



ازاي تبقى Backend Engineer شاطر؟

يا جماعة، لو انت Backend Engineer أو عايز تبقى واحد محترم في المجال ده، لازم تبقى فاهم شوية حاجات أساسية. النهاردة هكلمكم عن 9 حاجات لو فهمتهم كويس، هتبقى من أحسن المهندسين في الشغل! 🚀

1 Communication Protocols - إزاي الكمبيوترات بتتكلّم مع بعض

ده أول حاجة لازم تفهمها كويس. تخيل إنك عايز تبعث رسالة لصاحبك، في كذا طريقة تعمل كده صح؟

TCP vs UDP

- **TCP (Transmission Control Protocol)**: ده زي المكالمة التليفونية، لازم يكون في اتصال مضمون وثابت (Connection-oriented). كل packet بيوصل بالطبط وفي الترتيب الصح، بس بطيء شوية في البداية بسبب ال 3-way handshake.
- **UDP (User Datagram Protocol)**: ده زي الواتساب فويس نوت، أسرع بس مش مضمون يوصل 100%. (Connectionless). بيستخدموه في الستريمينج والألعاب اللي السرعة فيها أهم من الدقة.
- **QUIC Protocol**: ده بروتوكول جديد اتعمل من Google، بيجمع أحسن حاجة في TCP (الموثوقية) مع أحسن حاجة في UDP (السرعة)! HTTP/3 بيستخدم QUIC بدل TCP، وده بيخلي الويب أسرع بكثير خصوصاً في الاتصالات الضعيفة.
- **HTTP Protocol**: لازم تفهم HTTP كويس لأنه الأساس في كل حاجة على الويب:
 - **HTTP/1.1**: بيفتح connection واحد لكل request
 - **HTTP/2**: بيستخدم multiplexing (طلبات كتير على نفس ال connection)
 - **HTTP/3**: بيستخدم QUIC بدل TCP، أسرع وأكثر أماناً
- لو مش فاهم الفروقات دي، ممكن تعمل أبليكيشن بطيء جداً (زي مثلاً لو فاتح وقافل TCP connections كتير في لوب).
- **WebSockets**: ده البروتوكول اللي بيخلي في تواصل مستمر في الاتجاهين (Full-duplex communication). محتاجه في أبليكيشنز زي الشات، الإشعارات اللحظية، أو الألعاب اللي بتحصل live.
- **gRPC**: بروتوكول جديد من Google بيستخدم HTTP/2 و Protocol Buffers، اتعمل عشان يحل مشاكل REST APIs. بيستخدموه كتير في ال microservices لأنه أسرع وأكفأ.



2 Web Servers - الخوادم

الويب سيرفرز دي شغلته الأساسية إنها تخدم المحتوى (Content).

إزاي ال **Web Servers** بتشتغل؟ الويب سيرفر بيستقبل HTTP requests من الكلاينت (المتصفح مثلاً)، بيعالج الطلب ده، وبعدين بيرد بـ HTTP response فيه المحتوى المطلوب. العملية دي بتحصل من خلال:

1. استقبال الطلب على Port معين (عادة 80 للـ HTTP أو 443 للـ HTTPS)
2. تحليل ال Request headers
3. معالجة الطلب (قراءة ملف أو تنفيذ كود)
4. إرجاع ال Response مع ال headers المناسبة

:Dynamic vs Static Content

- **Static Content**: محتوى ثابت زي الصور والملفات والـ CSS والـ JavaScript. المحتوى ده بيتخزن على الديسك وبيتقرأ مباشرة من غير معالجة. هنا بييجي دور ال Caching لأن المحتوى مش بيتغير كثير.
 - **Dynamic Content**: محتوى بيتعمل أول بأول حسب الطلب، زي صفحة بروفايلك على الفيسبوك. كل مرة تفتحها بتتولد من جديد حسب بياناتك الحالية.
- ETags (Entity Tags)**: ال ETag ده زي الباركود أو البصمة للملف. لما الكلاينت بيطلب ملف، السيرفر بيعت معاها ETag. المرة الجاية لما الكلاينت يطلب نفس الملف، بيعت ال ETag اللي عنده. السيرفر يقارن:

- لو ال ETag نفسه → بيعت "304 Not Modified" (يعني استخدم اللي عندك من الكاش)
- لو ال ETag اتغير → بيعت الملف الجديد

ده بيووفر bandwidth ويخلي الموقع أسرع!

:Threading Models في فرق كبير بين السيرفرات:

- **Single-threaded non-blocking (Event-driven)**: زي Node.js، بيستخدم thread واحد بس بيشغل بطريقة asynchronous. فعال جداً للـ I/O operations
- **Multi-threaded**: زي Apache Tomcat أو Apache web server، كل request بياخد thread منفصل. قوي للعمليات الحسابية الثقيلة

:Nginx - Multi-Role Server سيرفر زي Nginx بيشغل كـ:

- Web Server (بيخدم static files)
- Reverse Proxy (بيحول الطلبات للسيرفرات الداخلية)
- Load Balancer (بيوزع الأحمال)

عشان كده محتاج تفهم HTTP protocol كويس جداً عشان تعرف تعمل configuration صح.

3 Database Engineering - هندسة قواعد البيانات

المجال ده معقد جداً وفيه فرص تخصص كثير!

:Relational vs NoSQL

- **Relational Databases**: زي MySQL و PostgreSQL، بتحافظ على الـ ACID properties وعلاقات قوية بين البيانات. مثالية للبيانات المنظمة (زي بيانات البنوك).
- **NoSQL Databases**: زي MongoDB و Cassandra، بتستخدم نماذج مختلفة (Document, Key-Value, Graph). أسهل في الـ Scaling بس بتضحى ببعض ضمانات الـ ACID.

ACID Properties: أي Backend Engineer لازم يفهم المبادئ الأربعة دي:

- **Atomicity (الذرية)**: إما كل العملية تنفذ أو ولا حاجة (All or Nothing)
- **Consistency (الاتساق)**: البيانات دايماً في حالة صحيحة ومتسقة
- **Isolation (العزل)**: العمليات المتزامنة ما تأثرش على بعضها
- **Durability (الاستمرارية)**: لما العملية تنجح، البيانات بتفضل محفوظة حتى لو حصل عطل

دي أساس قواعد البيانات العلائقية (Relational Databases).

ليه NoSQL؟ قواعد البيانات دي ظهرت عشان المهندسين لقوا إن الـ Relational Databases صعب scale-ها أفقياً (Horizontal Scaling). NoSQL بتضحى ببعض خواص الـ ACID (زي Strong Consistency) عشان تبقى أسرع وأسهل في التوزيع على سيرفرات كثير.

التخصصات الدقيقة: ممكن تتخصص في حاجات دقيقة جداً زي الـ LSM Trees (Log-Structured Merge Trees) - نوع من الـ Indexes بيستخدموه في (NoSQL).



ال Proxies، خصوصاً ال Reverse Proxies، بقت مهمة جداً دلوقتي بسبب انتشار ال Microservices.

Proxy vs Reverse Proxy - إيه الفرق؟

- **Forward Proxy (Proxy):** بيشتغل نيابة عن ال Client. الكلاينت بيقول لل Proxy "روح جيلي الصفحة دي"، وال Proxy بيروح يجيبها. السيرفر مش عارف مين الكلاينت الحقيقي (زي VPN).
- **Reverse Proxy:** بيشتغل نيابة عن ال Server. الكلاينت بيكلم ال Reverse Proxy، وال Reverse Proxy بيحول الطلب للسيرفر الداخلي المناسب. الكلاينت مش عارف مين السيرفر الحقيقي.

Reverse Proxy Applications (الاستخدامات):

- **Caching Layers:** تخزين الردود المتكررة
- **Load Balancing:** توزيع الطلبات على سيرفرات كتير
- **TLS Termination:** فك تشفير ال HTTPS مرة واحدة بدل ما كل سيرفر يعملها
- **Compression:** ضغط الملفات قبل إرسالها
- **Security:** حماية السيرفرات الداخلية من الهجمات
- **API Gateway:** نقطة دخول موحدة لل Microservices

Layer 4 vs Layer 7 Proxy

- **Layer 4 (Transport Layer):** بيشتغل على مستوى TCP/UDP، أسرع بس مش بي فهم محتوى الرسالة. بيوزع الأحمال بناءً على IP و Port بس.
- **Layer 7 (Application Layer):** بيشتغل على مستوى HTTP، بي فهم محتوى الطلب. يقدر يوزع الأحمال بناءً على URL أو Headers أو Cookies. أبطأ شوية بس أذكى.

Load Balancing Algorithms (خوارزميات توزيع الأحمال):

- **Round Robin:** بيوزع الطلبات بالدور (واحد واحد على كل سيرفر)
- **Least Connections:** بيعت الطلب للسيرفر اللي عليه أقل عدد اتصالات نشطة
- **IP Hash:** بيستخدم IP الكلاينت عشان يحدد أي سيرفر يروحله (بيضمن إن نفس الكلاينت يروح لنفس السيرفر)
- **Weighted Round Robin:** زي Round Robin بس السيرفرات الأقوى بتاخذ طلبات أكثر
- **Least Response Time:** بيعت للسيرفر اللي بيرد أسرع

Service Meshes: الأنظمة دي اللي بتستخدم في ال Microservices (زي Linkerd و Istio)، في الآخر هي مجموعة Proxies متقدمة بتشتغل جنب كل Microservice!

5 Caching - التخزين المؤقت

الـ Caching مهم في كل حدة في الـ Technology Stack!

امتى نستخدم **Caching**? استخدم الكاش لما يكون عندك:

- بيانات بتتقرأ كثير بس ما بتتغيرش كثير (زي قائمة المنتجات في متجر إلكتروني)
- عمليات حسابية ثقيلة نتيجتها مش بتتغير بسرعة (زي تقارير إحصائية)
- استعلامات قاعدة بيانات متكررة (زي معلومات المستخدم)
- محتوى ثابت (صور، ملفات CSS/JS)
- **API responses** اللي نتيجتها مش بتتغير كل ثانية

متستخدمش **Caching** لما:

- البيانات بتتغير باستمرار (زي أسعار الأسهم اللحظية)
- المعلومات لازم تكون Real-time (زي رصيد البنك في لحظة السحب)
- البيانات حساسة ومش المفروض تتخزن (زي passwords)

أنواع الـ **Caching**:

- **In-Memory Caching**: زي Redis أو Memcached، سريع جداً لأنه في الـ RAM
- **CDN Caching**: للملفات الثابتة في مواقع جغرافية قريبة من المستخدم
- **Browser Caching**: المتصفح بيخزن الملفات عند المستخدم
- **Application-Level Caching**: الكاش جوه الكود بتاعك

Stateful vs Stateless Caching

- **Stateless**: الكاش مش بيحتفظ بمعلومات عن الـ session، كل request مستقل
- **Stateful**: الكاش بيحتفظ بمعلومات الـ session (محتاج تخلي الكلاينت يروح لنفس السيرفر)

المشكلة الصعبة (**Cache Invalidation**): الـ Caching يعتبر من أصعب المشاكل في علوم الحاسب بسبب موضوع الـ Cache Invalidation:

- إزاي تعرف امتى تمسح الكاش؟ (TTL - Time To Live)
- إزاي تضمن **consistency** البيانات بين الكاش والداتابيز؟
- إيه اللي يحصل لو الكاش فيه بيانات قديمة؟

Phil Karlton قال: "There are only two hard things in Computer Science: cache invalidation and naming things" 😊

6 Messaging Systems - أنظمة الرسائل

الأنظمة دي بتنسق التواصل بين الأجزاء المختلفة في ال Architecture.

Message Queue (الطوابير): ال Queue ده نظام (FIFO (First In First Out - أول واحد يدخل أول واحد يخرج، زي الطابور العادي. كل رسالة في الطابور بتروح لمستهلك واحد بس (Consumer). أمثلة: RabbitMQ و ZeroMQ.

Pub/Sub (Publish/Subscribe): في النظام ده، الناشر (Publisher) بينشر رسالة على topic معين، وكل المشتركين (Subscribers) في ال topic ده بياخدوا نسخة من الرسالة. واحد ينشر، كتير يستقبلوا. مثال: Kafka و Google Pub/Sub.

امتى تستخدم Message Queue؟ استخدم ال Queue لما:

- عايز تضمن إن كل رسالة تتعالج مرة واحدة بس (exactly once processing)
- عندك مهام ثقيلة محتاج توزعها على workers كتير (زي معالجة الصور أو الفيديوهات)
- محتاج Load Leveling - يعني لو فيه طلبات كتير فجأة، الطابور يمسخهم ويوزعهم على مهل
- عايز retry mechanism - لو المهمة فشلت، ترجع للطابور تاني

امتى تستخدم Pub/Sub؟ استخدم Pub/Sub لما:

- محتاج أكثر من سيستم يستقبل نفس الرسالة (زي لما يحصل order جديد: سيستم المخزون يعرف، سيستم الشحن يعرف، سيستم الإيميلات يعرف)
- عايز Event-Driven Architecture حيث كل سيستم بيتفاعل مع الأحداث
- محتاج Decoupling بين الأنظمة - الناشر مش عارف ولا مهتم مين المستقبلين
- عندك Real-time notifications أو broadcasting (زي إشعارات الأخبار العاجلة)

المبدأ الأساسي: المهم تفهم ليه النظام ده اتعمل أصلاً وببطل إيه مشكلة:

- Queue: واحد لواحد، ضمان المعالجة
- Pub/Sub: واحد لكثير، توزيع الأحداث

7 Web API Frameworks - أطر عمل الـ API

API Authoring (تأليف الـ APIs): الـ Frameworks دي بتخليك تكتب كود ديناميكي وتفتح إمكانيات كتير، خصوصاً لبناء REST APIs أو gRPC APIs.

أشهر الـ Frameworks:

- **JavaScript: Node.js + Express:** على السيرفر، سريع ومناسب للـ Express. I/O intensive applications. هو أشهر framework فوق Node.js.
- **Framework: Django (Python):** كامل (Batteries included)، فيه كل حاجة جاهزة من ORM لـ Admin Panel. مثالي للمشاريع الكبيرة.
- **Flask (Python):** أخف من Django، بيديك حرية أكثر في الاختيارات.
- **FastAPI (Python):** حديث وسريع جداً، بيستخدم Type Hints وبيولد API documentation تلقائياً.

كل Framework ليه مميزاته، المهم تختار الأنسب لمشروعك!

8 Message Formats - صيغ الرسائل

الصيغ دي كلها عن التوازن بين إن الإنسان يقرأها بسهولة vs الكمبيوتر يعالجها بكفاءة.

XML (Extensible Markup Language): الصيغة الأصلية، اتعملت عشان الإنسان يقدر يقرأها، بس:

- حجم الملف كبير جداً (كتير tags)
- بطيء في الـ parsing
- مش efficient للكمبيوتر

JSON (JavaScript Object Notation): أشهر وأسهل من XML، وبقي المعيار الحالي:

- الإنسان يقدر يقرأها بسهولة
- الكمبيوتر يعالجها أحسن من XML
- حجم أصغر نسبياً
- سهل في الاستخدام مع JavaScript وأغلب اللغات

Protocol Buffers (Protobuf): Google's solution للكفاءة القصوى:

- حجم صغير جداً - بيصغر حجم البيانات لحد 10x من JSON!
- سريع في الـ **Serialization/Deserialization**
- مش Human-readable (بايتات binary)
- محتاج تعرف الـ Schema (التعريف) بتاع البيانات
- مثالي للـ Microservices Communication (زي gRPC)

متى تستخدم إيه؟

- **JSON:** للـ REST APIs العامة، التطبيقات البسيطة، لما الـ readability مهمة

mohamed diaa

الأمان مجال ضخم وفيه طلب كبير جداً في سوق الشغل!

:TLS (Transport Layer Security) & Encryption

- **TLS**: البروتوكول المسؤول عن تشفير الاتصال بين الكلاينت والسيرفر (ال "S" في HTTPS). يمنع هجمات زي:
 - **Man-in-the-Middle (MITM)**: حد يقعد في النص ويقرأ أو يعدل البيانات
 - **Replay Attacks**: حد يسجل طلبك ويعيد إرساله ثاني
- **Encryption at Rest**: تشفير البيانات المخزنة على الديسك
- **Encryption in Transit**: تشفير البيانات أثناء نقلها (زي TLS)

Credential Management (إدارة البيانات الحساسة): لازم تفهم إزاي تخزن البيانات الحساسة بشكل آمن:

- **Passwords**: استخدم Hashing algorithms قوية (زي bcrypt أو Argon2)، مش Encryption!
- **API Keys & Secrets**: خزنهم في Environment Variables أو Secret Management Tools (زي HashiCorp Vault أو AWS Secrets Manager)
- **Database Credentials**: ما تحطهاش في الكود أبداً!
- ما تعملش commit لل secrets في Git!

Firewalls (جدران الحماية): استخدام ال Firewalls عشان تتحكم في Network Traffic:

- قفل كل البورتات غير اللي محتاجها (Principle of Least Privilege)
- فتح (HTTP) Port 80 و (HTTPS) 443 بس لل Web Servers
- قفل Database ports - ما يكونش accessible من الإنترنت مباشرة
- استخدم Security Groups في ال Cloud (AWS, Azure, GCP)

Attack Mitigation (التعامل مع الهجمات): لازم تعرف أنواع الهجمات وإزاي تتعامل معاها:

- **DDoS (Distributed Denial of Service)**: هجوم بطلبات كتير جداً عشان يوقع السيرفر. الحل: Rate Limiting وCDNs
- **SQL Injection**: استخدام Parameterized Queries وORMs
- **XSS (Cross-Site Scripting)**: تنظيف وvalidation لل User Input
- **CSRF (Cross-Site Request Forgery)**: استخدام CSRF Tokens

SNI (Server Name Indication): تقنية أمان بتسمح لأكثر من موقع يستخدم SSL/TLS على نفس ال IP address. مهمة في عصر ال Virtual Hosting.

لو عايز تبقى Backend Engineer شاطر فعلاً، لازم تفهم الأساسيات دي كويس. مش لازم تبقى خبير في كل حاجة، بس لازم يبقى عندك فهم كويس لكل موضوع عشان تعرف امتى تستخدم إيه.

نصيحتي ليك:

1. ابدأ بموضوع واحد وافهمه كويس
 2. اعمل مشاريع عملية تطبق فيها اللي اتعلمته
 3. اقرأ Documentation وال RFC الخاصة بالبروتوكولات
 4. متخفش تغلط، الغلط جزء من التعلم
 5. تابع التطورات الجديدة في المجال
- وفكر دائماً: ليه الحاجة دي اتعملت؟ وإيه المشكلة اللي بتحلها؟

متابعة أسبوعية

كل يوم جمعة هنزل بوست مفصل عن واحد من المواضيع دي!

هشرح كل نقطة بالتفصيل الممل، مع أمثلة عملية وكود حقيقي. تابعني عشان ما يفوتكش أي حاجة، وقولولي في الكومنتات أنهي موضوع عايزين نبدأ بيه الأسبوع الجاي! 🔥

لو عندك أي سؤال أو عايز تناقش أي نقطة، اكتبلي في الكومنتات! 🙌