوم SQLi على استعلام غير آمن (مثلاً إدخال `' OR '1'='1` في حقل كلمة المرور لتخطي التحقق). ثم أصلح الاستعلام باستخدام معاملات Parametrized ولاحظ كيف يبطل الهجوم. هذه التجربة التعليمية ستترك أثرًا قويًا لفهم خطورة الثغرات وأهمية معالجتها.

ملاحظات:

اليوم 33: التشفير وتأمين البيانات الحساسة

شرح مختصر: شرح دور **التشفير (Encryption)** في أمن الـBackend. سنقسمه لشقين: **التشفير في النقل (In-Transit)** و **التشفير في التخزين (At-Rest)** . في النقل: أهمية استخدام **HTTPS/TLS** لكل الاتصالات الخارجية والداخلية أيضًا، لضمان عدم اعتراض البيانات الحساسة (مثل JWT أو بيانات المستخدم) أثناء انتقالها. نذكر إصدارات TLS الحديثة وضرورة إلغاء القديم (TLS1.0/1.1). في التخزين: نقاش حول تشفير قواعد البيانات أو على الأقل الأعمدة الحساسة (مثل أعمدة تحتوي أرقام بطاقات ائتمان أو معرفات شخصية) باستخدام مفاتيح. توضيح الفرق بين **Hashing** (تجزئة) لكلمات المرور مثلاً وعدم تخزينها بنص واضح، بالإضافة لاستخدام **Salt** لمنع قواميس القاموس. نعرّج على إدارة المفاتيح: لا ينبغي وضع المفاتيح السرية في الشيفرة مباشرة، بل استخدام مخازن أسرار (Vault, AWS KMS). سنشرح أيضًا مفهوم **Signing** لضمان سلامة البيانات (مثل توقيع JWT أو payloads) مقابل التشفير الذي يضمن السرية. وأخيرًا، ممارسات مثل **عدم إعادة اختراع التشفير** – بل استخدام المكتبات القياسية لتجنب الأخطاء. بعد هذا اليوم سيكون لديك فهم لكيفية حماية البيانات سواء "على السلك" أو "وهي مخزنة".

المصطلحات الأساسية:

- HTTPS/TLS (بروتوكول تأمين النقل)
- Encryption at Rest (تشفير البيانات وهي مخزنة)
- Hashing + Salt (تجزئة مع ملح)

- Key Management (إدارة المفاتيح)
- Digital Signature (توقيع رقمي)

فكرة التطبيق العملي: تأكد من أن بيئة التطوير لديك تستخدم اتصالات آمنة: مثلاً إذا لديك API تتواصل معه، جرّب تشغيله عبر HTTPS (يمكن استخدام شهادات محلية self-signed للتجربة). أيضًا جرّب استخدام مكتبة Hash (مثل BCrypt) لتجزئة كلمة مرور، ثم تحقق من أن التجزئة مختلفة عند تغيير الملح أو نفس الكلمة. ولاحظ كيف أن مقارنة كلمة المرور تصبح عبر مقارنة التجزئة وليس النص الأصلي. هذا التمرين يظهر لك عمليًا تأثير استخدام التجزئة والملح.

ملاحظات:

اليوم 34: أمن الخدمات المصغرة والاتصالات الداخلية

شرح مختصر: التركيز على تأمين البيئة الداخلية للخدمات (Microservices Security). عندما يكون لدينا عشرات الخدمات تتواصل عبر الشبكة، نحتاج ضمان أنها تتواصل بأمان أيضًا وليس فقط المستخدمين الخارجيين. نعرّف مفهوم **Service-to-Service Authentication** – مثل استخدام هويات خاصة للخدمات (Service Accounts) كي تتأكد خدمة A أن المتصل هي خدمة B المصرّح لها، وليس جهة دخيلة. نناقش الحلول مثل **mTLS (mutual TLS)** حيث يتم تبادل شهادات رقمية بين الخدمات للتحقق المتبادل [24] . نذكر أنظمة **Service Mesh** (مثل Istio) التي تسهّل إضافة طبقة أمان على اتصالات الخدمات دون تغيير كودها – بما فيها تشفير الاتصال داخليًا وإدارة الشهادات. سنتحدث عن **تفويض** بين الخدمات: مثلاً Service A لديها صلاحية محددة للوصول إلى بيانات من Service B (ممكن عبر توكين JWT خاص بالخدمات). أيضًا **أمن الـAPI Gateways** التي تقف أمام مجموعات الخدمات: تطبيق سياسات موحّدة، معدل طلبات، فحص توكين موحد. ولن نغفل **مبدأ الأقل امتيازًا (Least Privilege)** – كل خدمة أو مكون يجب أن يحصل فقط على الحد الأدنى من الصلاحيات التي يحتاجها (سواء بالاتصال بقاعدة البيانات أو نداء واجهة خدمة أخرى). بهذا اليوم، سنفهم أن الأمن ليس فقط عند الحافة (edge) مع المستخدم، بل أيضًا داخل النظام بين الأجزاء المختلفة.

المصطلحات الأساسية:

- mTLS (TLS Mutual Authentication)
- Service Account (حساب خدمة مميز)
- API Gateway (بوابة واجهات برمجية)
- Service Mesh (شبكة خدمات)
- Least Privilege (مبدأ أقل الصلاحيات)

فكرة التطبيق العملي: إذا سمحت لك الفرصة، جرّب إعداد اتصالات mTLS بين خادمين محليين: استخدم مثلاً **OpenSSL** لإنشاء شهادة ذاتية التوقيع لكل "خدمة" وقم ببرمجة اتصال HTTPS ثنائي التحقق (هناك أمثلة جاهزة بالويب). ستلمس كيف يجب تبادل الشهادات والتأكد منها. إذا تعذر ذلك، اكتب خطة موجزة لكيفية نشر mTLS في شركة لديها 50 خدمة – ما الأدوات التي ستستخدمها لإدارة الشهادات والتجديد، وكيف ستوزع المفاتيح بشكل آمن.

ملاحظات:

اليوم 35: DevSecOps – دمج الأمن في دورة التطوير

شرح مختصر: يتناول هذا اليوم منهجية **DevSecOps** أي جعل الأمن جزءًا لا يتجزأ من عملية التطوير والنشر، بدلاً من معالجة الأمن بعد وقوع المشكلة. نشرح أنشطة يجب إدخالها في خط أنابيب CI/CD: مثل **تحليل الثغرات** في التبعيّات (Dependencies) باستخدام أدوات (مثل OWASP Dependency Check أو npm audit) للكشف عن إصدارات المكتبات المعروفة بضعف أمني. أيضًا **تحليل الكود الثابت (SAST)** بأدوات مثل SonarQube أو GitHub CodeQL لاكتشاف أنماط كود خطرة (مثل حقن). كذلك **اختبارات الاختراق الديناميكية (DAST)** الآلية على بيئة الاختبار – مثل استخدام OWASP ZAP لفحص تطبيق الويب بحثًا عن الثغرات الشائعة. نعرّج على أهمية ثقافة أمنية: مثل مراجعات كود تركز على نقاط الضعف، وبرامج مكافآت لاكتشاف الثغرات (Bug Bounty) بعد الإنتاج. نتحدث أيضًا عن **Infrastructure as Code Security** – فحص ملفات إعداد البنية (Dockerfiles, Terraform) لضمان عدم وجود إعدادات خطرة (فتح منافذ غير ضرورية مثلاً). الهدف أن يصبح الأمن مسؤولية الجميع في الفريق منذ اليوم الأول وليس مهمة تضاف في النهاية.

المصطلحات الأساسية:

- DevSecOps (تكامل التطوير والعمليات والأمن)

- SAST/DAST (تحليل الكود الثابت/الديناميكي)
- Dependency Vulnerability (ثغرة في المكتبات)
- Security Pipeline (خط أمني في CI/CD)
- Code Review (مراجعة الكود مع اعتبار الأمان)

فكرة التطبيق العملي: أضف أداة بسيطة لفحص الأمان في مشروعك الحالي – مثلاً قم بتشغيل `npm audit` أو `pip list --outdated` مع البحث عن CVEs معروفة، أصلح على الأقل واحدًا من التحذيرات إن وجد. أو جرّب أداة مفتوحة المصدر مثل **gosec** (للغة Go) أو **bandit** (للغة Python) لتفحص مشروعك وتعرض أي نقاط محتملة. توثّق هذه النتيجة وأصلح ما يمكن إصلاحه. هكذا ستشعر بعملية *shift-left* للأمن خلال التطوير.

ملاحظات:

اليوم 36: مراقبة الأمن والاستجابة للحوادث

شرح مختصر: ختام محور الأمن بالحديث عن **الرصد الأمني المستمر** وكيفية الاستجابة عند وقوع حوادث. نشرح دور **سجلات التدقيق (Audit Logs)** التي تسجل العمليات الحساسة (كعمليات تسجيل الدخول، تغيير الصلاحيات، محاولات الوصول المرفوضة) وكيف يجب الاحتفاظ بها وتحليلها. نتحدث عن أنظمة **IDS/IPS** (كشف التسلل ومنع التسلل) التي قد تكون جزءًا من البنية وتراقب الأنشطة المشبوهة (مثل نمط طلبات معين يدل على محاولة اختراق) وتتخذ إجراء أو تنبه الفرق. كذلك **SIEM** (منصة إدارة معلومات الأمن والأحداث) التي تجمع سجلات شاملة من مختلف الأنظمة وتحللها تلقائيًا لاكتشاف الحوادث وتوليد التنبيهات. سنشير لمفهوم **التهديدات المتقدمة** وأنه يجب متابعة تحديثات الأمن والتقارير (مثلاً Zero-day vulnerabilities). في الاستجابة للحوادث (Incident Response)، نحدد خطوات أساسية: التعرف، الاحتواء، الاستئصال، الاستعادة، ثم **ما بعد الحادث** (Post-mortem) لتحسين النظام وسد الثغرات. سنؤكد على وثائق خطة الاستجابة للطوارئ حتى يعرف الفريق بالضبط ما العمل عند اكتشاف اختراق (من يتصلون، كيف يعزلون الأنظمة...). الهدف هو أن يكون للمهندس الخلفي دراية بأنه ليس كافيًا بناء أنظمة آمنة، بل أيضًا مراقبتها باستمرار ومعرفة ما يجب فعله إن تعرضت لهجوم.

المصطلحات الأساسية:

- Audit Log (سجل التدقيق الأمني)
- Intrusion Detection/Prevention (كشف/منع التسلل)
- SIEM (منصة إدارة الأحداث الأمنية)
- Incident Response (استجابة الحوادث)
- Post-mortem (تحليل ما بعد الحادث)

فكرة التطبيق العملي: اكتب **خطة استجابة لحادث أمني تخيّلي** : مثلاً اكتشاف أن خدمة ما تتعرض لـSQL Injection وتسريب بيانات. ما الخطوات الفورية؟ (قد تتضمن فصل الخدمة عن الشبكة، تغيير كلمات مرور قاعدة البيانات، إعلام فريق الأمن، تفعيل خطة اتصالات الأزمة). وما الخطوات اللاحقة؟ (تحليل السجلات لمعرفة مدى الاختراق، patch الثغرة، تواصل مع المستخدمين المتأثرين إن لزم، تحسين عمليات الاختبار لمنع تكرارها). مجرد كتابتك لهذه الخطة سيجعلك تدرك نقاط الضعف التنظيمية والتقنية التي يجب تحسينها قبل حدوث الواقعة الحقيقية.

ملاحظات:

المحور 7: الأداء والتزامن (Performance & Concurrency)

(الأيام 37–42)

وصف المحور: في هذا المحور سنركز على تحقيق أداء عالي في التطبيقات الخلفية وفهم معالجة **التزامن (Concurrency)** داخل التطبيق. سنغطي قياس الأداء وتحليله، تحسين الكود وفعالية الخوارزميات، مبادئ البرمجة متعددة الخيوط (Multithreading) والتحديات كظروف التسابق (Race Conditions) والتزامن الآمن، التصاميم المتوازية للتعامل مع الحمل العالي (مثل البرمجة غير المتزامنة Reactive)، وأدوات اختبار الأداء (Profiling, Load Testing) لصقل النظام.

اليوم 37: مقاييس الأداء وفهمها

شرح مختصر: مقدمة لمفهوم **أداء** التطبيقات وكيفية تعريفه وقياسه. نشرح أهم مقاييس الأداء على مستوى الـBackend: **زمن الاستجابة (Response Time/Latency)** – الوقت الذي يستغرقه الطلب ليعطي نتيجة؛ **الإنتاجية (Throughput)** – عدد العمليات أو الطلبات في وحدة الزمن؛ **استهلاك الموارد** (CPU, Memory, I/O). نوضح الفرق بين **Latency** و**Throughput** (قد يكون زمن الاستجابة جيد لكن المعدل منخفض أو العكس، والهدف تحقيق توازن). سنتعرف على مصطلحات مثل **P99 latency** (زمن الاستجابة عند الشريحة 99% من الطلبات، وهو مقياس لجودة الخدمة تحت الضغط). أيضًا نناقش **اتفاقيات مستوى الخدمة (SLA/SLO)** التي تحدد مستهدفات زمن الاستجابة أو التوفرية. ثم نتطرق لأدوات **Profiling** – التي تسمح لنا بقياس أين يقضي البرنامج وقته (مثلاً أي دالة تستهلك معظم CPU). نذكر أساليب بسيطة مثل إضافة توقيتات Logging بنقاط معينة، وأدوات متقدمة كالملفات الشخصية (profilers) الخاصة بلغات مختلفة (مثل *VisualVM* لجافا أو *cProfile* لبايثون). خلاصة اليوم: قبل تحسين الأداء يجب أن نفهمه ونقيسه كمياً.

المصطلحات الأساسية:

- Latency (زمن الاستجابة)
- Throughput (الإنتاجية/عدد العمليات)
- Percentiles (المؤشرات المئوية كـ P95, P99)
- SLA/SLO (اتفاقيات مستوى الخدمة وأهدافها)
- Profiling (تحليل الأداء التفصيلي)

فكرة التطبيق العملي: خذ جزءًا من كود لديك (أو اكتب خوارزمية بسيطة تقوم بعمليات كثيرة) وقم بقياس الزمن المستغرق باستخدام **أداة بروفيلر** إن أمكن. مثلًا في بايثون استخدم `cProfile` لمعرفة أي الدوال أبطأ. أو ببساطة، ضع نقاط طباعة توقيت قبل وبعد أجزاء مختلفة لترى أيها الأكثر استهلاكًا للوقت. اكتب تقريرًا قصيرًا بما وجدته – ربما تفاجأ أن جزءًا معينًا استغرق 80% من الزمن. هذه الخطوة الأولى دائمًا لتحسين الأداء: **قياس ثم فهم**.

ملاحظات:

اليوم 38: تحسين الأداء – هياكل البيانات والخوارزميات

شرح مختصر: التركيز على **كفاءة الكود** من ناحية اختيار **الخوارزميات وهياكل البيانات** المناسبة. نستعرض تذكيرًا سريعًا بمفاهيم التعقيد الزمني Big-O لأهم العمليات (البحث، الفرز، الإدخال...) لندرك أثر اختيار هيكل معين (مثل ArrayList مقابل LinkedList، أو Dictionary/Map مقابل البحث الخطي). نذكّر بأن مهندس الـBackend وإن كان لا يكتب خوارزميات معقدة يوميًا، إلا أن عليه فهم تأثير قراراته: مثلاً استخدام عملية فرز O(n log n) على مليون سجل، أو القيام بها بشكل غير ضروري متكرر قد يقتل الأداء. نناقش حالات شائعة: مثل قراءة بيانات من قاعدة البيانات ثم فرزها برمجيًا مقارنة باستخدام أمر SQL للفرز في قاعدة البيانات (Delegation to the right layer). كذلك نذكر تقنيات **Caching** (تم تغطيتها سابقًا في محور البيانات، لكن نؤكد استخدامها لتحسين الأداء بتجنب العمليات المتكررة المكلفة). أيضًا، التحسين الدقيق مثل تجنب حلقة داخل حلقة بدون داع، أو استخدام مكتبات/دوال مدمجة سريعة (مثلاً استخدام `map/reduce` المضمنة بلغة C عوض حلقة Python بحتة). الهدف أن تصبح **Performance Aware** عند كتابة الكود – ليس هاجسًا دائمًا، ولكن تعرف متى تحذر: أي جزء قد يكون عنق زجاجة وتتعامل معه بذكاء.

المصطلحات الأساسية:

- Big-O Notation (تحليل التعقيد الحسابي)
- Time Complexity (تعقيد زمني) / Space Complexity (تعقيد مكاني)
- Data Structures (هياكل بيانات: قوائم، خرائط، أشجار...)
- Algorithm Optimization (تحسين الخوارزميات)
- Premature Optimization (التحسين المبتسر وغير الضروري)

فكرة التطبيق العملي: استعرض جزءًا من كود قمت بكتابته سابقًا – ربما دالة معالجة بيانات – وحاول تحليل التعقيد: هل تحتوي حلقة داخل حلقة (O(n^2))؟ كم يمكن أن يكون حجم n الأسوأ؟ هل من وسيلة لجعلها أكثر كفاءة؟ مثلًا استخدام هيكل بيانات إضافي لتحويل بعض العمليات إلى O(n) بدلًا من O(n^2). حتى لو كان التفكير نظريًا، دوّنه. هذه المراجعة ستغرس مهارة تقييم الكود من منظور الأداء قبل وقوع المشاكل.

ملاحظات:

اليوم 39: أساسيات التزامن والبرمجة متعددة الخيوط

شرح مختصر: مقدمة لموضوع **التزامن (Concurrency)** داخل التطبيق. نشرح الفرق بين **Parallelism (التوازي)** وConcurrency: حيث التوازي يعني تشغيل مهام حرفيًا في نفس الوقت (على معالجات متعددة مثلاً) بينما التزامن يعني التعامل الذكي مع المهام المتعددة بحيث تبدو متزامنة (حتى لو على معالج واحد عبر تبديل السياق). نستعرض نموذج **Multithreading (تعدد الخيوط)** – تشغيل عدة خيوط threads ضمن عملية واحدة، وكيف يمكن أن يزيد الاستفادة من المعالج المتعدد الأنوية أو حتى لتحسين التأدية في مهام I/O (حيث ينتظر خيط I/O بينما آخر يعمل). نناقش مفهوم **Shared Memory** بين الخيوط وما يجره من تعقيدات. ثم نوضح **Thread Safety (أمان الخيوط)**: أي كتابة كود يعمل بشكل صحيح عند وجود خيوط متعددة دون تنازع على نفس البيانات في الذاكرة. نذكر هنا المشاكل الكلاسيكية: **Race Condition (حالة التسابق)** – عندما تحاول خيطان تعديل متغير بنفس الوقت مما يؤدي لنتائج خطأ [25]؛ **Deadlock** – عندما تنتظر خيوط موارد محتجزة لدى بعضها في حلقة مفرغة؛ **Starvation** – عندما لا تنال خيط معينة حصتها من التنفيذ. نشرح سريعًا المزامنة primitives: **Locks/Mutexes, Semaphores** وكيف يمكن استخدامها للتنسيق ولكن بحذر (لأنها قد تسبب deadlock إن أسيء استخدامها). الهدف اليوم فهم لماذا التزامن صعب وما الأساسيات التي يجب تذكرها عند كتابة كود متعدد الخيوط.

المصطلحات الأساسية:

- Concurrency vs Parallelism (التزامن مقابل التوازي)
- Thread / Process (خيط مقابل عملية)
- Race Condition (شرط التسابق) [25]
- Deadlock / Starvation (قفلة ميتة / تجويع الخيوط)
- Mutex / Lock / Semaphore (آليات القفل والتزامن)

فكرة التطبيق العملي: اكتب برنامجا صغيرًا (بلغة تدعم threads كـ Java أو Python threading) يقوم بزيادة عدّاد عالمي في حلقة بواسطة مثلاً 4 خيوط بالتوازي. اجعل كل خيط يزيد العدّاد 10000 مرة. بعد انتهاء الخيوط اطبع النتيجة، غالبًا ستجدها أقل من 4 *10000* بسبب ظروف التسابق. ثم حاول إصلاح ذلك باستخدام قفل *(Lock)* حول عملية الزيادة. قارن النتائج. هذا التمرين يجسد لك مشكلة *race condition* وحلها باستخدام *synchronization*.

ملاحظات:*

اليوم 40: مشكلات التزامن والحلول – حالات عملية

شرح مختصر: تناول أكثر تعمقًا لمشكلات تظهر في البرمجة المتزامنة وكيفية التعامل معها. سنناقش **حالة التسابق (Race Condition)** بالتفصيل مع مثال عملي (كما قد جربنا باليوم السابق) وكيف أن التتبع قد يكون صعبًا لأنها تعتمد على تواقيت غير متوقعة. ثم **السباق عند البداية (Startup race)** أو **Double-checked locking** ونشرح لماذا أحيانًا حلول تبدو سليمة قد تفشل بسبب أوامر المعالج أو ترتيب الذاكرة. نناقش **آلية الذاكرة Memory Model** للغات: لأن فهم كيف يتصرف المعالج مع الذاكرة مهم (concept of volatile variable في Java مثلاً لضمان رؤية التحديثات بين الخيوط). نتكلم عن **Deadlock** بتفصيل: كيف قد يحدث عند أخذ قفلين بترتيب مختلف في خيطين مختلفين، ونذكر استراتيجية لتجنبه (مثل ترتيب ثابت لاكتساب الأقفال، أو استخدام أقفال عالية المستوى مثل ReadWriteLock إن أمكن). كذلك **Livelock** (الخيوط تعمل لكنها لا تنجز شيئًا). ثم نسرد بعض **المكتبات والتجريدات** المفيدة: مثل **Concurrent Collections** (قوائم أو خرائط thread-safe جاهزة) التي تغني عن استخدام الأقفال اليدوية كثيرًا. ونشير إلى **Parallel programming frameworks** (مثل Fork/Join في Java أو Parallel LINQ في .NET) وكيف يمكن الاستفادة منها بدل خيوط خام. الهدف هو تجهيزك للتفكير بالسيناريوهات غير المرئية بسهولة التي يجب اختبارها عند بناء نظام متعدد الخيوط.

المصطلحات الأساسية:

- Memory Model (نموذج الذاكرة للغة)
- Volatile (متغير فولاتيل يضمن رؤية الخيوط)
- Deadlock (قفلة ميتة) / Livelock
- Thread-safe Collections (هياكل بيانات آمنة للخيوط)
- Atomic Operations (عمليات ذرية لا تنقسم)

فكرة التطبيق العملي: استكشف مكتبة جاهزة خاصة بالتزامن في لغتك. مثلاً في Java استخدم `ConcurrentHashMap` بدل `HashMap` في برنامج متعدد الخيوط للوصول للهيكل. أو في Python استخدم `queue.Queue` (وهي thread-safe) بدل قائمة عادية لمحاكاة طابور منتج/مستهلك. جرّب وضع عناصر من خيط والإخراج من خيط آخر باستخدام هذه البنية، وينبغي أن تعمل بسلاسة دون الحاجة لأقفال إضافية. بذلك تتعرف على قوة استخدام الأدوات الموثوقة بدلاً من تنفيذ كل شيء بنفسك.

ملاحظات:

اليوم 41: تصميم أنظمة عالية الأداء – البرمجة غير المتزامنة

شرح مختصر: نتناول أساليب معمارية وبرمجية لبناء أنظمة عالية الأداء تستفيد من التزامن دون التعقيدات الكبيرة، مثل **البرمجة غير المتزامنة (Asynchronous Programming)** أو **النموذج التفاعلي (Reactive Model)**. نشرح فكرة أساسية: ليس دائمًا نحتاج خيوط متعددة للتعامل مع آلاف الاتصالات، هناك نموذج event-loop (كما في Node.js) حيث خيط واحد يدور على أحداث I/O بشكل غير محظور (non-blocking I/O) مما يمكنه من خدمة أعداد ضخمة من الاتصالات طالما العمل الفعلي قليل (مثال: خادم Proxy بسيط). نوضح كيف أن هذه المقاربة تتجنب تكلفة تبديل السياق بين الخيوط وتحسن استخدام الموارد في حالات الـI/O الكثيف. نذكر أدوات مثل **Node.js**، **async/await** في لغات متعددة (Python, C# etc.) وكيف تجعل كتابة الكود غير المتزامن أسهل. أيضًا نعرّج على **Reactive Extensions / Reactive Streams** (مثل مشروع Reactor في جافا أو RxJS) التي توفر نمط برمجة يعتمد على تدفق الأحداث والتعامل معها بمشغلات. نناقش مفهوم **Backpressure** – السيطرة على معدل تدفق الأحداث حتى لا يغرق المستهلك البطيء. كذلك نذكر أنه يمكن المزج: أنظمة كبيرة قد تستخدم مزيجًا من الـmultithreading التقليدي مع الـasync event loops في مكونات مختلفة حسب طبيعة العمل. الهدف إدراك الخيارات المختلفة لبناء أنظمة ذات إنتاجية عالية، وأن thread-per-request ليس الحل الوحيد.

المصطلحات الأساسية:

- Non-blocking I/O (الإدخال/الإخراج غير الحاجز)
- Async/Await (برمجة غير متزامنة بأنماط حديثة)
- Event Loop (حلقة الأحداث)
- Reactive Programming (البرمجة التفاعلية)
- Backpressure (آلية التحكم في ضغط التدفق)

فكرة التطبيق العملي: إذا كنت تعرف Node.js أو أي بيئة أحداث، جرّب كتابة خادم HTTP بسيط جدًا وتعريضه لعدد طلبات كبير (يمكنك استخدام `ab - ApacheBench` لاختبار). ثم جرّب خادم تقليدي متعدد الخيوط (مثلاً في Python استخدم Flask مع `threads=True`). قارن كيف يتصرفان مع زيادة الحمل (مثلاً 1000 طلب متوازي). على الأقل نظريًا أو عبر قراءات، استنتج أيهما يستخدم موارد أقل لكل اتصال ولماذا. هذه التجربة تعطيك ملموسية لفكرة الـnon-blocking event loop.

ملاحظات:

اليوم 42: اختبار التحميل وتحليل الاختناقات

شرح مختصر: ختام المحور بالتركيز على **اختبارات الأداء** العملية: كيف نكتشف حدود نظامنا قبل وقوعه في الإنتاج. سنتعلم إعداد **اختبار تحميل (Load Testing)** باستخدام أدوات مثل **JMeter** أو **Gatling** أو خدمات SaaS (مثل k6). نحدد سيناريوهات مهمة: عدد مستخدمين/طلبات في نفس الوقت، نوعية العمليات (قراءة vs كتابة)، ونشغّل الاختبار ضد بيئة تشابه الإنتاج قدر الإمكان. نراقب المؤشرات أثناء ذلك (CPU, Memory, Latency percentiles) لتحديد **عنق الزجاجة (Bottleneck)**: قد يكون في المعالج (CPU 100%) أو في قاعدة البيانات (استعلامات بطيئة) أو في الشبكة (Bandwidth ممتلئة) أو حتى قفل معين يعرقل التوازي. نتحدث عن قراءة النتائج: مثلاً إذا زاد معدل الطلبات ولم تعد زيادة throughput وبدأت الـlatency ترتفع كثيرًا، فهذه علامة أننا وصلنا حد الطاقة الاستيعابية. نشرح أيضًا **Stress Testing** (اختبار أقصى قدرة ممكنة) و**Spike Testing** (زيادة مفاجئة في الحمل) و**Soak Testing** (حمل طويل المدى لاكتشاف مشاكل التسريب). نؤكد على توثيق نتائج الاختبار وتحسين النظام بناءً عليها: ربما تحتاج زيادة موارد، أو تحسين كود، أو إضافة Cache. الهدف أن لا تكون مفاجآت الأداء في الإنتاج، بل يتم التمرن عليها والتخطيط لها مسبقًا.

المصطلحات الأساسية:
- Load Testing (اختبار الحمل)
- Bottleneck (اختناق/عنق زجاجة)
- Capacity Planning (تخطيط السعة الاستيعابية)
- Stress/Spike/Soak Testing (اختبارات الإجهاد/الطفرة/الاستمرارية)
- Benchmark (معيار قياس الأداء)

فكرة التطبيق العملي: استخدم أداة بسيطة مثل **wrk** أو **ApacheBench (ab)** أو **k6** لعمل اختبار تحميل على أحد خدماتك أو تطبيق تجريبي. ابدأ بعدد صغير من المستخدمين (مثلاً 10 متزامنين) ثم زد تدريجيًا إلى 50، 100... راقب متوسّط زمن الاستجابة ومعدّل الاستجابات. دوّن متى يبدأ الأداء بالتدهور بشكل كبير. حاول أن تجد الموارد على جهازك أين بلغت 100% أثناء ذلك (CPU؟). هذه التجربة الواقعية ستوضح لك كيف تكشف اختبارات الحمل حدود التطبيق وتعطيك أرقامًا للمستقبل (مثلاً: خادمي الحالي يخدم 200 طلب/ثانية عند زمن استجابة مرضٍ).

ملاحظات:

المحور 8: تصميم واجهات برمجية متقدمة (Advanced API Design)

(الأيام 43–48)

وصف المحور: يناقش هذا المحور مبادئ تصميم واجهات الـAPI المتقدمة، خاصة RESTful APIs وبدائلها. سنتناول أفضل الممارسات في تصميم REST (مثل التصنيف والإصدارات)، التحميل الزائد مقابل المرونة (Over-fetching/Under-fetching) وحلول كـGraphQL، تصميم واجهات gRPC ذات الكفاءة العالية، إدارة دورة حياة الـAPI (إصدارات، إهمال Deprecation)، وتوثيق الـAPI وتجربة المطور (Developer Experience) لضمان أن الـAPI سهلة الاستخدام والصيانة.

اليوم 43: مبادئ تصميم RESTful API المتقدمة

شرح مختصر: مراجعة سريعة لمبادئ تصميم **REST API** الجيدة، ثم التعمق في أمور متقدمة. نؤكد على استخدام **الموارد (Resources)** وأسلوب **HTTP** verbs الصحيح (GET/POST/PUT/DELETE...)، وكذلك **حالات HTTP** الملائمة لكل نتيجة (200 نجاح، 201 إنشاء، 400 طلب خطأ، 500 خطأ خادم...). ثم نناقش مبادئ مثل **الاستقلالية الذاتية (Statelessness)** – الخادم لا يحتفظ بحالة العميل، كل طلب يكتفي بمعلوماته (إلا في حالات تتبع محددة). نتحدث عن **هيكلة الموارد** : متى تستخدم العلاقات في المسارات (`/users/{id}/orders` مثلاً) ومتى تبقيها منفصلة والاستعلام عبر معلمات. نركز على **الاستجابة المرنة** : تضمين روابط HATEOAS (رغم قلة استخدامه العملي، لكن مبدؤه مفيد) – أي توفير روابط بالمخرجات لتوضيح ما يمكن فعله تاليًا. نناقش أيضًا **معالجة الأخطاء** في الـAPI – إرسال جسم واضح مع رمز خطأ وتفسير يمكن للبرنامج العميل فهمه. بالإضافة إلى **Pagination** واستراتيجياتها (limit/offset vs cursors) للتعامل مع مجموعات كبيرة. هذه الأسس تجعل الـAPI أكثر **قابلية للاستخدام** ووضوحًا، وهي مهمة خاصة في السياقات المتقدمة حيث الواجهة ربما يستخدمها مطورون خارجيون وعلينا جعلها سهلة الفهم.

المصطلحات الأساسية:
- REST (نمط نقل الحالة التمثيلي)
- HTTP Methods/Status (أساليب HTTP والحالات)
- Stateless (عديم الحالة)
- Pagination (ترقيم الصفحات)
- HATEOAS (الارتباطات التشعبية كجزء من الحالة)

فكرة التطبيق العملي: اختر خدمة REST شهيرة (مثل GitHub API أو Twitter API القديمة) وألق نظرة على وثائقها. حاول استدعاء أحد endpoints باستخدام أداة كـcurl أو Postman. راقب الاستجابة: كيف تم هيكلتها؟ ما الحقول؟ هل هناك روابط مرفقة (مثلاً رابط الصفحة التالية)؟ هذا التحليل سيجعلك ترى مبادئ REST الجيدة مُطبقة واقعياً وتستلهم منها.

ملاحظات:

اليوم 44: إدارة نسخ الـAPI والتوافقية

شرح مختصر: كيفية التعامل مع تطور الـAPI مع مرور الوقت دون كسر التطبيقات التي تستخدمها. نتناول استراتيجيات **إصدار الـAPI (API Versioning)** : مثل تضمين الإصدار في المسار (مثال: `/api/v2/...`)، أو في ترويسة Header مخصص، أو الاعتماد على **المحتوى (Content Negotiation)** عبر media types مختلفة. نشرح مزايا ومساوئ كل نهج. ثم نتحدث عن **التوافق مع الإصدارات السابقة (Backward Compatibility)** : القاعدة العامة هي محاولة عدم كسر الـAPI الموجود – بمعنى يمكن إضافة حقول جديدة في الاستجابة دون إزالة القديمة، وهكذا يستطيع العميل القديم تجاهل الجديد بدون مشاكل. ولكن إن دعت الحاجة لكسر التوافقية (مثلاً تصميم غير صالح أو تغيير كبير بالمفاهيم) فالخيار إصدار جديد. نشرح عملية **إهمال (Deprecation)** الـAPI القديم: إيصال إشعارات للمطورين (ربما عبر headers تنبيهية أو في الوثائق) بأن هذا الجزء سيوقف دعمه مستقبلاً. نشدد على أهمية إعطاء فترة انتقالية كافية. كما نذكر أدوات أو وثائق مساعدة: مثل **Changelog** واضح لكل إصدار يذكر التغييرات. الهدف أن يدرك المهندس أن إصدار الـAPI عملية حساسة تتطلب تواصل مع مستخدمي الـAPI وتخطيط كي لا ينكسر التكامل مع الخدمات الأخرى.

المصطلحات الأساسية:

- API Versioning (إصدارات الواجهة البرمجية)
- Backward Compatibility (التوافقية مع الماضي)
- Deprecation Policy (سياسة إهمال الإصدارات)
- Semantic Versioning (الإصدارات الدلالية)
- Changelog (سجل التغييرات)

فكرة التطبيق العملي: اكتب خطة صغيرة لكيفية إصدار نسخة جديدة من واجهة API قمت ببنائها (حتى لو فرضيًا). ما الطريقة التي ستستخدمها؟ (مثلاً `/v2` في المسار). وكيف ستُعلِم المستخدمين؟ (تحديث الوثائق، ربما بريد لو كان مغلقًا، ...). وإذا كان تغييرًا يكسر التوافق، هل ستدعم الإصدارين جانبًا إلى جنب لفترة؟ حدد تلك الفترة. تدوين هذه الخطة سيجعلك تفكر بخطوات انتقال الـAPI بشكل منظم.

ملاحظات:

اليوم 45: GraphQL وبدائل REST

شرح مختصر: استكشاف **GraphQL** كنموذج جديد لواجهة APIs بالإضافة إلى REST التقليدي. نشرح مشكلات REST في بعض الحالات مثل **Over-fetching** (الحصول على بيانات أكثر من اللازم لأن endpoint معين يعيد حقل لا تحتاجه) و**Under-fetching** (الحاجة لطلب عدة endpoints للحصول على بيانات كافية، مثلاً جلب مستخدم ثم طلب آخر لجلب أصدقائه). GraphQL يحل هذه بإتاحة **استعلام مرن** يحدد العميل بالضبط البيانات المطلوبة، في مكالمة واحدة. نصف بنية GraphQL: تعريف **Schema** يحوي أنواع (Types) وعلاقاتها، ثم Queries وMutations. نوضح كيف يبدو استعلام GraphQL ومخرجه. نناقش أيضًا الاعتبارات: GraphQL ينقل بعض عبء إدارة الاستعلام إلى الخادم الذي يجب أن يحلل ويجمع البيانات – مما قد يكون معقدًا ولكنه يمنح مرونة كبيرة للعميل. أيضًا مسألة **Caching** تصبح أصعب قليلاً مقارنة بـREST الذي يمكن فيه تخزين GET بسهولة. نذكر بدائل أخرى أقل شيوعاً مثل **gRPC** (سنتناوله غدًا) و**JSON:API** و**specification** وغيرها. الفكرة اليوم تعريف بأن RESTful ليس وحيد، هناك سيناريوهات يفضل فيها GraphQL (مثلا واجهة واحدة لعدة تطبيقات عميل مختلفة تحتاج بيانات متنوعة)، وعلى المهندس معرفة الفرق ومتى قد يستخدم كلاً منها.

المصطلحات الأساسية:

- GraphQL (جراف كيو إل)
- Over-fetching / Under-fetching (جلب زائد/ناقص)
- Schema & Type System (مخطط GraphQL ونظام الأنواع)
- Query & Mutation (الاستعلام والتعديل في GraphQL)
- Resolver (دالة جلب البيانات في GraphQL)

فكرة التطبيق العملي: جرّب استكشاف GraphQL فعليًا: يوجد خدمات عامة توفر واجهة GraphQL يمكنك التجربة عليها، مثل واجهة GitHub GraphQL. استخدم أداة GraphiQL أو Playground وأرسل استعلام لجلب بيانات معينة (مثلاً اسم مشروع ووصفه ونجومه من GitHub). راقب كيف حددت بالضبط الحقول التي تريدها. إذا لم تستطع، على الأقل

اكتب شكل استعلام GraphQL تتخيله لنظام ما تعرفه (مثلاً استعلام getUser(id) يعيد {id, name, [friends {id, name}] جميعًا في طلب واحد).
ملاحظات:

اليوم 46: تصميم واجهات gRPC وواجهات ثنائية

شرح مختصر: التعرف على **gRPC** كوسيلة اتصال بين الخدمات (أو حتى بين عميل وخادم) تستخدم **بروتوكول ثنائي Binary Protocol** عالي الأداء. نشرح أن gRPC مبني على **HTTP/2** ويستخدم **Protocol Buffers (Protobuf)** لوصف **واجهات RPC** وتبادل البيانات بشكل مضغوط. نوضح كيف نعرّف ملف `.proto` يحدد الخدمات والرسائل، ثم توليد الكود client/server منه. مزايا gRPC: أداء عالي نظرًا لتنسيق الثنائي وضغطه، ودعم الاتصال **المتدفق (Streaming)** بسهولة (client streaming, server streaming, bidirectional). لهذا شاع استخدامه في بيئات microservices عالية الأداء وبين مكونات تحتاج latency منخفض (مثل داخل مراكز البيانات). نناقش متى يكون مناسبًا: مثلاً اتصال خدمة-بخدمة خلف الكواليس حيث كلا الطرفين تحت سيطرتنا ويمكنهما مشاركة الـproto، بالمقابل REST/JSON مناسب أكثر لوثوقيته في البيئات المفتوحة وبساطة التطوير/debug. نشير إلى أن gRPC يتطلب اهتمام بالتوافقية أيضًا – تغيير الـproto قد يكسر العميل، لذا اتباع إرشادات إضافة الحقول دون حذف. أيضاً نذكر بدائل تاريخية مثل **Thrift** أو **ASN.1** التي لها استخدامات أقدم. النتيجة: فهم أن ليس كل API يجب أن تكون نصية JSON – أحياناً الواجهات الثنائية (Binary) خيار ممتاز خاصة بين أنظمة داخلية لأجل الكفاءة.
المصطلحات الأساسية:
- gRPC (نظام RPC مبني على بروتوكول ثنائي)
- Protocol Buffers (بروتوكول بفرز، صيغة تسلسل بيانات)
- IDL (لغة توصيف الواجهة، هنا .proto)
- Streaming RPC (مناداة إجراءات متدفقة)
- HTTP/2 (بروتوكول الويب الذي يعتمد عليه gRPC)
فكرة التطبيق العملي: حاول كتابة أبسط خدمة gRPC إن أمكن: عرف .proto يحتوي خدمة Greeter مع إجراء SayHello يأخذ رسالة (اسم) ويرد بتحية. استخدم مولد gRPC للغتك (مثلاً Python أو Go) وفعّل الخادم والعميل، جرّب الاتصال. ستلاحظ قلة الجهد في كتابة serialization/HTTP. إن لم تستطع عملياً، اكتب تصورًا نصيًا لملف .proto لنظام تعرفه (مثلاً خدمة UserService فيها RPC GetUser(UserRequest) returns (UserResponse)). رؤية الـAPI من منظور بروتوكولي سيغير طريقة تفكيرك قليلاً مقارنة بـJSON.
ملاحظات:

اليوم 47: بوابة الـAPI وتجربة المطور (DevEx)

شرح مختصر: تسليط الضوء على دور **API Gateway** في البنى الحديثة، وكيف يحسّن تجربة استخدام الـAPIs. نشرح أن بوابة الـAPI هي خدمة تقع بين العملاء ومجموعة خدمات الـBackend، وتقدم عدة فوائد: **تجميع endpoints** متعددة من خدمات مختلفة في واجهة موحدة، تطبيق سياسات أمنية مركزية (توثيق، معدل طلبات)، ترجمة البروتوكولات (مثلاً قبول REST وتحويله لاستدعاءات gRPC داخلية)، وكذلك أشياء مثل **إضافة رؤوس موحدة** أو إدارة الـCORS لجميع الخدمات. نعطي مثال: منصة لديها عشرات الخدمات المصغرة، بدلاً من أن يضطر العميل لمعرفة عنوان كل واحدة، يتصل بالـAPI Gateway والذي يوجه الطلب للخدمة المناسبة (توجيه Routing). نذكر أدوات مشهورة: **Kong, Apigee, AWS API Gateway, Nginx** وغيرها تقوم بهذا الدور. بالإضافة إلى ذلك، نناقش مفهوم **Developer Portal** و **API Documentation** المتكامل: حيث توفر بوابة الـAPI واجهة وثائق (ربما تفاعلية مع Swagger/OpenAPI) ومفاتيح API لإدارة الوصول للمطورين الخارجيين، حتى يتمكنوا من تجربة الـAPI بسهولة. الجانب الآخر هو **Analytics**: تجمع البوابة إحصاءات الاستخدام (من أكثر endpoints استخدامًا؟ أوقات الذروة؟) والتي تفيد في تحسين الـAPI. الهدف فهم أن بوابة الـAPI ليست مجرد عاكس طلبات، بل عنصر استراتيجي لإدارة الـAPIs خاصة المفتوحة للمستهلكين الخارجيين، وتلعب دورًا كبيرًا في تحسين **تجربة المطور** الذي يتعامل مع خدماتك.
المصطلحات الأساسية:
- API Gateway (بوابة واجهات برمجية)

- Routing / Composition (توجيه وتكوين الطلبات)
- Throttling / Quotas (تحديد المعدل وحصص الاستخدام)
- Developer Portal (بوابة المطورين للـAPI)
- API Analytics (تحليلات استخدام الواجهة)

فكرة التطبيق العملي: إذا كانت لديك فرصة، جرّب استخدام بوابة API بسيطة – على سبيل المثال **Kong** (ذو مصدر مفتوح) يمكن تشغيله محليًا. عرّف خدمة واحدة خلف Kong وأضف Rate Limit Plugin بسيط (مثلاً حدّد 5 طلبات في الدقيقة). جرّب إرسال 10 طلبات سريعة ولاحظ كيف تبدأ البوابة بالرفض مع كود 429. إن تعذر ذلك، قم بإعداد سيناريو على الورق: ارسم مخطط فيه عميل يتصل بـAPI Gateway ثم 3 خدمات خلفية، حدّد كيف ستتم عملية التوجيه (مثلاً `/api/users` للبوابة يوجه إلى خدمة Users)، وفكّر ما السياسات التي يمكن للبوابة تطبيقها عمومًا (Auth، Logging موحد، إلخ).

ملاحظات:

اليوم 48: توثيق الواجهة وتحسين تجربة التكامل

شرح مختصر: التركيز على أهمية **توثيق API** جيدة وأدوات لتحسين **Developer Experience** عند التكامل مع الـAPI. نبدأ بأشهر مواصفات للتوثيق: **OpenAPI (Swagger)** – تسمح بوصف endpoints والطلبات/الاستجابات بشكل آلي قابل لتوليد صفحات وثائق وحتى أكواد عملاء تلقائيًا. نشرح عناصر ملف Swagger (المسارات، المعاملات، النماذج). نشجع تضمين أمثلة في الوثائق لكل endpoint لتوضيح الاستخدام. أيضًا نذكر **Postman Collections** كطريقة لمشاركة مجموعة طلبات جاهزة مع المطورين لتجربة الـAPI بسهولة. نتطرق لموضوع **SDKs** : أحيانًا لتجربة مطور ممتازة، توفر الشركة مكتبات جاهزة بلغات متعددة تغلف نداءات الـAPI (مثلاً AWS SDK) لتجعل التكامل أسهل. هذا يتطلب الحفاظ على هذه المكتبات مع تحديثات الـAPI. كذلك **Sandboxes** أو بيئات اختبارية مفتوحة للمطورين لتجربة الواجهة بدون التأثير على بيانات حقيقية. نشير إلى **معايير الخطأ الموحدة** ورسائل الخطأ الواضحة كجزء من تجربة المطور – حين يخطئ في الاستدعاء، رسالة خطأ جيدة تساعده على التصحيح سريعًا. نختم بالتأكيد أن الـAPI هي منتج بحد ذاته، وتجربة المطورين معها تؤثر على نجاح تبنيها، لذا يجب الاستثمار في جعل التعلم والتكامل سلسًا (وثائق واضحة، أدوات مساعدة، دعم مجتمعي أو منتدى للإجابة على الأسئلة...).

المصطلحات الأساسية:

- OpenAPI/Swagger (مواصفات توثيق API)
- API SDK (حزمة تطوير برمجية تربط بالواجهة)
- Postman Collection (مجموعة بوستمان)
- Sandbox Environment (بيئة تجريبية معزولة)
- Error Codes & Messages (أكواد ورسائل الخطأ)

فكرة التطبيق العملي: جرّب استخدام أداة Swagger UI: إذا كان لديك ملف OpenAPI (يمكنك كتابة واحد صغير يدويًا أو ربما استخدم مثال من الإنترنت)، افتحswagger editor أو UI لترى كيف يتم عرض الـAPI بشكل تفاعلي. حاول أيضاً استخدام هذه الواجهة لتجربة نداء (إذا متاح backend). ولاحظ كم يسهل ذلك فهم الـAPI بدون قراءة الكثير من النص. إذا لا يوجد، اصنع توثيقًا مقتضبًاendpoint تملكه – اكتب اسمendpoint، نوع الطلب، بارامتراته، وهيكل الاستجابة، ثم أعط الوثيقة لشخص آخر (فرضًا) لترى هل يتمكن من فهم استخدامه. هذا يبين لك قوة التوثيق الجيد.

ملاحظات:

المحور 9: تجربة المطور وممارسات التسليم (DevEx and Delivery Practices)

(الأيام 49–54)

وصف المحور: يركز هذا المحور على تحسين **تجربة المطور** نفسها وعمليات تطوير البرمجيات لضمان إنتاجية وجودة عالية. سنتطرق لأتمتة العمليات عبر CI/CD، تكامل الاختبارات وأنواعها، إدارة الشيفرة الجماعية عبر المراجعات، تحسين

بيئة التطوير (تشابه البيئات واستخدام الحاويات)، قياس الأداء الإنتاجي لفرق التطوير (مثل مقاييس DORA) والتطرق لثقافة DevOps وتحسينها المستمر.

اليوم 49: التكامل المستمر والنشر المستمر (CI/CD)

شرح مختصر: نظرة عملية على أدوات ومنهجيات **التكامل المستمر (Continuous Integration)** و**التسليم/النشر المستمر (Continuous Delivery/Deployment)** . نشرح أن CI تعني دمج تغييرات المطورين بشكل متكرر (عدة مرات يوميًا ربما) في الفرع الرئيسي مع تشغيل اختبارات تلقائية لضمان عدم كسر شيء. نذكر أدوات CI الشهيرة: **Jenkins، GitLab CI, GitHub Actions, CircleCI** ... والتي تقوم ببناء المشروع وتشغيل اختباراته مع كل دفع (push) أو على الأقل مع كل طلب دمج (Pull Request) جديد. ثم CD: بعد نجاح الاختبارات، تأتي مرحلة تسليم الإصدار أو نشره. نوضح الفرق بين **Continuous Delivery** (تحضير التغييرات للنشر التلقائي لكن نشرها فعليًا يحتاج قرار بشري عادة) و**Continuous Deployment** (النشر تلقائي فعليًا لكل تغيير يمر الاختبارات). نتحدث عن خطوات Pipeline نموذجية: بناء (Compile/Package)، اختبار (وحدات وتكامل)، أحيانًا مرحلة **Staging** ، ثم نشر في **Production** . نذكر أهمية **Infrastructure as Code** هنا – نشر بنى تحتية تلقائيًا (مثل Docker containers, Kubernetes YAML, Terraform scripts ضمن الخط). الهدف: جعل إطلاق الميزات وتصحيحات الأخطاء أمرًا روتينيًا قليل المخاطر، بدلاً من إصدارات ضخمة نادرة وخطرة. هذا يتطلب ثقافة وأدوات، لكنها أساس دور الـSenior Backend أن يساهم في بناء خطوط النشر الفعّالة.

المصطلحات الأساسية:

- Continuous Integration (التكامل المستمر)
- Build Pipeline (خط أنابيب البناء)
- Continuous Delivery/Deployment (التسليم/النشر المستمر)
- Rollback (التراجع عن إصدار)
- Blue-Green Deployment (نشر أزرق-أخضر)

فكرة التطبيق العملي: أنشئ (أو تخيل) مشروعًا بسيطًا على GitHub واستخدم **GitHub Actions** لإعداد CI: مثلاً تشغيل اختبار وحدة بسيط كلما تم دفع كود. توجد قوالب جاهزة تقريبًا لكل لغة. جرّب دفع commit خاطئ وانظر كيف تبين لك الـCI الكسر قبل الدمج. أما على صعيد CD، إن كان لديك تطبيق على منصة مثل Heroku أو Netlify، صلها بمستودع Git لتنشر كل تحديث تلقائيًا. مراقبة هذا السريان الآلي سيجعلك تدرك قيمة الأتمتة في التطوير.

ملاحظات:

اليوم 50: الاختبار التلقائي وضمان الجودة

شرح مختصر: التركيز على دور **الاختبارات الآلية** في بيئة الـBackend. نشرح مستويات الاختبار: **اختبارات الوحدة (Unit Tests)** التي تختبر أجزاء صغيرة من الكود (دالة أو وحدة)، **اختبارات التكامل (Integration Tests)** التي تختبر تفاعل مكونات متعددة معًا (مثلاً التكامل مع قاعدة البيانات الحقيقية أو خدمة خارجية وهمية)، **اختبارات المنظومة (System Tests)** أو **End-to-End** التي تحاكي استخدام النظام كاملاً كما يفعل المستخدم (مثل إرسال طلب HTTP ومراقبة السلوك عبر الطبقات). نناقش أن Senior Backend ينبغي أن يهتم ببناء **مجموعات اختبارات موثوقة** تضمن عدم انهيار الميزات القديمة عند إضافة جديد (Regression). نذكر ممارسات: استخدام **Test Doubles (Mocks/Stubs)** لعزل الوحدات أثناء الاختبار، seed بيانات معروفة لقاعدة البيانات للاختبارات التكاملية. أيضًا **اختبارات الأداء** كما ذكرنا ومحاكات الأحمال ربما كجزء من خطوط QA (ليس مع كل بناء، لكن بشكل دوري). نتطرق لمفهوم **Test Coverage** – تغطية الكود بالاختبارات كنسبة مئوية، مع الحذر أن الرقم ليس كل شيء ولكن يغطيته العالية تشير لثقة أكبر. نتحدث عن **إستراتيجية TDD (Test-Driven Development)** حيث تُكتب الاختبارات قبل الكود لضمان فهم المطلوب. الخلاصة: جودة البرنامج الخلفي تعتمد كثيرًا على هذه الاختبارات، فهي تحمي من الأخطاء عند التطوير السريع وتوفر توثيق حيّ للسلوك المتوقع.

المصطلحات الأساسية:

- Unit Test (اختبار وحدة)
- Integration Test (اختبار تكامل)
- End-to-End Test (اختبار شامل من النهاية للنهاية)

- Test Coverage (نسبة تغطية الاختبارات)
- TDD (تطوير موجّه بالاختبار)

فكرة التطبيق العملي: إذا لديك وحدة برمجية بدون اختبارات، حاول كتابة اختبار وحدة بسيط لها. اختر دالة تقوم بحساب/تحويل واكتب 2-3 حالات مختلفة للتحقق من المخرجات. شغّلها وتأكد أنها تمر. ثم عدّل الدالة عمدًا لتنتج خطأ، راقب كيف يلتقط الاختبار ذلك. هذا سيوضح لك بشكل مباشر قيمة وجود اختبارات تنبهك عند إدخال خطأ غير مقصود. إذا كنت بالفعل تكتب اختبارات، جرب تغطية حالة طرفية لم تغطها من قبل ولاحظ راحة البال بعد إضافة مزيد من الأمان.

ملاحظات:

اليوم 51: مراجعة الشفرة والتعاون الجماعي

شرح مختصر: تسليط الضوء على **ممارسات التطوير الجماعي** التي ترتقي بجودة الكود وتجربة التطوير. أولها **مراجعات الكود (Code Reviews)** : كمهندس Senior سيكون عليك ليس فقط كتابة كود جيد، بل أيضًا مراجعة كود الآخرين وإعطاء ملاحظات بنّاءة. نناقش أهداف المراجعة: اكتشاف الأخطاء المحتملة، ضمان اتباع المعايير (style, security considerations)، ومشاركة المعرفة بين الفريق. نشير لأدوات مثل **Pull Requests** على GitHub/GitLab وكيفية التعليق ضمن السياق. نؤكد على أسلوب النقد الإيجابي (لا تنتقد الشخص بل ركز على الكود). ثانيًا **معايير الترميز (Coding Standards)** : ربما باستخدام Linters أو Formatters تلقائية، لتوحيد شكل الكود (spaces vs tabs, naming...). هذا يقلل النقاشات غير الضرورية في المراجعات. ثالثًا **التطوير الزوجي (Pair Programming)** أحيانًا كوسيلة تعاون فوري ومشاركة المعرفة، أو على الأقل جلسات تصميم جماعية عند مواجهة مهمة معقدة. كما نذكر إدارة **الفروع (Branching Strategy)** – مثلاً GitFlow أو Trunk Based – لضمان أن الكود يندمج بسلاسة دون تعطيل عمل الآخرين. الهدف هو أن البيئة ليست فقط أدوات تقنية بل أيضًا **ثقافة** تضمن أن الكود المنتج جماعيًا أفضل من العمل الفردي المعزول، وأن الجميع يتعلم من بعضهم ويراجعون بعضهم للحفاظ على مستوى عالي من الجودة والثقة في المنتج النهائي.

المصطلحات الأساسية:

- Code Review (مراجعة الشفرة)
- Pull Request (طلب دمج)
- Coding Standards / Linters (معايير الترميز وأدوات الفحص)
- Pair Programming (برمجة ثنائية)
- Branching Strategy (استراتيجية التفريع في التحكم بالإصدارات)

فكرة التطبيق العملي: إذا كنت تعمل ضمن فريق، اقترح على زميل أن تراجع قطعة كوده أو يراجع هو كودك بشكل ودّي خارج السياق الرسمي. تبادلا الآراء. إن لم تكن، انظر إلى مشروع مفتوح المصدر مهتم به واعثر على *Pull Request* حديث واقرأ النقاش الدائر فيه بين المساهمين. لاحظ طبيعة الملاحظات وكيف يتم الوصول لتحسينات. سجّل ما تعلمته من ذلك – ربما معيار جديد أو طريقة تفكير مختلفة. هذه الممارسة توسع مداركك حول تقنيات ومبادئ قد لا تكون فكرت بها وحدك.

ملاحظات:

اليوم 52: بيئة التطوير والبنية التحتية ككود

شرح مختصر: التركيز على **تحسين بيئة التطوير ومواءمتها مع الإنتاج** لضمان أن المطورين يعملون بكفاءة ودون مفاجآت. نتحدث عن جعل إعداد المشروع سهلاً: **Docker** مثلًا لتوفير حاويات للخدمات التابعة (Database, Cache) حتى يمكن تشغيل المشروع محليًا بضغطة زر بنفس إصدارات البنية المتوقعة. نشرح فكرة **Infrastructure as Code (IaC)** في سياق التطوير أيضًا: فبدلاً من تعليمات يدوية لتشغيل خادم محلي، لدينا ملفات تكوين (docker-compose, Vagrant, Kubernetes YAML) تهيئ البيئة. هذا يضمن أن جميع المطورين يستخدمون بيئة موحدة وقابلة للتدمير والإنشاء بسهولة دون "يعمل على جهازي فقط". نذكر **بيئات التطوير المتقاربة** : كأن تكون بيئة الاختبار شبيهة جدًا بالإنتاج من حيث إعدادات الشبكة والخدمات، بحيث ما يُختبر هناك يعكس الواقع (ربما باستخدام نفس قوالب Docker Compose للإثنين). كذلك **أدوات التطوير** : مثل Hot-reload لإعادة تشغيل التطبيق تلقائيًا عند تغيير الكود، Debuggers فعالة، الخ... أي شيء يقلل وقت الحلقة (code -> test -> fix). نذكر أيضًا **إدارة التكوين (Configuration**

Management) – استخدام ملفات config منفصلة لكل بيئة (dev/test/prod) لكن إدارة قيمها عبر أدوات (مثل dotenv أو Vault للأسرار) بحيث لا يكون هناك تحوير يدوي متعب عند النقل. الهدف: إزالة المعوقات والاختلافات بين البيئات، حتى يقضي المطورون وقتهم في حل المشكلات الحقيقية وليس "لماذا لا يعمل على جهاز فلان؟".

المصطلحات الأساسية:

- Docker / Containerization (استخدام الحاويات Docker)
- Docker Compose (تنسيق تشغيل متعدد الحاويات)
- IaC – Infrastructure as Code (البنية التحتية ككود) [26]
- Environment Parity (تكافؤ البيئة بين التطوير والإنتاج)
- Configuration Management (إدارة ضبط الإعدادات)

فكرة التطبيق العملي: إذا لم تكن تستخدم Docker في مشروعك، جرّب كتابة `Dockerfile` بسيط لتشغيل تطبيقك (حتى لو تطبيق "Hello World"). ثم `docker-compose.yml` لإضافة مثلا قاعدة بيانات خدمة. شغّل كل شيء عبر `docker-compose up` وانظر أن تطبيقك يعمل ويتواصل مع القاعدة. الآن تخيّل زميلًا جديدًا انضم – بدلاً من تثبيت كل شيء يدويًا، سيكون لديه هذا الملف ويشغله. إن كنت تستخدم Docker فعلاً، تأكد أن README مشروعك محدث بوضوح بخطوات التشغيل المحلية. هذه التجربة تعمّق فهمك لقيمة IaC وتسهيل البيئة.

ملاحظات:

اليوم 53: قياس الأداء التطويري – مقاييس DevOps

شرح مختصر: نظرة على **أداء فريق التطوير** وكيفية قياسه وتحسينه. نقدّم **مقاييس DORA الأربع** الشهيرة [27] [28] : **وتيرة النشر (Deployment Frequency)** – كم مرة ينشر الفريق للإنتاج (فرق النخبة قد يكون يوميًا أو أسرع) [29] ؛ **مهلة التغيير (Lead Time for Changes)** – الوقت من كتابة الكود إلى نشره في الإنتاج [30] ؛ **معدل فشل التغييرات (Change Failure Rate)** – نسبة الإصدارات التي تتسبب في خلل/انقطاع وتستدعي إصلاحًا [31] ؛ **زمن استعادة الخدمة (MTTR)** – متوسط الوقت اللازم لاستعادة الخدمة عند حدوث خلل [31] . نشرح لماذا هذه المقاييس مهمة: تعطي صورة عن سرعة الفريق واستقراره، وقد وجدت أبحاث أنها ترتبط بأداء الأعمال أيضًا. نشرح كيف يمكن تحسين كل منها: مثلاً رفع وتيرة النشر عبر أتمتة أكثر واختبار آلي، تقليل مهلة التغيير بتقليل حجم كل دفعة عمل (batch size)، خفض معدل الفشل عبر اختبارات أفضل وعمليات مراجعة جيدة، سرعة الاستعادة عبر تحسين المراقبة وخطط الاستجابة. نشير أيضًا لمقاييس أخرى مثل **رضا المطورين** (Developer Satisfaction) أو **زمن إعداد عضو جديد**، ولكن تبقى DORA baseline معترف بها. الهدف: كـSenior ينبغي ألا تركز فقط على الكود نفسه، بل على عملية إنتاجه وتحسينها باستمرار، واستخدام البيانات لإقناع الإدارة مثلاً بتبني تغييرات (مثل الحاجة لزيادة الاختبارات أو إعادة هيكلة معينة) بدل الاعتماد على الحدس فقط [32] .

المصطلحات الأساسية:

- Deployment Frequency (وتيرة النشر) [29]
- Lead Time (مهلة التغيير من الكود إلى الإنتاج) [30]
- Change Failure Rate (معدل فشل الإصدارات) [31]
- MTTR (متوسط زمن الاستعادة) [31]
- DORA Metrics (مقاييس دورا للأداء التقني)

فكرة التطبيق العملي: قيّم وضعك الحالي (أو فريقك لو تعمل بفريق) مقابل مقاييس DORA: كم مرة تقريبًا ننشر في الشهر؟ أيام أم أسابيع؟ كم عادة يستغرق من كتابة ميزة حتى ترى النور؟ هل غالبية الإصدارات سلسة أم نضطر لإطفاء حرائق؟ وكم تستغرق إصلاحاتها؟ مجرد محاولة الإجابة ستظهر نقاطاً ربما يمكنك اقتراح تحسينها. إن رغبت بالتعمق، اصنع جدولًا صغيرًا يضم آخر 5 مهام أنجزتها: متى بدأت -> متى دمجت -> متى وصلت إنتاج، وأي مشاكل حدثت. هذا التحليل البسيط يضع عينك على العملية لا المنتج فقط، وهو منظور قيم للقيادة التقنية.

ملاحظات:

اليوم 54: ثقافة DevOps والتحسين المستمر

شرح مختصر: الحديث عن **الثقافة** والممارسات اللا تقنية التي تجعل فريق التطوير عالي الأداء. نشرح مفهوم **DevOps** كدمج للتطوير والعمليات – لم يعد المطور يعمل بمعزل عن نشر وتشغيل برنامجه، بل هو معني بالاثنين، وكذلك مسؤول العمليات (أو SRE) يتعاون معه عن كثب. نؤكد على إزالة العوائق بين الفرق: كفريق اختبار منفصل ينتظر تسليم نسخة – بدلاً من ذلك، الجودة مسؤولية الجميع. نتحدث عن **Post-Mortems** بلا لوم بعد الحوادث – ثقافة تعلم من الأخطاء بدل معاقبة، لتشجيع الشفافية. أيضًا **مشاركة المعرفة** : عبر مراجعات، عروض داخلية، تدوين وثائق خفيفة، بحيث لا تبقى المعرفة حرز كل فرد. نذكر **Automate everything** مبدأ – أي مهمة متكررة مرهقة (إعداد خادم، ترقية نسخة) الأفضل أتمتتها، لتقليل الخطأ البشري وتحرير الوقت للإبداع. نعرّج على **معنويات الفريق** : أن بيئة داعمة، تعطي زمام الاستقلالية للمطورين (Autonomy) ولكن أيضًا تدعمهم بالتدريب والتوجيه، تؤدي لنتائج أفضل (وهذا ما أشارت له أبحاث DevEx [32] – تجربة مطور جيدة تعني أداء أعلى). نشير لمفهوم **Blameless culture** : عند حدوث خلل لا نبحث عن المخطئ بل عن العملية التي سمحت بذلك لنصلحها. أخيرًا **التحسين المستمر (Continuous Improvement)** – سواء عبر سباقات (Retrospectives) دورية أو تحليل المقاييس، مع عقلية "كيف نجعل الشهر القادم أفضل من الحالي" باستمرار. الهدف هنا تذكير بأن التفوق التقني ليس فقط أدوات وإنما أسلوب عمل وروح فريق تعززه قيادة الـSenior Engineers.

المصطلحات الأساسية:

- DevOps Culture (ثقافة التطوير/العمليات)
- Blameless Post-mortem (نقاش ما بعد الحوادث بدون لوم)
- Knowledge Sharing (مشاركة المعرفة)
- Automation (الأتمتة)
- Continuous Improvement (التحسين المستمر)

فكرة التطبيق العملي: اقترح تحسين ثقافي صغير في عملك أو لنفسك: مثلاً، بعد إنهاء مشروع أو مهمة، خذ 15 دقيقة لكتابة ما سار بشكل جيد وما يمكن تحسينه بالمرة القادمة (هذه عمليًا *Retrospective* شخصية). أو نظم جلسة مع الفريق (لو متاح) لنقاش حادث أخير بطريقة تحليل الأسباب الجذرية وليس إلقاء اللوم، واكتب توصيات. حتى لو لم تكن قائد الفريق، هذه المبادرات تقدّر وغالبًا ما تقود لتغيير إيجابي. إن لم يكن لديك فريق، تخيل لو لديك مشروع مفتوح المصدر وتأثر بعطل: دوّن نقاط تفترض أنها كانت ستساعد (اختبارات؟ توثيق أكثر؟) – هذه العملية تعلمك التفكير بمنظور أكبر من الكود: **منظور المنظومة ككل**.

ملاحظات:

المحور 10: الوعي بالحوسبة السحابية والبنية التحتية (Cloud & Infrastructure Awareness)

(الأيام 55–60)

وصف المحور: في المحور الأخير، سنغطي المعرفة الأساسية التي يحتاجها مهندس Backend في مجال البنية التحتية والحوسبة السحابية. سنتعرف على مفاهيم البنية التحتية السحابية (خدمات IaaS/PaaS)، الحاويات وإدارة الحاويات (Docker & Kubernetes)، البنية التحتية ككود (مثل Terraform)، خدمات سحابية مهمة (التخزين، قواعد البيانات المدارة، Functions)، استراتيجيات نشر على السحابة (Blue-Green, Canary)، وأخيرًا إدارة التكلفة والسحابة متعددة/هجينة بإيجاز.

اليوم 55: أساسيات الحوسبة السحابية ونماذج الخدمات

شرح مختصر: مقدمة مفهوم **الحوسبة السحابية (Cloud Computing)** ولماذا هو مهم لمهندس الـBackend. نعرف السحابة كخوادم وموارد تُقدّم عند الطلب عبر الإنترنت. نشرح مستويات الخدمة: **Infrastructure as a Service (IaaS)** مثل AWS EC2, Azure VM – يعطيك آلات افتراضية وشبكات لتتحكم بها [26] ؛ **Platform as a Service (PaaS)** مثل

Heroku, Google App Engine – تعطيك منصّة تشغيل تطبيق دون إدارة الخوادم بشكل مباشر؛ **Software as a Service (SaaS)** – تطبيق جاهز تستخدمه عبر السحابة (أقل ارتباط بمسؤولياتنا كمطورين ولكن للفهم). أيضًا نذكر **Container as a Service (CaaS)** و**Function as a Service (FaaS)** كعروض أحدث. نتحدث عن **مزايا السحابة**: لا حاجة لمعدات خاصة، قدرة على التوسع السريع (elasticity)، نموذج الدفع حسب الاستخدام. نذكر **المناطق (Regions) وتوزيعها** – مناطق جغرافية تحتوي مراكز بيانات، أهمية اختيار المنطقة القريبة من المستخدمين لتحقيق زمن استجابة أفضل. كذلك **التوافرية** : السحابة توفر مفاهيم مناطق توافر Availability Zones لتوزيع الخدمة من أجل موثوقية. كما نوضح مفهوم **موفّر السحابة Cloud Provider** (AWS, Azure, GCP، وغيرها) وأن العديد من المفاهيم متشابهة بينهم وإن اختلفت التسميات. الهدف تحضير الذهن لفهم الخدمات والأدوات المحددة في الأيام المقبلة بوضع إطار عام لهذا العالم السحابي.

المصطلحات الأساسية:

- Cloud Computing (حوسبة سحابية)
- IaaS / PaaS / SaaS (بنية تحتية/منصة/برمجية كخدمة) [26]
- Region / Availability Zone (منطقة جغرافية/منطقة توافر)
- Elasticity (القدرة على التوسع المرن)
- Cloud Provider (مزود سحابة مثل AWS, Azure, GCP)

فكرة التطبيق العملي: اختر مزود سحابة (AWS أو غيره) وتصفح قائمة الخدمات المتاحة لديه وخرائط المناطق. حاول معرفة أي منطقة أقرب لبلدك. ربما قم بإنشاء حساب مجاني إن أمكن وجرب تشغيل مورد بسيط (مثلاً EC2 صغير أو خدمة Lambda function فارغة). راقب لوحة التحكم كيف تطلب مورد وتوقفه. إن لم تستطع، على الأقل شاهد فيديو تعليمي يشرح نشر تطبيق بسيط على Heroku أو AWS. الفكرة أن ترى خطوات تحويل كودك ليعمل "في السحابة" وكيف تختلف عن تشغيله محليًا.

ملاحظات:

اليوم 56: الحاويات وإدارة الحاويات (Docker & Kubernetes)

شرح مختصر: يوم مخصص لفهم **Docker** و**Kubernetes** على المستوى الذي يحتاجه مهندس الـBackend. نبدأ بـDocker: هو طريقة **لتغليف التطبيق مع كل ما يحتاجه** (مكتبات، إعدادات) في وحدة قابلة للتشغيل (Container) أينما كان [33] . نشرح الفرق بين **Container** و**Virtual Machine** : الحاوية أخف بكثير وتشترك بنظام التشغيل مع غيرها، مما يجعل تشغيل العشرات منها على نفس المضيف ممكنًا بدون أداء سيء [34] . نوضح صورة Docker (Image) وكيف تُبنى من Dockerfile، وتشغيلها كحاوية. ثم ننتقل لـ**Kubernetes (K8s)**: وهو نظام **لتنسيق وإدارة الحاويات** عندما يكون لديك العديد منها عبر عدة خوادم. نشرح المفاهيم الأساسية: **Pod** (وحدة نشر كوبرنيتيس، غالبًا حاوية أو عدة حاويات معًا)، **Node** (العقدة التي تشغّل الحاويات، قد تكون VM)، **Deployment** (وصف لإبقاء عدد معين من البودات تعمل)، **Service** (للشبكة والكشف عن البودات)، **Ingress** (لتوجيه حركة خارجية للداخل). نسلط الضوء على ميزة k8s: تلقائية إعادة تشغيل الحاويات عند فشلها، توزيع الحمل بينها، والقدرة على التوسع الآلي (Horizontal Pod Autoscaler). نذكر بدائل مبسطة: Docker Compose للاستخدام المحلي، أو أنظمة orchestrators أخرى (Docker Swarm, ECS...). كمهندس Backend، ربما لن تدير بنفسك عنقود Kubernetes لكن يجب أن تفهم ماذا يعني عندما يقول فريق العمليات "لقد نشرنا التطبيق في cluster مع 3 replicas" – وما هي تداعيات ذلك كالتزامن والملفات المشتركة الخ.

المصطلحات الأساسية:

- Docker Container (حاوية دوكر) [33]
- Image / Dockerfile (الصورة وملف البناء)
- Container vs VM (فروق الحاوية والآلة الافتراضية) [34]
- Kubernetes (كوبيرنيتيس)
- Pod/Node/Cluster (بود، عقدة، عنقود)
- Deployment/Service (النشر والخدمة في K8s)

فكرة التطبيق العملي: إذا لم تجرّب Kubernetes من قبل، يمكنك استخدام *Katacoda* أو *Play with Kubernetes* (بيئات تفاعلية تعليمية). نفّذ درسًا قصيرًا لنشر تطبيق Hello world على K8s (يتم عادة عبر `kubectl run` أو تطبيق

YAML). لاحظ كيف تحدد عدد النسخ وتعرض التطبيق عبر Service. إن تعذر ذلك، اكتب ببساطة تخطيط نشر: لديك تطبيق وب يحتاج 3 نسخ خلف موزع حمل، أين ستستخدم Docker وأين K8s وما وظيفة كل منهما. مجرد التخطيط يدويًا لما يحصل عند الأمر "انشر 3 نسخ" سيجعلك تفهم الدور المهم لأدوات orchestration.

ملاحظات:

اليوم 57: البنية التحتية ككود وأتمتة الإعداد

شرح مختصر: نتعمق في مفهوم **Infrastructure as Code (IaC)** الذي لمّحنا له من قبل [35] . هنا على نطاق السحابة والبنية التحتية الكاملة. نشرح أن IaC يعني تمثيل موارد البنية التحتية (خوادم، قواعد بيانات، شبكات، تخزين) كملفات تكوين يمكن إدارتها مثل الكود (مراجعة نسخة، إعادة تطبيق، إلخ) بدل إعدادها يدويًا من واجهة مزود السحابة. نعرّف أدوات بارزة: **Terraform** (الأشهر، متعدد المزودين) – نكتب ملفات تصف مثلاً "أريد 3 أجهزة EC2 من النوع كذا ضمن مجموعة توزيع"، ثم يطبقها terraform. أيضًا **CloudFormation** لـAWS، **ARM Templates/Bicep** لأزور، و**Pulumi** (تستخدم لغات برمجة عادية). نذكر فوائد: توحيد الإعداد بين البيئات، القدرة على إنشاء/تدمير بيئات بسهولة (مثلاً بيئة اختبار مؤقتة)، وتعقب التغييرات. نوضح فكرة **معاينة (Plan)** التي تعرض ما سيتغير قبل التطبيق لتجنب أخطاء فادحة. أيضًا **المعوّقات** : منحنيات تعلم أحيانًا، الحاجة لضبط حسابات السحابة والصلاحيات بحذر. ثانياً، **Configuration Management** (يختلف قليلاً عن IaC): أدوات مثل Ansible, Chef, Puppet – تتحكم بإعدادات على الخوادم (تثبيت حزم، تحرير ملفات كونفج)، هذا كان مهمًا قبل عصر الحاويات، ولازال يستخدم لتكوينات معينة. لكن في السحابة الحالية، كثير من ذلك يُستبدل بالحاويات أو صور VM جاهزة، رغم أن Ansible مثلاً لا يزال يستعمل حتى لإدارة Kubernetes أو تشغيل أوامر على مجموعة من الآلات. نؤكد أن Senior backend ليس بالضرورة خبير Terraform، لكن جميل أن يعرف قراءته ويفهم ما يعنيه، ليشارك في قرارات مثل "يمكننا رفع عدد الماكينات أو تغيير حجمها عبر تعديل الكود بدل تكوين يدوي".

المصطلحات الأساسية:

- IaC – Infrastructure as Code (البنية التحتية ككود) [35]
- Terraform (تيرافورم)
- CloudFormation (خدمة AWS IaC)
- Ansible (أنسبل)
- Immutable Infrastructure (بنية تحتية غير قابلة للتعديل – تُستبدل بدل تعديلها)

فكرة التطبيق العملي: حاول قراءة ملف Terraform حقيقي (ابحث في GitHub عن `.tf` مع AWS Terraform). حتى لو لم تفهم كل التفاصيل، حدد: كم مورد EC2 ينشئ؟ هل هناك شبكة VPC معرفة؟ بدلاً من ذلك، جرب أداة أبسط: اكتب ملف Ansible صغير ينفذ أمرًا على جهازك (إن كان لديك Linux أو حتى Windows WSL). مثال: استخدام Ansible لتنصيب Docker على VM نظيف، بدلاً من SSH يدوي. رؤية هذه الأتمتة ستشرح لك قيمتها. إن لم تستطع العملي، شاهد فيديو قصير يوضح نشر بنية تحتية عبر Terraform (سترى سطر الأوامر يخطط ثم ينشئ الموارد).

ملاحظات:

اليوم 58: خدمات سحابية أساسية يجب معرفتها

شرح مختصر: جولة في بعض الخدمات السحابية الأساسية المفيدة لمهندس الـBackend واعتبارات استخدامها. نبدأ بـ**خدمات التخزين**: مثل **Amazon S3** للتخزين الكائني (Object Storage) – تُستخدم لحفظ ملفات بأمان وتوفيرها عبر الويب مع قابلية توسع عالية (نشرح الفرق بينها وبين التخزين على قرص خادم عادي). ثم **قواعد البيانات المدارة**: RDS على AWS مثلًا يقدم قواعد بيانات SQL تقليدية (Postgres, MySQL) دون عناء الإدارة اليدوية (إعداد، نسخ احتياطي، ...)، و**NoSQL المدارة** (DynamoDB, Cosmos DB) – استفد منها بدل تشغيل مثلاً MongoDB بنفسك إلا لو كان سبب معين. أيضًا **خدمات الرسائل**: مثل **AWS SQS** (طابور بسيط) أو **SNS** (إشعارات/مواضيع) – يمكن استخدامها بدل تشغيل RabbitMQ بنفسك في بعض الحالات. **خدمات المراقبة**: (AWS) CloudWatch لجمع Logs وMetrics بسهولة، **خدمات التنبيه**. كذلك **خدمات CDN** (شبكة توزيع المحتوى) كـCloudFront لتسريع توصيل المحتوى الثابت للمستخدمين حول العالم، ومعالجة التخزين المؤقت Edge. نشير إلى **Lambda / Functions** – الحوسبة بدون خادم (Serverless) حيث يمكن تشغيل وظائف استجابة لأحداث دون إدارة خوادم، يفيد مثلاً في مهام متقطعة أو API بسيط. نوضح أنه غالباً كمطور

Backend ستستخدم مزيجًا: خوادم تطبيق + خدمات مدارة للمرفقات (ملفات، رسائل، بحث). المهم أن تكون على معرفة بوجود هذه الأدوات حتى لا تعيد اختراع العجلة، وتستغل قدرات السحابة في حل مشاكل شائعة (مثل إرسال بريد إلكتروني؟ استخدم خدمة SES/SendGrid بدلاً من بناء مخدم SMTP).

المصطلحات الأساسية:

- Object Storage (تخزين كائني – مثل S3)
- Managed Database (قاعدة بيانات مدارة)
- Message Queue Service (خدمة صف رسائل مدارة)
- Content Delivery Network – CDN (شبكة توصيل المحتوى)
- Serverless Functions (وظائف بدون خادم)

فكرة التطبيق العملي: انتق خدمة سحابية واحدة لم تستعملها من قبل واكتشف استخدامها: مثلاً أنشئ حاوية تخزين S3 وارفع ملفًا عبر واجهة الويب، ثم جرّب الوصول إليه عبر URL (ربما ستحتاج جعل الحاوية علنية لأغراض الاختبار). أو جرّب إنشاء قاعدة بيانات RDS صغيرة (يوفر AWS طبقة مجانية لهذا) وحاول الاتصال بها من تطبيقك كما تتصل بمحلية. الإحساس بأن مواردك موجودة "في مكان آخر" لكن يمكنك التعامل معها سيعطيك ثقة أكبر باستخدام السحابة عند الحاجة.

ملاحظات:

اليوم 59: إستراتيجيات النشر والإصدارات في الإنتاج

شرح مختصر: مناقشة طرق متقدمة في **نشر الإصدارات (Deployments)** لتقليل التوقف والمخاطر. نشرح **Blue-Green Deployment** : وجود نسختين من البيئة (زرقاء وخضراء)، واحدة تعمل حالياً (زرقاء) والأخرى تحتوي الإصدار الجديد (خضراء)؛ عند الجاهزية نقوم بتحويل حركة المرور للخضراء فجأة. مزايا: تراجع سهل (نعود للأزرق إذا حصلت مشكلة) وتقليل downtime (تكاد تكون معدومة لأن الخوادم الجديدة جاهزة قبل التحويل). بديل أكثر تدريجي: **Canary Deployment** – نشر الإصدار الجديد لجزء صغير من المستخدمين أولاً، مراقبة المؤشرات، ثم توسيع النسبة تدريجيًا. هذا يقلل خطر أن خطأ ما يؤثر على كل المستخدمين؛ إن رصدته مبكراً توقف التوزيع. نذكر أدوات لتوزيع Canary (في Kubernetes مثلاً باستخدام Istio أو Linkerd للوزن النسبي). أيضًا **Feature Flags** كنمط: السماح بتشغيل/إطفاء ميزات من خلال أعلام برمجية دون إعادة نشر الكود – هذا يمكن استخدامه لإطلاق ميزات لجزء من المستخدمين أو تعطيلها سريعًا لو اكتشفت مشكلة (صمام أمان). نتطرق لموضوع **التوافقية في النشر** : إذا لدينا خدمات متعددة تعتمد على بعضها، وكيف ننشر بدون كسر، ربما نحتاج دعم كلا الإصدارين لفترة (ما يسمى **Backward/Forward compatibility** في الانتقالات). نختم بالتأكيد على **الاختبار في الإنتاج** : رغم أنه ممنوع إحداث مشاكل للمستخدمين، لكن تقنيات كـCanary هي في جوهرها اختبار حقيقي تدريجي. الهدف أن يكون لدى المهندس تصور أن "النشر" ليس مجرد زر يُضغط، بل عملية مدروسة يمكن تنفيذها بطرق مختلفة حسب متطلبات توافر الخدمة ودرجة جرأة الفريق.

المصطلحات الأساسية:

- Blue-Green Deployment (نشر أزرق-أخضر)
- Canary Release (إطلاق الكناري التدريجي)
- Feature Flag (علم الميزة القابلة للتبديل)
- Rollback/Rolling Update (التراجع/التحديث المتدرج)
- Backward/Forward Compatibility (التوافق للخلف/الأمام أثناء التحديث)

فكرة التطبيق العملي: إذا تستخدم أي خدمة توفر خاصية Canary (بعض منصات CI/CD أو Kubernetes through Istio)، جرّب إعداد إطلاق لـ10% من الطلبات بإصدار جديد. إن لم يتسنّ، قم بمحاكاة السيناريو: اكتب خطوات كيف ستطلق نسخة جديدة لـAPI حساس دون قطع الخدمة: مثلاً "سننشر النسخة الجديدة على خوادم منفصلة، ثم باستخدام موازن الحمل نوجّه 5% من المستخدمين لها، نراقب معدل الخطأ. بعد 1 ساعة، إذا الأمور جيدة نزيد إلى 50%...". ثم فكر: لو ظهرت مشكلة عند 50%, ما خطة rollback؟ تدوين ذلك يبرز نقاط تحتاج التفكير المسبق لتجنب ارتباك أثناء الحادثة الحقيقية.

ملاحظات:

اليوم 60: إدارة التكلفة والسحابة المتعددة/الهجينة

شرح مختصر: اليوم الأخير يتناول بعض المواضيع الإدارية/الإستراتيجية في البنية التحتية السحابية. أولاً **إدارة تكلفة السحابة** : يوضح أن سهولة تخصيص الموارد على السحابة قد تؤدي لفواتير ضخمة إن لم يكن هناك حوكمة. نذكر ممارسات: مراقبة الاستخدام عبر أدوات المزود (مثلاً Cost Explorer في AWS)، وضع تنبيهات ميزانية. تصميم معماري مراعي للتكلفة: مثلاً استخدام instance types مناسبة (ليس دوماً الأكبر أفضل)، إيقاف البيئات غير الضرورية خارج أوقات العمل (للأنظمة الاختبارية). كذلك **الاستفادة من الخدمات المدارة** قد يوفر جهدًا، لكن أحياناً تشغيل شيء ذاتيًا على VM قد يكون أوفر بالمال – الموازنة مطلوبة. ثم **Multi-Cloud** : توضيح فكرة استخدام أكثر من مزود سحابة إما لتجنب الارتباط ببائع واحد (vendor lock-in) أو للاستفادة من كل واحد في جزء معين. نذكر أنه يحمل تعقيدات (تعدد المهارات المطلوبة، نقل البيانات بينهما). **Hybrid Cloud** : الجمع بين موارد سحابية وعند مقر الشركة (On-Premises) – يحدث لدى المؤسسات التي لديها مراكز بيانات قائمة وتريد التوسع للسحابة تدريجيا، أو اعتبارات سيادية للبيانات. مهندس الـBackend قد لا يتخذ قرارات متعددة السحابة لكنه يجب أن يفهم قيود بيئته: مثلاً إن كانت شركته تعتمد مزودًا واحدًا بالكامل، يجب أن يعرف خدماته جيدًا، وإن كانت هجينة ربما يتعامل مع latency أكبر بين مكونات موزعة، إلخ. وأخيرًا نذكّر بضرورة متابعة الجديد: التقنيات السحابية تتطور (functions, edge computing)، فكن دائم التعلم. **خاتمة:** هنيئًا لك إتمام 60 يوم من التطوير والتعلم المستمر! ابق شغوفًا، فمجالنا دائم التغير والتحديات الجديدة تنتظر دومًا.

المصطلحات الأساسية:

- Cloud Cost Management (إدارة تكلفة السحابة)
- Vendor Lock-In (ارتباط بمزود محدد)
- Multi-Cloud (سحابة متعددة)
- Hybrid Cloud (سحابة هجينة)
- Budget Alerts (تنبيهات الميزانية)

فكرة التطبيق العملي: إن كنت مطلعًا على تكاليف البنية التحتية لمشروعك، حاول تحديد **أغلى ثلاثة بنود** (مثلاً الخوادم vs قواعد البيانات vs خدمة خارجية). فكر كيف يمكن تحسينها (مثلاً تبديل نوع خادم، أو رفع كفاءة كود لتقليل الحاجة لموارد، أو جدولة إيقاف ليلاً). إن لم تكن، جرّب استخدام حاسبة تسعير لأحد المزودين: صمّم بنية افتراضية (مثلاً 3 سيرفرات تطبيق و2 قاعدة بيانات وحمل، و100GB تخزين) وانظر كم تكلف شهريًا على AWS. هذا يعطي منظورًا لحجم المال الذي قد يصرف، وبالتالي أهمية قرارات مثل استخدام مورد أقل قدرة أو خدمة مدارة. بالنسبة لـMulti-cloud، اقرأ قصة شركة تبنتها أو تركتها (تجد مدونات تناقش ذلك) لاستيعاب الدوافع.

ملاحظات:

تهانينا على إكمال خطة الـ60 يوم! أنت الآن على دراية بطيف واسع من المهارات المتقدمة في تطوير الـBackend. تذكّر أن التعلم المستمر والتطبيق العملي هما المفتاح لترسيخ هذه المعرفة. بالتوفيق في مشوارك الهندسي! [32]

[1]

---

[1] Monolith vs. Microservices: Trade-offs, Pitfalls, and How to Choose | by Stefano Alvares | Beyond the Brackets | Medium
https://medium.com/beyond-the-brackets/monolith-vs-microservices-trade-offs-pitfalls-and-how-to-choose-bef2e793961c

[2] [4] [5] [6] [7] What Is the CAP Theorem? | IBM
https://www.ibm.com/think/topics/cap-theorem

[3] [8] [9] [10] [11] SQL vs. NoSQL Databases: What's the Difference? | IBM
https://www.ibm.com/think/topics/sql-vs-nosql

[12] [13] Event-Driven Architecture
https://aws.amazon.com/event-driven-architecture/

difference between exactly-once and at-least-once guarantees [14]
https://codemia.io/knowledge-hub/path/difference_between_exactly-once_and_at-least-once_guarantees

Saga Design Pattern - Azure Architecture Center | Microsoft Learn [15]
https://learn.microsoft.com/en-us/azure/architecture/patterns/saga

Saga pattern - AWS Prescriptive Guidance [16]
https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/saga-pattern.html

Three Pillars of Observability: Logs, Metrics and Traces | IBM [17]
https://www.ibm.com/think/insights/observability-pillars

Circuit Breaker Pattern for Resilient Systems [19] [18]
https://dzone.com/articles/circuit-breaker-pattern-resilient-systems

Resilience in Microservices: Bulkhead vs Circuit Breaker | by Parser [20]
https://medium.com/@parserdigital/resilience-in-microservices-bulkhead-vs-circuit-breaker-54364c1f9d53

Circuit Breaker and Bulkhead Patterns for Resilience - Medium [21]
https://medium.com/@platform.engineers/circuit-breaker-and-bulkhead-patterns-for-resilience-2a8ae88ac717

Chaos engineering - Wikipedia [23] [22]
https://en.wikipedia.org/wiki/Chaos_engineering

Why developer experience is more important than productivity [24]
https://www.atlassian.com/blog/devops/developer-experience-more-important

What Is a Race Condition? - Akamai [25]
https://www.akamai.com/glossary/what-is-a-race-condition

What is Infrastructure as Code? - IaC Explained - AWS [35] [33] [26]
/https://aws.amazon.com/what-is/iac

DORA | DORA's software delivery metrics: the four keys [31] [30] [29] [28] [27]
/https://dora.dev/guides/dora-metrics-four-keys

What is developer experience? Complete guide to DevEx measurement and improvement (2026) [32]
/https://getdx.com/blog/developer-experience

Containers vs Virtual Machines | Atlassian [34]
https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms