Software Architecture

for Developers

Anees Hikmat

بسم الله الرحمن الرحيم

من مبرمج إلى معماري: دليلك لتصميم البرمجيات بذكاء (الجزء الأول)

دليلك العملي للتفكير كمعماري: من الشيفرة البرمجية إلى معمارية الأنظمة وطرق صياغتها، ومن مجرد مبرمج إلى صانع قرار في تصميم البرمجيات

تأليف: أنيس حكمت أبوحميد

الموقع الإلكتروني: 2nees.com

إهداء

إلى إخواننا المسلمين الصابرين من "**الإيغور**"؛ القابضين على دينهم في زمن المحنة... والذين سُلبت منهم أبسط حقوق الحياة، وجُرّدوا من الكرامة، وضُيّق عليهم في رحاب دينهم تحت وطأة الخوف والقتل والتعذيب والإرهاب...

وإلى فرسان العقيدة خلف الأسوار المظلمة، والقضبان الموجعة؛ الذين قضوا زهرة أعمارهم خلف الجدران الموحشة، بلا جرمٍ سوى أنهم صدعوا بيقينهم قائلين: "**ربنا الله**".

وإلى الأمهات اللواتي انتظرن بقلوبٍ كواها الشوق، والآباء الذين غابوا ولم يعودوا، والأطفال الذين كبروا يتلمسون ملامح آبائهم في ذاكرة النسيان...

نهدي لكم هذا الكتاب؛ لأن ثباتكم على دينكم رغم كل شيء؛ هو من أسمى معاني الحياة وأجل صور البطولة! هذا الكتاب لكم لأننا نبكي عليكم في كل يوم ولا نقدر على نصركم... لكن، إن غُلّت أيدينا عن نصرتكم، فإن قلوبنا لن تنساكم، وستبقى قضيتكم حية فينا ما حيينا... وحسبنا الله ونعم الوكيل.

المقدمة

الحمد لله ربّ العالمين، يُحب من دعاه خفياً، ويُجيب من ناداه نجيّاً، ويزيدُ من كان منه حيِيّاً، ويكرم من كان له وفيّاً، ويهدي من كان صادق الوعد رضيّاً، الحمد لله ربّ العالمين.

إن كنت تقرأ هذه الكلمات، فأظن أنك تشعر في قرارة نفسك بأن ثمة شيئاً ينقصك! ليس في قدرتك على كتابة الشيفرة البرمجية، بل في قدرتك على **رؤية الصورة الكاملة**؛ أن تنظر إلى النظام الذي تبنيه كبنيان متكامل يخدم هدفا، ويعيش في عالم حقيقي مليء بالقيود والمتطلبات والمستخدمين، لا كمجموعة ملفات وأسطر موزعة من الشيفرة البرمجية هنا وهناك!

هذا الشعور الذي تُحسّه هو ما دفعك إلى فتح هذا الكتاب، وهو أيضاً السبب الذي دفعني إلى كتابته! لأن الأدوار المعمارية -خصوصا في عصر الذكاء الاصطناعي- لم تعد حكرًا على الأدوار القيادية فقط، فكل مطور يحتاج أن يفهم كيف تُبنى الأنظمة، وكيف يتم توثيقها، وكيفية اتخاذ القرارات التصميمية الصحيحة.

لمن هذا الكتاب؟

هذا الكتاب لكل مطور يريد أن يتجاوز حدود المهام اليومية الروتينية، ويريد أن يفهم لماذا بني النظام بهذه الطريقة لا تلك! وهذا الكتاب لكل مطور سمع بمصطلحات كال

Microservices وال Scalability وال Architectural Drivers، فأومأ برأسه دون أن يدرك كيف تتشابك هذه الأفكار معاً لتُشكّل قراراً حقيقياً!

وهذا الكتاب لكل مطور يطمح أن يصير يوماً ما صاحب رؤية في فريقه...

ماذا ستجد في هذا الكتاب؟

لن تجد هنا وصفة سحرية ولا نهجاً ثورياً جديداً، بل ستجد دليلاً عملياً يسعى إلى **الوسطية!** تلك الكلمة العظيمة التي أكرمنا الله -سبحانه وتعالى- بها في كتابه العزيز حين قال: ﴿وَكَذَٰلِكَ جَعَلْنَاكُمْ أُمَّةً وَسَطًا﴾، فلا إفراط في التصميم المسبق الذي يُضيّع الوقت ويُعيق المرونة، ولا تفريط في التفكير المعماري الذي يتركنا نبني أبراجا على الرمال.

كيف تستفيد منه؟

اقرأ والقلم في يدك، وضع خطوطاً تحت ما يلفت انتباهك، واسأل نفسك عند كل باب: "**كيف ينطبق هذا على المشروع الذي أعمل عليه الآن؟**" فالمعرفة التي لا تتصل بالواقع تبقى حبراً على ورق، أما تلك التي تُسقطها على ما تعيشه يومياً فهي التي تُغير طريقة تفكيرك إلى الأبد.

المرجع الرئيسي لهذا الكتاب؟

المرجع الرئيسي لهذا الكتاب هو: **Software Architecture for Developers** لمؤلفه: Simon Brown...

وأخيراً، تذكّر دائماً أن هذا العلم هو في جوهره أمانة وإحسان، فمن الأمانة أن تُحسن ما تبني، ومن الأمانة أن تُوجّه من يعمل معك بشكل صحيح وأن تحسن إليهم، ومن الأمانة أن تتخذ القرار الصحيح في المكان الصحيح وإن خالف رغباتك...

نسأل الله -سبحانه وتعالى- التوفيق والسداد، وأن يجعل ما نتعلمه نوراً يهدينا ويهديكم إلى الحق بإذنه.

والله وليّ التوفيق

فائدة

علّم نفسك وعود لسانك على الإكثار من الأذكار والدعاء، خصوصًا آناء الليل وأطراف النهار، بقلب متواضع خاشع لله -سبحانه وتعالى-، وأجمل اللحظات هي تلك التي تجد نفسك فيها تذكر الله -سبحانه وتعالى- عند شرود ذهنك بدلًا من أن تجده يردد قبيح القول أو عديم نفعه...

- كتاب إلى الجنة زمرا، الصفحة ١٨٣

الفهرس

الفصل الأول: ال Architecture

إن أول سؤال بديهي يمكن أن نبدأ به في هذا الكتاب هو:

ما هي معمارية البرمجيات؟

إن الإجابة على هذا السؤال قد تكون فضفاضة، فهناك الكثير من التعريفات أو المصطلحات التي يمكن صياغتها للتعبير عن هذا المفهوم، لكن يمكننا القول أن معمارية البرمجيات -بالمفهوم العام- يمكن تعريفها من منظورين مختلفين، وهما:

١. المعمارية ك Noun -اسم-: تمثل أو تشير إلى هيكل -structure- النظام، أي كيفية تقسيم ال Product إلى مكونات أصغر -building block- وكيف تتفاعل مع بعضها البعض، أي أنها البنية الخاصة بالنظام والتي تتألف من software elements، بما في ذلك الخصائص الظاهرة لهذه ال elements والعلاقة فيما بينها...

٢. المعمارية ك Verb -فعل-: هذا المفهوم يشير إلى أو يركز على "كيفية بناء" النظام بطريقة منظمة وفعالة بدلا من مجرد وصف المكونات وعلاقاتها، لأن هذه العملية يجب أن تبقى مستمرة ومرتبطة بتصميم وبناء ال product طوال فترة المشروع، وهذا يشمل اتخاذ

القرارات المتعلقة بال structure والمكونات والتفاعل فيما بينها، ويشمل الاختيارات المتعلقة بالاتجاه التقني وال standards وال guidelines التي توجه تطوير ال product.
إن تأثير هذه التعريفات أو العلم بها يجعلك تنتقل من دور إلى دور، ومن مجرد مصمم يقوم بإنشاء مخططات -Noun- ثم يتوقف، إلى قائد تقني يمكنه أن يوجه الفريق باستمرار -Verb- ويشارك بفعالية في عملية اتخاذ القرارات اليومية... وبدلا من أن تضع المخطط أو التصميم في أول يوم ثم تتركه ليشيخ، فإنك ستحافظ على استمرارية مرحلة الشباب لهذا المخطط من خلال ضمان أن ال structure الذي وضع وصمم ما زال ملائما ومناسبا للتغييرات والتحديات والاحتياجات الجديدة... ثم إن هذا الاختلاف يبرز الفروقات بين المصمم والقائد، فالقائد لن يكتفي بمجرد التصميم، بل سيجد الأدوات الجيدة وآلية التواصل المناسبة لإيصال هذا التصميم إلى الفريق، وتوجيه الفريق لآلية العمل المناسبة مع هذا التصميم... لذلك فمهارات التواصل هنا لا تقل أهمية عن الجانب التقني...

إن عملية تطوير البرمجيات ليست مجرد عملية تتم لمرة واحدة في البداية ثم تنتهي، بل هي عملية مستمرة ومتكررة، لذلك فإن هذا النهج يتوافق مع ال Agility التي تعتمد على التطور المستمر بدلا من التخطيط المفرط!
إن التوازن بين التصميم والتنفيذ -بين ال none وال verb- يجب أن يكون توازنا صحيا حتى يتجنب الفريق مخاطر التصميم الكبير مقدما (Big Design UpFront) أو انعدام أي تصميم على الإطلاق.

لا نريد استعجال الحديث الآن، فما زلنا في البداية ^^...

ال Types of architecture

بناء على ما ذكرناه في التعريف العام لكلمة Architecture قد يتبادر لذهنك ثاني سؤال منطقي، وهو ما هي أنواع المعماريات التي تحتاج إلى تصميم داخل المجال التقني؟ والجواب حقيقة فيه الكثير من الخيارات وغالبا ما تكون تحت واحد من هذه الأبواب:

ال Security، وال Technical، وال Solution، وال Network، وال Data، وال Hardware، وال Enterprise، وال Application، وال System، وال Integration، وال IT، وال Database، وال Information، وال Process، وال Business، وال Software...

أظنك الآن في مرحلة تفكير، كثير من هذه المصطلحات فضفاضة! أليس كذلك؟
وجوابك للأسف صحيح، فبعض هذه المصطلحات قد تحمل أكثر من معنى، وبعض هذه المصطلحات يشير بعضها إلى بعض، ويعتمد بعضها على بعض... مثلا لو قلنا ماذا تعني فعليا لك ال Solution architecture؟ وهل أنت متأكد أن نفس ما أجبته سيكون إجابة الشركة س والشركة ص؟
بكل تأكيد لا، فالشركة س قد تخبرك أنها مرادفة لكلمة Software architecture في حين أن الشركة ص ستخبرك أن هذا مصطلح محدد للتركيز على تصميم حل شامل لمشكلة معينة متوقفا قبل الوصول إلى التفاصيل التنفيذية!
وكذلك الأمر ل Technical architecture فهل تشير إلى البرمجيات أم الأجهزة أم كلاهما معا؟

إن هذه الأنواع يمكن نسبها للأشخاص أيضا... فإذا قلنا Solution architecture فإن الموظف أو الشخص المسؤول عن ذلك يطلق عليه Solution architect... لذلك فما ذكرناه على النوع ينطبق على الشخص... وهذا يظهر جليا في اختلاف الوصف للوظائف المتعلقة بنفس المسمى الوظيفي باختلاف الشركات...

وبعيدا عن كل هذا الكلام، ما هو القاسم المشترك بين هذه الأنواع؟
على الرغم من اختلاف المصطلحات عند الإشارة لهذه الأنواع، فإن جميع هذه أنواع تشترك في عنصرين أساسيين، وهما:
١. ال Structure: تقسيم المشكلة إلى أجزاء أصغر (مكونات) وكيفية تفاعل هذه الأجزاء مع بعضها.

٢. الرؤية -Vision-: وجود خطة واضحة ومُدروسة لكيفية بناء الحل مع الأخذ في الاعتبار القيود والتحديات الواقعية.

ولنخرج من الكلام النظري، لنتحدث عن مثال عملي، تخيل أننا نريد أن نعمل على infrastructure architecture لإنشاء شبكة بين مكتبين يقعان في منطقتين جغرافيتين مختلفتين في نفس البلد، فما هي الحلول المعمارية الممكنة لنا؟
أحد هذه الحلول هو استخدام Cable واحد طويل يصل بين المكتبين *ـ*، وهذا حل معماري، ومن الناحية النظرية يمكن أن يعمل فعلا، لكنه إذا ربط في التحديات الواقعية فإن هذا الحل غير قابل للتنفيذ، أو على الأقل هو حل يصعب تحقيقه! والسبب يعود في ذلك

لأن الواقع يفيد بوجود عدة تحديات منها: وجود عقبات مثل الأنهار أو البحيرات أو الطرق أو القوانين التي تمنع مثل هذا العمل... إلخ
بينما في الحل الواقعي نحن ندرك أننا بحاجة للتعامل مع الكثير من ال building blocks الشائعة والتي تتعاون فيما بينها لتحقيق هذا الهدف، فمثلا سنحتاج إلى router و switches وال firewalls وأن ننضم للشبكة في المكتبين... وبذلك سنتمكن من التواصل سويا... هذا الحل وإن كان أكثر تعقيدا، إلا أنه الحل القابل للتنفيذ واقعيا!

لذلك، خذ هذه القاعدة المهمة: بغض النظر عن النظام الذي ستبني معمارية له، فإن الحل الناجح سيتطلب منك فهما للمشكلة ثم خلق رؤية واضحة يمكن إيصالها لكل من يشارك في بناء الـ end-product...

ال Application architecture

معمارية التطبيقات هي الجزء الذي يركز على بناء وتصميم وبناء التطبيق نفسه، بحيث يمثل وحدة قابلة للنشر، ومكتوب بتقنية محددة، وهذا النوع هو الأكثر شيوعا عند المطورين لأنه يتعلق مباشرة بالشيفرة البرمجية.

يمكن أن يكون التطبيق عبارة عن أي وحدة قابلة للنشر، مثل:

- تطبيق ويب SPA مصمم ب React.
- تطبيق جوال على iOS أو Android.

- تطبيق سطح مكتب مكتوب بتقنية .NET.

في هذا النوع من المعماريات يتم التركيز على التصميم الداخلي للتطبيق، وهذا يشمل كيفية تقسيم الشيفرة البرمجية إلى أجزاء صغيرة ومنظمة مثل:

- ال Components: أجزاء يمكن إعادة استخدامها في أكثر من مكان
- ال Layers: ترتيب الكود في طبقات وظيفية (مثل طبقة واجهة المستخدم، ومنطق الأعمال، والوصول إلى البيانات)
- ال namespaces
- ال packages

كما أنها تهتم بفهم ال Patterns وال Frameworks وال Libraries التي سيتم استخدامها في بناء التطبيق.

باختصار، يمكن اعتبار معمارية التطبيقات بمثابة "معمارية الشيفرة البرمجية"، لأنها تتعلق بشكل أساسي بتنظيم الشيفرة البرمجية لضمان أن يكون فعالا ومفهوما وقابلا للصيانة.

ال System architecture

تمثل معمارية النظم -System Architecture- مستوى -طبقة- أعلى من معمارية التطبيقات، فبينما تركز معمارية التطبيقات على تصميم وبناء تطبيق واحد (أو وحدة واحدة

قابلة للنشر)، تهتم معمارية النظم بتصميم نظام برمجي كامل يتكون من عدة تطبيقات ووحدات مختلفة (مثل: تطبيق هاتف، تطبيق ويب، وقاعدة بيانات)، ثم النظر في كيفية عملها وتكاملها معا، كما أن هذا النوع يأخذ في الاعتبار التكامل والتوافقية مع الأنظمة الأخرى، ويركز على فهم العلاقة بين البرمجيات وال Hardware، فال Hardware أيضا تفرض قيودا وتحديات يجب أخذها في الاعتبار عند تصميم النظام.

ولتبسيط الفكرة، تخيل أن لديك نظام برمجي مكون من تطبيق أندرويد يتواصل عبر API مع ال PHP Backend Server، وهذا ال PHP يقوم بالتواصل مع قاعدة بيانات من نوع MySql... كما تلاحظ فإن كل واحدة ذكرناها تمثل بذاتها وحدة مستقلة -Mobile App و Web App و database- مبنية باستخدام تقنية مختلفة، وبذلك لكل واحدة منها معماريتها الخاصة داخليا... لكن [٨٨]، حتى يعمل النظام الخاص بنا بشكل صحيح؛ يجب التفكير في كيفية جمع جميع هذه الوحدات القابلة للنشر معا، والنظر في البنية العامة كاملة وكيف ستتواصل فيما بينها...

إذا يمكن إيجاز الفرق بين معمارية التطبيقات ومعمارية النظم بما يلي:
معمارية التطبيقات تركز في المقام الأول على الشيفرة البرمجية نفسها، لذلك تجدها تهتم بال Frameworks وال Programming language وال Libraries... في حين تركز معمارية النظم على كل من الشيفرة البرمجية والتكامل والتوافق مع التطبيقات / الوحدات المختلفة، وال Hardware، وبذلك تمثل معمارية التطبيقات عمارة سكنية واحدة في حين تمثل معمارية النظم المدينة بأكملها [٨٨].

ملاحظة ١: قد تتسائل لماذا تهتم معمارية النظم بال Hardware مع أنها في كثير من الأحيان ذات تفاصيل مخفية أو لا ننظر إليها أو نحن غير مسؤولين عنها... إلخ، والجواب أن النظر إلى ال Hardware أمر مهم جدا، وذلك لعدة أسباب منها:

- أنك تريد ضمان توافقية الأدوات التي استخدمتها مع ال hardware الموجود.
- أنك ستجد بعض الشروط اللازمة من لغات البرمجة أو الأدوات أو قواعد البيانات التي تتطلب حدا أدنى من الموارد التي ينبغي عليك توفيرها...
- أنك قد تحتاج لمعرفة القدرة القصوى لتحمل ال hardware الحالي عند وجود ضغط عالي من التطبيق والذي يدعوك عند نقطة معينة لتحسين هذا ال hardware والارتقاء به...

ملاحظة ٢: هناك خدمات يمكنك استخدامها تبسط عملية تطوير البرمجيات وتخفي هذه التفاصيل وغيرها من البنية التحتية عنك وتجعلك تركز على كتابة الشيفرة البرمجية، ومع أنك في هذه الحالة لن تحتاج نظريا إلى النظر لل Hardware مثل نظرتك في حال كنت تقوم بذلك بيدك، إلا أنه ما يزال من الضروري لك أن تفهم القيود التي تفرضها هذه الخدمات... مثلا: من الخدمات التي يمكن استخدامها ال FaaS وال PaaS، إذا كانت هناك قيود زمنية لتنفيذ مهمة ما فعليك أن تعرفها، وإذا هناك زمن استجابة عالي بسبب بعد المسافة المكانية فعليك أن تعرف ذلك، وإذا كان هناك قيود على حجم الملفات فعليك أن تعرف ذلك... إلخ.

معلومة ١: ال PaaS هي اختصار ل Platform as a Service، وهي خدمة تقدم بيئة كاملة لتطوير ونشر التطبيقات، والفكرة من هذه الخدمة هي أن يركز المطورون فقط على كتابة

الشيفرة البرمجية في حين تتولى المنصة إدارة جميع الجوانب المتعلقة بالبنية التحتية مثل ال Servers وال OS وقواعد البيانات والتعامل مع الزيادات في عدد المستخدمين... إلخ، ومن أشهر الأمثلة على ذلك: Herkou.

معلومة ٢: ال FaaS هي اختصار ل Function as a Service، وهي خدمة تستخدم لنشر وظائف فردية -مهمة معينة- أو حتى لتنفيذ جزء صغير من الشيفرة البرمجية يصل لأن يكون function واحد [٨]، ويتم تشغيل هذه الوظيفة واستخدامها عند وجود حاجة إليها، وعادة ما يتم الدفع مقابل وقت التنفيذ الفعلي، ولعل من أشهر هذه الأمثلة على ذلك: AWS Lambda، ومن الاستخدامات الشائعة لهذا الأسلوب استخدام ال FaaS للقيام بالمهام المجدولة مثل إرسال تقارير يومية بالبريد الإلكتروني والقيام ببعض عمليات المعالجة على قواعد البيانات مثل تنظيف قاعدة البيانات من السجلات القديمة... ومن الأمثلة الشائعة أيضا استخدامها لمعالجة الصور والفيديوهات وإنشاء ملفات ال PDF... إلخ

ال Software architecture

على النقيض من ال Application architecture وال System architecture اللذان يُفهمان نسبيًا بشكل أفضل وأسرع، فإن مصطلح ال Software Architecture ليس كذلك، وهو أحد الأنواع ذات التعريفات المتعددة والمتنوعة عبر الإنترنت، لذلك، وبدلًا من التورط في التعقيدات والفروقات الدقيقة للعديد من تعريفات ال Software Architecture سنحاول تعريفه بشكل مبسط ووافي بنفس الوقت [٨].

يمكن القول إن ال Software architecture هو مزيج بين معمارية التطبيقات ومعمارية النظم، أي أنها تشمل كل ما يتعلق بالنظام البرمجي ككل، بدأ من أدق التفاصيل في بنية الشيفرة البرمجية حتى كيفية نشره على البنية التحتية المخصصة لذلك، وبذلك في تشمل كيفية تنظيم الشيفرة البرمجية وجودتها وما يتعلق فيها من أمان وأداء وقابلية للتوسع، والقيود الواقعية مثل متطلبات الأجهزة والبنية التحتية، وعادة ما يركز المطورين على الشيفرة البرمجية نفسها وطريقة بنائها، لكن من الذي سيركز على كيفية البناء وشكل البناء؟!، من سيركز على ما يلي:

- ال Cross-cutting concerns بما في ذلك ال Logging وال Exception handling...
- الأمان، خصوصا ما يتعلق سرية البيانات الحساسة...
- الأداء والقدرة على التوسع وال availability...
- التناسق الهيكلي بحيث نضمن أن جميع الأجزاء تعمل بانسجام...
- القيود الواقعية الخاصة بال environment...
- متطلبات ال operations والدعم والصيانة...
- اتساق النهج في حل المشكلات وتنفيذ ال Features عبر ال Codebase، أي وضع معايير يلتزم فيها فريق التطوير بأسلوب موحد وثابت في كتابة الشيفرة البرمجية وحل المشكلات في جميع أجزاء المشروع... ولعل من أشهر الأمثلة على عدم الاتساق -نهج سيء- معالجة الأخطاء في الشيفرة البرمجية الواحدة بطرق مختلفة في أماكن مختلفة، لذلك هنا نهتم بوجود نسق موحد لتحقيق أفضل جودة...

حتى تتخيل هذا المفهوم بشكل صحيح، عليك أن تبتعد قليلا عن ال Code وأدوات ال Development الخاصة بك، وأن تنظر نظرة شمولية للنظام ككل...

والآن حتى لا نفقد البوصلة بين هذه المصطلحات نخلص النتائج بما يلي:

- معماري التطبيقات يركز على بناء وتصميم الشيفرة البرمجية داخل التطبيق الواحد.
- معماري النظم يركز على ربط التطبيق مع التطبيقات الأخرى، والبنية التحتية لتحقيق التكامل والتوافق فيما بينها.
- معماري البرمجيات يلعب الدورين معا، فهو لا يكتفي بالنظر إلى كيفية تواصل الأنظمة مع بعضها البعض، بل يضمن أيضا أن الهيكل الداخلي لكل تطبيق في النظام يتوافق مع الأهداف العامة للنظام بأكمله.

ال Enterprise architecture – الاستراتيجية بدلًا من ال Code

إن معمارية المؤسسات تشير عموما إلى عملية تحليل وتنظيم هيكل المؤسسة وعملياتها وتقنياتها بهدف تنسيقها ومواءمتها مع أهدافها واستراتيجيتها، وهذا يشمل تنظيم وإدارة الأشخاص والأقسام، وال Process المناسبة حتى تعمل هذه العمليات المختلفة معا بكفاءة أكبر، والتقنيات التي يجب أن تستخدمها المؤسسة على المدى الطويل...

إن التركيز الرئيسي في هذه النوع من المعماريات ينصب على وضع خطط استراتيجية واسعة المدى لكيفية استخدام التقنيات المختلفة على مستوى المؤسسة، كما أنها تغطي مجموعة واسعة

من الجوانب مثل الهيكل التنظيمي للمؤسسة والعمليات التجارية والأنظمة التقنية، لكنها لا تتعمق في التفاصيل التقنية، لذلك تجد أن هذا النوع شمولي ويتعلق بكيفية عمل الأقسام المختلفة معا، وتنسيق جهودها لتحسين الكفاءة العامة... ومن هنا يظهر لنا الفرق بين معماري البرمجيات ومعماري المؤسسات.

ملاحظة: يرى بعض المطورين و المعماريين أن ال Enterprise architecture هي الخطوة المنطقية التالية في السلم الوظيفي، لكن عادة لا يحاول هؤلاء الارتقاء إلى هذه الوظيفة، والسبب في ذلك يعود إلى أن العقلية المطلوبة للقيام بال Enterprise architecture تختلف كثيرا عن الـ Software architecture، حيث تتبنى منظورا مختلفا تماما تجاه التكنولوجيا واستخدامها عبر المؤسسة... فال Enterprise architecture تتطلب مستوى أعلى من ال Abstraction، وتتطلب الاهتمام بالاستراتيجية أكثر من الـ Code، وتتطلب الاهتمام بال Breadth أكثر من Depth، ويقصد بهذا الاهتمام بفهم مجموعة واسعة من العمليات الخاصة بالمؤسسة بدلا من التعمق في تفاصيل نظام محدد.

ال Architecture vs design

كثيرا ما يحدث لغط بين مفهوم ال design ومفهوم ال architecture، وكثير من الناس يستخدم كلمة Design لوصف ال architecture... فهل فعلا أن هذا الاستعمال صحيح؟ أم هناك تفصيلات معينة أو فروق دقيقة بين المفهومين؟

حتى نصل للمعنى الدقيق دعونا نعرف هذه المفاهيم أولا:
يمثل الـ Design عملية اتخاذ القرارات لاختيار حل واحد من بين العديد من الحلول الممكنة لمشكلة ما، لأن كل مشكلة لها "مساحة قرارات محتملة" مليئة بالخيارات المتاحة، وعملية التصميم هي عملية تضييق هذه الخيارات لتحديد الأنسب، فعلى سبيل المثال، لدينا عشرات الطرق لإنشاء تطبيق هاتف محمول أو تطبيق ويب، لكننا تختار طريقة واحدة من بين هذه الطرق لبناء التطبيق الخاص بنا حسب السياق الذي نحن فيه، وبذلك فإن ال Design يمثل جميع القرارات التي يتم اتخاذها في عملية بناء أي نظام برمجي سواء كانت كبيرة أو صغيرة.

أما الـ Architecture فهي تمثل القرارات التصميمية المهمة التي يصعب تغييرها في المستقبل دون بذل جهد كبير، بحيث تقاس "أهمية" القرار بـ"تكلفة التغيير"، وهي بذلك تمثل مجموعة فرعية فقط من ال Design، وتحديدا تلك التي يصعب تغييرها بعد اتخاذها.

بناء على هذه التعريفات التي طرحناها، يمكننا القول أن كل Architecture هو Design، لكن ليس كل الـ Design هو Architecture.

بعيدا عن التعريفات هذه، يمكننا التفريق بين ال Design وال Architecture من خلال طرح السؤالين التاليين:
١. كيف سنبني هذا النظام؟
٢. ما هي المكونات الأساسية لهذا النظام؟

السؤال الأول إجابته تمثل Design، في حين تمثل إجابة السؤال الثاني Architecture ^^.
والآن لنأخذ أمثلة عملية على ما ذكرناه:
إذا أردنا أن نصنع سيارة، فإن ال Design سيشمل كل ما يتعلق بتصميم المحرك وشكل مقابض الأبواب ولون المقاعد ولون السيارة ونحو ذلك، في حين أن ال Architecture سيمثل هل السيارة كهربائية أم بنزين؟ دفع رباعي أم سيارة عائلية؟

وكذلك إذا أردنا بناء منزل، فلون الجدران وشكل الأبواب ونوع أرضية البيت تعد قرارات تصميمة، في حين أن المنزل هل سيكون من طابق واحد أو أكثر أو هل سيتم بناءه بالاسمنت أم الخشب فهذه قرارات معمارية...

وعلى نفس الطريقة في عالم البرمجة، فإن اختيار المعمارية الخاصة بمشروعنا لتكون Monolith أو Microservice هو قرار Architecture، في حين أن استخدام Pattern معين أو تسمية المتغيرات أو هيكلة الشيفرة البرمجية داخل Class ما هو قرار Design...

إن التفرقة بين المصطلحين مفيد لفهم من يجب أن يتخذ أي قرار، فالقرارات المعمارية يتخذها عادة ال Architects لأن عواقبها بعيدة المدى، أما القرارات التصميمية التفصيلية فيتخذها المطورون الذين يعملون على تلك الأجزاء من الشيفرة البرمجية، إن الفهم الصحيح لهذا الفرق يجعل من عملية تطوير البرمجيات أكثر كفاءة وأقل عرضة للمخاطر.

ولجعل الأمور أسهل، انظر إلى هذه القائمة التي ستساعدك في تحديد ال Architecture إذا رأيتها -قائمة على سبيل السرد لا الحصر-:

- الشكل العام لل Software system مثل:
  - Client-server
  - Web-based
  - Native mobile
  - Distributed
  - Microservices vs Monolith
  - Asynchronous vs Synchronous... إلخ.
- ال Structure لل Code داخل الأجزاء المختلفة من ال Software system.
- اختيار التقنيات المختلفة مثل لغة البرمجة المستخدمة وال development platform... إلخ.
- اختيار ال Frameworks.
- اختيار نهج التصميم المناسب للنظام مثل اتباع نهج نحتاج فيه القابلية للتوسع وال Availability وتحسين الأداء من خلال ال Caching وتأثيره على استرجاع البيانات...

إن القرارات المعمارية التي تتخذها لن تتمكن من التراجع عنها دون بذل جهد كبير، و سيكون من الصعب عليك إعادة هيكلتها في فترة قصيرة... لذلك يجب أن تحرص على النظر بطريقة شمولية وصحيحة قبل اتخاذ القرار، وبناء النماذج التي ستساعدك على اتخاذ القرار...

والسير بطريقة منهجية للوصول إلى النتيجة المرجوة... وسنتحدث عن ذلك لاحقا -بإذن الله-، لكن هذا يقودنا لسؤال آخر مهم، كيف يمكننا معرفة أن هذا القرار الذي سنتخذه هو قرار معماري مهم وليس مجرد قرار تصميي عادي؟؟

هنا نعود لنقطة البداية، وهي ضبابية الخط الفاصل بين القرارات التصميمية والمعمارية، لكن يمكن القول أن القرارات المعمارية عادة ما تكون:

- ذات تكلفة عالية عند التغيير، ولا يمكن التراجع عنها بسهولة
- مرتبطة بالخيارات التقنية الرئيسية مثل لغات البرمجة وال frameworks المستخدمة والنهج المستخدم لبناء النظام Microservice أم غير ذلك...
- مرتبطة بالهيكل العام للنظام مثل تقسيم النظام إلى طبقات أو خدمات.

أما القرارات المتعلقة في تنسيق الشيفرة البرمجية ونحو ذلك، فهي ليست قرارات معمارية نهائيا... لذلك، ليزول الضباب عن الخط الفاصل علينا فهم ما هو مهم لنا ولماذا في سياق المشروع.

مثال عملي آخر لنتخيل الفكرة وأثر القرارات المعمارية، تخيل أننا قمنا ببناء تطبيق ويب، وسيتم التواصل مع قاعدة بيانات من نوع MySql، وسنقوم باستخدام ال ORM ك Abstraction Layer للتواصل مع قاعدة البيانات...

حسب هذا السياق لدينا قراران هنا، وهما:

1. القرار الأول هو استخدام ال Mysql كقاعدة بيانات
2. القرار الثاني استخدام ال ORM ك Abstraction Layer

إن القرار الثاني فعليا هو ما يمثل قرار معماري حقيقي هنا! والسبب في ذلك يعود لأن إضافة ال Abstraction Layer هو القرار المعماري المهم، والذي من خلاله سيتم التواصل مع قاعدة البيانات، وسيكون من السهل استبدال قاعدة البيانات بواحدة أخرى، لكن سيكون من الصعب إزالة ال ORM! وبذلك، فإن اختيار قاعدة البيانات لم يعد قرارا معماريا مهما في هذه الحالة...

هذا المثال يهدف لتوضيح الفكرة، لكن في الواقع قد تقول أيضا أن Abstraction Layer التي قمنا بإضافتها ليست مهمة أيضا لأننا يمكننا أن نضيف Layer أخرى لفصل هذه الطبقة عن ال Business logic، لكنك هنا أيضا اتخذت قرارا معماريا آخر! ويمكنك التفكير في هذا إلى ما لا نهاية... إن العبرة الحقيقة هي في تحديد ما هو مهم بالنسبة لنا ضمن سياق المشروع -فما هو مهم بالنسبة لي في هذا المشروع قد يكون غير مهما في مشروع آخر-؛ فإن ضمنت ذلك استطعت الوصول لقرارات ناضجة ومفيـدة بإذن الله.

هل ال Software Architecture مهمة فعلا؟

الجواب باختصار، طبعا مهمة دون أدنى شك ^^... قد تتسائل لماذا ولدينا تقدم كبير في المنهجيات والأساليب المستخدمة لكتابة الشيفرة البرمجية؟ لماذا ونحن لدينا DevOps و Agile و cloud service و CI ونحو ذلك؟

والجواب يكمن في النقطة الأساسية لما كتبناه من كلامك حتى هذه اللحظة، المشاريع التقنية ليست مجرد شيفرة برمجية! والمشاريع الناجحة ليست مجرد شيفرة برمجية جيدة! إن الاهتمام بمعمارية المشروع يضع الأساس القوي والمتين لما سنقوم ببناءه، مما يضمن أن جميع الأعضاء

سيعملون نحو هدف مشترك وبطريقة متسقة، وسنضمن حينها أن هناك جودة لما يتم كتابته وبناءه، وسنحد من الفوضى التي يمكن أن تحدث من دون معمارية واضحة، كما سنتجنب مشاكل ال Code Complexity والذي عادة ما يتحول ل Spaghetti *-*... كما سنتجنب المشاكل المتعلقة بإهمال الجوانب التقنية المهمة المتعلقة بالتوسع أو الأداء ونحو ذلك... تخيل أن تقوم ببناء نظام يجب أن يخدم ملايين المستخدمين ثم لا يخدم هذا النظام أكثر من عشرات الآلاف!

والآن، انظر لأحد المشاريع التي لديك، واسأل نفسك هذه الأسئلة:

- هل لدى ال Software system الخاص بك Structure واضح ومحدد؟
- هل يقوم جميع أعضاء الفريق بتنفيذ ال Features بطريقة متسقة؟
- هل هناك مستوى متسق من Quality عبر ال Codebase؟
- هل يشترك أعضاء الفريق في نفس الرؤية لكيفية بناء البرمجيات؟
- هل يمتلك جميع أعضاء الفريق القدر اللازم من التوجيه التقني؟
- هل هناك قدر مناسب من القيادة التقنية؟

قد تكون بعض إجابات هذه الأسئلة "لا"، لكنه مشروع ناجح، وذلك عادة ما يعود لوجود فريق تقني ممتاز وحظ كبير ^^... لكن الحقيقة المرة أن كل "لا" تزداد تعني انحراف أكبر عن المسار الصحيح...

ملاحظة: ال القيادة التقنية -Technical leadership- هي الدور الذي يقوم به شخص أو مجموعة من الأشخاص داخل فريق التطوير لتوجيه القرارات التقنية وتوفير الدعم الفني المناسب، ولا تقتصر مهام هذه القيادة على إصدار الأوامر! بل تركز على مساعدة الفريق في بناء منتج عالي الجودة بشكل فعال، ومن المسؤوليات المناطة على من يقوم بهذا الدور وضع الرؤية التقنية للمنتج والتي يسعى الجميع لتحقيقها، واتخاذ القرارات الصعبة التي تؤثر على قرار المشروع مثل اختيار التقنيات المناسبة أو نمط التصميم، وتوجيه الفرق وتقديم الدعم لهم، وضمان جودة ما تمت كتابته وأنه فعلا يتبع أفضل الممارسات والمعايير التي تم الاتفاق عليها، وبناء جسر للتواصل بين الفريق التقني والمسؤولين أو العملاء لشرح القرارات التقنية بلغة بسيطة وواضحة، وتدريب الفريق وإرشادهم ونشر المعرفة بينهم... هذا الدور عادة ما يندرج تحت المسمى الوظيفي Technical Team Lead أو Lead Software Engineer أو ال Principal Engineer أو Chief Architect... إلخ، ومن أهم السمات التي على القائد التقني امتلاكها الكفاءة التقنية ومهارات القيادة والتوجيه والتواصل.

إن تطبيق معمارية جيدة على مشروع ما سيقدم لنا العديد من الفوائد، نذكر منها:

1. رؤية واضحة وخارطة طريق يمكن للفريق من تتبعها والعمل عليها، سواء تم وضع هذه الرؤية من قبل شخص واحد أو من قبل الفريق بأكمله.
2. قيادة وتنسيق أفضل: توفر قيادة تقنية قوية وتسهل التنسيق بين أعضاء الفريق.
3. تشجيع التواصل: يحفز وجود معمارية واضحة على طرح الأسئلة ومناقشة القرارات الهامة والجوانب التقنية الحرجة مع جميع الأطراف المعنية، كما أن هذا يسمح بالتواصل

الفعال على مستويات مختلفة ومع جماهير مختلفة (تقنين وغيرهم)، مما يفتح الآفاق والرؤى بشكل أكبر وأجمل وأدق.

4. إدارة المخاطر: توفر إطارا لتحديد المخاطر التقنية ومعالجتها قبل أن تصبح مشكلات كبيرة.
5. اتساق المعايير: هناك ضمان ممتاز على أن الشيرة البرمجية ستكون بطريقة موحدة ومنظمة، وهذا سيؤدي إلى شيفرة برمجية أسهل في الصيانة على مستوى المشروع.
6. بناء أسس متينة: تضع حجر الأساس للمنتج، مما يضمن أن يكون قويا وقابلا للتطوير في المستقبل.

هل كل مشروع برمجي يحتاج إلى Software architecture؟

يقول Simon «إن الإجابة الشائعة على هذا السؤال هي "It depends"، لكن الإجابة بلا شك هي نعم».

إن السبب وراء هذه الإجابة ل Simon: أنه لا يوجد مشروع برمجي لا يحتاج إلى معمارية برمجية، ومع ذلك، فإن الاختلاف يقع في مقدار التفكير المطلوب في هذه المعمارية من مشروع لآخر... هذا التفكير يمكن أن يظهر جزء منه على شكل تصميم مسبق (Up front design)، ولتحديد مقدار الجهد المناسب لهذا التصميم يجب على الفريق أن يأخذ في الاعتبار عدة عوامل وهي:

- حجم المشروع: فكلما زاد حجم المشروع زادت الحاجة لوجود هيكلية واضحة لتجنب الفوضى وصعوبة الصيانة، لذلك نحتاج لقدر أكبر من التصميم المسبق، في حين أن المشاريع الصغيرة قد نكتفي فيها بتصميم مبسط قليل التعقيد سهل التغيير.
- تعقيد المشروع: كلما زاد تعقيد المشروع زادت الحاجة لتغطية قضايا أكبر، وعليه زادت الحاجة لوجود تصميم مسبق.
- حجم الفريق: إذا كان حجم الفريق صغيرا فيمكن الاكتفاء بتصميم بسيط لأن كيفية التواصل فيما بينهم سهلة وسريعة، في حين أن الفرق الكبيرة خصوصا التي تزيد عن ثمانية أو عشرة أشخاص تحتاج لتصميم مسبق لتوحيد الرؤية والنسق فيما بينهم، وإلا وقعنا في فوضى عارمة.
- خبرة الفريق: يمكن للفريق ذو الخبرة العالية أن يبدأ بتصميم بسيط لأن لديهم القدرة على اتخاذ القرارات الصحية أثناء التنفيذ، في حين أن الأفرقة ذات المستوى المبتدئ والمتوسط تحتاج إلى دليل إرشادي واضح وقيادة تقنية توجههم أثناء العمل... مما يعني حاجة أكبر لتصميم مسبق في هذه الحالة.

يقول Dave Thomas: «التصميم الكبير في البداية شيء غبي، لكن عدم القيام بأي تصميم في البداية شيء أغبى» ^^... لو نظرنا لواقعنا الآن مع تطور أدوات الذكاء الاصطناعي، لوجدنا أن موضوع معمارية المشروع والتصميم المسبق قبل كتابة أي سطر برمجي أصبحت ترفا فكريا لدى الكثيرين -للأسف-! وهذه كانت بالأساس مصيبة موجودة ومنتشرة لكنها تضاعفت في الأعوام الأخيرة، ومع ذلك، فإن الإفراط في التصميم المسبق يعد أيضا أمرا غبيا يجب تجنبه والابتعاد عنه قدر الإمكان.

إن ما نريده ونسعى إليه أن نكون وسطا، وأن نجد نقطة التوازن بين الإفراط والتفريط، وهذه واحدة من أهم مواضيع الكتاب والتي سنتعامل معها طوال رحلتنا هذه... وفي سياق هذه النقطة: لا يمكن إلا وأن نتذكر قوله تعالى في كتابه العزيز: "وَكَذَٰلِكَ جَعَلْنَاكُمْ أُمَّةً وَسَطًا"، فسبحان من أعزنا بالإسلام، وجعله لنا نورا وضياءا نهتدي به...

فائدة

الإنسان إنما بعث لغاية عظيمة، وجب عليه العمل لأجلها، فمن جاء لينتزع هذه الغاية العظيمة من نفوس أصحابها وجب علينا محاربته وعدم الرضوخ له وعدم الميل للنفس وهواها!

- كتاب إلى الجنة زمرا، الصفحة ٥١٥

ال Architectural drivers

بعد أن تطرقنا إلى بعض المفاهيم المهمة وتعرفنا على المفاهيم العامة حول ال Architectural، يجب علينا أن ننتقل إلى ما يجب علينا التفكير فيه عند القيام بالعمل على أي معمارية لأي نظام برمجي وبغض النظر عن ال Process المستخدمة في عمليات تطوير التطبيقات، أكانت Traditional and Plan-driven أم كانت Lightweight and Adaptive...
باختصار إن أهمية التفكير في ال Architectural لا تتغير سواء كنت تعمل على مشروع يخطط لكل شيء مسبقًا أو مشروع يتم تطويره بشكل مرن ويتكيف مع التغييرات.

ومن هنا ننطلق إلى مفهوم ال Architectural drivers، وهو يشير إلى العوامل التي تؤثر على القرارات المتخذة أثناء تصميم المعمارية، وبذلك فهي تحدد شكل ال Architectural للنظام الخاص بنا... أو بطريقة أخرى هي مجموعة القوى أو المحفزات التي تدفع أو تؤثر على عملية بناء النظام وتحديد شكله النهائي، أو الأسباب التي تجعلك تختار نمطا معينا من التصميم أو تقنية ما بعينها دون غيرها... ومن الأمثلة على هذه المحفزات الأداء، مثل أن النظام يجب أن يستجيب لطلبات المستخدمين خلال أقل من ٥٠٠ جزء ثانية أو ال Availability يجب أن تكون ٩٩.٩٩٪ أو القابلية للتوسع يجب أن تكون متوفرة وتدعم عدد كبير من المستخدمين ونحو ذلك، وبهذا تكون الترجمة المناسبة لهذا المفهوم -من وجهة نظري- هي: "المحفزات المعمارية" أو "العوامل المؤثرة على المعمارية" أو "العوامل المُحرِّكة للقرارات المعمارية".

ملاحظة ١: ال Traditional and Plan-driven يمثل إحدى المنهجيات المتبعة في تطوير البرمجيات، وتعرف أيضا بالمنهجيات الشلالية -Waterfall- أو التتابعية -sequential-، وهي تعتمد من اسمها على التخطيط الدقيق والمسبق لكل مرحلة من مراحل التطوير، مثل جمع المتطلبات الخاصة بالنظام ثم تصميم النظام ثم تنفيذ النظام ثم اختبار النظام ثم صيانة النظام... إلخ، وهناك أكثر من أسلوب يمكن استخدامه داخل هذا النهج، فال Waterfall المشهور هو أحد هذه الطرق، وهناك طرق أو نماذج أخرى مثل ال V-model أو ال Spiral Model...إلخ

ملاحظة ٢: ال Lightweight & Adaptive Methodologies تمثل إحدى المنهجيات المتبعة في تطوير البرمجيات، وتعرف أيضا بال Agile، وتتميز بالمرونة والتكرار طوال مراحل التطوير، فبدلا من التخطيط الكامل مقدما، يتم تقسيم العمل إلى Sprints يسمح بالتغييرات في أي مرحلة من مراحل المشروع، ومن الأمثلة عليها ال Scrum وال Kanban.

والآن بعد هذه المقدمة الشيقة، لننطلق إلى المحفزات المعمارية ^^:

١. ال Functional requirements:

تعتبر ال Functional requirement واحدة من أهم المحفزات وحجر الزاوية في أي معمارية لأي نظام! فهي تمثل الأساس الذي يجب أن يبنى عليه كل شيء، وبهذا فهي تمثل ما يجب على النظام أن يفعله من وجهة نظر المستخدم النهائي والأهداف التي يجب أن يحققها

هذا النظام ^^... وقد تتساءل الآن: أليس هذا السلوك بديهي لدينا؟ والحقيقة نعم! إن السلوك الطبيعي لدينا هو فهم ما ينبغي علينا تقديمه للمستخدمين وما هي حاجتهم أو دافعهم لاستخدام النظام الخاص بنا... لكن، مع أن هذا يبدو بديهيا لنا إلا أن هناك كثير من الفرق التقنية تبدأ بتصميم وبناء البرمجيات من دون فهم واضح حتى للميزات الأساسية أو ال User Stories أو أنواع المستخدمين في النظام، وهذا يعد نوعا من أنواع الحماقة في العمل؛ حتى لو أطلق البعض عليها اسم Agile! لأن ال Agile لا يعني أن تبدأ دون أن تفهم، بل أن تبدأ مع وجود فهم عالي المستوى على الأقل للمزايا الأساسية ونحو ذلك من المكونات المهمة... إن الفرق الأساسي هنا هو بين "عدم وجود خطة ثابتة" و "عدم وجود خطة على الإطلاق" ^^

٢. ال Quality Attributes (non-functional requirements)

إن لكل نظام مجموعة من السمات التي تقاس من خلالها جودة النظام، ومن دون هذه السمات قد يعمل النظام بشكل صحيح من الناحية الوظيفية، لكنه غير آمن وبطيء أو صعب الصيانة؛ مما يجعل هذا النظام غير صالح للاستخدام على المدى الطويل، وهذا بكل تأكيد ما لا نريده! لذلك، تعد "سمات الجودة -Quality Attributes-" من أهم المحفزات المعمارية...

إن العديد من هذه السمات يُشار إليها غالبًا بمصطلح "-ilities"، أي الكلمات التي تنتهي ب: "ibility" مثل Scalability و Reliability و Maintainability... إلخ، والتي يمكن ترجمتها للعربية إلى "قابلية" أو "إمكانية" مثل "قابلية التوسع" أو "قابلية الصيانة"... وبهذا قد تتسائل:

لماذا كان عنوان هذا الجزء non-functional requirements -المتطلبات غير الوظيفية- فكيف ستؤثر على النظام أو لماذا ينبغي علينا أخذها بعين الاعتبار في أثناء تصميمنا معماريتنا؟!

وهذا سؤال ذكي جدا، ويشير إلى انتباه عال...

حتى نكون متفقين، فإن كثيرا من الخبراء يفضلون استخدام مصطلحات مثل "سمات الجودة -Quality Attributes-" أو "خصائص النظام -system characteristics-" بدلا عن "المتطلبات غير الوظيفية -non-functional requirements-"، والسبب في هذا يعود إلى التساؤل المنطقي الذي قمت بطرحه؛ فمصطلح متطلبات غير وظيفية تحمل دلالة سلبية قد يفهم من خلالها أن هذه المتطلبات ليست مهمة أو ليست أساسية أو لا تؤدي وظائف أساسية في النظام! وهذا أمر خاطئ تماما، بل على النقيض من ذلك، فهذه المتطلبات حيوية ومهمة لنجاح أي نظام، لذلك تعتبر المصطلحات الأخرى أكثر دقة، فهي تصف ما يمتلكه النظام من خصائص مثل الأمان والأداء، وأكثر إيجابية فهي تؤكد على أهمية هذه الخصائص في تحديد جودة النظام وقيمته النهائية... وبهذا نكون أجبنا عن شطر من السؤال، وننتقل إلى الشطر الثاني منه ^^

تُعد سمات الجودة من المتطلبات التقنية في ذاتها، ولها تأثير كبير على البنية التحتية النهائية للنظام، فعلى سبيل المثال: إذا كان هدفنا هو بناء نظام ذا أداء عال أو أمان قوي، فإن القرارات المعمارية الرئيسية مثل اختيار التكنولوجيا، وهيكل الشيفرة البرمجية يجب أن تأخذ في الاعتبار منذ البداية؛ لأن إهمالها يعد خطأ فادحا بسبب صعوبة إضافتها إلى نظام موجود

بشكل مسبق، كما أن كلفة هذا التغيير مرتفعة... باختصار هذه السمات هي جزء أساسي من هيكل النظام الكلي وليست مجرد Feature يتم إضافتها ببساطة ^^.

والآن، لنذهب سويا لنستعرض أهم سمات الجودة الشائعة...

١. ال Performance:

إن الاهتمام بالأداء أمر ضروري لأي نظام، وبغض النظر عن نوعه أو حجمه! فإذا كان المستخدمون يشتكون من أن برنامجك بطيء = فهذا يعني أنك قد أهملت أحد أهم الجوانب في مشروعك والذي قد يكلفك الكثير، وستدرك حينها أهمية الأداء ووجوده!

إن الأداء عادة ما يشير أو يتعلق بسرعة النظام، وغالبا ما يقاس الأداء بمقاييسين رئيسيين وهما:

أ. ال Response time: وهو إجمالي الوقت الذي يستغرقه النظام للاستجابة لطلب المستخدم، ويبدأ القياس من لحظة قيام المستخدم بإجراء ما -مثل النقر على زر موجود بالصفحة- وينتهي عند حصول هذا المستخدم على النتيجة التي يريدها...
على سبيل المثال: إذا نقرت على رابط في موقع إلكتروني للدخول إلى صفحة ما، فإن زمن الاستجابة هو الوقت الذي يستغرقه الموقع حتى يتم عرض هذه الصفحة مكتملة ^^.

ب. ال Latency: وهو الوقت الذي تستغرقه البيانات أو الرسائل للانتقال من نقطة إلى أخرى داخل النظام، فإذا نقر المستخدم على رابط في موقع إلكتروني للدخول إلى صفحة ما، فإن ال Latency يكون هو الوقت الذي تستغرقه البيانات للانتقال من الـ server إلى الجهاز الخاص بي -بشكل عام-...

٢. ال Scalability:

قابلية التوسع هي قدرة النظام على التعامل مع الزيادة في الزخم على جزئية ما، وهذا يشمل الزيادة في عدد المستخدمين، أو حجم البيانات، أو عدد الطلبات...، وهذا يرتبط ارتباطا وثيقا بالتزامن -Concurrency-.

إن مفهوم التزامن مفهوم مهم جدا، ويشير إلى القدرة على تنفيذ عدة مهام في نفس الوقت، لذلك، حتى يكون النظام قابلا للتوسع فيجب أن يكون قادرا على التعامل مع المزيد من الطلبات المتزامنة في نفس اللحظة، مما يسمح له باستيعاب أكبر عدد من المستخدمين المتواجدين في نفس اللحظة.

سؤال: هل تعاملت مع نظام من قبل كانت فيه هذه المشكلة؟ كيف كان شعورك حينها؟ لعل من أشهر الأمثلة على هذه المشكلة هي مواقع عرض نتائج الثانوية العامة أو مواقع التسجيل للمواد الجامعية عند بداية كل فصل... وكثير من الناس تقول بالعامية: "الموقع معلق، كل الناس فاتحيته"، وهم بذلك فعليا أشاروا لوجود مشكلة في النظام تتعلق بالقابلية للتوسع وخدمة هذا العدد -المتوقع- من الأشخاص ^^.

٣. ال Availability:

ويقصد بهذا المفهوم قدرة النظام على الاستمرار بالعمل لأطول مدة زمنية ودون انقطاع بالخدمات، ويتم الحديث عن هذه الأرقام وتمثيلها من خلال النسبة المئوية مثل 100% والتي تعني أن النظام لن يسقط (Zero Downtime)، وهذا تجده موثقا في الأنظمة للشركات الكبيرة ضمن اتفاقية ال SLA، وال SLA هي اختصار service level agreement، والتي بدورها تمثل اتفاقية بين الزبون ومقدم الخدمة لبيان مدة استمرارية عمل النظام دون توقف، فمثلا ستجد المواقع تضع ال SLA uptime لديها يساوي أو أكبر من 99.9%، وقد تظن لوهلة أن 99.9% أو 99.99% أو 99.999% هي أرقام للغايات التسويقية فقط، لكن في الحقيقة هناك قيمة مهمة تختفي خلف هذه الأرقام، شاهد هذا المثال:

لو افترضنا أن Availability كانت 99%، فإن أقصى مدة زمينة لل Downtime يجب ألا يتجاوز ال 14.4 دقيقة يوميا والتي تعادل 3.65 يوم في السنة، ويمكن حسابها ببساطة من خلال هذه المعادلة -باعتبار أن الزمن المطلوب بالدقائق-:

MinutesInDay - MinutesInDay * Availability = downtime

وهذا يعني: 1440 - 1440 * 99% = 14.4 دقيقة يومي (14.4 * 365 = 3.65 يوم سنوي)، والآن لنشاهد الفرق بوجود تسعات أكثر بعد الفاصلة، وأنها ليست مجرد عملية تسويقية! فلو افترضنا أن Availability كانت 99.99% فالنتيجة ستكون 1440 - 1440 * 99.99% = 0.144 (وهذا يعادل 8.64 ثانية يوميا)، وسنويا (0.144 * 365 = 52.56

دقيقة)...، لاحظ الفرق بين النتائج، انخفض أقصى وقت لل downtime من 3.65 يوم إلى 52.56 دقيقة!

لذلك، كلما زادت أهمية النظام، زادت الحاجة إلى "تسعات" أكثر لضمان أنه متاح للعمل في أي لحظة، وهذا سيقودك إلى التفكير بطريقة مختلفة أثناء تصميم المعمارية لمثل هذه الأنظمة لضمان أن عمليات الصيانة المجدولة والتحديثات وحالات الفشل غير المتوقعة لا يجب أن تزيد عن الوقت الذي تم تحديده من خلال ال "تسعات" ^^، فاحرص على تذكر هذا في الأنظمة التي تعمل عليها...

٤. ال Security:

إن موضوع الأمان مهم وضروري لأي مشروع برمجي، حتى أبسط التطبيقات تحتاج إلى بعض إجراءات الأمان... إن إهمال موضوع الأمان في مرحلة تصميم المعمارية للنظام يمكن أن يؤدي إلى ثغرات خطيرة! كما أن موضوع الأمان في الأنظمة البرمجية لا يقتصر على جانب واحد، بل يغطي مجموعة واسعة من الجوانب الحيوية، مثل:

- ال Authentication: التحقق من هوية المستخدم
- ال Authorisation: منح الصلاحيات للمستخدمين بعد التحقق من هويتهم، على سبيل المثال: تحديد ما إذا كان المستخدم لديه الإذن للوصول إلى بيانات معينة أو صفحة معينة أو له الصلاحية على تنفيذ إجراء ما.
- ال Confidentiality: ضمان بقاء البيانات آمنة والحفاظ على المعلومات الحساسة، سواء أكان ذلك أثناء نقلها عبر الشبكة أم أثناء تخزينها في قواعد البيانات.

ملاحظة: يمكنك الاطلاع على ال Open Web Application Security Project (OWASP))، فهو يمثل نقطة بداية ممتازة ورائعة لفهم موضوع الأمان في عالم الويب.

٥. ال Privacy:

لا يمكن فصل الخصوصية عن الأمان، فهناك ارتباط ما بينهم دائما، وهي تتعلق بحماية المعلومات الشخصية للمستخدمين، وكيفية جمعها ومعالجتها وتخزينها... وهي بذلك تضمن أن البيانات الحساسة لا يمكن الوصول إليها أو استخدامها إلا من قبل الأشخاص المصرح لهم لذلك -.-، والجزئية المهمة هنا هي أن تكون على دراية بالتشريعات الحديثة، مثل: ال GDPR إذا كنت تتعامل مع بيانات لأشخاص تخص مستخدمين يقيمون في الاتحاد الأوروبي.

إن الفرق بين الأمان والخصوصية هو أن الأمان يركز على حماية البيانات من الاختراق والوصول غير المصرح به، بينما تركز الخصوصية على حقوق المستخدمين في التحكم ببياناتهم.

وقد تتسائل الآن، كيف لهذا المفهوم أن يؤثر على التصميم الخاص بمعماريتنا؟ أو لماذا يعد من السمات الجودة الأساسية؟
والجواب يكمن في أن إدراكك لمثل هذه الأمور وأهميتها أو وجود لوائح قانونية مثل ال GDPR سيجعلك تصمم النظام بطريقة تخدم المستخدمين وتحفظ خصوصيتهم وتجنبك العقوبات والغرامات الضخمة التي قد تفرض عليك... مثلا: يجب أن يكون النظام قادرا على

إثبات التزامه بقواعد الخصوصية المنصوص عليها، وهذا ما يطلق عليه بالشفافية وقابلية التدقيق -Transparency and Auditability-، والتأثير المعماري هنا هو الحاجة إلى تصميم نظام قادر على تسجيل الأحداث وتدقيق كل عملية وصول إلى البيانات الحساسة أو تعديلها... وهذا يعني الحاجة لبناء النظام ك Event Sourcing بحيث يسجل كل event بشكل مستقل لا يمكن تعديله... أرأيت أهمية هذه النقطة!

٦. ال Disaster Recovery:

ما هي قدرة نظامك على التعافي من الكوارث؟ هل فكرت في هذا من قبل؟
يُعنى هذا المفهوم بوضع خطة للتعامل مع الأحداث الكارثية التي قد تؤدي إلى توقف النظام البرمجي بحيث يتم تحديد الإجراءات اللازمة لاستعادة البيانات والوظائف بسرعة وفعالية بعد وقوع كارثة، مثل فشل ال server أو مركز البيانات...

هذا المفهوم يرتبط ارتباطا وثيقا بالأنظمة التي تعتبر "Mission Critical"، وال Mission Critical تشير لأي نظام أو عملية تعتبر حيوية لنجاح المؤسسة أو استمرار عملياتها، وأي فشل في هذا النظام أو العمليات المتعلقة به يعني خسائر كبيرة إما بالأرواح أو الأموال أو بسمعة الشركة... ومن الأمثلة على ذلك القطاع المالي، فإذا توقفت معالجة العمليات المالية لبطاقات الإئتمان لسبب ما، فهذا يعني خسائر بالملايين في الدقيقة الواحدة! لذلك، تهتم هذه الأنظمة وتكون أولويتها القصوى هي ال Availability وال Disaster Recovery... وهذا سيقودنا إلى مفهوم آخر وهو: عمليات استمرارية الأعمال -Business Continuity Processes-،

وهي تمثل أيضا مجموعة الخطط والإجراءات التي تضعها الشركة لضمان استمرار عملياتها التجارية الحيوية بعد وقوع كارثة ما أو حادث طارئ ما، وهنا قد تتسائل، ما الفرق بين التعافي من الكوارث -Disaster Recovery- وعمليات استمرار الأعمال -Business Continuity Processes-؟

وهذا سؤال منطقي، ويمكن الإجابة عنه باختصار، بأن مفهوم التعافي من الكوارث يركز على الجانب التقني فقط، فتجده يحاول استعادة عمل الأنظمة التقنية بعد انقطاع الخدمة... أما عمليات استمرار الأعمال فهي أشمل وأوسع نطاقا، فهي تحتوي في داخلها التعافي من الكوارث + الجوانب غير التقنية مثل كيف سيتواصل الموظفين والأقسام مع بعضهم البعض أثناء حدوث هذه المشاكل... وتشمل خطة استمرارية الأعمال عدة عناصر رئيسية مثل تحديد العمليات الأكثر حيوية للمؤسسة والإجراءات الفورية التي يجب اتباعها عن حدوث المشكلة، وخطة التعافي من الكوارث لاستعادة الأنظمة والبيانات وخطة الاتصالات لتحديد كيفية التواصل بين الموظفين والعملاء والموردين ونحو ذلك.

والآن، هل عرفت أهمية هذا الجزء ولماذا وضع من ضمن سمات الجودة؟ ولماذا يعتبر من المحفزات المعمارية؟

نعم صحيح يا صديقي، إن التعافي من الكوارث يؤثر بشكل مباشر على القرارات المعمارية، فهو ليس شيئا يضاف إلى النظام "لاحقا"، بل هو أمر يتطلب تصميم النظام ومعماريته على أساس الاهتمام بال availability، وبهذا يكون التصميم مصمما للتعافي السريع من الكوارث، مثل: أخذ نسخ احتياطية من البيانات وتخزينها في مكان مختلف، والاهتمام بال

Redundancy بحيث إذا فشل أحد ال server عن تقديم الخدمة يقوم server اخر بتقديمها، وال Failover التي تسمح بالتبديل التلقائي إلى نظام احتياطي في حالة فشل النظام الأساسي، ونشر النظام على data center متعددة في مناطق جغرافية مختلفة للحد من أثر الكوارث الطبيعية أو الانقطاعات التي يمكن أن تحدث في منطقة أو قارة ما... وبهذا كله ندرك مجددا أهمية "التسعات" التي تحدثنا عنها بال Availability، وأنها ليست مجرد رقم ^^.

٧. ال Accessibility:

ال Accessibility عادة ما تشير إلى معايير مثل ال W3C Accessibility Standards، والتي توضّح كيف يكون برنامجك متاحا للأشخاص ذوي الاحتياجات الخاصة مثل ضعف البصر أو العمى...

وتعتبر هذه السمة من المحفزات المعمارية لأن التصميم يجب أن يبنى بطريقة نضمن فيها إمكانية استخدام النظام من جميع أنواع المستخدمين، بمن فيهم ذوي الاحتياجات الخاصة... هذا الاعتبار يعني الحاجة لفرض بعض المتطلبات عند التصميم في بنيتنا التحتية، مثل شكل البيانات وكيف سيتم عرضها في ال UI بطريقة تسمح للأجهزة المساعدة -كقارئ الشاشات- من فهم ما هو مكتوب، وهذا يتطلب استخدام Label واضحة لكل عنصر، وكتابة ال HTML بطريقة semantic... إلخ

كما أن هذا النوع من المعماريات يتطلب الأخذ بعين الاعتبار القدرة على تخصيص ال UI للمستخدم، مثل تغيير الخط أو الألوان ونحو ذلك، مثلا: إذا استخدم الموقع من هو مصاب

بعمى الألوان يمكنه تغيير الألوان، وإن استخدمه كبير السن فيمكنه تغيير حجم الخط... إلخ، كما أن التصميم يجب أن يراعي أن كل إجراء أو حدث موجود على الموقع يجب أن يكون متاحا للتحكم به من خلال ال keyboard فقط ودون الحاجة لاستخدام ال mouse ^^... وهذا كله يظهر تأثير هذه السمة على المعمارية الخاصة بنا ^^

٨. ال Monitoring:

المراقبة هي عملية جمع البيانات حول أداء وصحة النظام للتأكد من أنه يعمل بشكل صحيح، وهذا يشمل زمن الاستجابة، ومعدل الأخطاء، وقيم الاستخدام لموارد النظام... وتُعتبر المراقبة من المحفزات المعمارية لأنها تؤثر بشكل مباشر على القرارات التقنية وهيكلية النظام، ومن هذا التأثير الحاجة لإرسال تنبيهات عند حدوث مشكلة ما بشكل مركزي، وإذا كانت عملية المراقبة عبارة عن خدمة خارجية فأنت بحاجة إلى تصميم معماري يسمح بهذا الدمج، وهذا يعني الحاجة إلى إضافة Layer أو مكونات مبنية خصيصا لهذا الغرض... كما أن ال framework المستخدم أو لغة البرمجة ونظام التشغيل وغيرها قد يؤثر على القرار لأنك بحاجة لضمان عمل الأدوات المستخدمة معها... إلخ

٩. ال Management:

لقد تحدثنا في النقطة السابقة عن المراقبة، وكانت النتيجة التي رأيناها أن المراقبة تخبرنا بما يحدث في النظام، لكن إذا أردنا القدرة على التدخل وتغيير / تعديل ما يحدث فهذا ما يطلق عليه بالإدارة -Managment- ^^

إن الإدارة في هذا السياق تمنح فريق التشغيل القدرة على التفاعل مع النظام وتعديل سلوكه أثناء ال runtime ودون الحاجة لعمل deploy، ومن الأمثلة المشهورة والتي تعاملنا معها كثيرا تحديث الـ Configuration الخاصة بالنظام أثناء العمل، مثل تفعيل وضع الصيانة أو تعديل حدود ال rate limiter ونحو ذلك، ومن الأمثلة التي تعاملنا معها كثيرا أيضا ال Feature Toggle بحيث يتم إيقاف أو تشغيل الميزات للمستخدمين دون الحاجة إلى عمل deploy... إلخ

١٠. ال Audit:

ال Audit -التدقيق- هو أحد السمات أو المحفزات المعمارية المهمة في معمارية الأنظمة، فوجودها يعني الحاجة إلى الاحتفاظ بسجل تفصيلي لجميع الأحداث أو التغييرات التي تحدث في النظام، والهدف من ذلك هو إنشاء سجل تدقيق دقيق يوضح:

- من قام بالتغيير؟
- ومتى حدث التغيير؟
- وماذا كان التغيير؟ (مثل القيمة قبل التغير وبعد التغير)

- ولماذا تم التغيير؟

وبكل تأكيد يتضح جليا سبب أهمية هذه السمة في تصميم المعمارية: الأنظمة المالية إنموذجا! والتي ستستخدم فيها هذه السجلات لاحقا لحل النزاعات المحتملة وإخلاء المسؤولية ونحو ذلك...

لكن، لحظة... هل تدرك ما يعني تطبيق هذه السمة فعلا على النظام / المعمارية وعلاقتها بالسمات الأخرى؟

إن وجود هذه السمة يتعارض مع سمة أخرى مثل الخصوصية، لأن تخزين كل ما يحدث يمكن أن يتعارض مع ال GDPR والتي تطلب حذف بيانات المستخدم عند طلب ذلك! كما أن تصميم هذا النظام يحتاج لجهد أكبر لكونه أكثر تعقيدا من النموذج التقليدي... وبهذا يزداد تأثير هذه السمة على المعمارية ويزيد من الحاجة لأخذها بعين الاعتبار منذ البداية ...^^

ملاحظة: قاعدة مهمة تذكرها دائما، وسنبقى نذكرها في كل سياق: التصميم الخاص بالأنظمة أو المعماريات التي قمت أو ستقوم ببنائها تمثل مجموعة من المقايضات والتنازلات للحصول على أفضل شكل يخدم النظام، وليس بالضرورة أنه سيخدم أي نظام آخر... بمعنى آخر، إنك تختار أفضل السيء، فلا وجود للحل المثالي في العالم الواقعي ^^.

١١. ال Flexibility:

إن كلمة "المرونة" مجردة تعتبر من المصطلحات الغامضة في عالمنا، إن لم يتم ربطها بشيء حقيقي... فمثلا كم مرة سمعت أحد الأشخاص يطلب منك بناء نظام مرن؟، لكنه في نفس الوقت لم يحدد ما نوع المرونة المطلوبة! ولم يحدد ما الذي يجب أن يكون مرنا أصلا! ومن الذي سيقوم بالتعديل أو من هو المخول بهذا التعديل!

لذلك يرتبط هذا المفهوم بشكل كبير بالسياق الذي أتت به، لذلك، يمكننا أن نقول هنا وفي هذا السياق أنها قدرة النظام على التكيف مع المتطلبات المتغيرة ودون الحاجة إلى إعادة تصميم، أي القدرة على تأدية نفس المهمة بطرق مختلفة حسب الحاجة أو القدرة على تأدية مهام متعددة وليس مهمة واحدة جامدة.

تعال معي لنفصل الأمر أكثر حتى تتضح هذه التعريفات والمفاهيم التي بداخلها ^^...

تخيل معي أنك قمت ببناء موقع تجارة إلكتروني يستخدمه أشخاص من دول مختلفة، هذا الموقع فيه قاعدة تجعل من الشحن مجانيا إذا كانت فاتورة المشتري أكثر من ١٠٠ دينار، ثم جاءت مناسبة ما وأرادت الإدارة تخفيض المبلغ -تغيير القاعدة- إلى ٥٠ دينارا، فما الذي سيحدث عندك الآن؟

الخيار الأول أن يكون النظام بُنية بطريقة جامدة بحيث تحتاج إلى تعديل الشيفرة البرمجية ومن ثم عمل build و deploy -.-! والخيار الثاني أن يكون النظام مرنا فيمكن للإدارة من تغيير هذه القاعدة بأيديهم من خلال لوحة تحكم ودون الحاجة لمبرمج ^^ وهذا مثال على: "القدرة على تأدية نفس المهمة بطرق مختلفة حسب الحاجة".

لو افترضنا أن هذا الموقع -موقع التجارة الإلكترونية- سيقوم بإصدار فاتورة للمشترين، فماذا سيحدث عندك الآن؟
الخيار الأول أن يكون النظام بنية بطريقة جامدة يعتمد على نموذج فاتورة واحد ثابت، ولا يدعم العملات المتعددة أو الضرائب المتغيرة! أو الخيار الثاني أن يكون النظام مرنا يمكنه إصدار فواتير بعملات مختلفة، وتطبيق ضرائب متغيرة حسب الدولة، واختيار نماذج فواتير مناسبة لكل نوع من المستخدمين، وهذا سيسمح لك باستخدام الفاتورة في سياق محلي أو دولي، لأفراد بضريبة أو دون ضريبة ونحو ذلك... وهذا مثال على: "القدرة على تأدية مهام متعددة وليس مهمة واحدة جامدة"

أظن أن تأثير هذه السمة على المعمارية أصبح واضحا جدا، لكن، حتى نتعلم أكثر فهناك أنماطا معمارية يمكنك استخدامها إذا كانت المرونة إحدى متطلبات النظام، مثل ال Data-Driven Design، وهو نهج معماري في تصميم الأنظمة البرمجية يعتمد على جعل "البيانات" هي التي تُحدّد سلوك النظام، بدلا من أن يكون السلوك مُضمنا في الشيفرة البرمجية... وبمجرد اعتمادك لمثل هذا النمط فهذا يعني أنك ستكون قادرا على تحقيق مجموعة من السمات التي تحدثنا عنها سابقا مثل ال Management...

وهنا قد يخطر على بالك سؤال منطقي وقوي، وهو ألم نتحدث عن هذه المزية في ال Management؟ أو على أقل تقدير ما هو الفرق بينهم! ^^
أحسنت يا صديقي، هذا سؤال مهم، لكن الإجابة المختصرة هي أن المرونة والإدارة مفهومين متداخلين بشكل كبير، لكن لدى كل واحد منهم أهداف مختلفة! فالإدارة تتعلق بالتحكم في

التفاصيل التقنية، بينما المرونة هي تصميم النظام ليكون قابلاً للتكيف مع التغييرات في متطلبات العمل، لذلك إذا كان ال configuration الذي تكلمنا عنه في موضوع الإدارة يحتوى على business rules فإن تعديله يعتبر مرونة، لكنه إذا احتوى على معلومات تقنية مثل ال API endpoints أو ال Database connection فإن تعديله يعتبر إدارة، لأنه لا يغير ال business logic، وبهذا تكون المرونة سهولة التكيف مع التغييرات في حين أن الإدارة سهولة التشغيل وإدارة النظام وصيانته...

١٢. ال Extensibility:

ويقصد بها قدرة النظام على إضافة ميزات جديدة أو وظائف إضافية في المستقبل بسهولة، دون الحاجة إلى تعديل الشيفرة الأساسية، وبهذا في تختلف عن المرونة بكونها تسمح للنظام بأداء مهام جديدة تماما لم يتم تصميمها للقيام بها منذ البداية...^^ ولعل من أشهر الأمثلة على هذه المزية ال APIs التي يتم تصميمها لخدمة المطورين، مثلا إمكانية إنشاء تطبيق متكامل يتعامل مع الخدمات التي نقدمها من خلال ال APIs أو من خلال تثبيت بعض ال Plugins، ومن الأمثلة المشهورة على الأسلوب الأول ال Android، فهو يسمح للمطورين بإنشاء تطبيقات جديدة واستخدام خصائص الجهاز مثل الكاميرا والموقع الجغرافي من خلال APIs معرفة... ومن الأمثلة المشهورة على الأسلوب الثاني هو تحميل Keyboard جديد على هاتفك المحمول بالرغم من أنها ليست جزء من نظام التشغيل الأساسي، ومع ذلك فإنها تتعامل مع نظام الأندرويد بسلاسة... كما أن الشركات المختلفة تقوم بتخصيص نسخ الأندرويد وتقوم بإضافة ميزات جديدة خاصة بها دون الحاجة لتغيير الشيفرة البرمجية الأساسية، وهذا

يجعل الأجهزة التابعة للشركات تظهر بأشكال مختلفة بالرغم من أن الجميع أندرويد ^^...
وهذه كلها أمثلة على ال Extensibility، وبهذا يظهر أهمية هذه السمة وأثرها على النظام إذا كانت من متطلبات النظام، ولا يمكن تجاهلها...

١٣. ال Maintainability:

تشير قابلية الصيانة إلى مدى سهولة تعديل وتصحيح وتحديث الشيفرة البرمجية في المستقبل، إنه مقياس لمدى صعوبة أو سهولة التعامل مع الشيفرة البرمجية بمجرد اكتمال بنائه الأولي... وهذا يذكرني بواحدة من الطرائف التي قرأتها قديما، وهي: «إن الطريقة الوحيدة الصالحة لقياس جودة الشيفرة البرمجية هي عدد المرات التي تقول فيها "شو هالغباء!!!" في الدقيقة الواحدة»... هذا النص وهذه السمة تثير نقطة حيوية ومهمة، وهي التفكير في الشخص الذي سيقوم بصيانة الشيفرة البرمجة! فهذا الشخص قد يكون أنت نفسك بعد عدة أشهر، أو مبرمجا انضم إلى الفريق حديثا ^^، فإذا كانت الشيفرة البرمجية غير قابل للصيانة فسيجد هذا الشخص -أنت أو غيرك- صعوبة في إصلاح الأخطاء لأنك ستستغرق وقتا أطول لإيجاد المشكلة وستستغرق وقتا أطول حتى تفهم الشيفرة البرمجية أو أن إضافة أي مزية جديدة ولو صغيرة قد تتسبب في كسر أجزاء أخرى من النظام مما يهدد سلامة النظام والموظف *-*

لذلك، ولتحقيق هذه السمة -والتي يصعب قياسها- يتم تطبيق مبادئ التصميم والتطوير في مراحل العمل، ومن ذلك ال Clean code من خلال كتابة شيفرة برمجية واضحة ومقروءة

ويمكن فهمها، ومن خلال التصميم الجيد عن طريق استخدام أنماط تصميمة تسمح أو تسهل من إضافة وظائف جديدة أو من خلال توثيق القرارات المعمارية الرئيسية...

والسؤال المعتاد الآن، لكن كيف ستؤثر هذه على تصميمنا الخاص بالنظام؟!
والجواب: إن هذه السمة تقودنا لاتخاذ بعض الإجراءات المهمة في التصميم مثل ال Separation of Concerns، بحيث تكون كل ميزة في النظام مستقلة عن غيرها قدر الإمكان، وهذا يسهل على المبرمج فهم جزء معين من الشيفرة البرمجية دون أن يحتاج إلى فهم الشيفرة البرمجية كلها... وهذا يقودنا لهذه القاعدة: "التنظيم وفصل المكونات = قابلية للصيانة عالية = صيانة أسهل، وتكلفة وجهد أقل".

١٤. ال Legal, Regulatory and Compliance:

بعض الصناعات تخضع بشكل صارم للقوانين المحلية أو الجهات التنظيمية، لذلك يجب أخذ هذا الموضوع المهم بعين الاعتبار، وهذا يشترك مع ما تحدثنا عنه سابقا في موضوع الخصوصية عند تطرقنا لل GDPR... ومن الأمثلة أيضا اللوائح المتعلقة بالضرائب على الخدمات الرقمية المقدمة ونحو ذلك.

١٥. ال Internationalisation (i18n):

هذا المفهوم كثيرا ما تعاملنا معه، وبنينا أنظمتنا باعتبار أنه جزء أساسي وركيزة لا يمكن البدء في المشروع من دونها... وهذا السلوك الطبيعي لن يذهب سدى، بل سنزيد عليه من خلال اعتباره سمة أساسية من سمات الجودة...

أما لمن لم يتعرف على هذا المفهوم من قبل: فهو مفهوم يشير إلى تصميم وتطوير النظام ليكون قابلا للتكيف مع مختلف اللغات والثقافات والمناطق الجغرافية دون الحاجة إلى تغيير الشيفرة البرمجية الأساسية، وأشهر استخدام له: الترجمة، مثل إنشاء موقع يعمل باللغة العربية والإنجليزية، وعادة ما ستسمع "i18n" أثناء عملك مع مبرمجين آخرين، وهم بذلك يريدون هذا المفهوم، فالاسم "i18n" هو اختصار لـ "Internationalisation"؛ حيث يرمز الرقم 18 إلى عدد الأحرف بين حرفي الـ "i" و "n"... ويركز هذا المفهوم على جانبين رئيسين وهما:

- اللغات: ويقصد بها القدرة على عرض المحتوى بلغات متعددة
- المناطق: ويقصد بها القدرة على التكيف مع الفروقات الثقافية والديموغرافية مثل التاريخ والوقت ونظام القياس وتنسيق العملات... إلخ، فمثلا إذا كان الجمهور المستهدف من اليابان فسنقوم بعرض العملة بالين -¥- الياباني بدلا من الدولار -$-... وإذا كان نظام القياس المعتمد عندنا هو المتري -كيلو جرام، متر- فهناك دول كالولايات المتحدة تستخدم نظام القياس الإمبراطوري -باوند، قدم-...

إن ال i18n تعتبر واحدة من المحفزات المعمارية المهمة لأنها تُملي علينا اتخاذ قرارات تصميمية جوهرية في مرحلة مبكرة من المشروع، فعلى سبيل المثال: قد يكون تصميم موقع

يعمل من اليسار إلى اليمين -LTR- بلغة واحدة أمرا سهلا، ولكن الوضع يصبح أكثر صعوبة عند إضافة لغات أخرى للموقع مثل اللغة العربية؛ لأنها تحتاج لترجمة النصوص، ونحتاج لبناء واجهة يتم عرضها من اليمين إلى اليسار -RTL-...إلخ، هذه الأمور كلها إذا كانت من ضمن معماريتنا فهذا الأمر ممتاز لأنه يعيننا على تصميم النظام بطريقة تمكننا من إدارة مكان تخزين النصوص وكتابة الشيفرة البرمجية بطريقة تدعم اللغات المتعددة وتدعم الاتجاهات المختلفة للواجهات، وتراعي نوع ال Encoding المستخدم لضمان عرض جميع اللغات بشكل صحيح...

أي هذه السمات التي تكلمنا عنها مهم بالنسبة لك؟

ليست كل هذه السمات بنفس الأهمية، وليست كل السمات يمكن تطبيقها معا، ولا يمكن التركيز على تطبيق كل هذه السمات في نفس الوقت! فلكل مشروع سياقه الخاص الذي سيدفعك للاهتمام بسمات على حساب سمات أخرى! وهذا الأمر طبيعي، بل هو الصحيح! وتذكر هنا أن الوسطية شعارنا، فلا إفراط ولا تفريط ^^

من الأمثلة على ذلك، إذا كنت تعمل على نظام يقدم خدمات مالية، فإن ال Availability والأمان سيكونان من أهم السمات الخاصة بهذا المشروع، في حين لو كنت تعمل على موقع إخباري فإن الأداء وال i18n قد تكونان من أهم السمات التي يجب أن تهتم بها... وهنا لا نتكلم عن إهمال السمات الأخرى، بل تصميم النظام بطريقة قائمة على ترتيب أهم السمات المطلوبة في سياق المشروع الذي تعمل عليه...

الخلاصة: "القرارات المعمارية يجب أن تكون موجهة من خلال أهداف العمل المحددة، وليس من خلال قائمة شاملة من السمات النظرية!"

العمل مع ال Non-Functional Requirements

هناك العديد من التحديات أثناء فترة العمل على المشروع، ومن أهم هذه التحديات هي جمع المتطلبات الخاصة بالمشروع، والناس هنا عادة ما يكونون متحمسين لتقديم قائمة بالمتطلبات الوظيفية التي يحتاجونها -أي ما يجب على النظام فعله-، وعملية توثيق هذه المتطلبات سهلة ويمكن استخدام العديد من الطرق لهذا الغرض مثل ال User Stories وال User Cases وغيرها... لكن هذا الأمر لا ينطبق على المتطلبات غير الوظيفية -كيف يجب أن يعمل النظام؟-، فمن الصعب جدا جمع سمات الجودة -Quality Attributes- وتحديدها، وذلك لعدة أسباب:

- غياب الوعي، فغالبا ما يرى المستخدمون أو حتى مديري المنتجات أن هذه المتطلبات أمر مفروغ منه، ولا يفكرون في أهميتها...
- صعوبة التعبير عنها بعبارات بسيطة أو بشكل دقيق
- لا يفكرون فيها تلقائيًا
- يركزون على "الميزات" أولا

مثال عملي: قد يظن الناس أن الأمان أو الأداء مفاهيم مفهومة من تلقاء نفسها، فيكتفون بالإشارة إلى: "نحتاج إلى موقع سريع في معالجة الفيديوهات"، لكن السؤال ماذا تعني كلمة

سريع هنا؟ هل يقصد بها أقل من ٥ ثواني أم ١٠ ثواني أم أجزاء من الثانية؟ لأن إجابة هذا السؤال بالنسبة لنا من العميل أمر مهم جدا! تخيل أن تكون معالجة فيديو مدته ١٠ دقائق بدقة عالية قد أخذت ١٠ ثواني، فيخبرك العميل أن المعالجة بطيئة وكان يتوقع سرعة أقل من ٥ ثواني لهذا العمل...

لذلك، يجب على المعماري أن يكون استباقيا في طرح الأسئلة الصحيحة لاكتشاف هذه المتطلبات، لأن المستخدمين لن يذكروها تلقائيا... مثل: كم عدد المستخدمين الذين سيستخدمون الموقع بشكل متزامن؟ وهل هناك بيانات حساسة تحتاج إلى تشفير؟ وما مدى توفر النظام الذي تريده؟ 99%؟ 99.99%؟... إلخ
لكنك قد تواجه مشكلة جميلة عليك حلها *-*: إذا أخبرك العميل أنه يحتاج لنظام يعمل بنسبة ١٠٠٪ أو أنه يريد أن يكون النظام سريعا أو أنه يريد أن يكون النظام آمنا، فماذا ستفعل؟ وكيف تتوقع أن تتعامل مع مثل هذه المواقف؟ وما هي الإجراءات التي يمكنك اتخاذها للحصول على الإجابة الصحيح؟

إجابة هذه الأسئلة السابقة تكون من خلال تحويل المتطلبات غير الوظيفية الغامضة والسطحية إلى متطلبات محددة قابلة للقياس، وبهذا فبدلا من السؤال بطريقة سطحية كم مقدار ال Availability المطلوب للنظام؟ ثم الحصول على الإجابة اللولبية: بنسبة ١٠٠٪ أو ٢٤ ساعة على مدار ٧ أيام، فإننا يمكن أن نسأل نفس السؤال بالطريقة التالية: كم من التوقف عن العمل يمكننا تحمله، وماذا سيحدث إذا فشل النظام في وقت الذروة؟، وماذا سيحدث لو فشل النظام خارج أوقات الذروة؟

إن ما نحاول فعله هنا هو استكشاف المتطلبات والوصول إلى النقطة التي تفهم فيها ما هي القوى الدافعة حتى يحتاج النظام إلى أن يكون متاحا كل هذا الحد؟
ونفس الأمر إذا كنا نبحث عن أعلى درجة أمان ممكنة، فالسؤال هو: ما الذي نحاول حمايته بالتحديد؟، وكذلك فيما يتعلق بالأداء، فالسؤال يمكن أن يكون: كم عدد المستخدمين المتزامنين اللذين يجب أن يدعمهم النظام في المتوسط؟ وماذا عن أوقات الذروة؟، وما هو زمن الاستجابة المقبول؟ وهل هذا ينطبق على جميع أجزاء النظام أم ميزات محددة فقط؟

إن إجابات مثل هذه الأسئلة تعبر أن قيم كمية يمكن قياسها، فإذا تمكنا من ربطها مع المتطلبات غير الوظيفية فهذا يعني القدرة على كتابة Acceptance Criteria واختبارها بشكل موضوعي ^^... هذه العملية هي ما يطلق عليه ال Refine.

إن هذه العملية مهمة جدا جدا، فمن خلالها يتم تحويل قائمة المتطلبات من مجرد أمنيات وأحلام إلى محفزات معمارية حقيقية، فإذا قلنا أن هناك مليون مستخدم متزامن فهذا قد يدعونا لاختيار بنية معمارية قابلة للتوسع بشكل Horizontal أو سيدعونا ذلك لاستخدام نوع قاعدة بيانات معين...

تحدي "كل شيء ضروري"

عندما يتم سؤال العملاء أو المستخدمين عن احتياجاتهم، يميلون إلى طلب كل ميزة ممكنة، مما يؤدي إلى تصنيف كل المتطلبات على أنها: يجب أن تكون موجودة -Must Have-، وهذا يجعل من عملية تحديد الأولويات -مثل استخدام منهجية MoSCoW- غير فعالة.

بدلا من ذلك، يمكننا اتباع نهج مختلف في التعامل مع هذه المشكلة، وهو عرض المقايضات والتكاليف -وهنا يتراجع أغلب الناس عن السمات / المزايا غير الضرورية حقيقة-، فمثلا إذا طلب العميل نسبة Availability عالية مثل ١٠٠٪، فيمكننا إخباره أن هذا الأسلوب يمكننا تصميمه، لكن التكلفة الإجمالية لتغطية متطلبات هذا العمل ستصل إلى مليون دولار، في حين أن بناء نظام أبسط بنسبة Availability مرتفعة ستكون تكلفته مئة ألف دولار... هذا الحل يهدف إلى خفض التكلفة وإيجاد الحل الأكثر ملائمة للمشروع...

إن شرح أن كل شيء يتم استخدامه أو إضافته أو طلبه هو عبارة عن Trade Off؛ يساعد في زيادة الوعي وثقافة العميل ويساعدنا على إيجاد أفضل حل ممكن في السياق المعطى لنا...

معلومة: ال MoSCoW هي احدى التقنيات المستخدمة لتحديد أولويات المتطلبات عند إدارة المشاريع، وتستخدم بشكل كبير عند اعتماد ال Agile كأسلوب لتطوير المشروع... وتمثل هذه الكلمة اختصارا لأربع فئات من الأولويات وهي: يجب أن تكون -Must

Have- وينبغي أن تكون -Should have- ويمكن أن تكون -Could Have- ولن نحتاجها حاليا -Won't have for now-...

فائدة

فلْتعلمْ يا أخي أن الإنسان يجب عليه أن يستعين بالله -سبحانه وتعالى- ويتوكل عليه في كل ما يصيبه، فإن رأى من أسباب التوفيق ما رأى فيجب أن ينسبه لله -سبحانه وتعالى-، لأن هذا التوفيق ما كان إلا بفضل من الله -سبحانه وتعالى-!، وفي هذا حفظ للنفس من الكبر والغرور، فلا يتكل الإنسان على نفسه ولا يميل إليها ولو شيئًا قليلًا لما يراه من مفاخر تحققت له!، ونستأنس هنا بما روي عن أنس بن مالك -رضي الله عنه- عن الرسول -صلى الله عليه وسلم- قال لِفاطمةَ رضي اللَّه عنها: "ما يمنعُكِ أن تسمَعي ما أُوصيكِ به ؟ أن تقولي إذا أصبحْتِ وإذا أمسَيتِ : يا حيُّ يا قيُّومُ برحمتِك أستغيثُ، أَصلِحْ لي شأني كلَّه، و لا تَكِلْني إلى نفسي طرفةَ عَيْنٍ"، فلا تكلني لنفسي طرفة عين، أي فلا تترك إعانتي يا الله ولو لحظة صغيرة، بل أعني بقدرتك وقوتك دومًا، فإننا نستعين بك، ونستهديك، وما توفيقنا إلا بك، فاللهم لا تكلنا لأنفسنا طرفة عين.

- كتاب إلى الجنة زمرا، الصفحة ٢٦٤

ال Constraints -القيود والمحددات-:

القيود ليست مجرد تحديات نواجهها! بل هي محفزات معمارية أساسية، فهي تُجبرك على التفكير في الحلول التي تناسب الإطار الذي سيعيش أو سيبنى فيه المشروع، وسيؤثر هذا بشكل مباشر على القرارات التقنية، فعلى سبيل المثال: إذا كان هناك قيود صارمة على الميزانية، فقد تضطر إلى اختيار تقنية مفتوحة المصدر بدلاً من الحلول مدفوعة.

إن أي مشروع رقمي نقوم ببنائه سيعيش في العالم الحقيقي لا الرقمي! وبهذا فإن وجود قيود ستصيب هذا المشروع أمر لا بد منه، أكانت هذه القيود عالمية، أو على مستوى الدولة وتشريعاتها، أو حتى على مستوى المؤسسة التي تعمل بها... لذلك، هناك مجموعة مختلفة من القيود التي ستصادفك، بأشكال وأحجام مختلفة ^^

قيود الوقت والميزانية -Time and budget constraints-:

هذه واحدة من أكثر القيود التي نعرفها جميعا ويعيش فيها المطورون ^^، فيمكنها أن تُجبرنا على اختيار أسرع الحلول وأبسطها، أو أن تجبرنا على التخلي عن بعض الميزات لضمان إطلاق المنتج في الموعد المحدد.

قيود التقنية -Technology constraints-:

هناك الكثير من القيود التقنية التي تُفرض علينا والتي نواجهها عند بناء أي مشروع تقني -خصوصا في الشركات الكبيرة-، هذه القيود يجب أخذها بعين الاعتبار! مثلا: لا يمكنك البدء بمعمارية مشروع بلغة برمجة س في حين أن الشركة تسمح باستخدام ص!

هناك العديد من القيود التقنية التي ستواجهها، نذكر منها:

- قيود التقنيات المعتمدة -Approved Technologies-: العديد من الشركات الكبيرة لديها قائمة بالتقنيات المعتمدة عندها، والهدف من هذه القائمة هو توحيد التقنيات المستخدمة لتقليل تكاليف الدعم والصيانة، لذلك، يفرض هذا القيد على المطورين استخدام تقنيات محددة، وفي حال رغبتهم باستخدام تقنية غير مدرجة، فسيضطرون إلى اتباع إجراءات طويلة للحصول على استثناءات خاصة بهم، أو حتى من خلال اللجوء إلى حلول مخادعة -غير مباشرة- للالتفاف على هذه القيود...

- قيود الأنظمة الحالية -Existing Systems- وقابلية التشغيل المتبادل أو المشترك -Interoperability-:

معظم المؤسسات لديها أنظمة موجودة وجاهزة، ونحتاج إلى دمج برامجنا الجديدة معها، وغالبا ستكون الخيارات المتوفرة لدينا محدودة للقيام في عملية الدمج هذه، وفي أحيان أخرى تكون هذه الأنظمة هي التي تحتاج إلى الاتصال بما نبنيه نحن، لذلك قد تجد أن هناك قيودا على مستوى المؤسسة تحدد البروتوكولات والتقنيات التي يُسمح لك

باستخدامها عند نقاط الاتصال... على سبيل المثال، إذا كانت الأنظمة القديمة تستخدم الـ XML، فإن النظام الجديد سيضطر إلى استخدام نفس الطريقة للتكامل مع النظام القديم حتى لو لم يكن هذا الأسلوب هو الأفضل أو الأحدث، أو سيضطر لاستخدام وسيط يعمل كمترجم أو محول بين الطرق المختلفة، مثل التحويل من JSON إلى XML...

باختصار، ما يحدث هنا هو: "عملية بناء شيء جديد، لكنه مقيد بالبنية التحتية القديمة"، وذلك يشبه بناء منزل جديد ذا مزايا متقدمة في أرض قديمة فيها أنابيب مياه قديمة... الأنابيب الجديدة المتوفرة بالسوق لا تتناسب معها، لذلك ستضطر لعمل محول معقد ليتمكن النظام القديم والجديد من التكامل معا، أو تعديل التصميم كله ليتناسب مع الأنبوب القديم...

- منصة النشر المستهدفة -Target Deployment Platform-:

تعد هذه واحدة من أهم القيود التقنية التي ستؤثر على قراراتك المعمارية، خصوصا في المشاريع الجديدة -greenfield software system-، فالمنصة التي ستنشر عليها مشروعك لديها مجموعة من المحددات التي وضعتها والتي يجب عليك الامتثال لها، ولا يمكنك تجاهلها لأنك قد تضطر لبناء أو إعادة بناء أجزاء مهمة وكبيرة من المشروع، فعلى سبيل المثال، تخيل أنك قمت ببناء مشروع يعمل على جهازك الشخصي، يقوم على معالجة الصور ومن ثم حفظ الملفات في مجلد اسمه upload، ويتم حفظ البيانات داخل ملف sqlite، ثم ذهبت واستخدمت احدى ال PaaS لنشر هذا العمل، فما الذي سيحدث؟؟

بكل بساطة ستفقد بياناتك ^^، وستضطر لإعادة كتابة جزء من الشيفرة البرمجية ليتمكن من التواصل مع واحدة من خدمات التخزين، واستبدال طريقة حفظ البيانات والتعامل معها...

هناك الكثير من الأمور التي يجب أن تفهمها عن منصة النشر، مثل: القيود على حجم الشيفرة البرمجية واستهلاك الذاكرة وقوة المعالج ونوع أنظمة التشغيل المدعومة ونحو ذلك.

باختصار: إن المنصة المستخدمة للنشر ليست مجرد "مكان" لتشغيل مشروعك، بل هي وسيلة تقنية لديها محدداتها وقيودها التقنية التي يجب عليك أن تراعيها...

ملاحظة: ال Greenfield software system مصطلح شائع في عالم البرمجيات ويطلق على أي مشروع برمجي يتم بناؤه من الصفر؛ دون قيود نابعة من وجود نظام قديم أو شيفرة برمجية ما، وفي المقابل هناك مفهوم آخر وهو: Brownfield software system ويقصد به عملية التطوير على نظام قديم موجود بالفعل، وهنا أنت مقيد بالتقنيات القديمة والبنى التحتية المستخدمة... إلخ

- نضج التقنية -Technology maturity-:

يشير هذا المفهوم إلى مستوى استقرار التقنية -نضجها- المراد استخدامها في المؤسسة، وهذا يعد أحد القيود المعتبرة في اتخاذ القرارات المعمارية، فالمؤسسات عادة ما يمكن تصنيفها إلى:

- مؤسسات متحفظة: ويقصد بها المؤسسات التي تفضل تجنب المخاطرة وتختار التقنيات التي أثبتت فعاليتها على مر السنين، حتى لو كانت هذه التقنيات ليست الأحدث، فاهتمامها منصب على الموثوقية والاستقرار وحجم المجتمع الخاص بها.
- ومؤسسات تحب المخاطرة: ويقصد بها المؤسسات التي تحب المخاطرة واستخدام أحدث التقنيات بالرغم من كونها ليست ناضجة بشكل كامل... هذه التقنيات قد توفر ميزات قوية أو أداء ممتازا، إلا أنها تحمل مخاطر كبيرة، مثل: نقص الدعم، وعدم وجود مجتمع كبير للمطورين، واحتمالية وجود أخطاء غير معروفة.

هذا القيد سيفرض علينا كمعماريين أن نختار التقنية بحسب ميول الشركة وتفضيلاتها وبحسب طبيعة المشروع وخطورته، فمثلا قد تختار الشركة استخدام نسخة Laravel الإصدار الحادي عشر، بالرغم من كون أن الإصدار التجريبي الثاني عشر قد تم الإعلان عنه... والعكس صحيح ^^.

معلومة: هل سمعت بال Bleeding Edge Technology من قبل؟ هذا المفهوم هو الأسلوب الذي تستخدمه المؤسسات التي تحب المخاطرة -التي ذكرناها في هذا الموضوع-، وهو مفهوم يشير إلى استخدام تقنية جديدة جدا لدرجة أنها قد تتسبب في مشاكل لمن سيقوم باستخدامها أولا، والمعنى الحرفي لها هو معنى ساخر ^^ يقصد به: "هذه التقنية قد تتسبب لك في جرح يجعلك تنزف دما، فهل أنت مستعد لذلك؟"... ويتم تشبيه هذا المفهوم بمن يمسك سكينة حادة جدا لدرجة أنها قد تجرح من يمسكها،

وهذا المفهوم يقودنا لمفهوم آخر وهو ال Cutting Edge، وهو يشير إلى أحدث ما توصلت له التقنية، لكنها جاهز نسبيا ومستقرة وناضجة بما يكفي -بما يكفي وليس تماما- لاستخدامها في المشاريع التجارية، ويمكن تشبيهها بمن يمسك سكينة حادة جدا لكنها تستخدم بكفاءة ^^... وكمثال عملي، لو أردنا تصنيف ال Generative AI ضمن هذه المفاهيم، فأين سيكون؟

الإجابة على هذا السؤال ستكون حسب السياق، فلو قلنا أنك ستستخدم ال Gemini أو GPT-4 فيمكننا القول أنه Cutting Edge وليس Bleeding، لأنها متاحة للعامة والناس تستخدمها، وتعمل بشكل موثوق، ويستخدم في آلاف الشركات، وتقوم عليها شركات كبيرة وليست ناشئة لم تجرب أعمالها من قبل... لكنها بنفس الوقت ليست مستقرة تماما -Stable- لأن النماذج تتغير من فترة لأخرى، ولا يمكن التنبؤ بدقة في كيفية تصرفها في بعض الحالات ووجود الهلوسات، وحتى النتائج قد تتغير مع نفس المدخلات من فترة لأخرى لسبب ما...

أما لو كان السياق استخدام ال AI في المجال الطبي للمساعدة في عمليات جراحة الدماغ، فسيكون هذا Bleeding لأن التقنية هنا ما زالت جديدة جدا والأخطاء الناتجة عن ذلك قد تكون كارثية ولا يوجد أي ضمان على دقة النتائج ولا يوجد تجارب عملية ودراسات كافية حول الموضوع...

فما رأيك في هذه المفاهيم؟ ومن أي نوع أنت؟ وهل يختلف أسلوبك المفضل باختلاف المشروع وطبيعة المرحلة التي تعيش فيها؟ وكيف تتعامل مع التقنيات الجديدة والتي حلت مشاكل لم تكن محلولة من قبل في نظامك القديم، لكن لا

يمكنك ترقية النظام القديم لاستخدام هذه التقنية جديدة أو الإصدار الجديد؟

- البرامج مفتوحة المصدر -Open source-:

بالرغم من المزايا العديدة للبرامج مفتوحة الصدر، إلا أن العديد من الشركات تضع قيودا على استخدامها أو تمنع استخدامها، ويعود السبب في ذلك لعد أسباب من أهمها:

  - أن كثير من الشركات تريد مكانا "لتصرخ فيه" إن حصل أي فشل أو مشكلة، بحيث يمكنها أن تحمل كائن ما المسؤولية المترتبة على هذا الفشل، كما أن كثير من الشركات تحب أن تتعامل مع جهة يمكنها الاتصال بها للحصول على الدعم الفني اللازم...
  - غموض بعض التراخيص المستخدمة في المصادر المفتوحة مثل ال GPL، فلو كان لديك مشروع كبير وتم استخدام مكتبة صغيرة مثل ال GPL قد تجد نفسك أمام دعوى قضائية تلزمك بنشر الشيفرة البرمجية الخاصة بك لتصبح أيضا Open Source! كما أن النصوص القانونية الموجودة في تراخيص ال GPL لا يمكن فهمها عادة من قبل المطورين والمدراء، وإنما تحتاج لشخصية قانونية يمكنها البت باستخدامها من عدمه... وهذا ما يسمى بالغموض، أما إذا كانت التراخيص من نوع MIT و Apache 2 فالموضوع سهل وبسيط، ولا يوجد غموض.

يؤثر هذا القيد كما ترى على القرارات التقنية، فعلى سبيل المثال: قد يُجبر المعماري على استخدام منتج تجاري باهظ الثمن حتى لو كان هناك بديل مجاني ومفتوح المصدر بنفس الجودة أو أفضل، وفي بعض الحالات، قد يكون البديل التجاري ما هو إلا

نسخة مغلفة من نفس البرنامج مفتوح المصدر، وهذا يعيدنا لفكرة ستجدها متجددة مع كل نقطة أو موضوع نذكره، وهي أن القرارات المعمارية لا تُبنى دائما على أساس تقني بحت، بل تتأثر أيضا بعوامل أخرى، مؤسسية ومالية...

- علاقات البائعين -"Vendor "relationships:-

إن العلاقات الشخصية أو التجارية تؤثر بشكل كبير على الجودة الخاصة بالمشروع وحتى على الكفاءة التقنية، وذلك لتأثرها بالقرارات الناتجة عن وجود هذه العلاقات... فمثلا لو كان المسؤول عن اتخاذ القرارات التقنية لديه علاقة شخصية أو عاطفية مرتبطة بشركة ما، فيمكن أن يجبر الفريق التقني على استخدام التقنيات المقدمة من هذه الشركة دون غيرها، ولو لم تكن الأفضل... وكذلك لو كان للمؤسسة علاقة تجارية وطيدة بينها وبين أحد البائعين، فغالبا ما ستقوم الشركة بإجبار الفريق على استخدام التقنية المرادة من هذا البائع لا من غيره وإن لم تكن الأفضل... وبهذا نعود للنقطة التي ذكرناها سابقا: وهي أن القرارات التقنية كثيرا ما ترتبط بعوامل خارجية غير تقنية... وكمثال عملي على هذه العاطفة الجياشة بين البائعين والشركات ^^ ما حصل عند ظهور الحوسبة السحابية، فكثير من الشركات انتقلت إلى Azure ليس لأنه أفضل من AWS أو GCP، بل لأن الشركة كانت تستخدم Windows Server و SQL server منذ فترة طويلة، وهذا جعلها تفضل الانتقال ل Azure -أي التفضيل بسبب الثقة-، كما أنها كانت ترغب من الاستفادة من الخصومات السخية التي قدمتها لعملائها...

- الفشل السابق -Past failures-

هذا القيد يمثل أحد القيود الأساسية التي قد تدمر معماريتك قبل أن تبدأ ^^؛ حتى لو لم تكن هناك علاقة بين ما حصل من فشل في الماضي وبين الواقع الحالي!

ويقصد بهذا القيد أن حصول فشل ما عند استخدام تقنية معينة أو أسلوب معين قد يبقى في ذاكرة الفشل الموجودة في عقولنا، مما يدعونا لسلوك غير عقلاني يمنع من استخدام هذه التقنية أو الأسلوب مجددا حتى لو كانت هي الحل الصحيح لهذه المشكلة، يقول Simon في كتابه حول هذه النقطة: «حوالي سنة 2000 ميلادي، ذهبتُ إلى بنك مع مقترح لبناء solution باستخدام الـ Java RMI -وهي تقنية تتيح استدعاء الدوال عن بُعد بين الـ Java virtual machines-، لكن هذا الاقتراح قوبل باعتراض كبير لأن البنك "جرّب ذلك من قبل ولم ينجح"، وكان ذلك نهاية التصميم! ولم تغيّر أي مناقشة تقنية رأيهم، وبسبب ذلك تم حظر استخدام الـ Java RMI في هذا السياق، بسبب قيد الفشل السابق! وانتهى بنا الأمر إلى بناء إطار عمل يرسل الـ Java Object عبر الـ HTTP إلى مجموعة من الـ Java Servlets، وهذا في الأساس حل بديل لإعادة اختراع نفس العجلة».. أتدرك ما الذي قاله Simon هنا؟ فعليا ما يعنيه كلامه هو كتابة حل لمشكلة تم حلها فعلا من قبل وبشكل موثوق، لكن بسبب رفض البنك لهذا المقترح اضطر الفريق لإعادة اختراع العجلة، وهذا الحل البديل غالبا كان أكثر تعقيدا وأقل كفاءة، وأكثر تكلفة في الصيانة... وهنا سيكون البنك وكأنه اختار الـ Bleeding Edge مع أنه من النوع "المحافظ"، لكن القرار غير العقلاني ستكون نتائجه الظاهرية عقلانية ويخفي تحته نتائج قد تكون كارثية أو

خطرة...

ملاحظة مهمة: هناك فرق كبير بين قيد الفشل السابق وبين التعلم من الفشل، وهذا يظهر من خلال المنهجية والهدف والنتيجة!

- في قيد الفشل السابق: تكون المنهجية المتبعة هي اتخاذ قرار عاطفي وغير عقلاني، ولذلك يتم رفض التقنية بشكل قاطع لمجرد أنها فشلت في الماضي، دون تحليل الأسباب... ويكون الهدف هنا هو تجنب الألم أو الخوف من تكرار نفس التجربة السلبية وليس تحسين الأداء... والنتيجة هنا غالبا ما تكون تكرار للأخطاء السابقة أو اختيار حلول بديلة أكثر تعقيدا وأقل كفاءة...
- في التعلم من الفشل: تكون المنهجية المتبعة هي اتخاذ قرار عقلاني ومنهجي، ففيه يتم تحليل أسباب الفشل بشكل دقيق، ومن ذلك: هل كانت المشكلة في التقنية نفسها؟ أم في طريقة التنفيذ؟ أم في ظروف المشروع؟... لذلك، يكون الهدف هو فهم الأخطاء لتجنبها في المستقبل وتحسين الأداء، ونتيجة ذلك سنحصل غالبا على أنظمة أقوى وأكثر مرونة، متجنبين الأخطاء السابقة بشكل واع فعلا...

- الملكية الفكرية الداخلية -Internal Intellectual Property:

الملكية الفكرية الداخلية هي البرمجيات والمكتبات وال Frameworks والأنظمة التي طوّرتها الشركة بنفسها وتستخدم بداخلها، وهنا يظهر القيد، فقد تفرض عليك هذه الشركة عند تصميمك للنظام أن تستخدم هذه التقنيات أو المكتبات الداخلية، حتى لو

كان هناك ما هو أفضل منها، باختصار: إن "ما تملكه الشركة داخليا يجب أن يستخدم"!

إن مشكلة هذا القيد تكمن في تقييد اختيار المطورين، والتي غالبا ما ستؤدي إلى إعادة اختراع العجلة أو إهدار الوقت والجهد لحل مشاكل شائعة قد تكون محلولة فعلا في حلول أخرى! وهناك حاجة لمعرفة تفاصيل هذا الحل وإمكانية توافقه مع النظام الجديدة وكيف سيتم التوافق، وهل صمم بطريقة يمكننا من خلالها الربط أو التطوير أم أنه مجرد نظام مغلق... إن هذه المشكلة تشبه شراء سيارة حديثة دون محرك، لأن لديك محرك سيارة جدك الذي ما زال يعمل ^^...

ملاحظة: إن الملكية الفكرية الداخلية ليس مشكلة في ذاتها! وإنما تصبح مشكلة وقيدا إذا كان استخدامها نابعا من قرار أعمى! حتى لو وجدت أدوات أفضل وأكثر كفاءة! في حين أنها تعد مزية ونقطة قوة إن كان هذا المنتج الداخلي يقدم ميزة تنافسية غير موجود عند غيره، أو يحل مشكلة معقدة لا يتوفر حل لها باستخدام الحلول المتاحة، أو حين يتعلق الأمر في الأمان أو القدرة على التخصيص والتحكم التي قد لا تتوفر من خلال ال vendors المتواجدين... وبهذا فهي تعد أداة قوية إذا تم استخدامها بشكل استراتيجي، لكنها تصبح عائقا وقيدا يمنع التطور إذا تحولت إلى قاعدة لا تقبل النقاش!

ال People constraints:

إن الموارد البشرية تؤثر بشكل كبير على القرارات المعمارية، وهي عامل مهم لا يمكن تجاهله، فكيف ستقوم بتصميم نظام بمعمارية معينة والفريق الموجود لا يستخدم التقنيات الموجودة في هذه المعمارية أو لا يعرف شيئا عنها! مثلا: إذا كان الفريق خبيرا بتصميم المواقع الإلكترونية في لغة ال PHP، فمن غير المنطقي أن أطلب منهم بناء الموقع الجديد بلغة الجافا! إن تجاهل هذا القيد قد يؤدي إلى بناء نظام لا يستطيع أحد فهمه أو صيانته، وهذا باعتبار أن النظام سيكتمل أصلا!

لذلك، هذا القيد يشمل عدة جوانب، وهي:

- حجم الفريق ومهاراته: هل لديك عدد كاف من المبرمجين؟ وهل يمتلكون المهارات اللازمة للتقنيات التي تريد استخدامها؟
- سهولة التوظيف والقدرة على التوسع: هل يمكنك توظيف متخصصين جدد بسرعة إذا لزم الأمر؟ وهل لديك قدرة على تحمل ما ينتج عن توسعة الفريق من تبعات؟
- الدعم والتدريب: هل يمكنك الحصول على تدريب أو استشارات خارجية لتعزيز مهارات الفريق؟
- فريق الصيانة: هل يمتلك فريق الصيانة المستقبلي نفس مهارات فريق التطوير؟ فإذا كان فريق الصيانة لا يمتلك نفس المهارات التي يتقنها فريق التطوير، فإن هذا قد يؤدي إلى حدوث مشاكل أثناء عمليات الصيانة... كما أن وجود مهارات مشتركة فيما بينهم يغني عن التكاليف المترتبة عن إعادة التدريب أو التوظيف ونحو ذلك.

ال Organisational constraints:

إن القيود التنظيمية من العوامل غير التقنية المطالبين بالاهتمام بها، لأن أثرها كبير على القرارات التقنية! إن هذه القيود لا تتعلق بالأشخاص أو التقنية المستخدمة؛ بل تتعلق بالسياسة أو البيئة الداخلية للشركة... فكم مرة تم إجبارنا على استخدام تقنية معينة أو تصميم معماري معين لإرضاء إدارة معينة أو قسم معين في الشركة؟!

هذا السؤال يعبر عن أحد القيود التنظيمية الموجودة على أرض الواقع، وهو: السياسات التنظيمية -Organisational Politics-.

والآن، هل النظام الذي تقوم بتصميمه سيكون من أجل تقديم حل تكتيكي أم حل استراتيجي؟ إن إجابة هذا السؤال مهمة أيضا، لأن الإجابة عن هذا السؤال قد تضيف قيودا على معماريتنا أو تزيلها، وحتى نعرف لماذا، علينا أن نفهم ما يعنيه كل من:

- الحلول التكتيكية: هي الحلول قصيرة الأجل، والتي تبنى بسرعة لحل مشكلة فورية أو عاجلة؛ لذلك تكون القيود هنا أقل في الغالب، وقد يُسمح لك بتجاهل بعض سمات الجودة مثل قابلية الصيانة على المدى الطويل في سبيل إنجاز المشروع بسرعة.

- الحلول الاستراتيجية: هي الحلول طويل الأجل، والتي تبنى ليكون أساسًا من أساسات الشركة لسنوات عديدة؛ لذلك هذا النوع من الحلول يأتي مع قيود أكثر صرامة، فمثلا يتم الاهتمام بسمات الجودة مثل قابلية التوسع وقابلة الصيانة، مما يتطلب جهدا أكبر

في مرحلة التصميم.

هل كل القيود -ال Constraints- سيئة؟

غالبا ما يُنظر إلى القيود على أنها شيء سلبي، لكن الحقيقة أن الأمر ليس كذلك دائما! بل على العكس؛ فهي غالبا ما تُفرض لسبب وجيه، وتساعد في توجيه القرارات المعمارية والهندسية... -راجع ما تحدثنا عنه في الملكية الفكرية الداخلية على سبيل المثال-.

هناك العديد من الفوائد الناتجة من التقييد، من ذلك: "الحد من التعقيد"، فالقيود تقلل من عدد الخيارات المتاحة؛ مما يسهل عملية اتخاذ القرارات! فعلى سبيل المثال: وجود قائمة محددة بالتقنيات المعتمدة يمنع المطورين من قضاء وقت طويل في البحث عن تقنيات جديدة، ويضمن أن النظام يمكن دعمه وصيانته بسهولة، كما أن هذه القيود قد تضمن الاتساق، فيتم تقييد الفرق لاستخدام أدوات ومنهجيات موحدة، وهذا كله مما يسهل التعاون بين المطورين والفرق، ويقلل من التعقيد على المدى الطويل ^^...

لكن، ألم يلفت انتباهك أمر ما؟ ألا تجد أن ما نتحدث عنه نحن الآن من ضوابط وسمات ومفاهيم ما هي إلا مجموعة من القيود التي تمت صياغتها بشكل جميل يخفي الشكل الحقيقي للقيود؟! ألا ترى أننا زينا الأصفاد بالورود؟!

نعم يا صديقي، إن المعمارية التي نتحدث عنها الآن ما هي إلا عملية إدخال للقيود! لكنها بطريقة منهجية علمية قائمة على مجموعة من المقايضات والأنظمة والتشريعات والسياسات...

إلخ، فمثلا: إن استخدام نمط معماري معين دون آخر أو تقنية محددة دون أخرى ما هي إلا قيد تم فرضه على المطورين! لكن هذا القيد سيساعد في الحفاظ على المشروع وحياته، وينظم العمل بين الأقسام المختلفة، وبين المطورين أنفسهم... وبهذا كان قيدا مفيدا ^^، لذلك: ستجد أن بعض القيود قد تحد من الإبداع على المدى القصير، لكنها ستساهم في بناء أنظمة أكثر استقرارا وأكثر قابلية للصيانة على المدى الطويل.

ال Constraints can be prioritised

إن القيود التي تحدثنا عنها ليست بذات الأهمية والأولوية دائما، فهذه القيود قد تكون لها أولويات مختلفة بحسب وضع المشروع أو سياقه وزمانه ومكانه... وهذا الأمر يمكن أن يستغله المعماري الناجح أو المطور الخبير لمصلحته حتى يتخلص من بعض القيود التي ستعود بالفائدة على جميع الأطراف... وهذا يشبه ترتيب الأولويات للمتطلبات الوظيفية، فليست كل المزايا المطلوبة في المشروع لها نفس الأهمية، وبذلك يمكننا الاستغناء عن ميزة في مرحلة من مراحل المشروع في مقابل الانتهاء من المزية الرئيسية... وعلى نفس النمط، إذا كان قيد الوقت هو أهم قيد، فيمكننا التخلي عن قيد من قيود التقنيات المستخدمة واستبدالها بأخرى تعيننا على تسليم المشروع بشكل أسرع...

لقد ذكر Simon مثالا عايشه بنفسه، وهو نظام أو مشروع لقياس وتحليل المخاطر التقنية التي قد يتعرض لها البنك، وكان لدى البنك قيدين أساسيين، وهما:

١. قيد التقنية: قدم البنك قائمة بمجموعة التقنيات المعتمدة مثل Java EE

٢. قيد الوقت: تسليم المشروع في موعد محدد وبشكل صارم
في هذه الحالة، أدرك Simon وفريقه أن قيد الوقت كان أكثر أهمية للبنك من قيد التقنية، وبناء على ذلك قدموا مقترحا يتجاوز قيد التقنية من خلال اقتراح استخدام تقنيات من خارج القائمة، لضمان تلبية قيد الوقت، وكانت النتيجة هي قبول المقترح، وهذا يدل على أن التنازل عن قيد أقل أهمية يمكن أن يساعد في تحقيق قيد أكثر أهمية، وبهذا قد تكون القيود عقبات يجب أن نجد طريقة للتعامل معها في بعض الأحيان، وفي أحيان أخرى يمكننا الموازنة فيما بينها...

ال Principles:

«إن ما يجب عليك فعله أو فرض عليك فعله، يختلف عما تختار أنت فعله أو فرضه»
إن هذا الاقتباس الذي قرأته هو عملية إعادة صياغة لمفهوم مهم، مع القدرة على التفريق بين هذا المفهوم وآخر قريب منه... هذان المفهومان هما: المبادئ -Principles- و القيود -Constraints-!

لقد تحدثنا عن القيود في الباب السابق، لذلك فإن أول جزء من الاقتباس يشير إلى القيود، في حين أن الجزء الثاني من الاقتباس يشير إلى المبادئ ^^... وبهذا أعتقد أن المفهوم صار واضحا، وصار معلوما الفرق بينهم ^^

ومع ذلك، ما هي المبادئ؟ ومن يختارها؟ ولماذا سنختارها أساسا؟ ولماذا سنفرضها؟ وما هي أنواعها... إلخ؟

إن المبادئ تمثل مجموعة القواعد أو التوجيهات التي يختار الفريق تبنّيها من أجل إدخال أساليب قياسية تضمن الاتساق والجودة في الطريقة التي تُطور بها البرمجيات، وهناك العديد من المبادئ الشائعة في هذا العالم، منها ما يتعلق بعملية الـ Development ومنها ما هو متعلق بال Architecture، وكلاهما معا يمثلان العملية الصحية الكاملة لتطوير البرمجيات...

ال Development Principles:

عند حديثنا عن المبادئ المهمة في عالم تطوير البرمجيات، غالبا ما يتراود إلى أذهاننا جميعا القواعد أو الأساليب التي سنقوم باعتمادها أثناء برمجة وتطوير التطبيقات مثل:

- ال Coding standards and conventions: أظن أن هذا أول مبدأ سيخطر على بال أي مبرمج عند الحديث عن مبادئ التطوير، وهو وضع قواعد محددة لكتابة الشيفرة البرمجية مثل تسمية المتغيرات وتنسيق الشيفرة البرمجية ونحو ذلك، وكل ذلك يهدف إلى جعل الشيفرة البرمجية قابلة للقراءة بشكل سهل وبنسق موحد لجميع الأعضاء بالفريق. -وأرى كثيرا من المطورين يتوقفون عند هذا المبدأ فقط... وتنتهي المبادئ بالنسبة لهم، لهذا يجب عليك الانتباه لما هو قادم ^^-.

- ال Automated unit testing: إن عملية أتمتة الاختبارات لم تعد رفاهية، ولم تعد تعني أننا نتحدث عن جزء يقوم باختبار الشيفرة البرمجية فقط... بل نتحدث عن جزء أساسي من عملية تطوير البرمجيات وما يدور حولها، وتعد هذه من المبادئ المهمة لأن

الشيفرة البرمجية لا ينبغي لها الذهاب إلى ال Production أو إلى Branch محمية دون أن يكون ما سيتم تصديره وإنتاجه يعمل بشكل صحيح! لذلك، وحتى نتأكد من أنه صالح ونقلل من نسبة الخطأ يجب أن تكون الاختبارات موجودة لتثبت صحته = وهذا سيقلل من عدد المشاكل الممكنة وخطورتها، وهذا يرتبط فيه أيضا مجموعة من المبادئ الأخرى مثل ال TDD وال CI...، ووجود هذا المبدأ مرتبط ارتباطا وثيقا بزيادة الثقة عند العمل من قبل المبرمجين، لأن الاختبارات الموجودة غالبا ما ستحمينا، وسيستطيع الفريق أن يعمل مع بعضه البعض دون تخريب أعمالهم عن قصد أو دون قصد، فإذا حصل خلل ما؛ فسيتم ملاحظته مباشرة وقبل أن تكبر الفجوة... إلخ.

وقبل أن ننتقل من هذه النقطة، أحب أن أشير إلى أن هذا المبدأ يشبه تماما المبدأ الذي يتبعه مهندسوا السلامة في الإنشاءات المعمارية، فهم لا يسمحون بدخول عامل دون ارتداء الخوذة أو أن يتدلى دون حبل أمان؛ مع أن هذا العامل قد لا يصاب بأي مكروه -وهو الأصل الغالب-، لكن إن حصل مكروه فسيكون أثره وخيما وأكبر من دون وسائل السلامة؛ في حين أن وجود وسائل السلامة قد تقلل من خطر الإصابة أو تحميك منها مسبقا... وهذا من باب الأخذ بالأسباب وحفظ النفس التي كرمها الله -جل في علاه-، وعلى نفس هذا النمط من التفكير فكر في هذا المبدأ في عالم البرمجيات...

- ال Static analysis tools: ويقصد بها مجموعة الأدوات التي يمكن استخدامها لتحليل الشيفرة البرمجية والتأكد من أنها تحقق معايير الجودة المتفق عليها؛ أكانت على مستوى

ال Code أو Test... ولعل من أشهر الأمثلة التي نعرفها جميعا ال ESLint...، بحيث يمكننا التحكم والتأكد من أن الشيفرة البرمجية لها نسق واضح ومعلوم، ويصل هذا الأمر لمستوى الفواصل والمسافات، وال Single Quote ومكان استخدامها، وال Double Quote ومكان استخدامها... وهذا المبدأ يعني أن المطور لن يكون قادرا على إدخال تعديلاته على الشيفرة البرمجية إلا إذا كانت شيفرته البرمجية نظيفة وتتوافق مع القواعد المتفق عليها ^^... ومن جميل هذا الموضوع أن هذا الأمر يمكن أن يتم بشكل تلقائي وأن يتم التصحيح بشكل تلقائي، بمجرد كتابة القواعد! وهذا يذكرنا أيضا بفضل الله -سبحانه وتعالى- ونعمته علينا مع تطور هذا العلم وسيرنا في مراحله الزمنية المختلفة، ولا أدري إن كنت أدركت العمل على المحررات النصية القديمة أم لا! لكن يكفيك أن تجرب العمل والتطوير من خلال Notepad -مثلا- لتشاهد الفرق مع ما نعيشه اليوم من وسائل وأدوات، فسبحان من علم الإنسان ما لم يعلم.

ال Architecture Principles:

بعد حديثنا عن مجموعة من المبادئ الخاصة بالتطوير في الباب السابق، ننتقل الآن إلى مبادئ أخرى قليل من المطورين من يفكر فيها بشكل افتراضي...

هذه المبادئ صممت لتوجيه القرارات التصميمية لمعمارية التطبيقات، لضمان الاتساق وجودة النظام -لاحظ أن ضمان الجودة والاتساق هو هدف هذه المبادئ، لذلك كان التعريف في شطره الثاني متشابها مع ال Development Principle-... وبهذا يمكننا أن نستنتج الاختلاف في المبادئ المعمارية عن مبادئ التطوير، وذلك بأنها تتعلق بكيفية هيكلة

النظام ككل، وليس الجزء المتعلق بالشيفرة البرمجية فقط! ومن هذه المبادئ -أمثلة متنوعة متعددة-:

- ال Layered Architecture: هذا المبدأ يقسم النظام إلى طبقات مستقلة، مثل طبقة واجهة المستخدم -ال -UI، وطبقة منطق الأعمال -ال Business Logic-، وطبقة الوصول إلى البيانات -Data Access-...

إن هذا الفصل بين الطبقات هدفه تعزيز المرونة؛ فمثلا يمكن تغيير قاعدة البيانات دون التأثير على طبقة منطق الأعمال أو طبقة واجهة المستخدم، لأن واجهة المستخدم تتعامل مع طبقة منطق الأعمال وطبقة الأعمال تتعامل مع ال abstraction الخاصة بطبقة ال Data Access، وذلك للحصول على ال Data، ولا تتعامل هذه الطبقة مباشرة مع قاعدة البيانات!

إن عملية التقسيم هذه تجعل من الطبقة العليا تستخدم الطبقة السفلى، لكن الطبقة السفلى لا يمكنها استخدام العليا، فكل طبقة لا تعرف شيئا عن الطبقة التي تعلوها... ومن هنا سنتمكن مثلا من تغيير قاعدة البيانات من MySql إلى PostgreSQL إذا احتجنا لذلك، ودون الحاجة لتعديل طبقة منطق الأعمال [٨٨].

ملاحظة طبقة الأعمال -ال Business logic layer- يقصد بها الطبقة التي تحتوي في ثناياها على القواعد والعمليات -الحسابات- والقرارات الذكية التي تميز النظام، وليست مجرد المنطق التجاري بالمعنى الضيق! وبذلك فإن هذا المفهوم يمكن تشبيهه بالدماغ لدى الإنسان، فعيون الإنسان تمثل الواجهة التي يرى بها وتنقل المعلومات إلى

الدماغ... بعد ذلك يقوم الدماغ بتحليل البيانات المتعلقة بالصورة التي أرسلتها العين والتحقق منها ودراستها وإصدار قرار ما يتناسب مع ما أرسلته العين... طريقة التفكير هذه = منطق الأعمال، لنأخذ مثالا تقريبيا: لو رأيت أسدا جائعا، فإن الدماغ سيقوم بالتحقق من أن ما تراه العين أسد جائعا فعلا وليس مجرد تمثال خشبي!، ثم سيقوم بتحليل الموقف وحساب الاحتمالات الممكنة للنجاة، ثم اتخاذ القرار بالمواجهة أو الهرب أو الاستيقاظ من النوم :P ^^... هذا هو منطق الأعمال، الرؤية تمثل ال UI والدماغ وما قام به يمثل منطق الأعمال، وما بعد اتخاذ القرار يذهب لطبقة اخرى، ونفترض حينها أن الركض يمثل ال Data Access ^^.

- ال Placement of business logic: هذا المبدأ يركز على ضرورة وضع منطق الأعمال في مكان واحد، وهذا المبدأ مهم لأنه سيقودك لاختيار المكان حسب طبيعة المشروع، وتأثيره السلبي كبير لو لم يتم تطبيق هذا المبدأ، فعلى سبيل المثال: في تطبيقات الهاتف المحمول، غالبا ما يتم وضع المنطق -الدماغ- على ال server، في حين أن الأنظمة التي ستتكامل مع أنظمة قديمة موجودة بالفعل، فقد يتم الحفاظ على منطق الأعمال في النظام القديم! تحتاج إلى توضيح أكثر؟ حسنا ^^
تخيل أن لديك تطبيق للبنك على هاتفك المحمول، وقمت بسحب مبلغ ألف دينار من حسابك، فأين تتوقع أن تتم عملية التحقق والتحليل واتخاذ القرار؟ إذا كانت على الجهاز نفسه فهذا يعني أنك انتهكت واحدة من أهم نقاط الأمان، وصارت عملية اختراق هذا النظام وتجاوزه سهلة... لكن إن كان على ال server فهذا الأفضل بكل تأكيد...

والآن تخيل أن لدينا شركة لبيع المنتجات الصناعية قائمة منذ ٢٠ سنة، وصممت بلغة برمجة قديمة، والنظام معقد وفيه الكثير من العمليات، وقمنا ببناء نظام ويب أو هاتف محمول جديد لهذه الشركة، فهنا الأفضل أن يتكامل هذا النظام مع النظام القديم لا أن يحتوي بداخله نفس المنطق الموجود بالقديم، وهذا يضمن أن النتائج الجديدة متوافقة مع المنطق الذي قد لا نفهمه، ونتجنب التكرار، ونبقي على المنطق في مكان واحد، وبذلك إن عدلت إحدى القواعد على النظام القديم فسينعكس هذا التعديل أيضا بشكل تلقائي على النظام الجديد...

- ال High cohesion, low coupling, SOLID: تمثل هذه مجموعة من المبادئ التي تركز على بناء مكونات برمجية ذات مسؤولية واحدة محددة وواضحة -وهذا ما يطلق عليه ال High cohesion-، وفي نفس الوقت مستقلة قدر الإمكان عن بعضها البعض -وهو ما يطلق عليه ال low coupling-... هذه المبادئ جنبا إلى جنب مع ال SOLID principle ستقودك لتحقيق ال Separation of Concerns، مما يسهل من صيانة النظام ويزيد من قابلية التعديل وإعادة استخدام المكونات...
- ال Stateless Components: إن هذا المبدأ واحد من أهم المبادئ التي ستحتاجها عند تصميمك للأنظمة التي تحتاج إلى قدرة كبيرة أو قابلية عالية على التوسع! لأن وجود مكونات stateless يساعد على التوسع الأفقي -Horizontal Scaling-، والسبب في ذلك يعود للقدرة الكبيرة على عمل نسخ متعددة من هذا المكون... إذا كان النظام قابلا للتوسع بشكل كبير، فيجب إعلام جميع أعضاء الفريق بذلك؛ حتى يتم بناء جميع المكونات على نفس هذا النمط قدر الإمكان لتجنب أي مشاكل غير متوقعة وللهروب من ال Bottlenecks المستقبلية [٨٨].

للمزيد من التفاصيل حول هذا الباب يمكنكم مراجعة كتاب System Design - من مستخدم إلى مليون مستخدم، الفصل الأول.

- ال Stored Procedures: قد تتعجب من هذا المبدأ ^^، وقد تظن أنه يحثك على استخدام ال Stored Procedure للتعامل مع قواعد البيانات! لكن ليس هذا المقصود من هذا المبدأ! بل المقصود هو أن تستخدم ال Stored Procedure في المشروع أو لا تستخدمه! وهذا يعني أن المطورين عليهم أن يتفقوا على استخدام أسلوب واحد للتعامل مع قواعد البيانات، إما من خلال ال Query عن طريق ال ORM مثلا أو من خلال ال Stored Procedures!

إن الهدف هنا هو وضع سياسة عمل تضمن الاتساق، وبذلك يكون تركيز هذا المبدأ على الاتساق لا التقنية، فمثلا يمكن أن تصاغ قاعدة لفريق المطورين: "جميع العمليات المعقدة تُكتب كـ Stored Procedures" أو "كل تفاعل مع قاعدة البيانات يتم عبر ORM".. قد يكون هناك استثناءات لبعض الحالات الضرورية، والتي يكون فيها استخدام الأسلوب الآخر مقبولا، مما يدعونا لكسر القاعدة في تلك الحالة! -بعد أن يتم نقاش هذه الحالة وتوثيقها-... لكن لا يمكن أن نقبل وجود مطورين ينجزون أعمالهم من خلال ال ORM وآخرون من خلال ال Stored Procedure في نفس المشروع دون سياسة واضحة تضبطهم جميعا! لذلك يقول Simon -بتصرف-: "أحب ذلك أو اكرهه، هذا ليس مهما، لكن المهم هو أن يكون لدى الفريق سياسة واضحة وثابتة"... لذلك، حتى لو سمعت مبرمجين يقولون: "هذه التقنية أفضل شيء في هذا العالم!" وسمعت آخرين يقولون عنها: " هذه أسوأ تقنية في العالم، لا أريدها في مشروعي أبدا!"... فهذا الأمر ليس مهما لأن لكل أسلوب منهما سلبياته وإيجابياته، وكل فريق

منهم محق من الزاوية التي نظر بها؛ لكن القرار الصائب ينبغي أن يكون من خلال النظر إلى ما نحتاجه حقا أو ما يناسبنا حقا ثم تعتمده أسلوبا ومبدأ لنا.

- ال Domain model - rich vs anaemic: قبل البدأ بشرح المبدأ لنأخذ فكرة عامة أولا، إن ال Domain Model هو abstraction لمنطق عمل النظام برمجي، ويمثل طريقة لتنظيم البيانات والسلوكيات المتعلقة بمجال معين، أي أنه يمثل خريطة تُوضح كيف تتفاعل ال Entity في النظام مع بعضها البعض، وهو أساسي لبناء أنظمة قوية وقابلة للتطوير... وكأمثلة واقعية، لو افترضنا أن لدينا نظام تجارة إلكترونية فإن ال Domain Model سيتكون من مجموعة من ال Objects التي تمثل Entity حقيقية في هذا المشروع، وبذلك سيكون لدينا عدة Objects مثل ال Product وال User وال Order ^^... وكل Object فيها يحتوي على:

١. البيانات -Data-: وهي مجموعة الخصائص التي تصف كائن ما مثل productName و productPrice... إلخ
٢. السلوك -Behavior- وهي مجموعة ال Method التي تحدد ما يمكن لل Object فعله مثل addToCart() أو CalculatePrice()... إلخ.

وهنا سيظهر لنا مفهومين، وهما ال Rich Domain Model وال Anemic Domain Model، ويقصد بال Rich Domain Model وهو النموذج الذي يجعل ال Object يحتوي على كلا المكونين، البيانات والسلوك معا، أي أن هذه ال

Objects لن تقوم بتخزين البيانات بداخلها فقط، بل ستحتوي أيضا على مجموعة من ال Method التي تنفذ العمليات المتعلقة بهذه البيانات! لذلك سمي بالنموذج الغني ^^، في حين أن ال Anemic Domain Model يمثل ال Objects التي تخزن البيانات فقط، ويمكنك تخيلها كحاوية للبيانات فقط! وهنا يتم فصل منطق الأعمال والسلوك ويوضعان في Classes أخرى يطلق عليها Services! ومن الأمثلة على ذلك مثلا إنشاء Class Account يحتوي بداخله على ال balance و accountNumber، بينما يوجد هناك AccountService يحتوي على ال method مثل ال withdraw()...

بعد هذه المقدمة الجميلة، ما هو المراد من هذا المبدأ إذا؟!
بكل بساطة نفس ما أردنا تحقيقه من المبدأ السابق! وهو الاتساق لا التقنية! أي يجب على الفريق اختيار أحد النموذجين والالتزام به في جميع أجزاء النظام، لأن عدم الاتساق سيؤدي إلى نظام غير متجانس يصعب فهمه وصيانته.

- استخدام ال HTTP Session: هذا المبدأ يشير إلى أن قرار استخدام ال HTTP session لتخزين المعلومات يجب أن يكون مدروسا بناء على استراتيجية التوسع المتوقعة! فإذا كان النظام بحاجة إلى التوسع، فقد يكون من الأفضل تجنب استخدام الـ session والانتقال لأسلوب آخر مثل Sessionless، ولعل أشهر مثال على ذلك ال JWT، أما الأسباب التي تدعو لتجنب استخدام الـ Session فهي: زيادة التعقيد بشكل كبير، ومشكلة ال Session-backed objects والتي تتمحور أن ال Session

الخاصة بالمستخدم قد تخزن على ال server 1 ويذهب الطلب إلى ال server 2، فإذا كانت ال session مخزنة فقط على ١ فإن العملية ستفشل، وبذلك نحن مطالبون بحل هذه المشكلة أيضا من خلال استخدام ال Sticky Session أو ال Session Replactions... وغيرها

ومع ذلك، يبقى قرار استخدام ال Session من عدمه قرار معماري يرتبط بالمشروع ومجموعة المقايضات التي سيتم دراستها، ويبقى الجزء المهم هنا أن يتم تطوير المكونات بنفس الآلية والنسق التي تم الاتفاق عليها مع الاتفاق على سياسة واضحة لحل المشاكل المتعلقة بالأسلوب المحدد...

- ال Always consistent vs eventually consistent: هناك نوعان من الاتساق: اتساق دائم وفوري -Always Consistent- واتساق متأخر -Eventually Consistent-، ويقصد بالاتساق هو مدى تطابق البيانات في النظام! أي أن الاتساق يضمن أن جميع نسخ البيانات متطابقة أو متوافقة، وأن أي تحديث يتم على نسخة منها سيتم تطبيقه على جميع النسخ الأخرى، ومن هنا نستنتج أن هناك حاجة في بعض الأنظمة للحصول على اتساق فوري ودائم بين جميع النسخ مثل أنظمة البنوك! فلا يمكن تقبل فكرة سحب الرصيد مثلا ثم الانتظار ٥ ثواني حتى يتم تحديث هذه المعلومة في جميع النسخ! في حين يمكن تقبل ذلك في موقع قبيلة مقابل الحصول على أداء أفضل وقابلية للتوسع أكبر! والمهم هنا أن عملية نشر أي منشور يجب أن تصل للجميع وأن تصبح البيانات متطابقة في نهاية المطاف، ولا يشترط بالضرورة أن يكون هذا في نفس اللحظة!

إن هذا المبدأ مهم جدا في الأنظمة الكبيرة والموزعة، ويعد من سمات الجودة الأساسية التي تؤثر على القرارات المعمارية... وكما هي العادة فاختيار نوع الاتساق المطلوب يمثل trade-off بين الأداء والقابلية للتوسع من جهة، ودقة البيانات من جهة أخرى!

احذر من نفخ ال Best Practice!

قد تتعجب من هذا العنوان، لكن هذه حقيقة مهمة! احذر من نفخ ال Best Practice! إن المبادئ المعمارية التي تحدثنا عنها والتي تبدو مثالية وتقدم حلولا سحرية للوهلة الأولى، في حقيقتها ليست سوى حلولا تتناسب مع المواقف المحددة بحسب سياق المشروع! أي أنها قد تكون حلا سحريا في مشروعك وقد تكون اللعنة التي تقتل روح العمل وتأخر التسليم في مشروع آخر! لذلك يجب أن يتم تطبيق واختيار المبادئ بحكمة بناء على سياق المشروع الذي بين أيدينا!

إن السياق الذي يتواجد فيه المشروع -Project Context- يجب أن يكون الأساس لأي تصميم معماري! ويجب أن يكون حجم النظام وتعقيده والقيود التي تفرض مثل الوقت والميزانية هي التي تحدد المبادئ التي يجب اعتمادها! وهذا سيدعو أي معماري أو خبير للاستماع للتغذية الراجعة من الفريق، خصوصا إذا تعلق الأمر بمبادئ صارت تمثل عائقا أمام سير العمل، ففي هذه اللحظة عليك العودة لإعادة تقييم هذا المبدأ واتخاذ الإجراء المناسب بناء على نتيجة التقييم...

والآن، لماذا علينا الحذر من نفخ ال Best Practice؟ ولماذا لا تكون هذه المبادئ دائما هي الأفضل؟!

الجواب على هذه الأسئلة المهمة يمكن تجزئته إلى:

١. المبالغة في التعقيد: لأن تطبيق مبادئ معقدة مثل استراتيجية الطبقات -layered architecture- على مشروع بسيط وقصير الأجل -tactical solution- قد يكون إهدارًا للوقت والجهد، وفي هذه الحالة، يمكن أن تكون البنية الأبسط أكثر فعالية وجدوى! فلا يمكن تقبل بناء مشروع HR داخلي لمؤسسة فيها ١٠٠ عضو ليكون Scalable ويضمن الاتساق الفوري و stateless مع أنه يمكن أن يتم إنجازه بأبسط من ذلك ولن يحتاج لكل هذه المبادئ التي ستصبح خسارة كبيرة وإهدارا للموارد!

٢. التأثيرات الجانبية: حتى المبادئ الجيدة قد يكون لها تأثيرات سلبية غير مقصودة، مثلا: قد يزيد الفصل المبالغ فيه بين المكونات من التعقيد ويزيد وقت التطوير... ولكل مبدأ إيجابياته وتحدياته التي تنتج منه...

٣. تغيّر السياق: ما يُعتبر أفضل ممارسة اليوم قد لا يكون كذلك غدا! بسبب التغييرات في التقنيات المستخدمة أو متطلبات العمل! وهذه تدعونا دوما للتوقف واحترام النفس قبل التفوه بكلمات مثل: "ما هذا الغباء!" أو "ما هذا التصميم!" دون أن تدرك ما هي الظروف التي كتبت فيها الشيفرة البرمجية أو صمم النظام عندها! ولا تقس ما تراه الآن من وسائل تقنية وتطورات مذهلة وتحاكم مصمما أو مطورا ما، على ما كتبه قبل ١٠ سنوات -مثلا-!

من الأهمية بمكان، أن نفهم تأثير كل ما ذكرناه

من المهم فهم كل ما تحدثنا عنه في العوامل المعمارية -Architectural Drivers-! وهذا يشمل المتطلبات الوظيفية -Functional Requirements- وسمات الجودة -Quality Attributes- والقيود -Constraints- والمبادئ -Principles-! وستجد أن هذا الفهم سينعكس عليك عند مواجهتك لأي نظام جديد لتعمل عليه... لإن إسقاط ما تعلمناه على أرض الواقع يعني أننا بحاجة ل High Level من المعرفة -على أقل تقدير- حتى نبدأ العمل بشكل صحيح! وهذا الفهم وإن كان عاما إلا أنه يوضح لنا حدود التصميم من خلال تحديدنا للعوامل المعمارية الموجودة عندنا!

لذلك، إن الفهم المجمل يمثل المستوى الأساسي من المعرفة الذي نحتاجه دوما لنبدأ في اتخاذ قرارات التصميم المعمارية... فاحرص على فهم كل ما تعلمناه، واحرص على اكتساب المعرفة العامة -على أقل تقدير- اللازمة للإحاطة بهذه العوامل المعمارية؛ وذلك كله سيساعدك على **تقليل عدد الخيارات المتاحة** أمامك لأن كثيرا من الحلول التقنية ستخرج تلقائيا من التي في رأسك ^^... ويمكن تلخيص ما ذكرناه هنا ب: "إن تقليل عدد الخيارات والحلول المتاحة في التصميم = واحدة من أهم تأثيرات العوامل المعمارية علينا وعلى الأنظمة".

هل الأثر المهم فقط هو ما ذكرناه؟

الإجابة بكل تأكيد: لا! فهناك أثر مهم لفهم العوامل المعمارية لا يمكن تجاوزه، **وهو اتخاذ قرارات مدروسة تناسب أهداف المشروع وسياقه!** وهذا واحد من أهم الآثار المترتبة على فهم العوامل المعمارية وفهم المشروع وسياقه... **اتخاذ قرارات مدروسة**، يا لها من عبارة!
إن إهمال العوامل المعمارية له أثر خطير على قراراتنا، فكيف سنبني مشروعا لم نفهم سياقه أو لم نفهم أهدافه؟! وكيف سنتخذ قرارا على معلومة ليست موجودة أساسا! أو على معلومة ناقصة لا تغطي الفهم العام لما نريده؟!
إن نتيجة هذا الإهمال ستكون كارثية، والنتيجة ستكون واحدة من اثنتين:

١. تصميم مبالغ فيه (Over-Designed): فإذا افترضت أن البيانات ضخمة وهي في الواقع بسيطة، وأن النظام قابل للتوسع وهو في الحقيقة لا يحتاج للتوسع؛ فإنك ستُضيع الوقت والجهد والمال في بناء حل معقد لا داع له.

٢. تصميم ناقص لا يغطي المتطلبات (Under-Designed): إذا افترضت أن البيانات بسيطة وهي في الواقع ضخمة، وأن النظام غير قابل للتوسع وهو قابل للتوسع؛ فإن النظام سيفشل عند التشغيل، مما يتطلب إعادة عمل مكلفة ومزعجة.

خلاصة هذا الباب: "إن فهم العوامل المعمارية في وقت مبكر من المشروع يساعد على اتخاذ القرارات بشكل صحيح، مما يجنبنا إعادة العمل المكلف في المستقبل، فلا تتجاهل هذا!".

فائدة

...فإن كانت أحوال الإنسان تتقلب في مدة قصيرة، فيها من الجهد والتعب ما فيها، فإن من باب أولى أن هذه الحياة ستتقلب أحوالها حتى تزول، كما زال الإنسان الذي عاش فوقها، ومن أدرك هذا أيقن أن لهذا الكون خالق عظيما، ليس كمثله شيء، وأيقن أن هناك حساب، وأن هناك ثواب وعقاب، ونعيم وعذاب!، فمن أراد النجاة من جهد الدنيا ومشقتها، فعليه أن يتبع أوامر الله -سبحانه وتعالى- حتى تستقر نفسه وتصبر، وتشكر وتبصر، وعليه أن يكابد نفسه في الشكر على النعم، والصبر على النقم، وعليه أن يكابد مصائب الدنيا فيصبر ويشكر، ويعمل ويخلص، فاللهم أكرمنا بنعيمك الذي لا يزول، ونعوذ بك من عذابك الذي لا يزول، ولا حول ولا قوة إلا بك.

- كتاب إلى الجنة زمرا، الصفحة ٧٤٤

الفصل الثاني: ال Architects

في هذا الفصل سنبدأ بالحديث عن دور -وظيفة/مهمة- المعماري، وما نوع المهارات التي يحتاجها، ولماذا تعد البرمجة ويعد الإرشاد والتوجيه والتعاون أمورا مهمة في عالم المعماريات وحياة المعماري ^^...

ال The software architecture role:

لا يوجد خط واضح يفصل بين كونك مطور برمجيات أو معماري! لذلك ستجد بعض الناس يعتبرون المعماري مجرد امتداد طبيعي لعملية التصميم التي يقوم بها المطور، وأن وظيفة المعماري لا معنى لها أصلا ولا وجود لها! بينما يراها آخرون دورا لا يقوم به إلا المبرمجون "الساموني" الذين لديهم هوس بال Abstraction، وهم في الحقيقة يتجنبون الخوض في تفاصيل التنفيذ الحقيقة والمشاكل العملية التي تواجه المطورين في حياتهم اليومية... وبين هذا وذلك، هناك أهل الوسط، أنضج الناس عقولا وأكثرهم اتزانا وأقربهم للصواب في أعمالهم...

لقد تعلمنا في الفصل الأول أن ال Software Architecture يتعلق بعدد من الأمور المختلفة؛ بدءا من تنظيم الشيفرة البرمجية، وصولا إلى امتلاك رؤية شاملة للنظام الذي يتم بناؤه من عدة زوايا مختلفة، واتخاذ القرارات التصميمية المهمة لضمان نجاح هذا النظام... هذا الفهم الذي تعلمناه يمثل معرفة عامة وواسعة، لكنه لا يصف حقا دور المعماري والمهام

المترتبة عليه! ولم نفهم أيضا كيف يمكننا الانتقال من كوننا مطوري برمجيات لنصبح معماريين؟! وكيف سنعرف إن كنا حقا معماريين جيدين أم سيئين! وكيف يمكننا أيضا نحن المطورين أن نعرف المعماري الجيد ذو التصميم الجيد من غيره؟

إن أول وأهم قاعدة عليك معرفتها هنا هي: "لن تصبح معماري جيد إلا بمرور الوقت! وذلك سيكون نتيجة لاكتساب الخبرات بشكل تدريجي، والثقة التي ستدفعك لاتخاذ القرارات المناسبة"... إن هذا الأمر لا يمكن أن يتم أو أن يحدث بين ليلة وضحاها! وهذا يذكرنا بقول أحمد شوقي -رحمه الله-: "وَمَا نَيْلُ المَطَالِبِ بِالتَّمَنِّي، وَلَكِنْ تُؤْخَذُ الدُّنْيَا غِلَابًا"!

ثاني قاعدة عليك معرفتها هي: "أن المعماري هو دور -Role- وليس مجرد رتبة في السلم الوظيفي -Rank-"، وهذا يترتب عليه الكثير من الأمور، منها أن المعماري قد يكون فردا أو فريقا -فهو Role-، وأنك لن تصبح معماري بعد ١٠ سنوات بمجرد عدد سنوات الخبرة! ويترتب على هذا إمكانية أن يقوم بهذا الدور موظف يمتلك الخبرة المناسبة بعدد سنوات قليل ولا يمكن لآخر لا يملك المهارة المناسبة لهذا الدور أن يقوم بذلك ولو كانت خبرته ١٠ سنوات...

حتى تكون الأمور واضحة بشكل أكبر، علينا أن نجيب عن هذا السؤال: ماذا الذي يفعله مطور البرمجيات؟

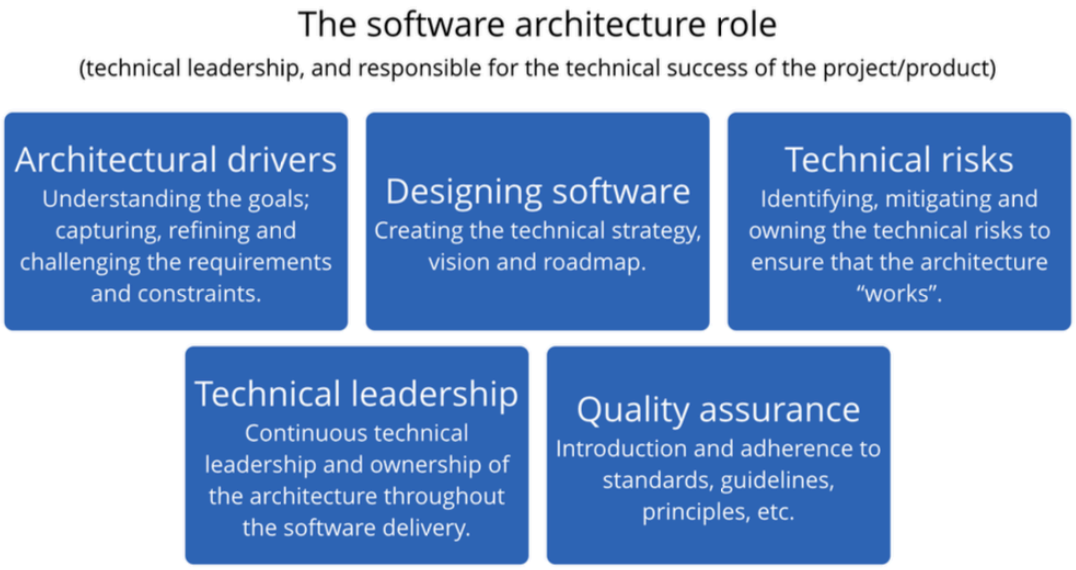
تتعلق مهام مطور البرمجيات بالرؤية الشاملة للنظام، ويركز على تنفيذ المهام وكتابة الشيفرة البرمجية بشكل يومي، لذلك تجد عادة أن مهام المطور واضحة ومفهومة نسبيا لكونها مرتبطة ب

Story أو تطوير لمزية معينة... بينما يبتعد المعماري عن تفاصيل الشيفرة البرمجية اليومية ليقوم بما يلي:

- تنظيم الشيفرة البرمجية والمكونات بحيث يحدد هيكل النظام حتى يحقق التناسق بين الأجزاء والمكونات المختلفة... وهذا يشمل تحديد آلية توسع النظام وضمان التوافق مع مبادئ التصميم التي تم اعتمادها... إلخ.
- القيادة التقنية: يقدم التوجيه لفريق التطوير مثل اختيار التقنية المناسبة... ملاحظة خاطفة: القيادة التقنية وإن كانت تبدو وظيفة ورتبة تقنية؛ إلا أنها جزء لا يتجزأ من دور المعماري...
- حمل المسؤولية: يضمن النجاح التقني للمشروع...

إن من قرأ الأبواب السابقة وسار معنا حتى هذه اللحظة، غالبا قد أدرك المقصود في كل هذه المفاهيم.. وأصبح الخط الفاصل واضحا له -وإن تشاركت بعض الخصائص والسمات-... ويمكن تشبيه النتيجة النهائية بأن المطور يركز على الشجرة بينما يركز المعماري على الغابة [٨٨].

والآن، شاهد الصورة التالية:



إن هذه الصورة حسب Simon تمثل الأدوار أو المهام الرئيسية التي تقع على عاتق المعماري، وملخصها هو ما ذكرناه في السؤال السابق... والآن لنذهب في زيارة لكل دور من هذه الأدوار ^^

١. ال Architectural drivers:

إن أول دور للمعماري هو فهم وإدارة العوامل المعمارية -Architectural Drivers-، وقد تحدثنا عن ذلك سابقا، وذكرنا العوامل المعمارية الأربعة:

- المتطلبات الوظيفية (Functional Requirements): ما يجب على النظام أن يفعله.
- سمات الجودة (Quality Attributes): كيف يجب أن يؤدي النظام وظائفه.
- القيود (Constraints): العوامل الخارجية التي تُفرض على المشروع (مثل الميزانية، والوقت).

- المبادئ (Principles): القواعد التي يتبناها الفريق لضمان الجودة والاتساق.

وجميعها تحدثنا عنها سابقا ^^... هذه العوامل ستُحدد الشكل النهائي للنظام، لأنها تعتبر بمثابة القوة الدافعة -driving forces- للمعمارية الناتجة والتي رسمت الشكل النهائي لمعمارية النظام...

هذا الدور يعتبر أول وأهم وأوضح دور للمعماري، وهذا الأمر يجب أن يكون واضحا للجميع، لأن ذلك سيجعل المسؤولية المتعلقة بهذه العوامل مناطة لشخص ما، أو فريق ما؛ مما يتيح أخذ هذه العوامل في الاعتبار بشكل استباقي -proactively considered- ومنذ البداية...

خلاصة هذا الدور:

عندما يعمل المعماري على مشروع ما، فإن من مهامه:

- فهم الأهداف -understanding the goals-: وفيها على المعماري أن يسأل "لماذا؟" بدلا من الاكتفاء بتسجيل ما يحتاجه العملاء فقط! وهذا لفهم الأهداف الحقيقة وراء كل مطلب، فمثلا إذا طلب العميل: "أريد نظاما سريعا"، فقد يكون هدفه هو تقليل وقت انتظار المستخدمين حتى يحصلوا على الخدمة...
- التقاط المتطلبات والقيود -Capturing-: وهي عملية جمع المتطلبات والقيود الخاصة بالمشروع، وذلك من خلال الاجتماعات وورش العمل وقراءة الوثائق أو الاطلاع على الأنظمة الموجودة أو من خلال الملاحظة!
- الصقل -refining-: بعد جمع المتطلبات ومعرفة القيود؛ على المعماري أن يقوم بصقل المتطلبات الغامضة -تحويلها- إلى متطلبات قابلة للقياس ومحددة! فمثلا بدلا من عبارة:

"إن النظام يجب أن يكون آمنا" فقد تكون نتيجة عملية الصقل هي: "يجب أن يتم تشفير جميع البيانات التي سيتم تخزينها في جدول المستخدمين من خلال خوارزمية AES-256"...

- التحدي -Challenging-: وهي الخطوة الأخيرة أو المهمة الأخيرة للمعماري في هذا الدور، وهنا ينظر المعماري هل جميع المتطلبات والقيود ضرورية ولازمة؟ ما هي أولوية كل واحدة منها؟ وهل يمكن تحقيق المطلوب ضمن القيود المتاحة؟ وهل يمكن التنازل عن بعض المتطلبات أو القيود الأقل أهمية أو غير المهمة في مقابل تلبية متطلب مهم أو أكثر أهمية؟... هذه الخطوة هي "التحدي" للمعماري ليصل إلى إجابة صحيحة على هذه الأسئلة...

٢. ال Designing software:

إن عملية التصميم تعد واحدة من المهام الأساسية لأي معماري، وهذا الأمر معروف بديهة! وتمحور وظيفة المعماري في عملية التصميم على إيجاد الحلول للمشكلات التي تظهر بسبب العوامل المعمارية -Architectural Drivers- التي تم تحديدها مسبقا...

إن عملية التصميم تتأثر بالأساليب والمنهجيات التي سيتم اعتمادها لتطوير التطبيقات، فمثلا لدينا ال Agile الذي يرتكز على التعاون والتكيف المستمر مع التغييرات في فترات زمنية قصيرة؛ لكن وعلى الرغم من ذلك فإن المعماري سيحتاج إلى تخصيص وقت للتفكير بعمق في كيفية استجابة البنية المعمارية للمتطلبات التي يتم تحديدها... فالتكيف المستمر لا يعني العمل بلا منهجية أو بلا تفكير! بل يعني العمل على التصميم ضمن رؤية تتناسب مع الأسلوب

المتبع! ويعتبر هذا الخطأ -العمل بلا منهجية أو تفكير مسبق- واحدا من أكثر الأخطاء شيوعا حول ال Agile! وعواقبه وخيمة -.-...

كما أن اختيار التقنية المناسبة للمشروع أو إطار العمل من أدوار المعماري في التصميم، وتعد واحدة من المهام المحبوبة والممتعة لمعظمنا؛ لكنها أيضا قد تصبح اللعنة التي تقيد المعماري وتصعب عليه الحياة إذا ارتبطت بالكثير من القيود -وتحدثنا عن ذلك سابقا-...

وخلاصة هذا الدور هي: إنشاء وتصميم الاستراتيجيات التقنية المراد استخدامها واعتمادها، وتحديد رؤية واضحة وخارطة طريق لتحقيق الأهداف المرجوة بطريقة منظمة... أي أنها عملية تحويل المشكلات والأهداف التي تم تحديدها في النقطة الأولى -دور تحليل العوامل المعمارية- إلى حل معماري منظم وقابل للتنفيذ [٨٨]

٣. ال Technical risks:

إن التعامل مع المخاطر التقنية من أهم ما يقع على عاتق المعماري، فمجرد التركيز على بناء حل جيد لا يعني النجاح! ومجرد التصميم النظري الناجح وحده لا يضمن النجاح! لذلك، لا يقتصر دور المعماري على اختيار التقنيات الأفضل -ظاهريا- فقط! بل يقع على عاتقه أن يتأكد من أن هذه التقنيات ستعمل فعلا كما هو متوقع منها في سياق النظام الخاص بنا! وهذه نقطة مهمة جدا، لأنها تحذرنا من نفخين كبيرين، وهما:

أ. فخ التهويل: كثير من التقنيات تأخذ ضجة تسويقية أكبر من حجمها أو كثير من مندوبي المبيعات يقدمون وعودا ضخمة لمنتجاتهم... وهذا قد يجعلنا نرتكب خطأ كبيرا باختيار تقنية غير مناسبة أو لا تتناسب فعليا مع سياق ومتطلبات النظام، وذلك بسبب التهويل أو شهرة أو نجاح المندوبين بتقديم وعود غير حقيقة، لذلك، عليك دوما اختبار التقنية وسؤال نفسك: هل هذه التقنية تعمل بالفعل بالطريقة التي نحتاجها؟

ب. فخ اللا مخاطر: إن وجود المخاطر والوقوع بها أمر وارد ومقبول وطبيعي! لذلك فلا وجود لشيء اسمه حل دون مخاطر! لكن، هناك عملية لإدارة المخاطر وهذه هي ما يحتاجه المعماري، فعملية اختيار التقنية بذاتها هي مخاطرة تحتاج لإدارة! وهدفها هو تقليل المخاطر المحتملة عند وجود تعقيد أو عدم يقين ما... وهذا يجعلنا نتقبل المخاطر عندما تكون الفوائد المحتملة تستحق فعلا هذه المخاطر التي سنواجهها...

وهذا يأخذنا إلى السؤال التالي:
كيف سيتم إثبات فعالية المعمارية التي قمنا بتصميمها؟ وهل هي حقا معمارية ناجحة؟ وما علاقة ذلك بالمخاطر التقنية؟
إن الجواب على هذه الأسئلة ينطلق من "الاختبار"، أي كما أننا نقوم باختبار الشيفرة البرمجية والتحقق من نتائجها فعلينا أن نقوم باختبار المعمارية لإثبات فعاليتها ونجاحها... ومن هنا يمكننا أن نعتبر المعمارية ناجحة إذا كانت:

- إذا كانت تستجيب للمؤثرات المعمارية: أي يمكن اعتبار أن هذه المعمارية ناجحة إذا نجحت في تلبية كل ما يفرض عليها من متطلبات وظيفية وسمات جودة وقيود ومبادئ...
- إذا كانت توفر أساسا متينا وموثوقا لجميع المطورين: أي يمكن اعتبار أن هذه المعمارية ناجحة إذا نجحت في بناء أساس صلب وقوي وموثوق يمكن لجميع المطورين من استخدامه كأساس معماري ليضعوا عليها لبناتهم حتى يكتمل البناء بنجاح، وليتحقق هذا؛ فعلى المكونات المختلفة أن تكون غير متضاربة، ولا يجب أن يكون لدينا تصاميم متضاربة ومتناقضة... كما أن ذلك يعني تحديد مبادئ التصميم وتوحيدها لضمان الاتساق...
- إذا كانت تحقق القيمة التجارية للمشروع: فإن المعمارية الناجحة يمكن ضمان نجاحها إذا حققت النتيجة النهائية المرجوة من المشروع، وبرزت القيمة التجارية لأنها قدمت الحل الذي بني النظام لأجله... فالمعماري لا يصمم من أجل اختبار التقنيات بحد ذاتها! بل يصمم لتحقيق الهدف الأساسي للمؤسسة، وهذا سيضمن أن قرارات التصميم تخدم استراتيجية العمل في المقام الأول...

إن ما تحدثنا عنه من معايير، يمثل النقاط الأساسية التي يمكنك قياس نجاح تصميم معماريتك بها...

إن التطبيقات البرمجية عادة ما تكون معقدة يصعب تصور خصائصها وسماتها بمجرد النظر إلى مجموعة من الرسوم أو حتى الشيفرة البرمجية؛ لذلك يجب على المعماري أن يختبر تصميمه مبكرا

كمثل الطاهي الذي يتذوق طعامه حتى يضمن النتيجة النهائية له! وهذا كله لتقليل المخاطر الإجمالية للمشروع، وتجنب الفشل في المراحل المتأخرة...

إذا، خلاصة هذا الدور هي: على المعماري تحديد المخاطر التقنية ذات الأولوية العالية ومعالجتها بشكل استباقي، لضمان النجاح التقني للمشروع وتحمل المسؤولية عن ذلك... وهذا يعني أن المعماري لن يكتفي باختيار أفضل التقنيات الموجودة، بل يجب عليه أن يتحقق من أنها ستعمل فعلا عند وجودها في المشروع.

٤. ال Technical leadership:

المعماري ليس مجرد مصمم! بل هو قائد تقني مسؤوليته توجيه الفريق طوال فترة التطوير وما بعد التسليم! وهو بذلك يعتني ويتابع متطلبات العمل المتغيرة والتغذية الراجعة من الفريق... أما المعماري الذي يقوم بتسليم المخططات والابتعاد دون متابعة، فهذا يدق ناقوس الخطر...

٥. ال Quality assurance:

من المهام التي تقع على عاتق المعماري هي ال QA، ويقصد بها ضمان الجودة للنظام الذي نرغب بتسليمه، فمجرد التسليم لا يعني أننا قمنا بتسليم شيء جيد! بل على العكس قد يكون ما سلمناه سيئا حتى لو كانت المعمارية جيدة! وهنا يظهر دور المعماري في التحقق من جودة النظام والتأكد من تسليم النظام بأفضل جودة ممكنة -تذكر لا يمكن ضمان كل شيء-.

والسؤال الآن، كيف يمكن أن يضمن المعماري ذلك؟
أ. من خلال وضع الأسس والمعايير: وذلك من خلال إنشاء مجموعة واضحة من القواعد التي يجب على الفريق اتباعها، مثل معايير كتابة الشيفرة البرمجية، ومبادئ التصميم، واستخدام أدوات محددة... هذا كله يوفر أرضية صلبة للبناء عليها.

ب. الامتثال والتوافق المعماري أو ما يطلق عليه ال Architectural Compliance أو Conformance: وذلك من خلال التأكد من أن التصميم المعماري الذي تم وضعه يتم تنفيذه بشكل متسق في جميع أنحاء المشروع، كما يجب على المعماري أن يضمن أن الرؤية التقنية -Technical Vision- التي أنشأها مفهومة ويلتزم بها جميع أعضاء الفريق.

إن عملية ضمان الجودة عملية واقعية وليست خيالية أو نظرية، فلا يمكن الجزم ولا يمكن ضمان كل شيء في النظام! لأن ذلك يرتبط ارتباطا وثيقا بالقيود الموجودة مثل الوقت والميزانية، والمعرفة والخبرات الخاصة بالفريق / الفرق التي تعمل على النظام... لذلك يتم هنا التركيز على الجوانب الأكثر أهمية مثل:
أ: الاهتمام بالأجزاء ذات الأهمية المعمارية والتي تمثل مكونات النظام الأساسية.
ب: الاهتمام بالأجزاء الحرجة ذات الأهمية العالية للمنطق التجاري والتي تؤثر مباشرة على نجاح العمل.
ج: الاهتمام بالأجزاء المعقدة أو عالية الظهور، وهي المناطق التي تمثل المناطق التي من المحتمل أن يحدث بها الأخطاء أو التي تؤثر على تجربة المستخدم بشكل كبير...

إذا، خلاصة هذا الدور هي: ضمان الجودة من خلال إدخال الضوابط اللازمة لضمان أن تنفيذ الشيفرة البرمجية يتطابق مع التصميم الاستراتيجي للنظام، وهذا يشمل المعايير والإرشادات والمبادئ والقيود والقواعد والوظائف والسمات...إلخ وليس مجرد المراجعة التقليدية للشيفرة البرمجية ذاتها.

ال Software architecture is a role, not a rank:

لقد أشرنا لهذه النقطة سابقا، لكن نريد أن نزيد بعض التفاصيل المهمة الآن... إن المسمى الوظيفي Software architecture يمثل Role -دور- للعمل، أي مجموعة من المهام المناطة على المعماري ليقوم بتنفيذها... وهي ليست رتبة يتم ترقية الموظف إليها...
ومع ذلك، فإن بعض المؤسسات الكبيرة تمنح أحيانا مكافأة أو ترقية إدارية لموظف ما ليكون "معماري" بناء على عدد سنوات الخدمة أو الخبرة أو كوسيلة لزيادة الراتب باعتبارها رتبة تمت ترقية الموظف إليها... هذه الممارسة تُفضي إلى مشكلة خطيرة وهي حصول شخص ما على لقب معماري دون أن يكون مؤهلا حقيقية للقيام بهذا الدور، وهو ما يُشار إليه أحيانا بـمبدأ بيتر -Peter Principle-.

يقول Simon -بتصرف-: «كم مرة قرأت على لينكد إن السؤال التالي: مرحبا، لقد تمت ترقيتي لأصبح Software Architect، ولكني لست متأكدا مما يجب أن أفعله، ساعدوني! وما هي الكتب التي يجب أن أقرأها؟»

إن جوهر مهمة المعماري هي القيام بقيادة تقنية ناجحة ومستمرة للفريق، واتخاذ القرارات التصميمية الهامة وضمان نجاحها... مما يعكس الخبرة والكفاءة والثقة التي اكتسبها الشخص خلال مسيرته المهنية؛ حتى يصبر معماري ناجح وقائد تقني حقيقي...

لذلك، احذر من هذا الفخ الكبير؛ المعماري ليس رتبة بقدر ما هو دور، فإن كانت المؤسسة تعامل هذا المسمى باعتباره رتبة، فيجب أن يتم ترقية المعماري الحقيقي لهذه الرتبة وليس أي شخص اعتمادا على قواعد مغلوطة مثل عدد سنوات الخدمة!

ملاحظة: مبدأ بيتر -The Peter Principle-: يشير هذا المبدأ إلى ظاهرة تحدث في التسلسل الهرمي للمؤسسات، بحيث يميل الموظفون إلى الارتقاء في مناصبهم بناء على أدائهم الناجح في وظائفهم السابقة، حتى يصلوا إلى منصب تنعدم كفاءتهم به... وكما ترى، فإن الفكرة الأساسية في هذا المبدأ أن المهارات التي جعلت الموظف ناجحا في منصب معين -مثل أن يكون مطورا ممتازا- قد لا تكون هي نفسها المهارات المطلوبة في المنصب الأعلى التالي -مثل أن يكون مديرا أو مهندسا معماريا-...
لذلك، ذكرنا في الشرح السابق أن خطورة الترقية لوظيفة معمارية باعتبارها رتبة ترتبط بعدد سنوات الخبرة أو الخدمة، لأن المطور الناجح والممتاز ليس بالضرورة أن يكون معماريا ناجحا! كما أن هذا الأمر يعني التوقف الوظيفي؛ فبمجرد الترقية إلى منصب يتطلب مهارات مختلفة لا يمتلكها الشخص الذي تمت ترقيته، فإنه سيتوقف عن الأداء الفعال، وسيستقر في هذا المنصب الجديد كـ"موظف غير كفء"، وهذا يعني فشل المشروع أو طرد هذا الموظف الكفء سابقا غير الكفء حاليا...

دور المعماري غامض؟ أنشئ تعريفك الخاص به ووثقه وشاركه!

كثير من المؤسسات لديها مسميات وظيفية معروفة، يتفق معظم الموظفين الموجودين فيها على مهام وأدوار هذه المسميات، وذلك: مثل Developers أو testers أو Product Owner... إلخ، لكن ليس الجميع على نفس الدرجة من الاتفاق فيما يتعلق بدور المعماري... لذلك: إن توثيق ما تتوقعه المؤسسة من هذا الدور وما هي صلاحياته ومهامه أمر مهم وسيحل هذا الإشكال، وسيجعلنا نقوم بتكليف الشخص الصحيح لأداء هذا الدور، وبغض النظر عن التسمية التي ستكون لهذا الشخص (Tech Lead أو architect أو Principal Designer... إلخ)... إن هذا الأمر سيحد من تضارب الفهم والغموض الذي قد يؤدي إلى فشل المشروع وتضارب الأشخاص وعدم القدرة على اتخاذ القرار، أو هروب الموظفين من إحدى المهام لأنها للمعماري وليس له... إلخ، لذلك، احرص على الاتفاق على مهام ودور المعماري ثم وثق هذه المهام كدور للمعماري في مؤسستك.

فائدة

فلْتعلمْ يا أخي أن الله -سبحانه وتعالى- لن يتركك، ولن يضيع أمرك، ولن ينساك، فأحسن الظن بربك -جل في علاه-، وارض بقضائه، واصبر لحكمه، وسترى خيرًا عظيمًا بإذنه -سبحانه وتعالى-، خيرًا قد يكون فيما قد ظننته شرًا!، وخيرًا ستجده في قلبك ومحياك لصبرك في الدنيا ونعيم في الآخرة!، فاللهم رحمتك نرجو، ولا حول ولا قوة إلا بك.

- كتاب إلى الجنة زمرا، الصفحة ٧٤٧

ال Technical leadership:

لقد تحدثنا عن القيادة التقنية، وأشرنا لأهميتها بشكل مختصر وسريع... والآن، علينا أن نتحدث بشكل أكبر وأكثر تفصيلا فيه هذا الباب...

إن أحد الأجزاء الأساسية في دور المعماري هو دوره كقائد فني، وهذا الأمر يبدأ من الفريق الذي يتكون من شخص واحد :P -أنت وعقلك وجهازك الحاسوبي- وصولا إلى الفرق الكبيرة التي تضم مئات المطورين ^^... لكن يبقى السؤال المهم، لماذا نحتاج إلى هذه السمة المهمة -القيادة التقنية- في شخصية المعماري؟

إن الهدف الرئيسي للقيادة التقنية هو التحكم في الفوضى -Controlling Chaos-، لأن التفكير الواعي في معماريتنا سيوفر نظاما يساهم بشكل كبير في حل المشاكل الشائعة التي تعاني منها الفرق العشوائية والفوضوية... وهنا سيظهر تساؤل مهم جدا، أليس هذا التحكم هو عبارة عن Micromanagement؟!

والجواب هو: كلا!

إن عملية التحكم لا تعني بالضرورة الإدارة التفصيلية والتدقيق المفرط -Micromanagement-؟! ولا تعني تقييد الإبداع الخاص بالموظفين في المهام اليومية! لكنه يتعلق بالتحكم -وضع- إطار عمل منظم يوجه الفريق، وهو ما سنناقشه بالتفصيل -بإذن الله- في هذا الفصل...

ملاحظة: لاحظ أننا قلنا "لا تعني بالضرورة"، وذلك لوجود حالات قد يكون هناك داع للدخول في التفاصيل الدقيقة مثل وجود أزمة مالية ومشروع على حافة الانهيار... لكن هذا

الاستثناء للحالات النادرة، ويكون مقيدا في فترة زمنية محددة، ويكون موجها... لا كأسلوب دائم.

توفير الوجيه والسعي نحو الاتساق (Provide Guidance, Strive for Consistency)

إن سمة القيادة التقنية تحوي في ثناياها العديد من الأدوار والمهام التي ينبغي على المعماري إدراكها وتحقيقها، ومن ذلك توفير توجيه -Guidance- والسعي نحو الاتساق -Consistency-، وهذا يعني أن على المعماري إدخال مجموعة من الضوابط والتي تمثل "تحكم" لضمان الاتساق والترابط بين الفرق والنظام ومكوناته... وهذا يشمل:
المبادئ والإرشادات والقواعد لكيفية التعامل مع أجزاء النظام المختلفة، بحيث تساعد المطورين أو الفريق على اتخاذ القرارات الصحيحة دون الحاجة إلى سؤال شخص آخر في كل مرة... بحيث يتبع كل جزء ومكون من النظام نسقا أو أسلوبا أو نمطا موحدا، مثل كيفية إعداد المشروع أو تسمية الملفات أو آلية الاتصال مع قاعدة البيانات... إلخ.

وهذا سيقودنا إلى سؤال آخر مهم، أليس هذا ما تحدثنا عنه سابقا حول الاتساق وأهميته؟! والجواب، أحسنت، هذا دليل على تركيزك فعلا... إننا هنا نتحدث عن الاتساق، لكننا سنذهب بشكل أعمق للإجابة عن سؤال خطير، وهو كيف سنحقق الاتساق أصلا؟
إن الاتساق لا يحدث بشكل تلقائي، بل يتطلب مجموعة من الإجراءات، وهي:

- التحكم: وذلك من خلال اتخاذ قرارات معمارية واضحة، مثل سنستخدم ORM أو أننا لن نستخدم ال ORM...

- الانضباط (Restraint): مقاومة الرغبة في "تجربة شيء جديد" لمجرد أنه يثير فضولي كمطور أو لكون التجربة بذاتها ممتعة!
- الحدود (Boundaries): وهي عبارة عن هيكل واضح ومنسق لل codebase، بحيث ترتسم الحدود والخطوط الفاصلة بين المكونات وأجزاء المشروع، وهي تحدد "من يملك هذا" و "من يتحدث مع من"، مثل: طبقة ال Service يجب ألا تعرف شيئا عن طبقة ال UI أو أن ال Microservice يجب أن تتواصل مع بعضها من خلال ال APIs وليس من خلال قواعد بيانات مشتركة... إلخ.
- الإرشادات: توصيات عملية وتطبيقية تساعد الفريق على تطبيق المبادئ في سياقات محددة، ويمكن تعديلها أو استثناؤها عند وجود سبب وجيه، وهذا مثل اعتماد استخدام ال async/await بالجافا سكربت بدلا من ال callback functions أو تحديد أكبر حجم لدالة ما ب ٢٠ سطرا أو تسمية الملفات ب kabab-cases أو استخدام ال Preitter لتنسيق الشيفرة البرمجية تلقائيا...
- المبادئ: وهي القواعد الأساسية العامة التي تعبر عن القيم الجوهرية أو الفلسفة التصميمية للفريق أو النظام، وهي ثابتة نسبيا ولا تغيرها الظروف بسهولة أو يتم استثنائها... مثلا: النظام يجب ألا ينهار كاملا إذا حدث خطأ في خدمة معينة -مقاومة الفشل- أو أن الشيفرة البرمجية يجب أن تكون قابلة للاختبار أو اعتماد الـ Single Responsibility...

إن وجود مشروع دون هذا التوجيه والضوابط، سيجعل المطورين يتجهون إلى مساراتهم الخاصة = مما يؤدي إلى نظام غير متماسك ولا يتوافق مع الرؤية المعمارية! وسينتهي بنا

المطاف إلى حالة من الفوضى، لذلك يجب على القائد التقني ضمان أن الفريق بأكمله يلتزم بالمبادئ المتفق عليها...

إذا خلاصة هذه النقطة هي: إن الفرق الجيدة التي تعمل على أنظمة واسعة و Codebase كبيرة لا تترك الأمور "عشوائية"، بل تفرض وتضع "توجيه" و"مبادئ" لضمان أن الشيفرة البرمجية ومكونات النظام ستبقى متسقة ومترابطة، وسهلة الفهم والصيانة...

ملاحظة: قد تتسائل عن الفروق الدقيقة بين المصطلحات التي ذكرناها، فمثلا ما هو الفرق بين الإرشادات والمبادئ والحدود والانضباط والتحكم؟ فجميعها قد يتداخل... وهذا صحيح، هناك ضبابية في بعض هذه المفاهيم، لذلك تم ربطها بأمثلة مباشرة حتى تصل الفكرة... ومع ذلك، سأحاول الآن التفريق بين هذه المفاهيم من خلال هذه الأمثلة أو الملاحظات:
إن التحكم يجيب عن سؤال: **كيف نتأكد** من أن الجميع يتبع المبادئ والحدود والإرشادات التي تم وضعها؟ وهل هناك انضباط كافي؟ ... وهذا يعني أن التحكم يمثل الوسيلة الرقابية أو التنفيذية ^^ في حين أن الحدود تجيب عن سؤال: **أين** ينتهي دور هذا الجزء ويبدأ دور الجزء الآخر؟ وهو يمثل الهيكل أو التنظيم التي ترسم الخطوط العريضة والحدود في النظام ^^ في حين أن المبادئ تجيب عن سؤال: **لماذا**، مثل: **لماذا** نكتب الشيفرة البرمجية بطريقة تجعله قابلا للاختبار؟ وإجابة السؤال: لأننا نريد أن نتأكد من أن النظام يعمل بشكل صحيح اليوم وفي الغد! وهذا يعني أننا نتحدث عن البعد الفلسفي أو السبب الحقيقي الذي يجعلنا نتبع قاعدة معينة ^^ في حين أن الإرشادات تجيب عن سؤال **كيف**، مثل: كيف سنقوم بعملية الاختبار؟ وهي بذلك تمثل الوصفة العملية -الأدوات- التي نتبعها لتطبيق المبادئ ^^ وأخيرا

الانضباط الذي يمثل مقدار التزام الأشخاص فيما ذكرناه مثل الالتزام بكتابة الاختبارات حتى مع ضيق الوقت أو الالتزام بكتابة الدالة بحيث لا تتجاوز ٢٠ سطرا برمجيا حتى لو كانت تعمل بشكل ممتاز على ٢٥ سطر ^^...

أظن أن الأمور أصبحت واضحة الآن -بفضل الله-... لذلك، لننطلق للموضوع التالي ^^...

القيادة التقنية التعاونية -Collaborative- ليست سهلة

لقد ذكرنا سابقا أن المعماري قد يكون شخصا واحدا وقد يكون فريقا متعدد الأطراف، وهذا هو موضوعنا في هذا الباب، فالقيادة التقنية التعاونية تعني أن تصميم البنية المعمارية للبرمجيات ليست مسؤولية شخص واحد! بل هي جهد جماعي يشترك فيه جميع أعضاء الفريق، وكل بحسب خبرته ومعرفته... لكن، هل هذا الأمر ممكن أصلا؟ وهل هو أمر يمكن تحقيقه بسهولة؟ وهل معظم الفرق التي نراها ونعمل معها أو سنعمل معها هي أفرقة تعاونية محترفة قادرة على اتخاذ القرارات الصحيحة والتخلص من عقدة إثبات الذات وفرض الرأي والانحياز للخيار الأفضل حقيقة؟

إن الإجابة باختصار، إن هذا الأمر ممكن لكنه صعب! ومعظم الفرق لن يكون هذا حالها إلا بعد تدريب وجهد وإرشاد... وكثير من الفرق التي تدعي أنها منظمة ذاتيا هي أفرقة تقوم على إدارة الفوضى وإخفاء المشاكل بطريقة منظمة، لذلك تجد أن كثير من هذه الأنظمة تحتوي في ثناياها الكثير والعديد من المشاكل مثل البطء أو صعوبة الصيانة أو عدم الاتساق أو مشاكل الأمان... إلخ.

كما أن القيادة التقنية التعاونية تتطلب توازنا دقيقا بين الخبرات التفردية والتخصصات المختلفة والتحكم في النفس لاختيار الأفضل أو الأصح -لا ما أراه فقط- والقدرة على العمل الجماعي بصدق!

وهنا يبرز لنا تحدي آخر، وهو وجود فرق agile ذاتية التنظيم (Agile & Self-Organising)، فكثير من الشركات والفرق الآن تعتمد هذا النهج وتتبناه وتسعى لأن تكون ذاتية التنظيم، وهذا يعني قدرة الفريق على اتخاذ قراراته بنفسه دون وجود مدير تقني مركزي، لكن هذا يعيدنا للسؤال المهم: إذا لم يكن هناك معماري محدد، فمن يتحمل مسؤولية التصميم المعماري الذي خرجنا به؟ والجواب الطبيعي في هذه الحالة هو: الجميع! فالكل قد شارك في اتخاذ القرارات المعمارية المهمة والكل يتحمل المسؤولية... لنأخذ مثالا عمليا:
تخيل أن لديك فريقا صغيرا فيه:

١. مطور FE

2. مطور BE

٣. مطور قواعد بيانات

وكل منهم خبير في مجاله ولديهم خبرة ممتازة في الـ full stack، من الناحية النظرية ستقول: هذا فريق مثالي ^^، لكن دعنا نتابع: إذا أردنا تصميم ميزة بسيطة مثل عرض ومعالجة مجموعة من البيانات على واجهة المستخدم، فما الذي سيحصل؟

إما أن يبدأ كل واحد منهم بالعمل بناء على رأيه وخبرته وفلسفته مثل أن يقول مطور ال FE نريد مرونة عالية للتعامل مع البيانات فأعطوني إياها JSON ويمكنني إدارتها بسهولة، ويقوم

ال BE بالتركيز على المنطق التجاري حتى يتمكن من حماية البيانات بأفضل شكل ممكن مع هيكلة الشيفرة البرمجية لتكون قابلة لإعادة الاستخدام، وبذلك فالمعالجة للبيانات ستكون بال server، ويقوم ال مطور ال DB باستخدام ال stored procedure لإرجاع البيانات جاهزة وذلك لتقليل حركة البيانات! وهذا كله يعني أن النتيجة النهائية ستكون غير متوافقة!

وإما أن يقوم الجميع بالتواصل بشكل مسبق والتحدث عن التحديات والمزايا ومناقشتها = وهذا يعني إخراج الافتراضات إلى العلن في أقرب وقت لتجنب المفاجآت السيئة ^^... إن هذا النهج سيبني فهما مشتركا ويدعم فكرة الملكية الجماعية للشيفرة البرمجية؛ مما يعين على بناء فريق تعاوني يتحدى الافتراضات ويكشف الاختلافات في الرؤى بشكل مكبر، وهذا سيجعل كل عضو في الفريق لديه القدرة على رؤية الصورة الكبيرة للمشروع، وسيحول الجميع إلى معماريين حقيقين لديهم القدرة على اتخاذ القرار.
إذا وصلنا لفريق يمكن لكل مطور فيه من الانتقال من تفاصيل الشيفرة البرمجية إلى البنية العامة للنظام، ووجدنا الجميع يفكرون في المتطلبات غير الوظيفية مثل الأمان والأداء وقابلية التوسع، ولا أحد يقول "هذا ليس من مسؤوليتي"... فهنا نقول لك: مبروك! لديك فريق ذاتي التنظيم حقيقة، وقادر على اتخاذ قرارات معمارية سليمة دون الاعتماد على شخص واحد، وهو بذلك يملك قيادة تقنية تعاونية فعالة...

إن القيادة التقنية التعاونية مفيدة ورائعة والجميع يطمح لتكوين فرق ذاتية التنظيم -self-organising-، لكن حدوث هذا الأمر وتوفر هذه الفرق غالبا ما يكون أمرا نادرا وصعبا، وذلك بسبب وجود المعوقات التي ذكرناها! إن الهدف من كل هذا الكلام هو: أن

يفهم الجميع ما يفعله المعماري، ويساهموا فيه معا؛ لا أن يكون الجميع معماريا! وهذه نقطة مهمة ^^.

معلومة: الفريق ذاتي التنظيم self-organising هو الفريق الذي يقرر بنفسه كيف سيُنفذ العمل، وليس فقط ماذا سيفعل! فلا يُعطى الفريق تعليمات مفصلة من مدير أو معماري حول "من يفعل ماذا" أو "كيف تكتب الشيفرة البرمجية"، بل يتحمل الفريق مسؤولية تنظيم نفسه مثل: توزيع المهام، واتخاذ القرارات التقنية، وحل المشكلات، وتحسين الأداء باستمرار... وهذا لا يعني الفوضى! فذاتية التنظيم لا تعني أن الفريق يعمل كما يشاء، بل تعني أنه يعمل ضمن إطار واضح تم رسمه يشمل أهداف المشروع والمبادئ التقنية ونحو ذلك، ولا يعني ذاتي التنظيم غياب القائد! بل يعني وجود قائد أو قائد خدمي وظيفته دعم الفريق وتوجيهه لا التحكم في الفريق وقراراته بشكل تفصيلي... وكمثال عملي: في بداية كل Sprint يتم إنشاء المهام المطلوب إنجازها، ويقوم الفريق بالاتفاق بأنفسهم على من سيعمل على كل جزء، وسيتعاون المطورين لعرض الطرق الممكنة لإنجاز مهمة ما؛ ليكون الفريق على اطلاع على ما يجري بشكل مبكر، واقتراح ما هو مفيد أو السؤال عنه... إلخ.

معلومة: هناك نموذجين مشهورين للتعامل مع الشيفرة البرمجية، وهما: ال Collective Ownership وال Individual/Module Ownership، وهما يمثلان مرتكزا مهما في موضوع القيادة التقنية التعاونية:
إن الملكية الجماعية للشيفرة البرمجية -collective ownership- تعني أن كل عضو في الفريق يملك الحق والمسؤولية في قراءة وفهم وتعديل أو إصلاح أي جزء من قاعدة الشيفرة البرمجية

(codebase) وليس الأجزاء التي كتبها هو فقط! بمعنى آخر لا يوجد جزء من الشيفرة البرمجية يعتبر "ملكا خاصا" لمطور واحد، فيقال: هذا الملف كتبته أنا، فلا يلمسه أحد! وفوائد هذا الأسلوب كبيرة من أهمها تقليل الاعتماد على أفراد محددين، فلو غاب هذا الفرد أو غادر الفريق لن يتعطل المشروع، كما أن الشيفرة البرمجية ستصبح أفضل لأن هناك أكثر من شخص يراجع ما كتب ويحسن عليها، وسيساهم هذا في نشر المعرفة داخل الفريق ومشاركة الخبرات وتعزيز روح الفريق حقيقة بشكل عملي، وسيصبح الجميع شركاء في النجاح أو الفشل! لذلك تجد أن طريقة التفكير السائدة في هذا النوع من الفرق هي: "كيف سنصلح هذا؟" بدلا من: "هذا ليس من شأني!"، ومع ذلك هناك تحديات مهمة هنا هي الخوف من "كسر شيء ما -بالعامية نفقع الشغل-" خصوصا إذا لم تتواجد اختبارات حقيقة كافية!، كما أن وجود أكثر من مطور يعمل على نفس المكان قد يدخل أكثر من أسلوب لحل المشكلة مما سيربك الآخرين، لذلك يلزم وجود سياسة واضحة وأدوات وممارسات تضبط هذا الأمر، كما أن وجود هذا التعاون قد يسرق من الوقت مثل الوقت "الضائع -مجازا-" في مراجعة الشيفرة البرمجية، مما سيبطء من العمل... وهذا كله يمكن ضبطه من خلال ثقافة تقنية صحيحة وحوكمة مناسبة... أما علاقتها مع القيادة التقنية فهي كبيرة لأن القيادة التعاونية لا تتحقق إلا بفهم الجميع للصورة العامة أو الكبيرة للمشروع، والملكية الجماعية للشيفرة البرمجية هي خطوة عملية لجعل هذا الفهم ممكنا!

أما الملكية التقليدية -Individual/Module Ownership- فهي تعني أن كل جزء من الشيفرة البرمجية مثل: Page, Service, Component سيكون مسؤولا عنها مطور واحد فقط أو فريق صغير محدد! ولا يُعدل على هذا الجزء أي أحد آخر دون إذن أو تنسيق مسبق... وهذا الأسلوب مناسب في المشاريع ذات الطبيعة الحساسة أو المعقدة والخطيرة

مثل وجود خبير وحيد أو فريق متخصص لإدارة عملية تحويل الأموال في بنك ما... وهذا لمنع أي خطأ كارثي يمكن أن يحدث، وهو مفيد إذا كان الفريق الذي يعمل على المشروع متفاوت في خبراته بشكل كبير وليس لديهم النضج الكافي في بيئة العمل، فهذا سيحمي الشيفرة البرمجية من الخراب وسيقلل من خطر كسر المهام الحيوية -فقع الشغل-، وهذا الأسلوب أيضا مفيد في المشاريع السريعة أو قصيرة الأمد والتي لا يكون فيها متسع من الوقت لبناء ثقافة تعاونية أو عند وجود أنظمة قديمة غير موثقة... أما العيوب فهي واضحة لا داع لذكرها :P

وهنا نشير إلى ملاحظة مهمة، وهي أن هذا الباب لا يعني أبيض وأسود فقط! أي أن هذا الكلام لا يعني أن عليك الاختيار بين واحد منهم فقط، بل يمكنك الجمع بين الأسلوبين ببساطة! وهذا ما يحدث على أرض الواقع في كثير من المشاريع! لكن العبرة في سياق المشروع وضوابط العمل.

والآن، لنسأل أنفسنا مجددا، هل المفهوم المثالي الذي طرحناه للأفرقة ذاتية التنظيم هو ما يتواجد على أرض الواقع؟

والجواب بكل تأكيد لا كما أشرنا لذلك في بداية هذا الباب!

إن الكثير من الشركات والفرق تبدأ في محاولة الركض قبل أن تتعلم المشي! لذلك، وجب على هذه الشركات والفرق أن تطبق مفهوم قيادة مرن، أي أن أسلوب القيادة يرتبط ارتباطا وثيقا بمستوى نضج الفريق، وهذا يمكن تصنيفه إلى ثلاث مستويات من النضج وهي:

١. البقاء / الفوضى -Survival/Chaos-: ويقترح أن أسلوب القيادة المناسب هنا هو الأمر والسيطرة -Command & Control-، أي أن هناك قيادة مباشرة لتوجيه الفريق.

٢. التعلم -Learning-: يقترح أن أسلوب القيادة المناسب هو التدريب -Coaching-، أي أن القيادة تقوم بتوجيه وإرشاد الفريق.
٣. ذاتي التنظيم -Self-Organising-: ويقترح أن أسلوب القيادة المناسب هو التيسير -Facilitation-، أي أن القيادة التقنية وظيفتها الحفاظ على التوازن وإزالة العوائق والسماح للفريق بقيادة نفسه...

هذه المستويات الثلاث هي نموذج اقترحه Roy تحت نموذج اسمه Elastic Leadership، وهو كما ترى إطار فلسفي هدفه مساعدة القادة على تنمية وتطوير فرق ذاتية التنظيم، وذلك من خلال إبراز أن القيادة تتمدد أو تنكمش حسب مستوى نضج الفريق! وبناء على هذا الكلام: تجد أن المشاريع التي لا تمتلك مطورين ذو خبرة معمارية كافية ومتفاوتة في خبراتها بشكل كبير تكون عادة في حالة فوضى برمجية كبيرة، لذلك تحتاج هذه الفرق إلى شخص يقوم بدور المعماري لفرض النظام وقيادة الجودة... ثم يصبح دوره تدريب الآخرين تدريجيا ليصبح مع الوقت فريقا ناضجا...

هل تحتاج فرق الـ Agile إلى معماريين -Software Architects-؟

غالبا ما تنظر الفرق التي تتبنى ال Agile إلى دور الـ Software Architect كـ "شر لا بد منه" ^^، ويعود سبب هذا التصور السلبي إلى التجارب السابقة لهذه الفرق مع مفهوم التصميم الشامل المسبق (Big Design Up Front - BDUF) لا المعمارية والمعماري!

إن الكثير من الفرق التي تستخدم ال Agile، ترتكب خطأ شائعا من خلال التركيز الشديد على Agile لدرجة يتم فيها إهمال الجوانب الحيوية الأخرى للمشروع!، وهذا يؤدي ويقود إلى فوضى عارمة بدلا من الحصول أو الوصول إلى التنظيم الذاتي! وهذا يجعل الفرق تفترض أو تظن أن هناك تعارض بين ال Agile و ال Architecture، واعتبار كل واحدة منهما قوة متضادة، فلا يمكن الجمع بينهما! والحقيقة ليست كذلك!

إن العدو الحقيقي لل Agile ليس الـ Architecture ذاته! بل هو التصميم المسبق المبالغ فيه -BDUF-! وهنا تظهر أهمية ما تحدثنا عنه في المتطلبات الوظيفية وسمات الجودة ونحوها، واسأل نفسك الآن، هل يمكن لهذه القيود أن تختفي بمجرد استخدام منهجية Agile؟ بكل تأكيد لا!

إن الفرق التي تتبنى ال Agile تحتاج إلى المعمارية ولدور المعماري، لكن يكمن الاختلاف هنا في كيفية تنفيذ هذا الدور وليس في وجوده! وهنا يبرز دور من أدوار القيادة التقنية الفاعلة... ^^

ملاحظة: قد يكون من المبكر الحديث عن ال BDUF بشكل مفصل هنا، لكن يجب أن نشير إلى المفهوم حتى يتضح سياق الكلام لمن ليس له باع طويل في هذا المجال... إن ال BDUF يمثل أحد المنهجيات المستخدمة في تطوير البرمجيات والتي تقوم على مبدأ إكمال تصميم كامل أو كل الجوانب قبل كتابة أي سطر برمجي من الشيفرة البرمجية، وبهذا ينفق فريق التطوير وقتا طويلا في مرحلة التصميم والتحليل، ويوثق كل شيء بدقة من المتطلبات

وبنية النظام وصولا لواجهات المستخدم وسيناريوهات الاستخدام المحتملة! ويعد هذا النهج من الطرق الشائعة عند التعامل مع نماذج ال Waterfall -التي تحدثنا عنها سابقا-... هذا النهج له مزايا مهمة مثل وضوح الرؤية بشكل مبكر للمشروع ويسهل تخطيط الموارد والجدول الزمني، لكنه في المقابل غير مرن وأي تعديل يتطلب إعادة تصميم جزئية أو كاملة بحسب التغيير! كما أن هذا النهج يفترض أن كل شيء معروف أو يجب أن يكون معروفا بشكل مسبق وهذا كلام غير واقعي خصوصا في واقعنا المعاصر! وقد يؤدي إلى تصميمات معقدة لا تتوافق مع ما هو مطلوب حقيقة... وهذا كله لا ينفي أو يلغي أهمية هذا النهج، فبعض الأنظمة قد تحتاج هذا النهج أو يفرض عليك هذا النهج مثل المشاريع الحكومية التي تكون من متطلباتها وجود التصميم بشكل مسبق، وكذلك بعض المشاريع الحرجة والحساسة -الخطيرة- مثل المفاعلات النووية أو عند وجود نظام معقد في ترابط بشكل معقد... إلخ، لكن احذر ^^، فلا يعني ذهابك نحو التصميم الشامل المسبق أن التصميم لن يتغير أبدا! لذلك، كثير من الفرق الواعية تستخدم ما يعرف اختصارا ب JEDUF، أي صمم ما يكفي لبدء العمل بثقة، واترك التفاصيل للوقت المناسب... وهذا كما قلنا يرتبط بحساسية المشروع وطبيعته والشروط والقواعد الناظمة -يبدو أننا أطلنا الحديث هنا، لذلك يكفي هذا! سنترك كل باب إلى وقته *-*-.

تطوير البرمجيات ليست رياضة تتابع!

هل تعرف رياضة التتابع؟ تلك الرياضة التي يقوم فيها الرياضيين بالركض ويقوم كل واحد منهم بتسليم عصا لغيره عندما يصل إلى نقطة معينة! إن عملية تطوير البرمجيات عملية مغايرة

لهذا الشكل! فهل يمكن أن نقبل بوجود معماري قام برسم معماريته التي تمثل الحل النهائي ثم يقوم برمى عصاه -التصميم- على الفريق ليقوموا بتنفيذها دون أي مشاركة أو متابعة للتنفيذ؟! بكل تأكيد لا! هذا غير مقبول بتاتا! وهو ما يشير لواحد من أكثر الأخطاء شيوعا وهو: AaaS، وهي اختصار ل Architecture as a Service -أي المعمارية نكخدمة-!

إن هذا النهج السيئ يتمثل في أن المعماري يقوم بتصميم النظام وتوثيقه بالكامل، ثم يرمي الحل النهائي إلى فريق التطوير بمجرد انتهاء التوثيق، ثم ينتقل هذا المعماري إلى مشروع آخر دون أي مشاركة أو متابعة للتنفيذ، وهذا له الكثير من الآثار السلبية، منها:

١. عدم التبني: بحيث تميل فرق التطوير عادة إلى تجاهل تصميم هذا المعماري والقيام بشيء مختلف، لشعورهم بعدم ملكية الحل -أن الحل لا يمثلهم حقيقة وهم ليسوا أعضاء فيه-، وكثير من المطورين ستجدهم يتحايلون على الحلول المقدمة في التصميم بدواع عدة ودوافع متعددة...

٢. فقدان السياق: يؤدي تبادل الوثائق -العصا- إلى فقدان سياق قرارات التصميم الأصلية، مما يجعل التنفيذ صعبا أو غير مفهوما...

٣. تحميل سبب الفشل للمطورين! إن المعماري الذي يعمل بهذا الأسلوب ينظر إلى أن معماريته نظريا ناجحة، وأن نجاحها هو مجرد Implementation Detail، فإن فشلت فسبب الفشل هو المطور الذي لم يستطع تنفيذ التفاصيل المعمارية المقترحة نظريا من خلاله... وهذا يمثل جهلا حقيقا للواقع العملي!

إن دور القيادة التقنية للمعماري يقود المعماري بشكل مباشر للمتابعة المستمرة لما صممه نظريا، وهذا يشمل متابعة التنفيذ على الواقع ومشاعر الفريق حول التصميم! إن دوره هنا كدور ربان السفينة الذي يوجه السفينة طوال فترة الرحلة، لا مجرد تحديد لمسارها عند بداية الرحلة! ومن هنا إذا كان النهج المستخدم عند الفريق هو ال Agile؛ فإن الرؤية المعمارية يجب أن تكون نقطة انطلاق يتم توضيحها وتطويرها تدريجيا طوال فترة حياة المشروع، وهذا سيضمن تحول الرؤية إلى واقع ناجح.

خلاصة ما وصلنا إليه:

إن الفشل في إنشاء نقطة انطلاق وتحديد اتجاه تقني واضح سيقودنا إلى نتيجة حتمية = هي الفوضى المعمارية؛ حيث ينتج عن ذلك قواعد شيفرة برمجية ضعيفة البنية وغير متماسكة -Big Ball of Mud-؛ مما يجعل فهمها وصيانتها أمرا صعبا! هذا الفشل غالبا ما يؤدي إلى عدم تلبية خصائص الجودة الحرجة -كالأداء، وقابلية التوسع، والأمان-... لذلك، يجب أن نتذكر أن تطوير البرمجيات ليس مجرد عملية تمرير مهام - ليست مجرد رياضة تتابع-، والنجاح في التنفيذ ليس "تفصيلا هامشيا"! وبهذا، فإن القيادة التقنية المستمرة ضرورية لجميع الفرق، لضمان تحويل الرؤية إلى نظام يعمل بكفاءة.

احترس من الفجوة -Mind the Gap-: تجنب عزل المعماري عن المطورين

هناك ضرورة لعدم خلق فجوة كبيرة وغير ضرورية بين المعماري وبقية فريق التطوير؛ رغم اختلاف دور كل منهما، فلا يجب أن يُنظر إلى المعماري كشخص "من منصة عليا" يُصدر الأوامر دون المشاركة في التفاصيل!

إن عزل المعماري عن فريق التطوير يؤدي إلى مشاكل خطيرة أهمها عدم الاحترام وفقدان الدافع؛ مما يؤدي لتجاهل القرارات المعمارية وإن كانت صحيحة! وسيختفي الاهتمام بالصورة الكبرى للمشروع! مع أن كثير من الشركات تقوم باختيار معماري من فريق التطوير وترقيته، إلا أن هذه العملية إن جعلت هذا المعماري في طبقة منفصلة ومنعزلة عن الفريق، وفقدت روح العمل الجماعي فيما بينهم... وقعنا في فخ "الفجوة".

إن حل مثل هذه المشكلة بسيط وسهل للفرق الصغيرة والمتوسطة، فكل ما هو مطلوب هو بناء جسر التعاون والمشاركة فوق هذه الفجوة، وبهذا سيتمكن الجميع من العبور، وسيساهمون في تحسين الجسر الذي تم بناؤه -القرارات المعمارية والتصميمية-... إن المعماري الذكي يقوم بإشراك المطورين في القرارات المعمارية والتصميمة، ويوضح لهم ويشرح لهم **المنطق والنية** وراء هذه القرارات، وهذا سيشجعهم على تبني المعمارية وبناء مشاعر ملكية فاعلة مع هذا التصميم! كما أن هذا المعماري الذكي يمكنه الانخراط والمشاركة في الأنشطة اليومية لفريق التطوير مثل مراجعة الشيفرة البرمجية للاحتفاظ بفهم عملي والحصول على تغذية راجعة حول مدى فاعلية المعمارية الحالية وقراراتها على أرض الواقع.

وهنا ملاحظة مهمة، كما أن المعماري الذكي عليه مسؤوليات فالمطور / المطورين الأذكياء يقع على عاتقهم دور مهم في تقليص الفجوة وبناء الجسر بينهم وبين المعماري!
إن المطور الذكي يقع على عاتقه فهم الصورة الكبرى للمشروع، أي أن يخصص وقتا لفهم سياق المشروع وسياق القرارات المعمارية التي تم اتخاذها، وهذا سيساعده على استيعاب النظام ككل واستيعاب القرارات وسبب اتخاذها، ومن هنا يمكنه مناقشة وتحدي قرارات المعماري! وهذا يعني أن المطور إذا شاهد أو رأى خطأ ما فعليه التعبير عن رأيه، فالعملية تعاونية تشاركية وليست مجرد عملية روتينية! بل إن من الحركات الذكية التي يقوم بها بعض المطورين هي الانخراط بالعمل المعماري من خلال طلب السماح لهم بأن يكونوا مع المعماري في عمله، وهذا يعني المساهمة وتقديم الاقتراحات ومناقشة الخيارات الممكنة معه، مما يخدم كلا من المعماري والمطور، وهذا عادة سلوك يفرح به المعماري الذكي!

ملاحظة: قد تتعجب من كلامي في الفقرة السابقة حول: "تحدي القرارات المعمارية"، وأول ما سيتبادر إلى ذهنك أن هذا سيتسبب في مشاكل كثيرة وكبيرة! وكلامك صحيح إذا تم ذلك في بيئة غير صحية! أما إن تم ذلك بطريقة مهنية وبناءة فما تتحدث عنه سيكون مجرد خوف وانطباع مسبق! بل إن هذا التحدي يعد واحدا من أعمدة المعماري الذكي الضرورية لمراجعة واكتشاف الأخطاء بشكل مبكر، أو مراجعة التصميم والقرارات المعمارية حول جزئية ما، وإثراء القرارات المتخذة والتخلص من التصميم الزائد (قد يصمم المعماري حلا صحيحا لكن المطور يقترح حل أبسط وأقل تعقيدا فيقوم المعماري باعتماده بدلا من تصميمه)... إلخ، أما إذا سيطرت على المطور أن التحدي هو آلية لإثبات الذكاء والتعالي وتحدي المعماري، فهذه ستكون مشكلة! وتحدي المعماري بسبب اختيار تقنية أو أسلوب

لأن المطور يفضل تقنية ما عن غيرها دون الأخذ بالاعتبار العوامل المعمارية ستكون مشكلة، وإذا تحدى المطور القرارات المعمارية دون فهم واسع للسياق ودون وجود صورة كبيرة عن المشروع فهذه ستكون مشكلة أيضا، وإذا تأخر المطور وتحدى المعماري في آخر المراحل فهذه مشكلة أخرى أيضا لأنها ستؤثر على النتائج والتكلفة والوقت...، باختصار: التحدي البناء للقرارات المعمارية هو جزء أساسي من القيادة التقنية التعاونية، وحتى يكون صحيحا: يجب أن يكون التحدي في وقت مبكر، ومبنيا على فهم السياق الواسع للمشروع، ويهدف إلى تحسين الحل وليس مجرد إثبات خطأ للمعماري.

معلومة: لقد ذكرنا في هذا الموضوع هذه الفقرة: "إن المعماري الذكي يقوم بإشراك المطورين في القرارات المعمارية والتصميمة، ويوضح لهم ويشرح لهم **المنطق والنية** وراء هذه القرارات"، لكن ما المقصود بالمنطق والنية؟

إن المنطق -Rationale- يشير إلى السبب الموضوعي والتحليلي الذي دعانا إلى اختيار حل معين بدلا من حل آخر، وهو يمثل عملية توثيق اتخاذ القرار، وهو إجابة سؤال: "لماذا اخترنا هذا؟"... مثال: "لقد اخترنا استخدام قاعدة بيانات NoSQL بدلا عن ال SQL لأن منطق البيانات أظهر أننا نحتاج إلى قابلية توسع أفقية عالية جدا، وقدرة على تخزين بيانات غير منظمة، وهي متطلبات لا تستطيع قاعدة بيانات من نوع SQL تلبية متطلبات الأداء الخاصة بها".

في حين تشير النية -Intent- إلى الهدف الذي يسعى المعماري لتحقيقه من منظور جودة النظام أو الأداء والوظيفة، أي أنها توضح الغاية من هذا الاختيار، وهي إجابة سؤال: "ماذا نريد أن نحقق؟"... مثال: "كانت النية من وراء تصميم المكونات على أنها Stateless هي

ضمان سهولة التوسع الأفقي وتحمل الأعطال، بحيث يمكن إضافة خوادم جديدة أو إزالتها دون فقدان معلومات ال session".

وسؤالي لك الآن، هل لاحظت أهمية المنطق والنية على موضوعنا هذا؟ وهل لاحظت أثرها على سد الفجوة أو تقليلها وبناء جسر حقيقي بين المعماري والمطور؟ وهل لاحظت أثرها المطورين حتى يتمكنوا من القيام بتحد بناء؟!

فائدة

فلْتعلمْ يا أخي أن الشدة والألم والصعاب مهما بلغت من الإنسان مبلغها، فإنه سيأتي يوم يلازمه اليسر فيزيل العسر!، ألم تسمع قوله تعالى في سورة الشرح: "فَإِنَّ مَعَ الْعُسْرِ يُسْرًا (5) إِنَّ مَعَ الْعُسْرِ يُسْرًا (6)"؟!، فاللهم رحمتك نرجو، ولا حول ولا قوة إلا بك.

- كتاب إلى الجنة زمرا، الصفحة ٧٥٢

ال Software architects and coding:

الاستعارة المعمارية -Building Metaphor-:

هل سمعت من قبل بالاستعارة المعمارية؟ إذا لم تسمع بها من قبل، هل حدثت نفسك يوما عن أن ما تقوم به من مهام أثناء بنائك مشروع من الصفر يشبه ما يقوم به المهندسين المعماريين وعمال البناء؟ إن كان ذلك، فأنت قريب من المقصود من هذه الاستعارة ^^...

إن الاستعارة المعمارية تشير إلى استخدام وإسقاط مفاهيم وصور من عالم البناء والهندسة المعمارية -مثل التصميم، الأساسات، الهيكل، المخططات المعمارية..إلخ- لوصف أو فهم عملية تطوير البرمجيات، والسبب في ذلك يعود لعدة أسباب:

- كلاهما يحتاج إلى تخطيط مسبق
- كلاهما يتكون من أجزاء مترابطة مثل: جدران ↔ مكونات ووحدات برمجية.
- كلاهما يُفترض أن يكون متينا، قابلا للتوسع، وسهل الصيانة.
- كلاهما يستخدم مصطلحات مثل:
  - Foundation (الأساس - مثل البنية التحتية)
  - Blueprint (المخطط - مثل وثائق التصميم)
  - Structural Integrity (السلامة الهيكلية - مثل الموثوقية في الشيفرة البرمجية)

لكن مع ذلك، فهل هذا الاسقاط أو هذه الاستعارة تنطبق بشكل كامل أو تعمل دائما معنا ضمن سياق البرمجيات؟ الجواب بكل تأكيد لا! والسبب في ذلك يعود إلى:

- في البناء المادي يصعب تغيير الجدران أو الأساسات في حين أن البناء البرمجي يتغير باستمرار ويتوقع تغييره باستمرار.
- في البناء المادي يجب أن يسبق التصميم البناء في حين أن التصميم والتنفيذ قد يحدثان بالتوازي في عالم البرمجيات خصوصا في عالم ال agile -إذا كان النهج المتبع في البناء البرمجي هو تصميم شامل مسبق فقد يتشابه البناء المادي والبرمجي هنا-.
- في البناء البرمجي يقوم المطورون بكتابة أو إعادة كتابة أجزاء من الشيفرة البرمجية باستمرار في حين أن عامل البناء المادي لن يقوم بإعادة بناء أجزاء المبنى كل فترة كما في حال البرمجيات...
- إن التجريب والتكرار والتغذية الراجعة هي جوهر عملية تطوير البرمجيات الحديثة، وهي عملية مستمرة طوال فترة حياة المشروع، في حين أن البناء المادي يختلف عن ذلك في بعض الجزئيات، فلا يمكن تجريب وضع ٣ شمعات بدلا من ٤ شمعات، لأن الأخطاء الناجمة عن ذلك قد تكون كارثية!

بناء على ما ذكرناه، فإن الاستعارة المعمارية مفيدة جدا لشرح بعض الجوانب البرمجية وتقريب الأفكار من الواقع، لكنها أيضا محدودة لأن البرمجيات ليست شيئا ماديا ثابتا، بل رقمي متغير وقابلة للتغيير ويعتمد على التفاعل مع المستخدمين!

قد تتساءل الآن، ما علاقة هذا كله بموضوعنا! وسأقول لك، انتظر قليلا، فلم ننتهي، دعنا نتابع مواضيع البناء المادي ^^

هل سمعت من قبل عن البَنّاء الرئيسي -master builder-؟ مثلا هل سمعت ب"المعماري سنان"؟

لو عدنا قليلا إلى الوراء حتى عصورنا الذهبية، سنجد أن هناك بنائين، ومشرفين، وكبار المعماريين، ومهندسين، وهناك كبار المهندسين الذين يمثلون طبقة فيها العديد من أشهر وأفضل المهندسين والذين يترأسهم شخص يطلق عليه كبير المعماريين...

إن كبار المهندسين كانت وظائفهم تصميم المباني والمعالم الحضارية المهمة وفهم المواد والقوى الفيزيائية والتقنيات المتاحة، وقد يشارك في التنفيذ بيده في المراحل الحرجة أو التفصيلات الدقيقة التي تحتاج لتدخله، ولا يفصل بين التصميم والتنفيذ، وكان يتدرج من كونه عامل بناء وطالبا في هذا المجال حتى يصير مشرفا أو حتى ينجح في مشروع يبرز إبداعه فيصبح كبير مهندسين... وهذا كله يعني أن هذا البّناء القديم كان مهندسا ومنفذا في آن واحد ^^...

مجددا، لماذا أتحدث عن هذه المواضيع؟ تعال معي إذا لأجيبك حتى لا تسأم ^^

إن كثير من الناس يعتقد أن دور معماري البرمجيات يتحور حول التفكير المجرد أو عالي المستوى فقط -High level abstract thinking-! لذلك ربما سمعت بمصطلحات شهيرة مثل PowerPoint Architect أو ال Ivory Tower Architect... وهذه مصطلحات تشير عادة إلى أشخاص يصممون معماريتهم دون أن يأخذوا التفاصيل العملية بعين الاعتبار! وهذا يتنافى ما جوهر المعماري أساسا! وهذا ما أردنا إيصاله من خلال المقدمة التي طرحناها...

إن العلوم المختلفة تنقل تجاربها الخاصة بها، لكن يمكن للعلوم الأخرى التعلم والتفكر فيما عند غيرها حتى تختصر على نفسها سنوات من البحث والضياع!

إن المعماري الرئيسي يجب أن يكون خبيرا حقيقيا في مهنته! لا مجرد مرتبة وظيفية نالها دون خبرة مهنية أو معرفة بما يقوم المطورين بفعله، وما هي تحديثاتهم ومشاكلهم المتوقعة ونحو ذلك! وكما كان عمال البناء يتعاملون مع الأحجار بشكل فعال للحصول على أجمل بناء بأقوى المميزات فعلى المطور أن يبدع في شيفرة البرمجية للحصول على أفضل النتائج!
وكما كان هناك كبير المعماريين فلدينا اليوم المعماري الأساسي أيضا -Chief Architect-!
وكما كان يتعدى دور كبير المعمرين مراقبة عمال البناء إلى مسؤوليته عن النجارين وصانعي الزجاج وكل من يعمل في الموقع فإن المعماري الرئيسي لدينا يقع على عاتقه ذلك من ناحية رقمية! وكما كان كبير المعماريين متمكنا وقائدا فنيا وتنفيذا فعلى المعماري الرئيسي اليوم أن يكون كذلك! متمكنا من التصميم وعلى دراية عالية في التنفيذ!

الخلاصة

إن معمارية البرمجيات ليست مجرد رسومات UML أو شرائح PowerPoint، بل هي فن وتقنية تتطلب أن يظل المعماري فيها قريبا من الشيفرة البرمجية! وذلك حتى يتمكن من اتخاذ قرارات مبنية على واقع التنفيذ لا على مجرد افتراضات نظرية! وكما كان كبار المهندسين يوازنون بين الفن والهندسة، يجب على معماريو البرمجيات أن يوازنوا بين الرؤية التقنية والممارسات اليومية للتطوير..

وهذا سيأخذنا للتساؤل الذي كتبت لأجله هذه الكلمات، وهو:

هل يجب على المعماري أن يبرمج -أن يكتب شيفرة برمجية؟-؟

الجواب باختصار: نعم!

إن المعماري الذي يقوم برسم مربعات وخطوط على اللوح يجب أن يعي أن هذه الخطوط والمربعات يجب أن تتحول في النهاية إلى شيفرة برمجية حقيقة، أما المعماري الفاشل فهو الذي يفعل كما فعل أتباع فرضية التطور الحمقى؛ الذين رسموا خطوطا افتراضية بين الكائنات ثم قالوا هكذا حدث التطور -الحمد لله على نعمة الإسلام-!

لذلك، بصفتك معماري فهذا يعني أن هناك حاجة ملحة لتحافظ على مهاراتك التقنية، ومن المنطق الصحيح أن تحافظ على مهاراتك من خلال كتابة الشيفرات البرمجية باعتبارها إحدى أفضل الطرق لتحقيق هذه الغاية!

لكن هذا لا يعني أن المعماري عليه أن ينغمس في المهام البرمجية اليومية للمطورين في الفريق، وإنما يعني الانخراط باستمرار في عملية التسليم والمشاركة بشكل فعال طوال فترة حياة المشروع! مع العلم أن هناك معمارين ناجحين ليس لهم خبرة وباع طويل في عالم البرمجة أو تطوير البرمجيات، لكنهم قلة مقارنة بالواقع، وقد تجدهم يركزون على تركيب المكونات ومعالجة القيود والمشاكل ونحو ذلك ويتركون الشيفرة البرمجية وقراراتها للمطورين، ومع أنني لم أعمل مع أناس من هذا النوع من قبل، إلا أنني شاهدت نقاشا بين المطورين وتجاربهم حول هذا النوع من الناس، والمشاكل التي يرونها ونحو ذلك... لذلك، أميل لما قاله Simon: "إن معظم أفضل المعماريين الذين أعرفهم لديهم خلفية قوية في الـ Software Development، ولا يزالون يستمتعون بكتابة الشيفرة البرمجية حتى اليوم".

إن كل ما ذكرناه سيقودنا إلى هذا الاستنتاج:
يفضل أن يكتب المعماري شيفرة برمجية حقيقة تصل إلى ال Production، وذلك لضمان أن تكون تصاميمه واقعية وقابلة للتنفيذ! لأن هذا الانخراط سيمنح المعماري فهما عمليا للتحديات التي يتعامل معها الفريق التقني وسيسهل عليه مراقبة التزام الفريق بالمبادئ المعمارية التي تم وضعها... وهذا يعني معالجة المشاكل المعمارية وأخذ التغذية الراجعة بشكل مباشر ودوري! وهذا بدوره سيبني ثقة كبيرة بينه وبين المطورين، وسيقلل من الفجوة بينه وبين المطورين... وهذا سينعكس أيضا على القيادة التقنية، فالمعماري الذي يمارس ويكتب الشيفرة البرمجية مع الفريق ستكون قيادته أكثر فعالية... ويُستثنى من هذا الكلام المشاريع الضخمة التي تتطلب وقتا كبيرا للقيادة الاستراتيجية ونحتاج فيها إلى صورة كبرى عن المشروع وأجزائه!

وهذا سيقودنا إلى سؤال آخر، وهو: إذا كنت معماريا ولم أكن قادرا على كتابة الشيفرة البرمجية لأي سبب مثل القيود التي قد تضعها الشركة أو بسبب ضيق الوقت أو كبر حجم المشروع، فما هو العمل في هذه الحالة؟
أولا تذكر أن الهدف من هذه العملية أن يحافظ المعماري على انخراطه العملي مع الفريق، وكتابة الشيفرة البرمجية ومشاركة الفريق بعض الأعمال ستكون وسيلة ممتازة لهذا الغرض، لكن إن لم نتمكن من ذلك لأي سبب، فهناك عدة طرق يمكننا استخدامها للانخراط مع الفريق، مثل بناء النماذج الأولية (Prototypes)، والأُطر (Frameworks)، والأسس (Foundations)، ومراجعة الشيفرة البرمجية، والتجربة ومواكبة المستجدات...

- النماذج الأولية: بناء النماذج الأولية يساعد على التحقق من أن القرارات المعمارية الرئيسية ستعمل فعلا كما هو متوقع، وذلك مثل اختيار تقنية معينة أو نمط تصميم، وليكون الموضوع أكثر واقعية لنأخذ مثالا حقيقيا، إن القرار النظري سيكون اختيار تقنية معينة لل Caching ولنقل أنها Redis لتقليل زمن الاستجابة... إذا اكتفى المعماري بهذا فسيكون قرارا نظريا ولن يحقق ما نصبو إليه، لكن لو قام ببناء نموذج أولي PoC يحاكي وجود ضغط عالي لاختبار زمن الاستجابة عند جلب البيانات من Redis فهنا أصبح للقرار المعماري مرجعية واضحة تُبرز أن الهدف المراد من هذه التقنية سيتم تحقيقه عند بناء المشروع: وهو التحقق من أن Redis سيوفر الأداء المطلوب لتلبية متطلبات وقت الاستجابة الذي نحتاجه...
- بناء الأُطر والمكتبات والأسس (Frameworks & Foundations): وذلك من خلال المساهمة في تطوير الأدوات المشتركة (Shared tools) والمكتبات الأساسية التي سيعتمد عليها الفريق.

لكن، احذر، فهناك أيضا معماريين يأخذون هذا الباب لبناء أدوات كثيرة لا معنى لوجودها أصلا! أو يبنوها ثم يرمونها على الفريق ليقوم باستخدامها دون مشاركتهم ما أراد بناؤه وكيف سيتم بناؤه واستخدامه... إلخ، وهذا المعماري يطلق عليه: Ivory Tower Architect... أي المعماري الذي يصمم بمعزل عن الواقع -المعماري صاحب البرج العاجي-، وسبب التسمية هذه نابع من أن هذا المعماري يخطط ويصمم من مكان عالي أو موقع مرتفع وهو البرج العاجي ^^... لذلك المعماري الذي يضع قرارات نظرية ومثالية جدا لكنها غير عملية أو صعبة التنفيذ ضمن القيود التي تحدثنا عنها سابقا = معماري عاجي ^^

والذي يصمم ويخطط دون أي مشاركة مع الفريق أو متابعة لتفاصيل التنفيذ = معماري عاجي ^^، والذي يفشل في الحصول على تغذية راجعة من المطورين الذين يعانون صعوبات في تطبيق تصميمه = معماري عاجي ^^.

- مراجعة الشيفرة البرمجية: تعد مراجعة الشيفرة البرمجية والاطلاع على طريقة كتابة الشيفرة وماهية التقنيات المستخدمة وكيفية استخدامها واحدة من الطرق التي تجعل المعماري منخرطا مع الفريق حتى ولو لم يكتب بيده شيفرة برمجية... لكن يجب أن ألا تتبع هذه الاستراتيجية لمدة طويلة، بل إن هذا الأمر له تأثيرات سلبية قد تقلل من احترام المعماري مثل إعطاء نصائح تقنية للفريق حول موضوع لم يمارسه حقيقة أو لم تكن لديه خبرة واقعية فيه أو التركيز على التفاصيل الدقيقة التافهة أو غير المهمة والتي تتعلق بالأسلوب وليس القضايا المعمارية الحقيقة وهو ما يطلق عليه: Nitpicking -وهذا قد يقع فيه المطورين أيضا أثناء مراجعتهم-... إن هذه التأثيرات السلبية تضر بمصداقية المعماري لأنها ستظهره بمنظر المتزمت غير العملي، ويجبر الفريق على إصلاح قضايا لا قيمة لها في الحقيقة -على الأقل في الوقت الراهن-، كما أن كل نصيحة في غير محلها ستجعل الفريق يفقد الدافع لاتباع إرشاداته وقراراته؛ وهذا سيقلل من الاحترام المخصص للقرارات التقنية الصادرة منه...
- التجربة ومواكبة المستجدات: من الضروري أن يحافظ المعماري على معرفة تقنية عالية ليتمكن من تصميم حلول ذات كفاءة عالية حتى ولو لم يُسمح له بالمساهمة في الشيفرة البرمجية داخل الشركة، ومن ذلك النشاط خارج العمل مثل تجربة مكتبات معينة والمساهمة في المشاريع ال open source أو تجربة أحدث التقنيات وتقييمها...

كما أن قراءة الكتب وحضور المؤتمرات ونحو ذلك من الأمور المفيدة... وهذا كله سيضمن بقاء المعماري على دراية في الواقع التقني وسيجعل قراراته قريبة من المطورين...

والآن، بعد هذا الكلام الجميل، هل دور المعماري يمثل اختيار وظيفي / مهني للانتقال من مرحلة كتابة الشيفرة البرمجية لمرحلة جديدة لا يوجد فيها كتابة للشيفرة؟
بكل تأكيد لا! إن هذا الاعتقاد الشائع هو أحد الوجوه الخاطئة في نظرتنا لوظيفة المعماري، بل العكس هو الصحيح، فدور المعماري يتطلب مهارات تقنية قوية جنبا إلى جنب مع الخبرة والمعرفة العامة، وهذا الكلام تجده عند الأشخاص الذين يتمتعون بصفة ال Generalising Specialists -أي المتخصصون الذين لديهم معرفة عميقة في مجال ما، ولكنهم قادرون أيضا على فهم الصورة العامة-، وهم من يمكنهم الإجابة بشكل حقيقي وفعال على سؤال: هل سيعمل هذا التصميم فعلا؟
لذلك، **إن مما يطلب منك كمعماري هو الموازنة بين الحفاظ على المهارات التقنية اللازمة لك من خلال كتابة الشيفرة البرمجية، وبين الوفاء بالمسؤوليات الاستراتيجية والقيادية المناطة بك كمعماري...** إذا كان المعماري سيكتب الشيفرة البرمجية طوال الوقت، فمن سيقوم بدور المعماري؟!

التوتر والضغط النفسي بين كتابة الشيفرة البرمجية ووجودي في أحد المناصب العليا (Being Senior)

قد تتعجب من العنوان، لكن الحقيقة أن هذا العنوان الغريب هو أحد التناقضات الشائعة في الشركات الكبيرة -خصوصا- بين مكانة المعماري باعتباره من الطبقات الإدارية العليا وبين القيمة التي تُمنح حقيقة لكتابة الشيفرة البرمجية...

إن العديد من الشركات الكبيرة تعتبر أن كتابة الشيفرة البرمجية مجرد نشاط ذا قيمة قليلة، ويمكن توكيله لأي فريق حتى لو كان فريق خارجي لا يعمل بالشركة، بينما تنظر للأدوار القيادية على أنها أدوار يجب أن تركز على المهام الإدارية والتصميم النظري فقط! وهذا الفصل الخطير يعطينا انطباعا مباشرا عن وجود تناقض كبير! فكيف تطلب من معماري تصميم حلول معمارية ثم تمنعه من المشاركة فيها ومن المشاركة في إنجاح عملية التسليم؟!

يرى Simon أن المعماري يجب أن يدافع عن فكرة كتابة الشيفرة البرمجية بوضوح، وذلك من خلال طرح هذه النقاط:

1. الحفاظ على المهارات: إن كتابة الشيفرة البرمجية تضمن الحفاظ على مهارات المعماري محدثة، ولديه وعي بالتحديات التقنية الحقيقية.
2. ضمان الواقعية: إن كتابة الشيفرة البرمجية تضمن أن التصاميم واقعية وقابلة للتنفيذ، وليس مجرد نظريات.
3. القيادة بالقدوة (Leading by Example): إن كتابة الشيفرة البرمجية تُبرز أن دور المعماري هو دور قيادي، وأن المشاركة في الشيفرة البرمجية هي أفضل طريقة لضمان نجاح التسليم...

وهنا لدي تعليق بسيط، من وجهة نظري الشخصية:

إن اعتبار كتابة الشيفرة البرمجية أو أن الشيفرة البرمجية شيء قليل القيمة = أمر خاطئ وكارثي، وأراه حقيقة من الأسباب الشائعة والمزعجة لتدهور جودة المشروع عبر الزمن! لكنني كذلك أرى أن كتابة الشيفرة البرمجية هي الجزء السهل من عملية تطوير البرمجيات، وأقصد بذلك أن عملية تحويل الأفكار والمنطق والخوارزميات التي تم التفكير بها من شكل نظري إلى شكل ترميز يترجم الأفكار هي عملية سهلة إذا ما قورنت بما قمنا به قبل كتابة الشيفرة البرمجية... فالمبرمج قبل أن يبدأ بكتابة أي سطر برمجي يجلس ويفكر وينظم أفكاره ويكتب ما يحتاجه من خطوات وتقنيات ثم ينتقل لكتابة الشيفرة البرمجية التي تمثل مجموع الأفكار التي دارت بخلده... لكن من يبدأ بكتابة الشيفرة البرمجية مباشرة دون تفكير حقيقي = كارثة على المشروع في كل المقاييس!

إن ما أرغب في الإشارة إليه أن النظرة إلى كتابة الشيفرة البرمجية تختلف باختلاف مكان النظر، فإذا كانت النظرة من الشركة أو الإدارة العليا للشيفرة البرمجية قليلة القيمة أو أنها شيء سهل يمكن إحالته لأي أحد بغض النظر عن الخبرة أو مستوى التعقيد ونحو ذلك؛ فاعلم أن هذه كارثة وأن المشروع سيدخل في مرحلة انهيار أو صعود في مستوى التعقيد غير مسبوق! في حين إن كانت النظرة قليلة القيمة من قسم التقنية فهذه مصيبة تشير إلى فقدان الشغف أو القدرة على تمثيل الأفكار أو فهم أهمية ما أحيل إليهم من أعمال! أما إن كانت النظرة هي فقط أن الشيفرة البرمجية هي الجزء السهل مباشرة دون تحليل وتفكير، فهذا دليل على وجود مشكلة خطيرة في المنطق الخاص بهذا المطور! إن الشيفرة البرمجية هي الجزء السهل عند قولبة الأفكار من منطق إلى ترميز، من نظري إلى عملي... وهنا ستجد المتعة الكبيرة،

وستجد ثمرة ما مر من مراحل منطقية حتى وصلت للحظة تحويل الأفكار إلى أفعال...
وستجد أن هذه السهولة لا تعني أن أي شخص يمكنه فعلها! بل تعني أن هذه مرحلة جني ثمار ما بذلته من مجهود... وهذا مثل من يبني بيتا، فينتهي من وضع الأساس والشمع، ثم يأتي لمرحلة صب السقف، إنها مرحلة لها مشقتها وتحدياتها، لكنها مرحلة الثمرة لإتمام ما بُني في الأساس!

خلاصة سريعة لما سبق:
إن العمارة التقليدية انقسمت إلى مهندس إنشائي ومعماري بسبب النضج والتراكم المعرفي، لكن لا تزال صناعة البرمجيات حديثة وسريعة التغير! لهذا، يجب على المعماري أن يجمع بين دور المهندس الإنشائي والمعماري، لهذا فإن كتابة الشيفرة البرمجية ضرورة لضمان بقاء مهارات المعماري محدثة، وضرورة للتحقق من أن التصميم قابل للتطبيق، وضرورة لبناء الثقة مع الفريق، وهذا كله ما يضمن نجاح المعمارية ومنها نجاح المشروع في بيئة تقنية متقلبة...

والآن، لننتقل إلى آخر موضوع لدينا في هذا الفصل، وهو:

المهارات والعلوم والخبرات المطلوبة من المعماري

هناك اعتقاد شائع عند كثير من المطورين بأن ال Software Architecture ليست سوى مرحلة تالية من المسار التقني (Post-Technical)، أو أن معمارية النظم ما هي إلا مسار إداري بحت، لكن الحقيقية -وكما أشرنا من قبل-: إن دور المعماري لا يقتصر على مجرد رسم مجموعة من المربعات والأسهم على لوح أبيض ومن ثم البدء بإصدار الأوامر للفريق! بل

يتطلب هذا الدور من المعماري أن يمتلك مزيجا متوازنا من المهارات التقنية والمهارات غير التقنية -مثل القيادة والاتصال-، لضمان تصميم وبناء نظام ناجح.

ومن هنا سنطلق للحديث عن هذه المهارات المطلوبة من المعماري بشيء من التفصيل ^^، أي سنشير إلى المعرفة التي ستكون في عقل المعماري حتى نقول عن شخص ما أنه يقوم فعلا بدور المعماري بشكل صحيح ^^

١. المهارات التقنية -Technology skills-:

إن معظم المعماريين الرائعين الذين تعاملنا معهم كانت لديهم خلفية جيدة أو قوية في عالم تطوير البرمجيات، لقد كانوا وما زالوا مطورين أكفاء حتى اللحظة ^^... وهذا لا يعني بالضرورة أنهم أكثر الناس إبداعا في كتابة الشيفرة البرمجية أو أنهم أفضل من يكتب شيفرة برمجية في الفريق! بل يعني أن لديهم العلم الكافي والمهارة المناسبة للتنقل بين التفاصيل التقنية الدقيقة أو العميقة والصورة الكلية للمشروع...

إن المعماري الرائع يجمع بين مفهومين مهمين وهما: ال Deep Technical Specialism وال Breadth of Knowledge، بطريقة أخرى فالمعماري الناجح عادة ما يمتلك تخصص تقني يفهمه بعمق وماهر فيه مدعوما باتساع معرفي اكتسبه في سنوات طويلة من عالم تطوير التطبيقات، وبهذا يجتمع العلم والخبرة ليشكل ركيزة أساسية لأي معماري ناجح أو يرغب بالنجاح.

معلومة ١: إن ال Deep Technical Specialism تعني أو تترجم إلى "تخصص تقني عميق"، وهي تشير إلى امتلاك شخص ما مستوى متقدم وخبرة عميقة في مجال تقني ضيق ومحدد، وهذا يشمل الفهم التفصيلي للمفاهيم الأساسية وآليات العمل الداخلية والتحديات المتقدمة وأفضل الممارسات في هذا المجال، لذلك يمثل هذا المفهوم الجانب العمودي أو الرأسي من المهارات، وهو الأساس الذي يبنى عليه دور المعماري، ومن الأمثلة على ذلك: خبرة شخص ما بالجافا سكربت وقواعد البيانات من نوع mysql في مجال تصميم مواقع الإنترنت.

معلومة ٢: إن ال Breadth of Knowledge يمثل الجانب الأفقي من المهارات، وترجمته هي: "اتساع المعرفة"، ويشير إلى المعرفة العامة والسطحية نسبيا التي يمتلكها شخص ما في عدة مجالات أو في مجالات واسعة مثل فهم ال Agile ووجود مهارات قيادية ومعرفة عامة بعدد جيد من لغات البرمجة وقواعد البيانات المختلفة... وهذا مهم لأنه يمكن هذا الشخص من فهم الصورة الكبرى لنظام ما وكيف تتفاعل المكونات المختلفة فيما بينها، ويساعده على فهم العمليات التجارية ومنطق الأعمال واحتياجات العملاء والقيود الزمنية والميزانية، وهذا كله يساعد في اتخاذ القرارات التقنية التي تخدم أهداف النظام...

بناء على المقدمة السابقة وما تم طرحه قبل قليل، يمكن تصور أن الخط العمودي والأفقي سيلتقيان في نقطة معينة، وهو ما له انعكاس مرئي في عالم المعماريين وهو: T، فكما تلاحظ يمثل حرف T باللغة الإنجليزية تصور مرئي لما تحدثنا عنه، وكثيرا ما ستشاهد هذا الانعكاس البصري أو هذه العبارة: "T-Shaped Profile" والذي يعني الملف الشخصي للمعماري على

شكل حرف T ^^، وقد تجد مصطلح آخر يشير لنفس المفهوم وهو: ال Generalising Specialist، أي أن هذا المعماري يمكنه الغوص في التفاصيل المعقدة في مجاله، وهو قادر أيضا على الانخراط في المناقشات المعمارية على نطاق واسع...

ملاحظة: يستخدم ال T-Shaped Profile وال Generalising Specialist عادة في نفس السياق ولنفس المعنى، وكلاهما يشيران لنفس النموذج الذي تحدثنا عنه، لكن إن أردت بعض الفروق الدقيقة بينهم فيمكن القول إن ال Generalising Specialist يستخدم عند الإشارة إلى المسمى الوظيفي والوصف الخاص به -أي ماذا يفعل هذا الشخص-، أما ال T-Shaped Profile فعادة ما تستخدم كآلية لتمثل أو وصف مهارات هذا الشخص... ومع ذلك، لا تركز كثيرا على هذه الجزئيات الصغيرة، فهي ليست ذا قيمة كبيرة، وإنما ذكرتها هنا في سياق الحديث حتى تدرك ماذا يمكن أن يدور حولك إذا عملت كمعماري ^^

والآن، لأننا مطورين فنحن نمتلك عادة معرفة بأشياء مشتركة مثل:

- ال Programming Language Syntax
- ال API's
- ال Frameworks
- ال Design Patterns
- كتابة ال Testing
- وغيرها من الأنشطة التقنية اليومية

هذه المعارف هي نفسها المعارف الأساسية التي يحتاجها المعماري، وذلك يعود لسبب رئيسي وهو للحصول على إجابة صحيحة وواقعية حول أسئلة مثل:

- ما هي المقايضات المرتبطة بهذا القرار التقني؟
- وهل هذا الحل قابل للتنفيذ حقا؟
- وهل ما تم رسمه من مربعات وأسهم للفريق يعكس الطريقة التي سيبنى فيها النظام بشكل واقعي؟

.. إلى آخره.

والآن، سأسألك سؤالا، هل أنت ذو عقلية متفتحة؟

انتظر، بكل تأكيد لا أسألك السؤال الساذج والرديء الخاص بالنسوية أو أشباه الرجال، بل أسألك أأنت ذو عقلية متفتحة على اكتساب خبرة جديدة مثل استخدام لغة برمجة أخرى غير التي تعلمتها في مسيرتك المهنية داخل مشروع ما لأنها الأنسب لهذا المشروع؟ إذا كان جواب نعم، فهذا ممتاز، وسأقول لك لماذا بعد قليل... لكن إذا كانت إجابتك لا فهذا ناقوس خطر، راجع نفسك، فهناك مثل يقول: "إذا كان كل ما تملكه مطرقة، فإن كل شيء ستراه، سيبدو لك وكأنه مسمار"...

إن انغلاق عقولنا عن تقبل خيارات أخرى يندرج تحت باب: خطر التخصص المفرط، والذي عادة ما نراها ضمن ظاهرة: "Technology Lock-in" والتي يقصد بها الانغلاق داخل التكنولوجيا الواحدة بحيث يصبح فيها مستخدم النظام أو العميل أو المطور مرتبطا بتقنية معينة أو بائع معين لدرجة أنه يجد صعوبة كبيرة أو تكلفة باهظة أو حتى استحالة التبديل إلى بديل آخر! وطبعا نحن هنا لا نقول أن كل ارتباط وثيق هو ارتباط سيء! لكننا

نقول أنك يجب أن تكون واعيا بما يترتب على هذا الارتباط ومعرفة تكلفته المستقبلية...
لكن، لماذا من أجاب هذا السؤال بنعم يمكن وصفه بصاحب العقلية المتفتحة؟
السبب يعود لأن النتيجة الطبيعية لأي منحنى تعلم خضناه هو الوصول إلى الخبرة العميقة في هذا العلم والذي سيقودنا لتحقيق إنتاجية أعلى وبجودة أفضل، فإذا كان المعماري يميل لاستخدام التقنيات التي يعرفها فقط ويفرض حلولا على الفرق بناء عليها -المطرقة-، فإن أي مشكلة يواجهها ستكون هي المسمار الذي يطرق! بينما صاحب العقلية المتفتحة لن يجعل خبرته العريقة وتخصصه العميق عائقا عن رؤية البدائل، فالقرارات المعمارية يجب أن تُتخذ بناء على متطلبات المشروع وخصائص الجودة، وليس بناء على تفضيل المعماري أو خبرته المجردة عما سبق!

وهنا سيظهر لدينا سؤال آخر مهم، لأن لدينا اتساع معرفي كمعماريين ولدينا عقول متفتحة، فهل يجب علينا فهم التفاصيل الدقيقة لل Technology Stack الذي قمنا باختياره؟
والجواب بكل تأكيد نعم! وهو جانب مهم جدا! فلا يعني اتساع المعرفة أو العقول المتفتحة ألا تفهم كيف تعمل مجموعة التقنيات المختارة أو كيف تتفاعل فيما بينها، وكمثال عملي، لو افترضنا أننا سنبني موقع إلكتروني قابل للتوسع، فعلينا أن نكون على قدر من الوعي يؤهلنا للإجابة عن مثل هذه الأسئلة:

- كيف سيتم عمل Scaling على ال Server-side؟
- كيف يتم عمل ال Replication لل HTTP Session عبر مجموعة من الخوادم لضمان المرونة عند فشل أحدها؟

- هل هناك مشاكل أداء أو قابلية توسع (Performance/Scalability Issues) مرتبطة بال Replication لل Session؟
- أم من الأفضل استخدام مخزن بيانات خارجي من نوع Key-Value Store لحفظ ال session الخاصة بالمستخدم؟
- ما هو أفضل Framework لبناء هذا التطبيق؟
- كيف سيتم عمل الـ Authentication في هذا التطبيق؟

...إلى آخره.

هذه الأسئلة كما تلاحظ تحتاج فيها إلى فهم تفصيلي للتقنيات المستخدمة حتى تستطيع الإجابة عليها، لكن، انتظر... لحظة، لن تنطلي علي حيتلك، فكيف تخبرني عن العقل المتفتح والفهم العميق وأنا قد أختار تقنيات جديدة أو تقنيات لا أعرفها! هذا تناقض واضح!!!

وهنا يسرني أن أخبرك أنك رائع يا صديقي، فكلامك منطقي ويدل على تركيز عميق، لكن هذا السؤال هو ما سيقودنا للجزء التالي، والذي يمكن اختصاره بالإجابة التالية:

"إذا كنت تخطط لاستخدام تقنية جديدة، فهناك احتمال كبير بعدم وجود خبرة كافية أو عميقة لديك حولها، لكنك كمعماري يجب أن تعرف بالضبط كيف ستكتسب هذه المعرفة!"...

إن التعلم السريع والتركيز على هذه المهارة واحدة من النقاط المهمة لأي معماري، لأن المواقف التي سيتعرض لها المعماري ستجبره على اتخاذ قرارات تقنية حول تقنيات قد تكون غير مجربة بعد أو أنه لا يملك الخبرة الكافية لذلك، وبهذا سيتحول تركيزك كمعماري على معرفة كيفية اكتساب ما تحتاجه من خلال الاستعانة بالحلول المناسبة لكل حالة، مثل بناء نموذج أولي للتقنيات المستخدمة وتخصيص وقت للبحث والتجريب أو من خلال الاستعانة

بخبير لفترة معينة يساعد على سد الفجوة... إلخ، لذلك تعد إدارة المخاطر واحدة من مسؤوليات المعماري! فعليه أن يعرف كيف سيسد الفجوة المعرفية حتى يضمن أن اختيار التقنية الجديدة لن يؤدي إلى فشل المشروع أو النظام بسبب نقص الخبرة!

وهذا كله سيقودنا مجددا إلى أهمية الاتساع في المعرفة التقنية، لأن ذلك سيمكننا من الإجابة على أسئلة مثل:

- هل التقنية التي اخترناها هي الأكثر ملاءمة لمشروعنا مقارنة بالخيارات الأخرى؟
- وما هي الخيارات الأخرى المتاحة؟
- وهل هناك نمط أو أسلوب معماري يجب أن نستخدمه؟
- وهل نفهم جيدا ما هي المقايضات في قراراتنا المعمارية؟
- وهل صممنا النظام بما يلبي سمات الجودة المطلوبة الآن وفي المستقبل؟
- وما هي المشاكل المحتملة عند اعتماد هذا التصميم؟
- وكيف يمكننا إثبات أن هذه المعمارية ستعمل على أرض الواقع؟
- وما مدى شعبية التقنية حاليا؟ وهل من السهل العثور على مطورين لديهم خبرة بها؟

... إلى آخره.

بناء على هذا كله، أريد أن أنوه إلى نقطة مهمة وخطرة، فعلى الرغم من أن المعرفة العامة في التصميم والتقنيات والأنماط والأساليب غالبا ما يمكن تطبيقها من خلال تقنيات أو أساليب متعددة، إلا أن عدم فهم كيفية تطبيقها بنجاح عند التنفيذ (Implementation Level) قد يؤدي إلى مشاكل خطيرة، وهذا إن حدث سيعرقل المشروع ويخلق فجوة بين التصميم والواقع! لذلك، خذ هذه القاعدة والتي تمثل الزبدة لهذا الكلام كله:

"المعرفة العامة تعطيك الخيارات المتاحة، لكن عدم فهم كيفية تطبيقها عمليا عند التنفيذ سيجعلها بلا قيمة".

٢. المهارات الناعمة -مهارات التعامل مع الآخرين، ال Soft skills-:

تمثل المهارات الناعمة (Soft Skills) مجموعة المهارات غير التقنية وغير الفنية والتي ترتبط بكيفية تفاعل الشخص مع الآخرين، وعمله في الفريق، وحل المشكلات، وإدارة النفس وسلوكياتها... وبذلك فهي تمكن المعماري من تحويل المعرفة التقنية إلى نجاح عملي من خلال العمل مع الناس! فمهما كانت مهاراتك التقنية عالية؛ ستكون دون جدوى أو كفاءة عالية إن لم ترتبط بمهارات التعامل مع الآخرين! لذلك، يمثل امتلاك جانب تقني قوى نصف القصة فقط! في حين أن هناك أمور يصعب قياسها تعد ذات أهمية بالغة، ومنها ما نتحدث عنه الآن...

وقد تتساءل: ماذا تشمل المهارات الناعمة؟ ويمكن تلخيصها بما يلي:

- القيادة والرؤية: إنشاء رؤية مشتركة وأخذ الفريق نحو تحقيقها، مدعومة بتقدير بالنفس لا الغرور.
- التواصل والتأثير: القدرة على توصيل الأفكار بفعالية لجميع الفئات المستهدفة، واستخدام مهارات التفاوض والتسوية للتأثير على الآخرين وتحويل أجنداتهم الفردية إلى هدف مشترك.
- التعاون والتيسير: العمل مع الفريق للوصول إلى حلول أفضل، وتيسير النقاشات عند الخلافات لضمان التوافق.

- التدريب والإرشاد والتحفيز: مساعدة أعضاء الفريق على النمو التقني (تدريب/إرشاد) وضمان بقاء روحهم المعنوية عالية لاتباع الرؤية الخاصة بالمشروع أو المؤسسة.
- المسؤولية والفهم السياسي: تحمل المسؤولية الكاملة عن جودة المعمارية والنتائج التقنية، مع فهم السياسات الداخلية للمؤسسة مع تجنب الانخراط فيها.
- التفويض: تعلم تفويض المهام عند الضرورة، مع العلم أن المسؤولية النهائية تظل على عاتقك كمعماري، وتذكر أن هناك خط رفيع يفصل بين تفويض كل شيء وبين فعل كل شيء، لذلك تعلم أن تُفوِّض المهام عندما يكون ذلك مناسبا، ولكن تذكر ألا تُفوِّض المسؤولية ذاتها...

الخلاصة: إذا كنت مسؤولاً عن تصميم نظام ما، فيجب أن تمتلك السلطة الكافية لاتخاذ القرار! وإلا فإنك ستواجه عقبات كثيرة! لإن وجود مسؤولية دون سلطة = الكثير من العقبات! كما إن القيادة التقنية التي لديك سيكون هدفها توحيد رؤية الفريق وعمله نحو هدف واحد -هدف المشروع-، وهذا كله يعتمد على المهارات الناعمة أكثر من السطوة وفرض القرارات! وهذا كله يجعل المعماري قدوة للفريق، خاصة وأن المطورين يتطلعون إليه، وهذا يزيد العبء عليه أيضا! فتصرفاتك كمعماري ستؤثر مباشرة على الفريق، فحماسك سيولد حماسا، وأي تشاؤم منك أو لامبالاة سيخلق دوامة سلبية لدى الفريق قد يصعب الخروج منها!

وهنا يقول Simon: «نادرا ما نتحدث عن الجانب الإنساني في دور المعماري، لكن المهارات الناعمة غالبًا ما تكون أصعب وأهم من المهارات التقنية! فالفريق السعيد هو الفريق الذي ينجز أعماله، وبصفتك قائد، ستقع على عاتقك مسؤولية الحفاظ على إيجابية الفريق، ولا ينبغي

التقليل من أهمية دورك في ديناميكيات الفريق ككل»، وهذا الكلام يذكرنا بما ذكرناه سابقا في الفصل الثامن من كتاب شرح ال micro-frontend باللغة العربية، نقتبس منه ما يلي: "لا يقتصر نجاح مشروع الـ Micro-frontends على كتابة الشيفرة البرمجية أو اتباع أفضل الممارسات التقنية فحسب؛ بل إن الجانب الأكثر أهمية والذي يجب أن نفكر فيه بعمق: هو الجانب البشري والتنظيمي... إن إهمال الجانب البشري غالبا ما يضيع قرارات صحيحة أو منطقية لأنها كانت تغطي الجانب التقني فقط؛ دون الاهتمام بالجانب البشري -العاطفي أو العقلاني-! لذلك تجد كثيرا من المطورين المبدعين بقوا في أماكنهم لأنهم لم يستطيعوا التعبير عن أنفسهم بشكل صحيح! وهناك كثير ممن أعرفهم شخصيا، كان لتسويقهم لأنفسهم الأثر الأكبر في وصولهم لمراتب أعلى من غيرهم -بعد فضل الله سبحانه وتعالى وتوفيقه لهم-؛ رغم أن هناك من هو أكثر منهم مهارة وعلما".

٣. إما التعاون أو الفشل -Collaborate or Fail-: أهمية إشراك ال Stakeholders

ملاحظة: قبل أن نبدأ نشير إلى أن ال Stakeholders هو مصطلح يستخدم ضمن سياق تطوير البرمجيات للإشارة إلى جميع الأطراف التي يجب أن يتعاون معها التقني لضمان نجاح المشروع ونجاح التسليم، والترجمة الحرفية لهذا المفهوم هو: المالكون أو أصحاب المصلحة، ومن الأمثلة عليهم: الإدارة العليا وفرق التطوير وال vendors الذي يقدمون تقنيات وأدوات مستخدمة لدينا...

علينا أن نعلم أن النجاح في تطوير البرمجيات أو في تصميم الأنظمة لا يعتمد فقط على التواصل مع المستخدمين النهائيين، بل على التعاون المنتظم مع جميع أصحاب المصلحة داخل المؤسسة! لأن ذلك سيقينا من خطر الانعزال ومن ثم افتراض افتراضات خاطئة بناء على تخيلاتنا نحن لا الحقيقة! بل إن المفاجأت التي قد تنتج عن العمل بمعزل عن الآخرين غالبا ما يعني = إعادة العمل وإلقاء المجهود المبذول في القمامة... ومن الأمثلة على ذلك تخيل أن تعمل على لغة برمجة لتنفيذ نظام ما ثم تكتشف أن هذا النظام غير مدعوم من نظام التشغيل أو من السيرفر! أو تخيل أنك تعمل على نظام لمنطقة تشترط سياسة خصوصية معينة مثل الاتحاد الأوروبي وأنت لم تراعي آلية حفظ البيانات واسترجاعها ونحو ذلك!

إن مثل هذه الأخطاء والمواقف = فشل خطير، لأن نتائجه قد تظهر متأخرة؛ أي عند التسليم!

من الأمثلة على الفئات التي يجب أن تبقى على تواصل معها لضمان أن النظام سيتكامل وسيعمل بشكل صحيح كما هو متوقع:

- فرق التطوير (الحالية والمستقبلية): وذلك لضمان الاتساق المعماري وفهم الأسباب وراء القرارات التصميمية.
- مديرو قواعد البيانات (DBAs): وذلك لضمان الاستخدام السليم للموارد، والأمان، والتصميم...
- فرق التشغيل والدعم: وذلك لتحديد كيفية النشر والمراقبة وتشخيص المشكلات.
- فرق الامتثال والمخاطر والأمان وخصوصية البيانات: وذلك لضمان التزام النظام باللوائح المؤسسية والقانونية (مثل GDPR).

الخلاصة التي نريد أن نصل إليها:
"الأنظمة البرمجية لا تعمل في عزلة عن العالم ولا عن الأشخاص! وعملية التصميم لهذه الأنظمة يجب أن تكون منصة للحوار والتعاون، فمحادثة قصيرة ومبكرة لمدة ٥ دقائق قد تمكنك من الكشف عن عوامل معمارية ضمنية حاسمة كانت ستودي بك للكثير من العمل أو التكلفة أو حتى للفشل"! = إذا لم تتعاون، **فتوقّع الفشل!**

٤. مجال/نطاق العمل وأهمية فهمه والإحاطة به -Domain knowledge:

إن معرفة نطاق العمل والفهم الواقعي لمجال العمل يعد أمر أساسيا لنجاح المعماري! خصوصا لأننا بحاجة لاكتشاف التعقيدات الخفية وفهم الأهداف الحقيقية للعمل الذي نريد بناء تصميم لخدمته! فهل يعقل أن تبني تصميم لسفينة وأنت لا تعرف عن البحر والموانئ شيئا؟!

إن هذه النقطة المهمة تعتبر واحدة من أهم التحديات التي قد تواجهها كمعماري! وقد تكون عشتها حقيقة بصفتك مطور برمجيات خبير انتقل خلال فترة عمله في عدة شركات لكل منها مجاله، فهذه شركة طبية وأخرى مالية وثالثة لوجستية... إلخ، وقد تكون عملت في شركة استشارات تقنية تقدم وتبني حلولا للشركات الأخرى! وكل هذا يعني أن الخبرة العميقة المتخصصة في هذا المجال لن تكون متوفرة، وهنا سيظهر لنا سؤال خطير ومهم، وهو كيف يمكن لنا أن نتعامل مع هذا التحدي؟
والجواب يمكن اختياره من أحد خيارين وهما:

- التخصص في مجال واحد فقط: هنا يقوم المعماري بتقييد نطاق عمله ليكون في مجال واحد -مثل القطاع الطبي أو المالي- لاكتساب خبرة عميقة وكبيرة فيه، وهذا سيضمن لهذا المعماري فهما ممتازا وخبرة حقيقة كبيرة مبنية على واقع عملي وتجارب كثيرة، مما يعطيه نقطة قوة في سوق العمل عند بحث المؤسسات عن معماري في هذا المجال... ولكن لهذا الأسلوب مشكلة خطيرة وهي: حده من التنوع المهني؛ مما سيقلل فرصه في السوق أيضا ويخلق له صعوبات في الانتقال لشركات تقدم حلول تقنية متنوعة!
- التنوع: هنا يكون المعماري مستمتعا بالتنقل بين عدة مجالات عمل مختلفة؛ وذلك لاكتساب قدر كافي من تنوع الخبرات في عدة مجالات، مما يعطي قدرة قوية لتبادل الأفكار بين الأعمال والصناعات المختلفة... وهذه نقطة قوة وتفوق قوية، لكنها أيضا تحتاج لمعالجة بعض المشاكل حتى يصبح هذا الأسلوب فعالا وناجحا! من ذلك على المعماري تطوير مهارات التحليل الخاصة به لدرجة أو مستوى عال، وذلك حتى يتمكن من الاستيعاب السريع للأجزاء الجوهرية والأساسية لأي نطاق عمل، وبذلك يصبح منتجا وفعالا في وقت قصير، كما عليه أن يتجنب الشلل التحليلي، والذي يقصد به معرفة متى يجب عليه التوقف عن جمع التفاصيل والبدء في العمل واتخاذ القرارات المعمارية... كما أن لهذا النهج نقطة سلبية وهي أن المعرفة المكتسبة في نطاق سنعمل عليه ستكون محددة ولن تكون عميقة كمن كان يعمل في هذا المجال بشكل كامل أو مستمر، ومع ذلك، يعتبر هذا الأسلوب للمعماري أفضل من الأسلوب الأول بشكل عام.

معلومة: هل سمعت من قبل ب: "Razor-sharp Analysis Skills"؟ إن هذه العبارة تمثل استعارة بلاغية للدلالة على وجود حاجة أو قدرة على التفكير التحليلي، وترجمتها الحرفية: "مهارات التحليل الحادة"! وهي تتطلب من الشخص القدرة على استخلاص جوهر المشكلة واقتناص الفرص بسرعة ودقة ووضوح، ودون أن تشتته التفاصيل الصغيرة غير الجوهرية ^^، لذلك تتضمن هذه المهارة التمييز بين الأهم والمهم والهامشي، بحيث يتم تحديد المتطلبات الحاسمة لاتخاذ أي قرار معماري فوري أو مؤثر على التصميم، وتأجيل ما يمكن تأجيله، وهذا سيقودنا لمهارة أخرى تتضمنها هذه العبارة وهي القدرة على الاستنتاج السريع من معلومات ناقصة أو غير مكتملة، فالمشاريع الجديدة في المجالات غير المألوفة ستحتوي على كثير من المعلومات، لكن ما سيقدم لك هو مجموعة البيانات أو المعلومات المتاحة لخدمة المشروع، وبهذا قد ينقص لديك العلم ببعض التفاصيل المهمة والضرورية، ومع ذلك فمهارة الاستنتاج السريع ستقودك لتحليل حاد للمتطلبات مما يمكنك من وضع افتراضات معقولة ستقودك للبدأ بالعمل دون انتظار الكمال، وهذا سيساعده مهارة ربط الأنماط المتشابهة بين الصناعات المختلفة، فمثلا نظام حجز المواعيد في العيادات قد يشبه نظام حجز المواعيد في تذاكر الطيران -من ناحية الحاجة إلى إدارة ال Availability ومنع التعارضات بالحجز ونحو ذلك-... إلخ.

إن هذه المهارة مهمة جدا للمعماري الذي سار بالنهج الثاني -التنوع- لأنه لا يملك رفاهية الوقت ليصبح خبيرا في كل مجال! وهو مع ذلك سيجبر على التنقل بين عدة مجالات عمل مصرفية أو صحية أو لوجستية ولكل منها لغتها ومعارفها وقيودها الخاصة...

الخلاصة: إذا كنت معماريا فعليك أن تملك معرفة عميقة في مجال معين أو أن تمتلك مهارة تحليل حادة تمكنك من استيعاب الأجزاء الجوهرية والأساسية في المجالات المتعددة؛ بسرعة ودون الوقوع في فخ التحليل المفرط أو التفريط المخل!

٥. الانتقال من مطور إلى معماري:

لقد ذكرنا سابقا أن الخط الفاصل بين دور المطور والمعماري ليس واضحا -ضبابي-؛ فكثير منا الأحيان يؤدي المطور أجزاء من دور المعماري، ودون أن يكون هذا هو مسماهم الوظيفي! وبذلك نؤكد على أن التحول من مطور إلى معماري هي عملية تطور في المهارات والخبرات عبر الزمن وبشكل تدريجي حتى تصبح جاهزا لأداء دور المعماري بنجاح!

لكن، ألا نحتاج آلية لتقييم جاهزيتنا كمعماريين؟!
الجواب بكل تأكيد نعم!
إن عملية تقييم دور المعماري لا تعني تقييم الجانب المعرفي التقني فقط! بل هي عملية تقييم شاملة تشمل كل من السلوك والمسؤولية والتأثير والمشاركة الفعالة عبر مراحل المشروع، وبهذا نستنتج أن التركيز لن يكون على على جزء صغير مثل: "هل يعرف kubernetes؟" بل سيكون على: "هل يستطيع هذا الشخص -من سيلعب دور المعماري- أن يُوجه النظام ككل؟ وأن يتحمل مسؤولية قراراته؟ وأن يتفاعل باستمرار مع الفريق لضمان نجاح التنفيذ؟"... ومن هنا يمكننا أن نبني جدولا أو نطرح مجموعة من النقاط التي ستساعدنا على تقييم أنفسنا كمعماريين أو تقييم من نقابل لوظيفة معماري أو عند اختيارنا لأحد المطورين

ليقوم بهذا الدور، ومن ذلك:

ملاحظة: سنفترض أننا في شركة ناشئة في مجال التعليم الرقمي، وتريد هذه الشركة بناء منصة جديدة لتقديم دورات تفاعلية، مع ميزات مثل: البث المباشر، ولوح للكتابة بشكل مشترك مع المعلم والطلاب، والقدرة على تسجيل الدروس، والتكامل مع أنظمة الدفع الرقمي...

- المحفزات المعمارية -Architectural drivers- والأسلوب الاستباقي -Proactive-: ويقصد بذلك أن المعماري الناجح يجب أن يتعامل مع مهامه بإسلوب استباقي؛ فيتوقع المشكلات قبل حدوثها، ويتخذ الإجراءات المناسبة لحل هذه المشكلات قبل أن تحصل، أما إذا كان سلوكه مجرد رد فعل فهذه نقطة سلبية في تقييم هذا الشخص... مثال:
  - شخص غير جاهز لأداء دور المعماري: وصلت لفريق التطوير ما تحتاجه هذه الشركة وبدأ الفريق بالعمل وما زال المعماري ينتظر أن يُطلب منه تصميم المعمارية أو أن تكتمل كل المتطلبات والمعلومات قبل أن يبدأ بشيء!

  - شخص جاهز لأداء هذا الدور: بمجرد معرفته حاجات هذه الشركة باعتبارها ناشئة؛ قام المعماري بشكل استباقي بطرح الأسئلة الاستراتيجية المهمة مثل كم عدد المستخدمين المتوقع في الذروة؟ وهل يجب أن يكون التسجيل متاحا فورا بعد انتهاء الدرس أم يمكن تأخير ذلك لوقت معين؟...إلخ، ثم قام هذا المعماري بتقديم مسودة أولية للمحفزات المعمارية -Architectural-Drivers.

وهذا يعني: أن المعماري الناجح سيقود التغيير ويقلل من المخاطر بشكل استباقي -Proactive-؛ بدلا من أن يستجيب للأزمات التي كان يمكن تجنبها -Reactive-.

- المحفزات المعمارية وتحديات سمات الجودة المعقدة: ويقصد بذلك فحص قدرة المعماري على التقاط سمات الجودة المهمة بشكل استباقي ومعرفة تحدياتها ومحاولة معالجتها بدلا من تجاهلها أو افتراض وجودها دون مقدمات حسية أو حقيقية... مثال:
  - شخص غير جاهز لأداء دور المعماري: قال المعماري للفريق: "إن الخصوصية مهمة جدا" واكتفى بذلك، ولم يحدد متطلبات ملموسة يمكن للفريق التعامل معها!

  - شخص جاهز لأداء هذا الدور: قال المعماري للفريق: "لقد قمنا بتعريف سمات الجودة التي نحتاجها، وهذه النقاط المهمة التي يجب عليكم مراعاتها: ففي جانب الخصوصية عليكم ألا تسمحوا بتسجيل الشاشة دون موافقة المعلم، ويجب على الأعضاء الآخرين أن يعرفوا أن المحاضرة في حالة تسجيل أو لا... كما أن لدينا جانب يعزز موثوقية النظام، فيجب أن يعاد الاتصال تلقائيا خلال ٥ ثوان كحد أعلى!"... لاحظ أن المعماري الجيد هنا قدم سمات الجودة بشكل قابل للقياس، ولم يتجاهل التعقيد! بل حول هذا التعقيد لمتطلبات قابلة للتنفيذ.

- تصميم البرمجيات: ويقصد بهذه النقطة قدرة الشخص الذي سيقوم بدور المعماري على تصميم الأنظمة الجديدة بالكامل -Greenfield- أو توسيع الأنظمة القائمة -Brownfield-، وذلك لأن الأنظمة التي سيتم بناؤها بشكل جديد بالكامل فيها تحديات أصعب -غالبا- من التعديل على نظام موجود، وكذلك المهارات التي ستحتاجها لجمع الأفكار ورسم المخططات وجمع البيانات وتحليلها ونحو ذلك... ومع ذلك فهذا الأمر غالبا وليس دائما...

    - شخص غير جاهز لأداء دور المعماري: إذا لم تكن لديه قدرة على تصميم نظام جديد بشكل صحيح وانطلق من الفراغ أو رفض / تجاهل التعامل مع أنظمة قديمة كان من الممكن أو من الأفضل صيانتها أو توسعتها أو الانطلاق من خلالها والتكامل معها... مثال: قام هذا الشخص بتصميم نظام جديد بالكامل وكأنه منعزل عن العالم ولم يراعي وجود نظام قديم قيد العمل الآن.

    - شخص جاهز لأداء هذا الدور: إذا كانت لديه القدرة على تصميم نظام جديد بشكل صحيح وخطط للتكامل مع النظام القديم أو وضع أسس وآليات الانتقال من القديم إلى الجديد، أو اعتمد على صيانة النظام القديم باعتباره مشروع قائم بنجاح والتصميم القديم يؤدي الغرض مع الحاجة لبعض التعديلات والتوسعة... مثال: قام هذا الشخص بتصميم نظام جديد بالكامل وخطط ليتكامل هذا النظام الجديد مع القديم من خلال واجهات API، ولعزل التعقيد قام بإنشاء adapter لهذا الغرض.

معلومة: ال Greenfield Project أو المشروع الأخضر هو ذلك المشروع الجديد الذي سيأتي على حقل أخضر مليء بالأعشاب لم يبنى عليه شيء من قبل، وهي بذلك تمثل منطقة سيتم البناء عليها من الصفر، وهذا يجعل من خصائصها الحرية الكبيرة فلا يوجد أنظمة يجب التكامل معها أو الالتزام بها، كما يمكن للفريق استخدام التقنيات الحديثة وأنماط التصميم دون قيود تذكر... ومع ذلك فهو يحتاج إلى رؤية معمارية قوية وقدرة على التصميم وربط المكونات عالية ^^، أما ال Brownfield Project فهو يمثل ذلك المشروع الذي يأتي على حقل قد امتلأ طينا من البناء أو مكوناته، وهو بذلك يمثل منطقة سيتم إضافة بناء جديد إليها أو صيانة وتعديل مبنى موجود فيها بالفعل، ومن خصائصها وجود قيود يجب الاهتمام بها للتعامل مع الأنظمة والشيفرات البرمجية القديمة، والقدرة على التكامل معها والالتزام بالبنية التحتية قدر الإمكان، وغالبا ما تكون المهام المرتبطة بهذا النوع إما إضافة مزايا جديدة أو إعادة تصميم أجزاء من النظام أو دمج النظام القديم مع أنظمة جديدة... وتحديه الأساسي هو ازدياد صعوبة العمل بسبب القيود والتحديات الناتجة بسبب ما ينبع من الشيفرة البرمجية القديمة وتصميمها وتوافقها مع الجديد ونحو ذلك، ومن هنا يظهر لنا أسلوب هجين يجمع بينهم يطلق عليه:
"Green-Brown Hybrid"، وهو نهج معماري يحاول الاستفادة من مزايا ال Greenfield والاستفادة أو التكامل مع ال brownfield... وهذا

الأسلوب تفضله المؤسسات الكبيرة بشكل جلي، وهو ما حولنا إظهاره في المثال السابق.

ملاحظة: على المعماري في حالة تصميم أنظمة ال Greenfield أن يكون على وعي كبير بحيث لا يقع ضحية الفراغ، لذلك هناك مصطلح يشار إليه في هذا الباب وهو: "تحدي الفراغ"، وهو قدرة المعماري على تحدي نقص القيود -لأنه حر ويمكن أن يستخدم ما يحب من تقنيات- فلا يفرط في التصميم فيصل لتعقيد لا داع له، أي لا نريد أن نصل إلى ال Over-engineering! وعلى المعماري في حالة تصميم أنظمة ال Brownfield أن يكون على مهارة كبيرة في الهندسة العكسية -Reverse Engineering-، أي قدرته على تحليل النظام القديم وفهم كيفية وآلية عمله داخليا وذلك ليتمكن من بناء تصميم يتكامل معه أو أن يكون قادرا على إعادة تصميمه أو تصميم جزء منه أو حتى جمع المتطلبات منه! وسيكون من المهم أن يكون قادرا على وضع قرارات تقنية تتماشى مع النظام القديم وتتعايش مع النظام الجديد، وهذا يعني تصميم يركز على التكامل والمرونة...

- المخاطر التقنية: ويقصد بذلك إثبات أن المعمارية ستعمل حقا، لا من خلال تمني ذلك والعيش في الأحلام الوردية ^^...

- شخص غير جاهز لأداء دور المعماري: قال هذا الشخص: سنقوم باستخدام ال WebRTC لأنها شائعة وكثير من التطبيقات تستخدمها لغايات البث المباشر!

- شخص جاهز لأداء هذا الدور: قام هذا الشخص بإجراء Spike لاختيار التقنية المناسبة للمشروع من بين ال WebRTC وال RTMP وقارن بينهم من خلال مقارنة ال latency وسهولة التكامل ودعم المتصفحات والأمان والتشفير وقابلية التوسع ونحو ذلك، ثم اختار التقنية المناسبة بناء على هذه التجربة.

معلومة: ال Spike هي تجربة قصيرة ومحدودة النطاق تُجرى لاستكشاف فكرة ما أو تقنية جديدة دون بناء نظام كامل، والهدف منها هو اختبار الجدوى التقنية ومحاكاتها على أرض الواقع قبل الالتزام بتصميم التطبيق بشكل كامل اعتمادا على قرار دون تجربة.

- القيادة التقنية: ويقصد بذلك الانخراط المستمر في تطوير المعمارية بدلا من تسليمها للفريق ثم تركهم يتعاملون مع المشكلات والابتعاد عنهم لشرب القهوة على شاطئ البحر -.-...

- شخص غير جاهز لأداء دور المعماري: يقوم المعماري بتسليم التصميم للفريق ثم يقول "نفذوا هذا يا شباب وأموركم طيبة"! ثم يختفي -.-...

- شخص جاهز لأداء هذا الدور: يقوم هذا المعماري بتسلم التصميم للفريق ومن ثم يحضر ال stand up ليبقى على اطلاع على آخر المستجدات، كما أنه يراجع بعض ال PR ويجري جلسات معمارية بشكل أسبوعي لمتابعة أو مناقشة أي تحديثات جديدة ناشئة...إلخ.

- ضمان الجودة: ويقصد بذلك الحاجة لتحديد مجموعة الإرشادات والمبادئ والمعايير لضمان الجودة بشكل مستمر، بدلا من عدم فعل أي شيء وافتراض أن الفريق سيقوم بعمل جيد ^^...
  - شخص غير جاهز لأداء دور المعماري: المعماري يبتعد بعد تسلم التصميم ويقول سيقوم الفريق بعمل جيد وسيكتب شيفرة برمجية نظيفة *-*...

  - شخص جاهز لأداء هذا الدور: قام هذا المعماري بتعريف مجموعة من المعايير الواضحة مثل كل خدمة سيتم كتابتها يجب أن يكون لديها testing يغطي ٨٠٪ كحد أدنى، كما لا يسمح باستخدام أي dependency من مصدر غير موثوق أو لم يتم تحديثها منذ سنة...إلخ.

النتيجة المهمة: من هنا نستنتج أن هذه النقاط هي نقاط تقييم تقييس **الاستباقية والواقعية والمرونة والمسؤولية والقيادة والانضباط**... وكل نقص في واحدة من هذه النقاط دليل على وجود جانب يقلل من جاهزية هذا الشخص ليكون معماري...

الخلاصة:

إن الفرق الجوهري بين المطور والمعماري يكمن في الفرق بين المساهمة في المعمارية وبين تحمّل المسؤولية الكاملة عنها! كما يتطلب دور المعماري سلسلة متصلة من المهارات المكتسبة بوعي، لذلك تشبه عملية الانتقال من مطور إلى معماري = عملية العبور من حد مليء بالضباب الذي يغشى العيون، لذلك: يتطلب هذا العبور قرارا شخصيا من المطور لتقييم خبرته الحالية ومعرفة أين يحتاج إلى تركيز جهوده ومهاراته في القيادة والمسؤولية.

ملاحظة: يقصد بال "القيادة والمسؤولية" في الفقرة السابقة بأن الذي يرغب بأن يصبح معماري ناجح عليه تحديد المجالات التي تنقصه الخبرة بها أو المهارة فيها، وأن يخصص جهده في زيادة مسؤولياته وقدرته القيادية فيها، وذلك من خلال تحديد فجوات المهارة أو المعرفة التي لديه، ثم تركيز جهده لزيادة سلطته ومسؤوليته فيها، وذلك من خلال العمل على سد الفجوات بشكل استباقي من خلال اكتساب المعرفة المطلوبة والخبرة اللازمة في هذا المجال، ويمكن أن يطلب الصلاحية المناسبة للقيام بهذا الدور أو من خلال الاتفاق مع فريقه لقيادة الفريق، أو أن يتطوع في كتابة التوثيق لمعماري خبير، أو أن يأخذ زمام المبادرة في حل الخلافات التقنية داخل الفريق، أو أن يحضر الاجتماعات التي يتم فيها جمع المتطلبات ونحو ذلك.

٦. التدريب والإرشاد = بناء قادة الغد:

تاريخيا كان البناؤون يتبعون نموذجا واضحا مكون من: متدربون وعمال رحّل وبناؤون رئيسيون، وهؤلاء هم من يقوم بنقل هذه المعرفة للأجيال القادمة! لكن صناعة البرمجيات تفتقر حاليا إلى نموذج تدرّب -Apprenticeship Model- محدد للتقدم من مطور مبتدئ -Junior- إلى معماري أو قائد تقني، وهذا النقص أو غياب نموذج التدرّب أدى إلى إهمال نشاطات التدريب -Coaching- والإرشاد -Mentoring-... وهو بذلك يؤثر على الشركة نفسها أو على انتقال المعرفة والخبرة للأجيال التالية في الشركة أو على مستوى أكبر...

إن المعماريون والقادة التقنيون يتحملون مسؤولية التدريب والإرشاد؛ لأنهم يمتلكون الخبرة التقنية العميقة اللازمة والمهارات الناعمة التي تؤهلهم لمشاركة ما لديهم! فالإرشاد يطور مهارات الأفراد التقنية والشخصية، ويضمن بقاء "المهنة حية" للأجيال القادمة [٨]، ومع ذلك، فمن المؤسف أن العديد من المؤسسات تدفع أفضل وأقدم خبرائها التقنيين للانتقال من المناصب التقنية إلى مناصب إدارية غير تقنية كطريقة وحيدة للتقدم الوظيفي! وهو ما يمكننا تسميته ب "مكافأة الخبراء التقنيين = بإبعادهم عن التقنية"[٨]، وهذا الأمر سيحرم فرق البرمجيات من قادتها التقنيين الأكثر قدرة على الإرشاد والتوجيه، مما يضيف عبئا على بقية الفريق... وكمثال عملي، تخيل أن لدينا شركة س، هذه الشركة لديها معماري ناجح قام بإنقاذ الشركة من العديد من المشاكل التقنية، كما أنه أشرف على ٥ مشاريع كبرى فيها، وبعد مضي عدد معين من السنوات، قامت الشركة بترقية هذا المعماري ليصبح Engineering manager، وستكون مهامه الوظيفية هي التركيز على الميزانية والتقارير واجتماعات الموارد

البشرية ونحو ذلك، مع منعه من المشاركة في تصميم الأنظمة أو الإشراف عليها، وسيتم تعيين معماري مبتدأ لتولي هذه المهام الفنية... إن نتيجة هذا القرار غالبا ما ستظهر في الشهور ال ٦ الأولى مثل أخطاء معمارية أو تعارضات تصميمة أو حتى بطء في الإنتاج... إن الشركة هنا كافأت المعماري لتميزه التقني، لكن بحرمانه من ممارسة التقنية وحرمان الفريق من مهاراته! والحل الصحيح يكون بتعريف مسار تقدم وظيفي يناسب هذا المعماري ليكون مسارا قياديا تقنيا يسمح لهذا المعماري المخضرم بالترقي مع زيادة في الأجر والتأثير ودون التخلي عن دوره الفني والإرشادي...

نصائح للـ Software Architects الجدد

في آخر موضوع لنا في هذا الباب، لا يمكننا أن تجاوز فكرة مهمة، وهي أن دور المعماري دور مليء بالتحديات، بل وتحدياته كبيرة ومختلفة ومتنوعة! لذلك فالنجاح من المرة الأولى قد لا يحصل إلا في أحلامنا الجميلة! لذلك هناك بعض النصائح التي ستساعدك على النجاح، وهي:

- افهم دور المعماري بشكل جيد: احرص على فهم مسؤوليات المعماري بدقة، واعرف مسؤولياتك اليومية كمعماري.

- ابحث عن مرشد -Mentor-: ابحث عن شخص لديه خبرة في المجال لمساعدتك في عملية الانتقال إلى دور المعماري، أكان هذا الشخص داخل مؤسستك -وهو

المفضل- أو خارجها...

- اطلب المشورة عند الحاجة: لا تتظاهر بمعرفة كل شيء! فلا أحد يعرف كل شيء مهما بلغ من العلم والخبرة! وزد على ذلك بأن إدراكك لحدود معرفتك = أول خطوة نحو الارتقاء، وأول خطوات تهذيب النفس للعمل في بيئة صحية وأكثر تعاونية... وكما ذكرنا سابقا: فإن دور المعماري يمكن أن يقوم به شخص أو فريق، فلا مانع من ذلك أبدا! لذلك: لا تحاول أن تتظاهر بأنك تعرف كل شيء، واطلب المشورة من داخل الفريق وخارجه عند الحاجة!

- افهم فريقك: جزء من الدور القيادي أو السمات القيادية هو أن تفهم فريقك أو الفريق الذي تعمل معه! مثل السياسات الداخلية ونقاط القوة والضعف لدى الأفراد وآلية التواصل فيما بينهم وكيف يتعاونون لحل المشاكل المختلفة ونحو ذلك... دون هذه المعرفة سيكون من الصعب اتخاذ القرارات الصحيحة لقيادة الفريق.

- لا تتجاهل فريقك: إياك أن تتجاهل ملاحظات الفريق، سواء قيلت لك مباشرة أو ظهرت من خلال تلميحات ضمنية؛ لأن المشاعر السلبية تجاه المعمارية قد تنتج مشكلات حقيقة لا يمكن حلها بسهولة! لذلك إذا شعرت أن هناك آراء سلبية ذات قيمة حقيقة فقم بمعالجتها...

- مارس بعض التحكم: لا تفرط في مفهوم "التنظيم الذاتي -Self Organising-" للفريق، لأن هذه الفكرة تعمل بكفاءة مع الفرق ذات الخبرة الكبيرة، لكن أغلب الفرق ليس هذا حالها، وهذا ما سينتج فوضى عارمة وشيفرة برمجية غير متسقة! لذلك: اضبط الحدود وضع المبادئ التوجيهية والإرشادات والمبادئ حتى يتبعها الفريق، ويفضل أن يتم ذلك بشكل تعاوني إلا إذا كانت الظروف غير مواتية وتتطلب الحزم.

- كن استباقيا وأصلح المشكلات مبكرا: عندما تلاحظ وجود تجاهل للمبادئ المعمارية أو وجود شيء يسير بشكل خاطئ فحاول إصلاح ذلك بشكل سريع؛ لأن المشكلات لن تُحل من تلقاء نفسها!

- اتخذ قرارات: تجنب التردد واتخذ قرارات تساعد على تشغيل محرك العمل والإنجاز -حتى لو كانت افتراضات مبدئية- بدلاً من عدم اتخاذها على الإطلاق! هناك مصطلحان يساعدان على فهم هذه النقطة، وهما: Last Responsible Moment و Stake in the sand... إن كثير من الأشخاص ما يستخدمون ال Last Responsible Moment لتبرير كسلهم لا الواقع الحقيقي، وكثير منهم من يرغب بالجلوس دون أن يتقدم نحو الأمام حتى بوجود افتراضات منطقية يمكنهم الانطلاق من عندها... مع أن هذا المصطلح يخبرك ب: "ألا تتخذ القرار قبل أن يكون لديك ما يكفي من المعلومات حتى تتخذه بذكاء، لكن لا تؤجل اتخاذ القرار أكثر من اللازم بحيث يضرك التأخير"! بمعنى آخر: "إن الانتظار المقبول هو الانتظار لآخر لحظة مسؤولة -لاحظ مسؤولة وليس ممكنة-، إذا تجاوزناها دون قرار سيبدأ المشروع بعدها بالتأثر

سلبا أو بالانهيار"! وهنا يأتي دور المصطلح الثاني Stake in the sand، وهو مصطلح قديم مرتبط بغرس أوتاد في الرمال لغايات معينة ومؤقتة، ويقصد به هنا هو اتخاذ قرار مبدئي وغير نهائي ليستخدم كمرجع للعمل ولتحريك الفريق نحو الهدف بدلا من الجمود والانتظار حتى اللحظة المثالية! وهذا القرار يكون ظاهرا بوضوح للجميع باعتباره قرارا مبدئيا وليس نهائيا... دعونا نشاهد مثالا عمليا على هذين المثالين: إذا قلنا أننا فريق يريد أن يبني موقع تجارة إلكتروني لكنه غير متأكد بعد من أن ال MySql أو ال MongoDb ستكون هي الأنسب، والتحليل يحتاج إلى ثلاثة أسابيع لتحليل كافة الاحتمالات... هنا ظهر المعماري الناجح ليقول: سنبدأ بال MySql ك Stake in the stand لأنها تدعم المرحلة الحالية بشكل جيد، وسنختبر الأداء عند ١٠ آلاف مستخدم، وإذا ظهرت لدينا مشاكل في ال scaling سنعيد التقييم في ال sprint السادس... ثم قال صديقنا المعماري الناجح للفريق: سيكون ال Last Responsible Moment لاتخاذ القرار النهائي هي قبل كتابة ال Data Access Layer؛ لأن تغيير هذا الجزء بعد ذلك سيتطلب إعادة كتابة جزء كبير من النظام، وبهذا فإن النتيجة هي أن القرار النهائي سينتظر حتى تُجمع بيانات أداء حقيقية، ولكنها في نفس الوقت لن تتجاوز هذا الـ LRM ^^...

ملاحظة: هناك خلط كبير يحصل عند كثير من الناس بين ال LRM و"آخر لحظة ممكنة"، فمثلا: لو قام معماري باختيار نظام Auth للتطبيق الخاص به قبل نشره بيومين فهذا سيكون تهور وانتحار! ولن يكون LRM بكل تأكيد! لأن تغيير ال auth في هذه اللحظات قد يدمر كل شيء! أما ال LRM الحقيقي ممكن أن يكون في هذا المثال: إن

القرار النهائي لاتخاذ نظام ال auth المرغوب هو قبل البدء بتطوير واجهات المستخدم التي تعتمد على بياناته... هل لاحظت الفرق الخطير؟

- وثّق القرارات الكبرى: يجب أن يكون هناك مرجعية دائما لمراجعة أي حل أو فكرة أو للحصول على تفاصيل مهمة أو لحل النزاع والمشكلات... إلخ، لذلك وثّق القرارات المهمة التي تكلف كثيراً عند تغييرها باستخدام أساليب خفيفة وسريعة مثل ال ADR.

- لا تكتب شيفرة برمجية طوال الوقت: كن واعيا بمقدار الوقت المناسب لهذه المهمة، واحجز وقتاً للمهام المعمارية!

- لا تغرق في التفاصيل ولا تتجاهل الشيفرة البرمجية: ابقَ على اتصال دائم بال Codebase ودون أن تفقد الصورة الكبرى للمشروع أو أن تفقد المحفزات المعمارية.

- تواصل باستمرار: تأكد من أن الجميع على دراية بالقرارات التي تم أو يتم اتخاذها والتقدم الحاصل بالمشروع، لأن المشكلات غالباً ما تنشأ من العمل الصامت!

- ابقَ على اطلاع: ابقَ على اطلاع على التقنيات الجديدة والاتجاهات الحديثة في عالم البرمجيات... بكل تأكيد ليس عليك فهم كل هذه التقنيات بالتفصيل! لكن من

المهم أن يكون لديك إلمام عام بما هو حديث منها ^^.

- تحمّل مسؤولية الإخفاقات، واحتفل بالنجاح كفريق: أي اترك غرورك عند الباب! فتطوير البرمجيات لعبة جماعية، والنجاح فيها عمل جماعي؛ فالأمر لا يتعلق بك وحدك أيا كنت! لذلك، تحمّل مسؤولية الإخفاقات، واحتفل بالنجاح كفريق!

فائدة

فلْتعلمْ يا أخي أن هناك أوجه متعددة لفعل الخير، فقد تكون في حال تقدر فيها على فعل الخير، فتفعله وتؤجر، وقد تكون في حال لا تقدر فيها على فعل الخير، لكنك لا تكل من حث الآخرين على ذلك!، وقد تكون ذو حظ عظيم، فتعمل الخير، وتحث عليه -اللهم اجعلنا منهم-!، ولْتعلمْ يا أخي أن هناك أوجه لفعل الشر، فإما أنك تقدر على الشر فتفعله، وإما أنك تحث على الشر ولا تقدر على فعله، وإما أنك تحث على الشر وتفعله، وهذا أشر الناس والعياذ بالله!، فاحرص على أن تكون من أهل الخير وخاصته!

- كتاب إلى الجنة زمرا، الصفحة ٧٦٢

الفصل الثالث: ال Architecting:

في هذا الفصل سنبدأ بالحديث عن العمل المعماري، وكيف ستتكامل المعلومات التي تعلمناها فيما بينها حتى هذه اللحظة، وما أثر ذلك على يومنا في عالم تطوير البرمجيات؟ ثم سنجيب على السؤال المهم الذي طرحناه في أول الكتاب، وهو: "ما مقدار التصميم المسبق -up-front design- الذي ينبغي علينا القيام به"؟

إذا كنت متحمسا لهذه الجولة الجديدة، فانطلق معي كأسد جائع ^^

إدارة المخاطر التقنية -Technical Risks-:

قبل أن نبدأ بالحديث بشكل مفصل، علينا أن نتعرف أولا على معنى المخاطر هنا -ال risks-، ويمكن القول ببساطة أن المخاطرة هي احتمال حدوث أمر سيء، هذا الحدث السيء قد يؤثر على المشروع أو النظام، ويأتي في أشكال وأحجام مختلفة ولا يجب حصرها في جانب واحد...

من خلال التعريف يمكننا أن نستنتج أن لدينا أبعاد مختلفة للمخاطر، وهذه الأبعاد يجب علينا أن نعيها كمعماريين، ويمكننا أن نفكر في ثلاثة أبعاد رئيسية للمخاطر، وهي:

1. المخاطر المتعلقة بالأشخاص -People Risk-: وهي المخاطر التي تتعلق بفريق العمل ومهاراته: مثل:

a. هل لدى الفريق العدد الكافي من الأفراد؟ وهل يمتلكون المهارات المناسبة؟ وهل يعمل أعضاء الفريق بانسجام؟
b. هل تُدار الأدوار القيادية كما ينبغي؟
c. هل يمكن توفير مستشارين للمساعدة في سدّ الفجوات المهارية عند الفريق؟
d. هل سنتمكن مستقبلًا من توظيف أشخاص بنفس المهارات لأغراض الصيانة؟
e. هل يمتلك فريق العمليات والدعم المهارات المناسبة لتشغيل البرنامج والعناية به؟
f. ما مدى احتمال مغادرة أحد الأعضاء الرئيسيين للفريق أو المنظمة؟

2. المخاطر المتعلقة بالعمليات -Process Risk-: وهي المخاطر التي تتعلق بكيفية تنفيذ المشروع، مثل:
a. هل يفهم الفريق منهجية العمل المتبعة؟
b. هل هناك توثيق يصف ويشرح كيف تسير ال process؟
c. هل هناك ميزانية كافية؟
d. هل يمكن أن تؤثر عوامل خارجية -تغييرات قانونية أو استحواذات ونحو ذلك- على سير العمل؟

3. المخاطر المتعلقة بالتقنيات -Technical Risk-: وهي المخاطر التي تتعلق بالحل المعماري نفسه مثل:
a. هل المعمارية المقترحة قادرة على تلبية سمات الجودة الرئيسية؟
b. هل التقنيات المختارة مستقرة وستعمل كما هو متوقع عند التنفيذ؟

بعد هذا الكلام الجميل، أظنك تتسائل، كيف يمكن لنا قياس خطورة -أهمية- المخاطر المحتملة أو الموجودة وكيف يمكننا ترتيبها؟

وحقيقة هذا أحد الأسئلة المهمة جدا، ويمكن الإجابة عنها باختصار من خلال: مصفوفة المخاطر! تريد المزيد من التفاصيل؟ إذا هيا بنا...

قياس وترتيب أولويات المخاطر -Quantifying and prioritising risks:-

إن المخاطر بطبيعتها ليست متساوية، فمن يهاجمه أسد ليس كمن يهاجمه كلب! ومن يهاجمه كلب ليس كمن تهاجمه قطة!

وهذا أيضا يفتح بابا آخر، وهو أولوية الخطورة التي تجعلنا نتخذ القرار الصحيح في تقييم الخطورة، فلو كنت في مدينة فيها الكثير من الكلاب، فاحتمالية أن تتعرض لهجوم من أسد تؤول إلى الصفر! فقط إن هرب من حديقة الحيوان ^^، في حين أن تعرضك لهجمات الكلاب أو القطط واردة...، وخطورة هجمات الكلاب أكبر فسيكون معدل الخطورة عندي للحذر من الكلاب أكبر... وعلى نفس هذا المنوال سنحاول التفكير في تقييمنا وترتيبنا للمخاطر المختلفة...

إن ما قمنا بفعله في المثال السابق يمثل بشكل ضمني مصفوفة المخاطر، ومصفوفة المخاطر هذه فيها شقين: الأول يمثل احتمالية حدوث الخطر، وهي البعد الأول في المصفوفة، ويطلق عليها Risk Probability! والثاني هو الأثر السلبي -العواقب- في حال حدوث هذا الخطر، ويطلق عليه Impact.

في هذين البعدين يمكن التعبير عن الاحتمالية والأثر بقيم لفظية أو رقمية مثل: منخفض، متوسط، عال الخطورة أو كتقييم من ١ إلى ٣ أو من ١ إلى ٥... ومن هنا يمكننا الانتقال لكيفية حساب ترتيب الأولويات -Prioritising-...

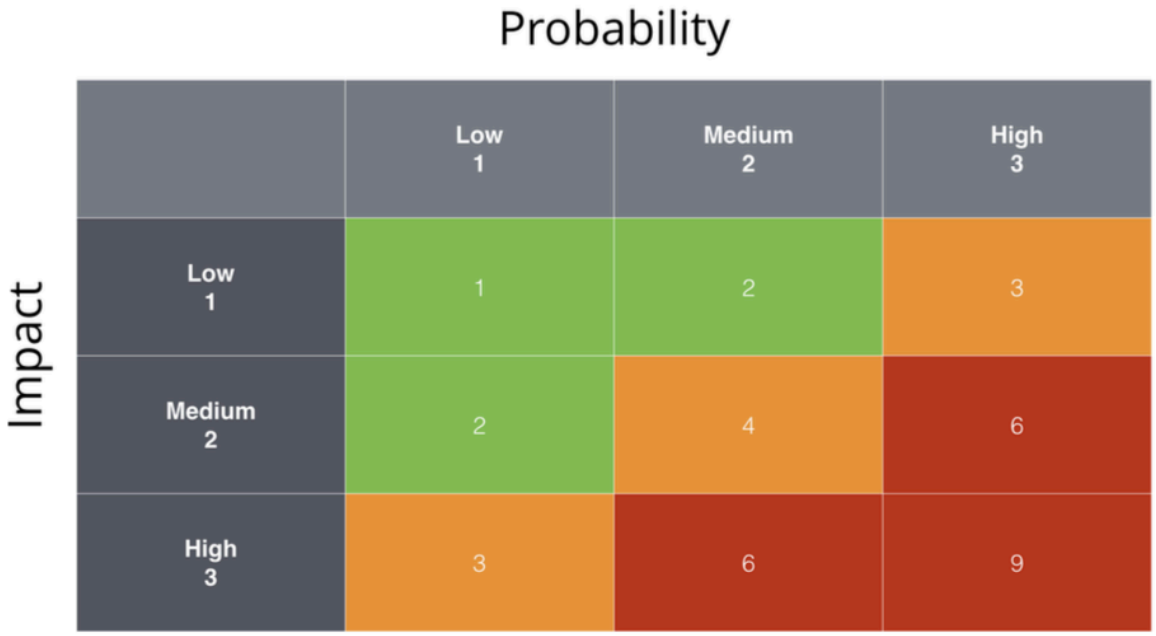
لحساب ترتيب الأولويات علينا أن نجد نتيجة المخاطرة -Risk score-، والتي يتم احتسابها بناء على المعادلة التالية:

الأثر -Impact- * الاحتمالية -Probability- = نتيجة المخاطرة -Risk score-

ثم يتم استخدام هذه القيمة لتحديد مستوى الأولوية من خلال استخدام نظام شائع من الألوان وهو: الأحمر والبرتقالي والأخضر بحيث يمثل: -افترض أن نظام التقييم من ١ إلى ٣-

- اللون الأحمر نتيجة عالية الخطورة تتطلب تدخلا فوريا، والقيم التي تمثل هذا اللون هي ٦ و ٩.
- اللون البرتقالي نتيجة متوسطة الخطورة تتطلب مراقبة وإجراء لتقليل خطورتها، والقيم التي تمثل هذا اللون هي ٣ و ٤
- اللون الأخضر نتيجة منخفضة الخطورة ويمكن قبولها ومراقبتها...

شاهد الصورة أدناه حتى تتضح الفكرة بشكل أكبر ^^



A probability/impact matrix for quantifying risk

ولعلك تتساءل الآن، هل هذه هي الطريقة الوحيدة لقياس المخاطر؟

الجواب بكل تأكيد: لا، بل هناك طرق قياس أخرى، لكن هذا الأسلوب الشائع في التطبيقات الشائعة... وهناك طرق أخرى تستخدم في بعض الأنظمة الحساسة والحرجة وأكثر رسمية، مثل:

- الأنظمة الطبية -تقييم مخاطر السلامة Safety Risk-: تخيل أن لديك فريق يطور برنامجا لوحدة العناية الحثيثة لضبط جرعات الأدوية المُعطاة تلقائيًّا عبر مضخة وريدية، وسيقوم البرنامج بحساب الجرعة بناءً على وزن المريض، وظائف الكلى، ونوع الدواء... تخيل وجود خطأ بسيط مثل استخدام وحدة ال "ملغ" بدلا من ال "ميكروغرام"؟ هل تعرف أن النتيجة قد تعني الوفاة؟ إذا ماذا يمكننا أن نعمل في هذه الحالة؟ هناك مقياس آخر مستخدم لهذه الحالة وهو: FMEA (Failure Modes and Effects

Analysis) ، وهو أسلوب قائم على تحليل نمط الفشل وتأثيره، ويتم استخدام هذا المقياس ضمن إطار ال IEC 62304، وهو إطار شامل ومعيار دولي وضع لدورة حياة البرمجيات الطبية... وباختصار يتم تطبيق هذا المقياس من خلال تحديد ٣ مكونات أساسية، وهي:

- الشدة -Severity-: من 1 (ضرر بسيط) إلى 10 (وفاة).
- الحدوث -Occurrence-: من 1 (نادر جدًّا) إلى 10 (متكرر).
- الكشف -Detectability-: من 1 (يُكتشف تلقائيًّا) إلى 10 (لا يُكتشف إلا بعد الضرر).

ولحساب نتيجة المخاطر هنا علينا بتطبيق المعادلة التالية:

الشدة * الحدوث * الكشف = نتيجة أولوية المخاطر (هنا يطلق عليها RPN أو رقم أولوية المخاطر)، وكلما زاد الرقم زادت الحاجة لمعالجة الخطر واتخاذ إجراءات وقائية... وليكتمل السياق بشكل عملي، لنتابع على مثالنا السابق... يعتبر ال RPN الأكبر من ١٠٠ عالي الخطورة وبعض التصنيفات فوق ٨٠ يعتبر عالي الخطورة -لأننا في المجال الطبي، قد تختلف القيمة في القطاع الصناعي مثلا-، بناء على ذلك إذا ظهر لدينا أي إجراء ذا قيمة أكبر من ١٠٠ سنقوم بعمل ما يلي:

أ. منع الإدخال اليدوي لأي وحدات غير معيارية.

ب. إنذار صوتي + بصري عند تجاوز حد أمان.

ج. طبقة تحقق مزدوجة.

بهذا نضمن أن الطبيب سيدرك أن هناك خطب ما إذا حصل أي خطأ، والجهاز لن يقوم بتنفيذ الجرعة حتى يشعر الطبيب بأن هناك خطورة محتملة وتأكيد أن ذلك آمن من خلال الطبيب... إلخ

لا تشغل نفسك الآن بالمثال السابق، فإنما الغاية منه هو توسعة معرفتك لتشمل طرق قياس أخرى... لكن، أرأيت كم كان ذلك جميلا؟

إن عملية تقييم المخاطر وتحديد أولويات معالجتها مهمة جدا، والخطأ هنا قد تكون نتائجه كارثية، وعلى أقل الأحوال خسائر مادية وإضاعة لأوقات الفريق بما لا طائل مباشر منه!

لكن، نحن تحدثنا عن كيفية قياس المخاطر وترتيبها، لكن كيف سأتمكن أصلا من تحديد المخاطر لقياسها!!!

تحديد المخاطر -Identifying Risks-: التعاون يمنع الرؤية المحدودة

هناك طريقتان للبدء في تحديد المخاطر الممكنة، الأولى فردية والثانية تشاركية... إذا ذهبت في اتجاه أن يقوم فرد واحد بهذه المهمة فحينها تأكد أن هذا هو أسوأ أسلوب يمكن السير فيه! وهو الأسلوب السائد في كثير من المؤسسات -للأسف-! والسبب يعود لكونه الأسلوب الأسوأ هو أننا سنحصل على رؤية محدودة تعتمد فقط على المنظور الشخصي لهذا الموظف، ومعرفته وخبرته، وبذلك سيكون تصور هذا الشخص للمخاطر ذاتيا، ولعل المثال الشائع على

هذه الحالة هو التخطيط لاستخدام تقنية معينة، فإذا كانت هذه المهمة موكلة لفرد واحد فستكون إجابة هذا السؤال هي نعم أو لا! نعم -مخاطرة- في حال لم يجربها هذا الشخص ولا -ليست مخاطرة- في حال جربها... لأن العوامل الشخصية مؤثرة بشكل كبير هنا!

إن النهج الأفضل هنا هو اعتماد رؤية تشاركية يشترك فيها الفريق كاملا لطرح وتحديد المخاطر المحتملة ومن ثم تقييمها وترتيبها، وهناك عدة طرق للقيام بذلك، وفكرة هذا العمل مشابهة لما نستخدمه من تقنيات لتقدير حجم مهمة معينة أو الوقت اللازم لإنجازها من خلال ال Planning Poker أو Wideband Delphi... ومن الطرق الفعالة لإتمام هذه المهمة في تحديد المخاطر وتقييمها تقنية ال Pre-Mortem، وهذا المصطلح يعني ما قبل الانهيار أو ما قبل الوفاة [٨]، وهو استراتيجية إدارية استباقية يقوم فيها الفريق بتخيل فشل المشروع كأمر واقع، ثم يعمل بشكل عكسي لتحديد جميع الأسباب المحتملة لهذا الفشل قبل وقوعه، وبهذا يمكن تشريح هذا المصطلح لثلاثة أجزاء:

1. التخيل: يتخيل الفريق أن المشروع قد فشل بالفعل.
2. العصف الذهني: يقوم الفريق بمناقشة جماعية حول جميع الأسباب المحتملة التي أدت إلى هذا الفشل.
3. التحديد: تُستخدم هذه الأسباب لتحديد المخاطر الحقيقية التي يجب التعامل معها في الوقت الحالي.

وكما تلاحظ، لعل الفائدة الرئيسية من اتباع هذا الأسلوب التشاركي هو الكشف عن مجموعة أكبر وأوسع وأكثر واقعية من المخاطر المحتملة والتي قد يتجاهلها الفرد إن كان لوحده...

على الهامش: ذكرتني هذه الكلمات ب "Devil's Advocate" والتي تعني محامي الشيطان، وتقوم فكرة هذا المفهوم على أن يقوم أحد الأشخاص بتقمص دور المحامي الذي سيعارض قرارات المجموعة أو يتبنى وجهة نظر مختلفة بشكل متعمد لاختبار قوة الحجج والأفكار المطروحة، وكذلك لاستكشاف الثغرات الممكنة والافتراضات الخاطئة، وكثيرا ما يستخدم في النقاشات الفكرية المؤثرة على صنع القرار، ولعلها تبرز بشكل قوي في التحليلات الاستراتيجية، العسكرية أو التجارية ونحو ذلك... وهذا يرتبط بما يطلق عليه الآن ب Red Teaming أو الفريق الأحمر والذي يعبر عن فريق يعمل في نفس الشركة أو لصالحها، يمثل جهة معادية لمحاكاة الهجمات المحتملة عليها، أكانت مادية أو رقمية! ويشتهر هذا المفهوم بشكل كبير في مجال الأمن السيبراني...

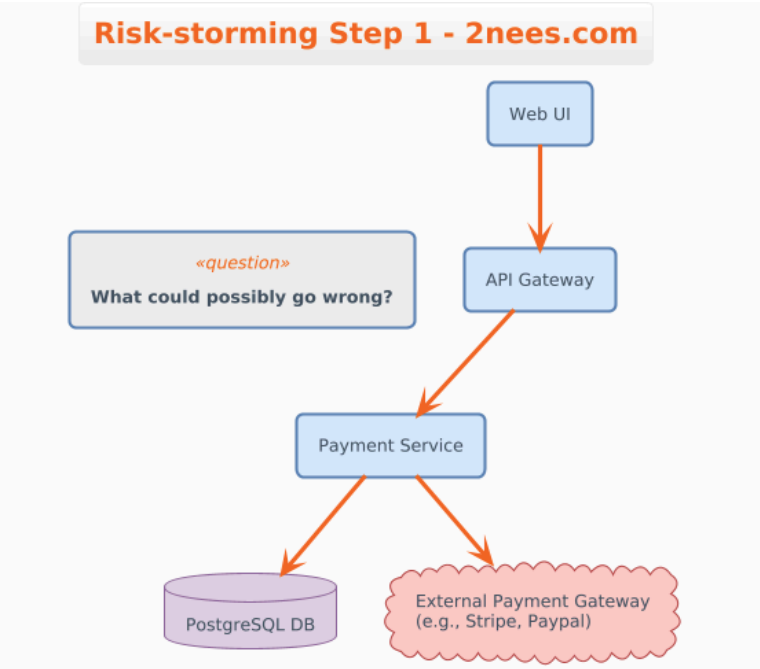
معلومة: مفهوم ال Wideband Delphi يشير إلى أسلوب يضم مجموعة من الخبراء في مجال معا يتشاركون في شكل جماعي لتقدير الجهد أو الزمن أو التكلفة في المشاريع أو الأعمال التي بين أيديهم، وتبرز بشكل قوي في مجال تطوير وتصميم البرمجيات وإدارة المشاريع التقنية، ولتنفيذ هذا الأسلوب يتم اتخاذ مجموعة من الخطوات تبدأ بتحديد المهمة وفيها يُعرف بوضوح ما الذي سيتم تقديره ثم يتم اختيار الخبراء المشاركين وعادة ما يكونوا من ٣ إلى ٧ أشخاص لديهم خبرة في المجال المطلوب ثم يتم عقد جلسة تمهيدية يتم فيها شرح الهدف والقواعد وطريقة حساب التقدير، وتوضيح نطاق المهمة بدقة ثم تبدأ الجولة من الأولى من التقديرات بحيث يقدم كل خبير تقديرا مبدئيا بشكل مستقل ودون نقاش ثم يتم عرض هذه التقديرات للجميع بشكل مجهول دون ذكر أسماء أصحابها ويحاول الفريق مناقشة أسباب التباين في القيم ثم تبدأ جولة ثانية من التقديرات بعد النقاش وعادة ما يتم تقديم تقديرات أكثر دقة هنا، ثم يتم

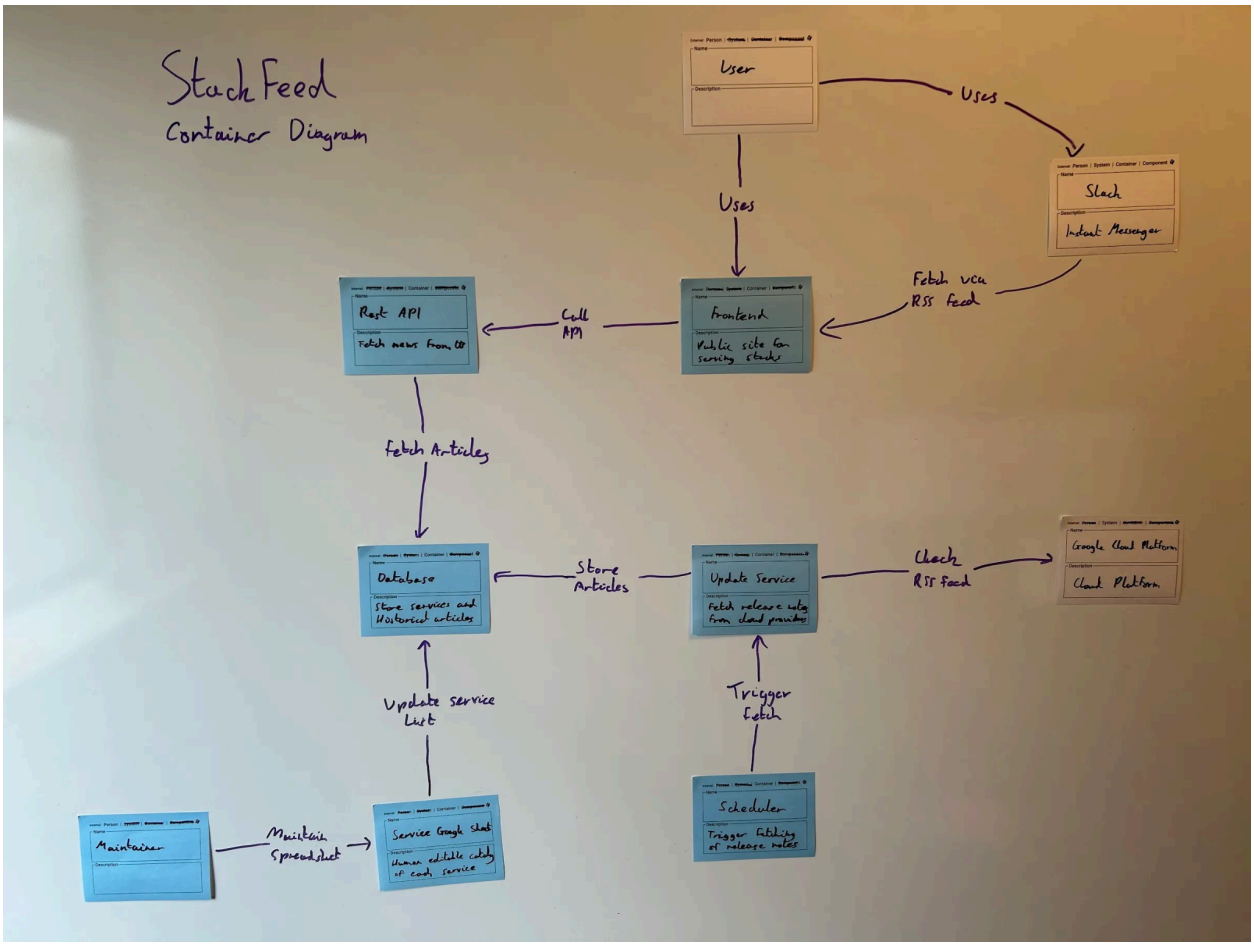
تحليل النتائج ويتم احتساب متوسط القيم أو اعتماد اجماع الخبراء على قيمة معينة أو تكرار العملية حتى الوصول لنتيجة مقبولة ثم يتم توثيق النتائج ويشمل هذا كل الافتراضات والقيود وأسباب التقدير للمراجعة عند الحاجة...

ال Risk-storming

ال Risk-storming هو نهج قام بابتكاره Simon، والمعنى الحرفي هو: "عاصفة / عصف المخاطر"، ويشير هذا الابتكار إلى نهج يمكن اتباعه لتحديد المخاطر في المشاريع البرمجية، من خلال آلية سريعة وممتعة وتشاركية وبصرية في آن واحد [٨]، والفكرة الأساسية لهذا المفهوم قائمة على تسخير الخبرة الجماعية للفريق بأكمله -يشمل ذلك المطورين وال QA والمعماريين ومدراء المشاريع...) للحصول على صورة شاملة للمخاطر الممكنة [٨]، ويتم ذلك من خلال أربعة خطوات رئيسية وهي:

1. رسم المخططات المعمارية -Architecture Diagrams-: في هذه الخطوة يتم البدء برسم مجموعة من المخططات معمارية للمشروع على لوح أبيض أو على أوراق كبيرة، ويجب أن تكون هذه المخططات على مستويات مختلفة من ال Abstraction، لأن كل مستوى من التفاصيل يساعد على تحديد أنواع مختلفة من المخاطر في المشروع، وستمثل هذه المخططات ما يسعى الفريق لبنائه أو تغييره... شاهد الصور التوضيحية:


Risk-storming Step 1 - 2nees.com
Web UI
«question»
What could possibly go wrong?
API Gateway
Payment Service
PostgreSQL DB
External Payment Gateway
(e.g., Stripe, Paypal)


StackFeed
Container Diagram
User
Uses
Uses
Slack
Instant Messenger
Fetch via RSS feed
Rest API
Fetch news from DB
Call API
Frontend
Public site for serving stacks
Fetch Articles
Database
Store services and historical articles
Store Articles
Update Service
Fetch release info from cloud providers
Check RSS feed
Google Cloud Platform
Cloud Platform
Update service list
Trigger fetch
Maintainer
Maintain Spreadsheet
Service Google Sheet
Human editable config of each service
Scheduler
Trigger fetching of release notes

2. تحديد المخاطر بشكل فردي -Identify the risks individually-: بعد رسم المخططات يتم إشعار الأفراد بالبدء بكتابة المخاطر التي يرونها، وهذه الخطوة تعتبر خطوة مهمة جدا لأنها تضمن وجود رؤية غير متحيزة، وإذا أخل فيها صارت النتيجة وكأننا انتقلنا من النهج التشاركي في تحديد المخاطر إلى النهج الفردي، فاحذر من ذلك!
للقيام بهذه الخطوة بشكل صحيح عليك مراعاة ما يلي:

a. السرد الصامت: يطلب من كل فرد أن ينظر إلى المخططات ويقوم بكتابة المخاطر التي يراها على ملاحظات لاصقة -Sticky Notes-، وتكتب مخاطرة واحدة على كل ملاحظة لاصقة، كما يتم التركيز على أن هذه العملية تتم بصمت -أي دون مشاركة للأفكار أو الحديث عنها بصوت عالي- لأن ذلك سيؤثر على بشكل سلبي على النتيجة المتوقعة؛ بسبب تأثير الأفكار الفردية على عقول الآخرين...
b. التقييم الفردي: يقوم كل شخص على تقييم كل مخاطرة يكتبها بناء على بعدين وهما: احتمالية الحدوث والتأثير -تحدثنا عنها سابقا في أول الباب-.
c. ترميز الأولوية: كما قمنا باستخدام الألوان لتحديد الأولويات فسنقوم باستخدام نفس الألوان في ورق الملاحظات، فتكون الورقة ذات اللون الأحمر صاحبة الأولوية العليا...إلخ
d. التوقيت: يتم وضع إطار زمني لإنجاز هذا العمل ويقدر ب ١٠ دقائق أو حتى ٥ دقائق، والسبب الرئيسي لذلك أن الهدف من هذه العملية هو السرد، أي جمع أكبر عدد ممكن من المخاطر الأولية وليس تحليلها والتعمق فيها...

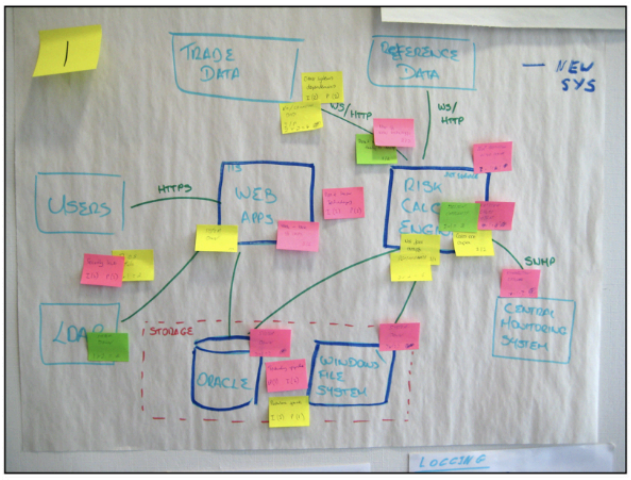
من الأمثلة على المخاطر التي يمكن أن يكتبها الفريق: وجود نقطة فشل وحيدة -Single Point of Failure- أو عدم توفر مساحة تخزينية كافية أو الفريق يفتقر للخبرة في تقنية معينة أو أن هذه التقنية معقدة جدا في التعامل أو التقنيات المستخدمة لا يمكن عمل scale لها أو بعض الأنظمة أو الاعتماديات الخارجية غير موثوقة أو تعمل ببطء... إلخ.

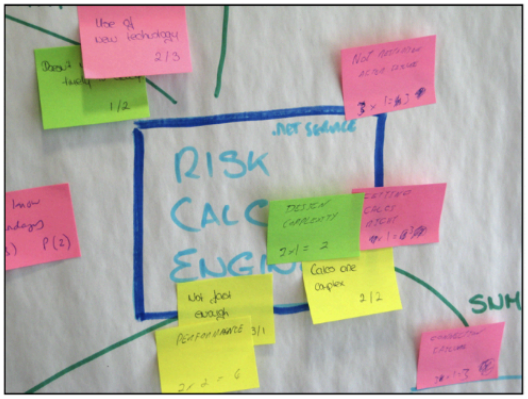
3. تجميع المخاطر على الرسومات المعمارية -Converge the risks on the diagrams-: في هذه الخطوة سنقوم بتحويل المخاطر المحددة بشكل فردي إلى خريطة بصرية مشتركة بين الجميع، وتتم هذه الخطوة من خلال خطوتين:

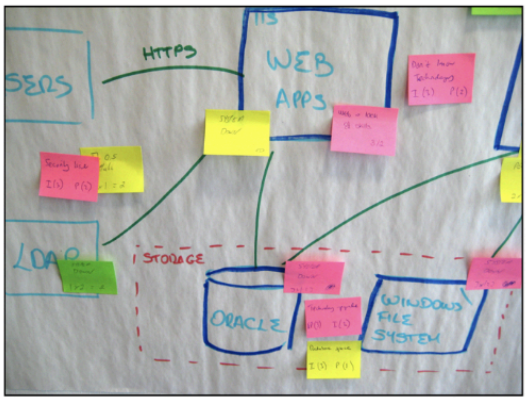
a. الوضع البصري: بحيث يضع كل فرد ملاحظاته اللاصقة على المخطط المعماري الذي قمنا برسمه -وهنا فائدة اللوح أو الورق الكبير-، وتحديدا يتم لصقها على مكان الخطر في الرسم المعماري.

b. اكتشاف التجمعات -Clustering-: بمجرد أن تنتهي الخطوة السابقة سيتمكن جميع الأفراد من مشاهدة المناطق الأكثر خطورة في المعمارية وبشكل سريع، فإذا قام عدة أفراد بتحديد مخاطر متشابهة ستجد العديد من الأوراق في نفس المكان أو متقاربة، وهذه ما يطلق عليها تجمع، وكلما زادت عدد التجمعات كان ذلك إشارة لوجود مخاطر محتملة في أكثر من مكان في التصميم، كما ستعطينا هذه الخطوة لمحة أخرى على أماكن الخطورة العالية والمتوسطة والمنخفضة... وهذه المعاني كلها تحمل لفتة جميلة، وهي وجود أماكن تتلاقى فيها أفكار الفريق ومشاعرهم الداخلية...

شاهد الصورة التوضيحية:







4. تحديد أولويات المخاطر -Prioritise the risks-: وهذه هي الخطوة الأخيرة، وفيها تبدأ مرحلة النقاش الجماعي والتوصل إلى توافق حول تقييم كل خطر محتمل تم وضعه، وآلية العمل هنا يمكن تقسيمها إلى:

a. مرحلة معالجة الملاحظات الفردية: وفيها يتم فهم الأسباب التي دعت صاحب الملاحظة لكتابة وتقييم هذا الخطر (الاحتمالية والأثر) ثم يتم التوافق جماعيا على التقدير النهائي لهذا الخطر أو يتم إزالة هذه الملاحظة من الرسم إذا كان الأثر = "لا شيء - أي الخطر غير موجود فعليا"، ومع ذلك يتم الاحتفاظ بها للتوثيق لاحقا.

b. مرحلة معالجة المجموعات اللاصقة: إذا كانت الاحتمالية والأثر متماثلتين في كل الملاحظات اللاصقة، فالأمر ممتاز ولا داع للقيام بأي إجراء، لكن إن كانت قيم التقدير مختلفة فهنا يجب الاتفاق بشكل جماعي على كيفية تقدير هذا الخطر، وهذا يشبه ما نقوم به بال Planning Poker أو Wideband Delphi... لكن ماذا إن كانت هناك قيم متطرفة؟ في هذه الحالة يجب فهم الأسباب وراء اختلاف تقدير الأشخاص بهذا الخطأ، ودراسة الأسباب التي دعت لهذا التباين ومن ثم الاتفاق على التقييم المناسب كما تحدثنا...

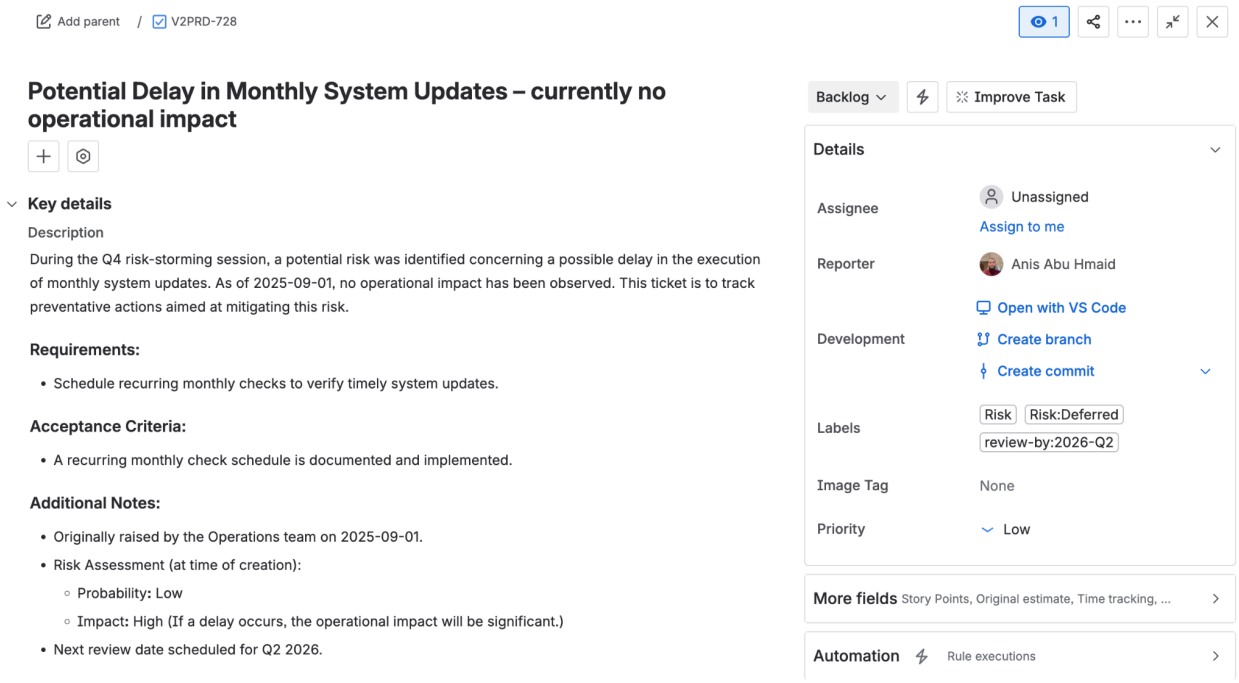
ملاحظة: قد تتسائل لماذا لم نقم باتلاف الملاحظات التي قمنا بإزالتها من اللوح، على الرغم من أن تأثيرها الآن هو = "لا شيء"! والسبب في ذلك يعود لعدة أسباب جميلة منها:

- التوثيق الاستباقي: حتى لو لم يكن الأثر واقعيا الآن أو أثره لا شيء = فإنه قد يتغير مع الأيام، مثال: ملاحظة عن احتمالية وجود تأخير طفيف في تحديث الأنظمة لكنه لا يؤثر على العمليات الحالية... في هذه الحالة يتم تسجيل هذه الملاحظة مع أن لا أثر لها، ويمكن اتخاذ إجراء استباقي عليها مثل مراجعة تحديثات النظام وأدائه بشكل شهري... بهذا تتحول الملاحظات الصغيرة أو غير المؤثرة إلى أصول معرفية يمكن استخدامها... مثال آخر؟ حسنا، ماذا عن انهيار شبكة الإنترنت أو فقدان فعاليتها وانقطاعها على نطاقات واسعة؟ خيار غير واقعي؟ نعم، لكنه يبقى موجودا! فلو كنت عسكريا فهذه نقطة مهمة وخطيرة!، وهذا ما حدث في حرب أوكرانيا وروسيا، فكان البديل الذي وفره الأنجاس للانجاس: الإنترنت من خلال الأقمار الصناعية...

- الشفافية وبناء الثقة: إذا أزيلت الملاحظة من الرسم دون توثيق، قد يشعر كاتب الملاحظة -خصوصا المبتدئين أو غير التقنيين- بأن رأيهم تم تجاهله، أو أن هناك تحيز ما، وسيفقد الأفراد الثقة والأمان على طرح أفكار جديدة وجريئة... وهذه كلها مشاكل خطيرة يجب ألا تصل إلى قلوب وعقول الأفراد... مثلا، يمكن كتابة التوثيق الآتي لملاحظة ما: "تم الاستيعاب: الخطر يفترض سلوك غير مدعوم في الإصدار الأول من المشروع".
- تحليل التغيير: إن أي تغيير قد يطرأ على المشروع لاحقا قد يعيد تفعيل خطر كان مستبعدا، وجود سجل مركزي يسمح بعمل مراجعة سريعة لهذه المخاطر وأثر هذه التغييرات عليها...
- تحسين عملية ال Risk-stroming نفسها: قد تمثل هذه الملاحظات المهملة وسيلة جيدة للحصول على تغذية راجعة، وستجيبنا عن أسئلة مهمة مثل: هل كنا نبالغ في التفاؤل؟ هل استبعدنا مخاطر أمنية حقيقة لأنها لم تمثل الجانب التقني بشكل كاف؟.. إلخ.

إن هذه الملاحظات غير المهمة عادة ما يتم توثيقها على confluence / wiki من خلال إنشاء جدول فيه اسم المشروع والخطر والاحتمالية والأثر والحالة... أو من خلال Jira تكت، بحيث يتم إنشاءها من نوع issue مع label = risk و review-by:2030-Q2 أو ملف CSV...

شاهد المثال في الصورة أدناه:



متى يمكننا استخدام ال Risk-storming؟

بعد أن تعرفنا على ال Risk-storming فهنا سيتبادر لنا سؤال مهم، وهو: متى يمكننا استخدام هذا الأسلوب؟

والجواب يمكن أن يتم تقسيمه لجزئين زماني ونطاقي كما يلي: ^^

1. زماني، وهو الجزء المتعلق ب "متى تستخدم التقنية؟": ويمكننا القول باختصار بأن ذلك ممكن في أي مرحلة أو فترة من فترات المشروع، أكان ذلك في بداية المشروع عند وضع تصميم المعمارية بنسخته الأولية أو خلال فترة المشروع -agile- مثل وقت ال sprint planning لتحديد المخاطر المحتملة والتي سيتم تنفيذها بال sprint القادم أو وقت الـ Retro لمراجعة المشاكل التي حدثت وتحديد المخاطر المتبقية للمستقبل.

2. نطاقي، وهو الجزء المتعلق ب"أين يمكن تطبيق هذه التقنية؟": ويمكن القول باختصار بأنه يمكن استخدام هذه التقنية على أي شيء يمكن تصويره بصريا وليس فقط الشيفرة البرمجية مثل الهياكل المعماري للمؤسسات وال business process وال workflows... إلخ.

التخفيف من المخاطر -Mitigating Risks-:

والآن يبرز لنا سؤال مهم، وهو كيف سنعالج ونخفف من المخاطر التي قمنا بتحديدها وترتيبها في مصفوفة المخاطر؟ كيف سنفعل ذلك في خطوات عملية لتجنب وقوعها أو تقليل أثرها؟ والجواب ببساطة من خلال وضع استراتيجيات لمعالجة المخاطر المحتملة باختلاف أنواعها، ومن الاستراتيجيات الشائعة:

- التعليم: يتم تطبيق هذه الاستراتيجية لمعالجة مخاطر الأشخاص والمهارات. بحيث تشمل هذه الاستراتيجية تدريب الفريق أو/و إعادة هيكلة الفريق أو/و توظيف أعضاء جدد؛ لملء الفجوات المعرفية، خصوصا في التقنيات الجديدة أو التقنيات التي يفتقد فريقنا المهارات اللازمة للتعامل معها...
- النماذج الأولية -Prototypes-: يتم استخدام النماذج الأولية لمعالجة المخاطر التقنية من خلال إنشاء نماذج مصغرة من الأعمال المراد إنجازها (PoC)، وتحديدا في الأجزاء غير المؤكدة من المعمارية؛ لإثبات أن التقنية أو التصميم سيعمل كما هو متوقع أو لبيان فشله مبكرا... بل إن الأمر يتعدى ذلك ويصل لدراسة هل فعلا هذه الفكرة مجدية وتستحق التنفيذ؟ فمثلا لو قلنا أننا نريد بناء نظام للتعرف على الوجوه، فهنا يمكننا

اختبار قيمة هذه الفكرة للمستخدمين، وهذه ما يطلق عليها: Concrete Experiments...

ملاحظة: تذكر بأن تقنية Risk-storming تسمح بتحديد "المسارات" التي تحتاج إلى نماذج أولية بسهولة ^^، كما أن استخدامنا لنتائج ال Risk-storming يساعدنا على تحديد مناطق أكثر قد تحتاج لتجارب أو بناء نماذج أولوية، ويشمل هذا أول مرة أو بعد كل تعديل... راجع الأبواب السابقة ^^.

معلومة: هناك فرق بين ال Poc وال Concrete Experiments، بحيث يختبر ال PoC قابلية الفكرة للتنفيذ، فلو عدنا لمثال التعرف على الوجوه فسنقول أننا سنجرب استخدام خوارزمية التعرف على وجوه ونرى نتائجها، وبهذا نعرف أن هذه الفكرة قابلة للتنفيذ من الناحية التقنية أم لا... في حين يمثل ال Concrete Experiments إذا كانت هذه الفكرة تستحق التنفيذ أساسا، وهل هي مجدية وسيقوم الناس باستخدامها لأنهم بحاجتها أم لا... إلخ، وبغض النظر عن هذه المفاهيم؛ فالمطلوب في استراتيجية النماذج الأولية هو إثبات أن شيء ما سيعمل أو لن يعمل، وذلك لتخفيف المخاطر التقنية المحتملة...

- إعادة العمل المعماري -Architecture Re-work-: يتم هنا إجراء تغييرات مباشرة على المعمارية نفسها لإزالة أو تقليل احتمالية/تأثير الخطر، ومن أشهر الأمثلة على ذلك: إزالة ال Single Point of Failure، أو إضافة ال Cache لتقليل تأثير أعطال الأنظمة الخارجية... إلخ، ثم يتم بعد هذه الخطوة إعادة تنفيذ ال Risk-Storming

للتحقق من فعالية التغييرات التي قمنا بها والتأكد من عدم إدخال مخاطر جديدة عالية الأولوية على تصميمنا ^^...

ملكية المخاطر -Risk Ownership-:

ذكرنا سابقا أن تحمل المخاطر هو جزء أساسي من المهام المناطة لدور المعماري -راجع موضوع ال Technical Risk-، وهنا نعيد التأكيد على أهمية هذا الجزء، فبعض الشركات تجعل ملكية المخاطر من المهام المسندة لمدير المشروع -وهو شخص غير تقني أو ذو معرفة تقنية عامة-، وهذا له أثر سلبي كبير على المشروع؛ خصوصا إذا كانت ملكية المخاطر فردية، وهي ما يطلق عليه: "مشكلة الملكية الفردية"، هذه المشكلة تقول أن مدير المشروع غالبا ما يمتلك سجل المخاطر بالكامل، لكنه قد يفتقر إلى فهم أو اهتمام كاف بالمخاطر التقنية الممكنة -وكثيرا ما رأيناها للأسف-، والحل يكون ببساطة من خلال التخلص من "فردي" والعمل كما ذكرنا إما من خلال الأسلوب التشاركي -الجماعي- أو من خلال تضمين هذه المهام للمعماري نفسه؛ بحيث يضمن المعماري -هو أو من كان يقوم بدور المعماري بشكل مشترك- أن هناك خططا لإدارة المخاطر التقنية الممكنة والاهتمام بها والعمل على حلها...
ممم... أظنك تقول أن هذا الكلام النظري ممل ومزعج وغير مفهوم ووووو...! انتظر لحظة قبل أن تقتلني ^^، سأقوم بطرح مثال يوضح الفكرة ^^... فهيا بنا *-*

تخيل أننا نريد بناء تطبيق للهاتف المحمول ليقوم بخدمة البنك س، هذا البنك قام بتعيين:

- مدير مشروع ذو خلفية مالية ممتازة، لكنه لا يقرأ الشيفرة البرمجية ومعرفته التقنية عامة...
- معماري أول يشارك في التصميم المبدئي للتطبيق
- ثلاثة فرق تطوير وهم: frontend و Core Banking و integration...

وتم تحديد مدة المشروع ب ١٢ شهرا، وسيتم إطلاق نسخة تجريبية بعد ٦ أشهر.

أثناء العمل قام أحد المطورين في فريق ال Core Banking خلال جلسة ال Risk-storming بالإشارة لوجود خطر يجب علينا معالجته وهو احتمالية استدعاء خدمة الرفع مرتين بسبب ضغط المستخدم مرتين على كبسة الدفع بشكل سريع، ودون آلية idempotency فقد تخصم الأموال مرتين...

وركز معي الآن، هنا قام مدير المشروع باعتبار أن هذه الخطورة هي متوسطة ولم يخصص وقت لمعالجتها في ال sprint، والسبب في ذلك أن هذا الخطر لم يحدث بعد وليس له أولوية في خارطة الطريق بالنسبة له... النتيجة: عند الإطلاق التجريبي ومع ٥٠٠ مستخدم من داخل البنك ١٩ حالة خصم مزدوج من الرصيد والحاجة إلى تدخل الدعم الفني وإصلاح هذه الأخطاء، وبناء على ذلك تم تأجيل الإطلاق!

عند مراجعة السجل التاريخي، وجدنا أن الخطر كان معروفا ومسجلا، لكن لم تكن هناك ملكية تقنية فعّالة لمتابعته وتحويله إلى متطلبات فنية مُلزمة! وهنا ظهر تغيير في النهج، وهو جعل المعماري المسؤول عن الملكية التقنية بشكل مباشر، في حين يبقى مدير المشروع يراقب المواعيد والموارد... ثم قام بتحديد المخاطر التي تصنيفها مسبقا ومن ثم تحويلها للمطورين للعمل عليها، وبهذا كانت النتيجة هي تخفيف المخاطر التقنية وعددها، وما تبقى من مخاطر تقنية لم يتم العمل عليها تعتبر مقبولة لا خوف منها حاليا...

هذا المثال يوضح لك كيف أن النظرة من زوايا مختلفة قد تحدد خطورة مشكلة معينة، والخطورة التقنية يجب أن تعطى أولوية وأن يتحملها فعلا المعماري أو القائد التقني لأنه أعلم وأدرى في هذا الباب من غيره، وبناء على ذلك يمكنه اتخاذ القرار التقني الصائب...

فائدة

ولا شيء أكثر حزنا من أن يتوهم الماجرياتي من أنه في قلب عملية التغيير وفقه الواقع وهو مجرد مراقب ومتفرج لا غير..

- كتاب ماجريات الصفحة ٣٢٨

إدخال معمارية التطبيقات في عملية التسليم -Software architecture in the delivery process-

لقد تعلمنا الكثير من المفاهيم المهمة بفضل الله -سبحانه وتعالى-، والآن سنسعى في هذا الباب للإجابة عن سؤال مهم وهو: كمية التصميم المسبق -Up-front Design- التي نحتاجها حقيقة أو التي ينبغي علينا القيام بها قبل الشروع في العمل؟

ولتحقيق هذه الغاية علينا أن ننظر أولا إلى بعض التصورات الخاطئة حول وجود تعارض بين معمارية البرمجيات وال Agile كأسلوب لتطوير المشاريع... وسيكون هدفنا هو إظهار أن معمارية البرمجيات بما فيها ليست عائقا أمام استخدام أساليب مثل ال Agile، بل يمكن دمج هذه التقنيات معا للحصول على نظام صحي يمكن أن يعمل لفترة طويلة... ومن هذه البداية يظهر أول سوء فهم! وهو سوء فهم جوهري!

إن سوء الفهم هذا يتعلق بنظرة المؤسسات إلى المعمارية كأحد العوائق التي تمنعها من تطبيق ال agile، لذلك تقوم هذه المؤسسات بالتخلص أو التحرر من الممارسات المعمارية المهمة مثل التصميم المسبق وال modeling والتوثيق! وهذا نابع من فهم مغلوط آخر وهو الاعتقاد بأن ال Agile = لا تخطيط، أي باشر بكتابة الشيفرة البرمجية مباشرة دون أي تفكير معماري مسبق!

وهذا سيجعلنا نعود للوراء قليلا، وتحديدا إلى أولى صفحات هذا الكتاب، حيث ذكرنا حينها أن المعمارية تتعلق بالبنية -structure- والرؤية -vision- مع التركيز على قرارات التصميم المهمة... وبناء على هذا الكلام لن يخلو أي مشروع من مخاوف ومواضيع وقضايا رئيسية يجب على المعماري أن يعالجها عند تصميمه للأنظمة -مثل الأمان والأداء وقابلية

التوسع...-، وهذا يشمل ما تحدثنا عنه أيضا في المحفزات / العوامل المعمارية بأقسامها المختلفة، الوظيفية وغير الوظيفية -سمات الجودة والمبادئ والقيود-... ويكمن التحدي حقيقة في القدرة على إيجاد التوازن الصحيح بين هذه التقنيات! وحتى نصل إلى التوازن الذي زعمناه، فعلينا أن نجد مواضع التعارض الخاطئ في تصورنا، لذلك لنبدأ بسردها:

التعارض الأول: هيكل الفريق -Team Structure-

هناك صورة سلبية عن المعماري منتشرة في النموذج التقليدي للعمل في المؤسسات المختلفة، هذه الصورة السلبية منبتها وسببها هو وجود موظف يقوم بدور المعماري في الفريق، فلا يشارك في كتابة الشيفرة البرمجية فهو منعزل عن العمل، ويقوم بتسليم وثائق تصميم كبيرة لفريق التطوير ثم يغادر تاركا الفريق غارقا بين الرسومات... هذه الممارسات الخاطئة قامت بإنشاء رد فعل عكسي خاطئ أيضا وهو رفض وجود المعماري أو القيام بدور المعماري في بيئة ال agile بدعاوى مختلفة، وأهم ما نسمعه هو: "نحن لا نحتاج إلى معماري، فكلنا معماريين"! وهذا الاعتقاد الخاطئ يجب معالجته، وكما تحدثنا سابقا، فدور المعماري يختلف باختلاف الفريق وخبرته، وهنا يأتي دور القيادة التقنية لتحديد النوع المناسب طبقا للفريق وطبيعته والمؤسسة وطبيعتها... وقد تحدثنا عن هذه المشاكل سابقا عند حديثنا عن المعماري ودوره والممارسات التي ينبغي عليه تعلمها وتجنب السيء منها...

ملاحظة مهمة: من الدروس المهمة التي تعلمناها من تطبيقات ال Agile وال Lean هي أن فرق التطوير يجب أن تسعى إلى تقليل العبء الناتج عن التواصل عبر المستندات

-Document Hand-offs-، فالهدف من هذه المنهجيات هو زيادة التعاون وتقليل الهدر! وذلك يتم من خلال بناء فرق صغيرة من ال Generalising Specialists -تحدثنا عن هذا المفهوم سابقا- يستطيعون أداء مهام عميقة ولديهم مهارات واسعة تساعدهم على التعاون مع الأعضاء الأخرين... ولأن ال Agile يدعو لتقليل عدد الوثائق وزيادة التعاون ودعم الفرق متعددة المهارات؛ ظهر لنا هذا التصور الخاطئ الذي جعل المطورين يقولون: "لا نحتاج إلى معماري" أو "كلنا معماري"؛ فهم رأو المعماري شخص يجلس بعيدا، يكتب وثائق كثيرة وكبيرة، ولا يكتب شيفرة برمجية، ويبطئ من سرعة العمل، وذو مهمة واحدة! وهو بهذا يتعارض مع ال Agile! لكنهم تناسوا أن القدرة على اتخاذ القرارات المعمارية وخصوصا الأساسية تحتاج إلى خبرة واسعة واطلاع كبير، وقدرة على اتخاذ القرار بناء على المعطيات المختلفة! كما أن اختلاف الفرق وتنوع خبراتها وخبرات أفرادها ومدى خبراتهم قد يجعل من توزيع هذا الدور على الفريق قرارا فاشلا سيقود الجميع للفشل... لهذا، فهم ما على المعماري فعله ووظيفته من قبل المعماري نفسه وفريق المطورين، وكيف سيعمل هذا المعماري ضمن منهجية ال agile = درء لهذا التعارض المزعوم.

معلومة: ال Lean هي منهجية من منهجيات تطوير المنتجات وتحسين الأعمال، وأصل هذه المنهجية من بيئة المصانع ثم انتقلت لاحقا إلى عالم تطوير البرمجيات، وفكرته الأساسية تركز على إزالة الهدر والتركيز على تحقيق أقصى قيمة يحتاجها الزبون باستخدام أقل موارد ممكنة، وإزالة الهدر تشمل: التخلص من الوقت ضائع، والعمل غير المهم، والخطوات غير ضرورية... لذلك يجتمع هذا المفهوم كثيرا مع ال agile بحيث يتم التركيز في ال agile على تطوير المنتج نفسه على شكل إصدارات متعددة بشكل مرن وبأسلوب تكراري، في حين يقوم ال lean على

تحسين العمليات الداخلية من خلال فعل الشيء الصحيح بأقل هدر ممكن، وذلك عن طريق تقليل الوقت الضائع بين الفرق وتحسين عملية سير العمل ونحو ذلك...

التعارض الثاني: ال Process وال Outputs

إن هذا التعارض يمثل الصراع الكبير بين المعمارية وال agile، وتحديدا بين ال process الخاصة بتسليم البرمجيات وبين ال outputs التي نسعى إليها من خلال ال agile... والسبب في ذلك يعود لأن إحدى الأهداف الأساسية لل agile هو تسليم قيمة حقيقية للعميل بشكل متكرر وعلى دفعات صغيرة، وبذلك فالمشروع والفريق يتحرك بسرعة، ويحصل على تغذية راجعة، ولديه قدرة على احتضان التغيير، والتحسين المستمر.

لكن، على الجانب الآخر ارتبطت المعمارية بشكل تقليدي مع وجود Big Design Up Front، وهذا يجعل العمل ذو طابع Waterfall! وفيه تسعى الفرق إلى التأكد من دراسة كل عنصر في تصميم النظام قبل كتابة أي سطر في الشيفرة البرمجية!

إن التعارض الوهمي الذي نراه هنا، ظهر من خلال وجود تصور خاطئ لمبدأ "Agile Manifesto" القائل: "الاستجابة للتغيير أهم من اتباع الخطة"، والذي يأخذه كثير من الناس على المعنى الحرفي، ومنه نتجت مقولات عجيبة مثل: "لا يجب أن نخطط" أو "لا يجب أن نبني تصميما بشكل مسبق"، ولتبرير هذه الكلمات يتم استخدام مصطلحات رنانة مثل: التصميم الناشئ -Emergent Design-، والمعمارية التطورية -Evolutionary Architecture-، واللحظة المسؤولة الأخيرة -Last Responsible Moment، لقد تحدثنا عنها سابقا-! بل يتجاوز الأمر هذه المصطلحات الرنانة ليذهب البعض إلى ال TDD فيقول بأن الاعتماد على

ال TDD يغني تماما عن الحاجة لوجود معمارية! وهذا كله سيتسبب بكل تأكيد بدوامة مستمرة من عمليات ال Refactoring والتي ستنتج نظاما فوضويا وغير مترابط ^^، كما أن النظام في مثل هذه الحالات غالبا ما يفشل في دعم العوامل المعمارية المطلوبة مثل الأمان أو الأداء أو الخصوصية أو القابلية للتوسع ونحو ذلك...
إن المشكلة الكبرى هنا هي الذهاب من أقصى اليمين لأقصى اليسار! وذلك بدلا من التوسط وتحقيق التوازن الحقيقي الفعال! إن ال Agile لا يرفض التصميم؛ بل يرفض ال Big Design Up Front! والـ Agile لا يعني أن "نبني المشروع أولا ثم نتحمل إعادة الهيكلة لاحقا"! بل يعني التفكير المعماري المبكر واتخاذ القرارات الضرورية لتجنب الفوضى، مع الاحتفاظ بمرونة التصميم المتبقي... وهذه الأسطر القليلة هي ما تزيل هذا التعارض المزعوم!

معلومة ١: يقصد بالتصميم الناشئ -Emergent Design- عملية التصميم التي لا تبني كل التفاصيل منذ البداية، بحيث لا يتم رسم التصميم الكامل بشكل مسبق! بل يترك التصميم يتطور تدريجيا مع تقدم المشروع -تطور المتطلبات وكتابة الشيفرة البرمجية-... والفكرة العامة من هذا النهج هو ألا نصمم أكثر مما نحتاج الآن، لكن مع الحفاظ على مرونة عالية في الشيفرة البرمجية بما يكفي ليتم تطوير المشروع دون الحاجة لكسر وتدمير ما قمنا ببنائه، مثلا نبدأ ببناء لعبة ما من خلال بناء الأرض وإضافة الشخصية مع التحريك والقفز، ثم نبدأ بزيادة المزايا الخاصة باللعبة والتحكم ونحو ذلك مع مراحل تقدم المشروع...

معلومة ٢: يقصد بالمعمارية التطورية -Evolutionary Architecture- عملية بناء أنظمة برمجية تكون فيها المعمارية نفسها قابلة للتطور مع تغيير المتطلبات والتقنيات، فبدلا من بناء

معمارية صلبة يصعب تغييرها يمكن أن يصمم النظام بطريقة تمكنه من التوسع أو إعادة تشكيله أو إضافة خدمات جديدة أو استبدال أجزاء من الشيفرة البرمجية دون تعطيل النظام كاملا، وهذا يشبه بشكل كبير لعبة الليجو، بحيث يمكن إضافة أو إزالة القطع أو مجموعة من القطع دون أن ينهار كل شيء! ومن الأمثلة العملية على ذلك الانتقال من تطبيق monolith إلى microservice بشكل تدريجي، بحيث يعاد توجيه بعض المسارات إلى الخدمات الجديدة التي تم بناؤها مع الحفاظ على التوافق -Backward Compatibility-.

وأظنك تتساءل الآن:

ما هو واقع المعمارية في عالم ال Agile؟ وما الذي يحدث على أرض الواقع الآن؟

وهذا حقيقة سؤال مهم، والشركات على ثلاثة أنواع رئيسية في هذا الباب، فمنهم من يرفض التصميم المسبق جملة وتفصيلا، ومنهم من يبالغ في التصميم المسبق حتى يصل إلى تصميم كبير راكضا وراء المثالية المفرطة، وفريق يدرك كم يحتاج حقيقة من تصميم مسبق ويبدأ من عنده، مع علمه بأن التصميم لن يكون مثاليا وليس جامدا... وهذا كله هو ما ذكرناه فعلا في الباب السابق، لكن سنعيد صياغته لتمثل خلاصة ما نريد إيصاله في هذا الجواب: "إن المشكلة الكبرى هي الذهاب من أقصى اليمين لأقصى اليسار! وذلك بدلا من التوسط وتحقيق التوازن الحقيقي الفعال! فبدلا من رفض ال Big Design Up Front بشكل كامل والانتقال إلى عدم القيام بأي تصميم مسبق إطلاقا؛ علينا أن نجد نقطة التوازن بينهما، وهي نقطة يسهل اكتشافها إذا كنت مستعدا لبناء تصميم مسبق ليس كبيرا وأنت موقن بأنك لن تبني تصميما مثاليا".. وهذا يعود بنا إلى أولى صفحات هذا الكتاب، وهو قول Dave

Thomas: «التصميم الكبير في البداية شيء غبي، لكن عدم القيام بأي تصميم في البداية شيء أغبى»...

وهذا سيقودنا لنقطة مهمة، وهي التفكير بالتصميم المسبق -Up Front Design- على أنه طريقة لإنشاء نقطة البداية لمشروعنا، وطريقة لرسم المسار الصحيح للفريق، وليس خطة جامدة صلبة يجب الالتزام بها حرفيا! وهذا سيقودنا للتأكيد على جزئية مهمة، وهي أن المعمارية وال Agile ليسا في صراع! بل هي منهجيات نحتاج لربطها واستخدامها بشكل صحيح لتحقيق أفضل النتائج، وهذا سيجعل الفرق تفكر بشكل صحيح وتقيم وضعها، وعليها أن تفهم وتطلب أو تشير إلى أسلوب القيادة التقنية المناسب لهم، مع تحديد مقدار التصميم المسبق الذي يحتاجه الفريق حقيقة؛ بناء على السياق الخاص بالمشروع...

التصميم التقني -Technical Design- في مقابل التصميم الوظيفي -Functional Design-

لقد ذكرنا سابقا أن كلمة تصميم -Design- هي كلمة واسعة الاستخدام وتحمل معاني كثيرة ومتعددة، لذلك اعلم أننا إذا تحدثنا الآن عن كلمة Design فإننا نقصد ال Technical Design!

إن التصميم التقني يتعلق ب "كيف" سيتم بناء البرمجيات -هيكلية النظام واختيار التقنيات...إلخ-، في حين أن التصميم الوظيفي فهو يتعلق ب "ماذا" ستفعل البرمجيات -الوظائف والميزات التي يقدمها البرنامج-؟

والآن، أريد أن أسألك سؤالا، هل تظن أن أغلب المشاريع ذات متطلبات واضحة أم غير واضحة؟

قد تتعجب من الإجابة، لكن أغلب الشركات الكبيرة ذات متطلبات واضحة ومعلومة منذ البداية، وهذا يشمل المتطلبات الوظيفية وغير الوظيفية -مثل الأمان وقابلية التوسع-، ويكون هذا أكثر وضوحا في الأنظمة الكبيرة خصوصا ال Enterprise منها... وذلك يعود لعدة أسباب منها وجود نظام قديم فعلا ونريد أن نستبدله، أو نريد إضافة مزية أو برنامج يخدم مهمة محددة بوضوح، أو بناء برنامج لدمج أنظمة أخرى... في حين أن الشركات الناشئة أو شركات المنتجات ذات متطلبات غير واضحة، لذلك في هذه الحالة لا يتم إمضاء وقت طويل في التصميم الوظيفي ويتم الاعتماد على تسليم ال features بشكل سريع ومستمر، ونحافظ على عملية جمع التغذية الراجعة بشكل مستمر لتغيير سلوك العمل عند الحاجة...

والنتيجة هي أنه وبعض النظر عن وضوح المتطلبات من عدمها؛ فإن الحاجة لتصميم تقني مسبق أمر لا بد منه للأشياء الكبيرة والمؤثرة وصعبة التغيير لاحقا... مثل اختيار لغة البرمجة وبيئة العمل وتحديد شكل هيكلية النظام -مثل monolith أو microservie-... إلخ، لهذا يبرز هنا أهمية كتابة شيفرة برمجية جيدة ونظيفة، لأن ذلك يساعد الفريق في ال Agile على التحرك فعلا بسرعة في المستقبل، بينما الشيفرة السيئة ستجعل العملية كلها أبطأ مع مرور الوقت، وهو ما يطلق عليه ال Big Ball of Mud.

معلومة: ال Big Ball of Mud تترجم حرفيا إلى كرة كبيرة من الطين، وفي ذلك إشارة إلى وصول النظام لمرحلة أصبح فيها ضخما ومتشابكا ويصعب فصل أجزائه كمثل كرة كبيرة تتجمع

أجزاءها فوق بعضها البعض بشكل غير منظم! وفوق ذلك هذا التضخم يكون فوضويا لأن كل دحرجة للكرة على الطين سيزيد من حجمها ومن اتساخها ويلتصق فيها أشياء كثيرة وبطريقة عشوائية، فالنتيجة النهائية هي كتلة مشوهة وغير مستقرة... وإذا أسقطنا هذا على الشيفرة البرمجية فإن هذا يمثل شيفرة برمجية غير مرتبة ذات أسماء عشوائية واعتماديات متشابكة، وخوف دائم من كسر أي جزء بعد القيام بتعديل ما، وحلول سريعة مكدسة فوق بعضها البعض لحل مشاكل مختلفة... إلخ.

المعمارية ترسم "حدود" المشروع البرمجي

إن هذا العنوان المهم يوضح أن المعمارية ليست بديلاً للممارسات التقنية الحديث، بل هي **الإطار** -**Framework**- الذي يسمح لهذه الممارسات بالنجاح ^^.

والسؤال الآن، كيف ترتبط تقنيات مثل ال TDD وال BDD وال DDD و ال RDD... إلخ بعمارية الأنظمة؟ وهذا السؤال يعني حقيقة: "هل تعد تقنيات ال xDD بديلا عن معمارية الأنظمة خصوصا داخل بيئة عمل اعتمدت على ال Agile كنهج لها؟"

والجواب ببساطة: "لا"، ليست بديلا ولا يمكن اعتبارها كبديل ^^.

إن عملية التفكير في معمارية الأنظمة يجب أن تتمحور حول وضع حدود واضحة ليتم بناء التطبيقات البرمجية باستخدام أي من ممارسات ال xDD وأي مبدأ Agile تفضله لكن ضمن الحدود التي تم رسمها... وكمثال عملي فإن ممارسة أسلوب مثل الـ TDD سيمثل نقطة ممتازة

لبناء شيفرة برمجية نظيفة ذات وظائف تعمل بشكل صحيح، لكنها لن تخبرك بكيفية تقسيم النظام إلى خدمات أو كيفية ضمان أمان البيانات عبر الشبكة... وبهذا نصل إلى هذه النتيجة: "إن معمارية التطبيقات تضع **الحدود -Boundaries-** والاتجاه العام = وداخل هذه الحدود يمكنك استخدام ال TDD أو ال Agile وأي ممارسة أخرى تفضلها بحرية".

والآن قد تتسائل، كيف سأتمكن من رسم هذه الحدود أصلا؟! وهذا سؤال ذكي ومهم، والإجابة تكمن عنه من خلال فهم المدخلات الأساسية الأربعة التي تضع هذه الحدود! وهذه المدخلات هي ما تحدثنا عنه سابقا، وهنا تظهر أيضا أهمية التفاصيل التقنية النظرية التي تكلمنا عنها، ولربط الأفكار فإن هذه المدخلات الأربعة التي ستضع الحدود تتمثل بما يلي:

- المتطلبات الوظيفية: والحد هنا يتمثل بالإجابة على سؤال: "ماذا سنبني؟"، أي أنك يجب أن تعرف ولو بشكل تقريبي ما الذي ستعمل عليه أكان ذلك من خلال User Story أو Use Case أو Requirement...إلخ، وهذا شيء بديهي، فالمتطلبات هي من تقود المعماري، وهذا يذكرنا فيما ذكرناه في نفس الموضوع أول هذا الكتاب: "وبهذا -المتطلبات الوظيفية- فهي تمثل ما يجب على النظام أن يفعله من وجهة نظر المستخدم النهائي والأهداف التي يجب أن يحققها هذا النظام... وقد تتساءل الآن: أليس هذا السلوك بديهي لدينا؟ والحقيقة نعم! إن السلوك الطبيعي لدينا هو فهم ما ينبغي علينا تقديمه للمستخدمين وما هي حاجتهم أو دافعهم لاستخدام النظام الخاص بنا"...

- سمات الجودة: تعد سمات الجودة من الأمور التقنية التي يصعب إضافتها لاحقا! مثل القابلية للتوسع والأداء والأمان ونحو ذلك، لذلك يتم دمجها وأخذها بعين الاعتبار عن وضع التصميم الأولي منذ بداية المشروع، وتجاهلها خطير لأنك بذلك ستقع ضحية إلى واحدة من مشكلتين أساسيتين وهما: ال Over engineered أو ال Under engineered، أي تصميم مبالغ فيه أو تصميم ضعيف هندسيا ^^... وهذا أيضا يعود بنا لأول الكتاب: "تُعد سمات الجودة من المتطلبات التقنية في ذاتها، ولها تأثير كبير على البنية التحتية النهائية للنظام، فعلى سبيل المثال: إذا كان هدفنا هو بناء نظام ذا أداء عال أو أمان قوي، فإن القرارات المعمارية الرئيسية مثل اختيار التكنولوجيا، وهيكل الشيفرة البرمجية يجب أن تأخذ في الاعتبار منذ البداية؛ لأن إهمالها يعد خطأ فادحا بسبب صعوبة إضافتها إلى نظام موجود بشكل مسبق، كما أن كلفة هذا التغيير مرتفعة... باختصار هذه السمات هي جزء أساسي من هيكل النظام الكلي وليست مجرد Feature يتم إضافتها ببساطة" ^^.
- القيود: إن معرفة الحدود التي تنشئ بسبب وجود قيود أمر بالغ الأهمية ولا يمكن تجاهل ذلك! فمثلا حجم الفريق أو المعايير والشروط الخاصة بالتكامل والاندماج مع نظام معين لا يمكن تجاوزها! لأن ذلك يعني أن البيئة والعمل الذي ستقوم فيه سيصطدم مع نفسه ومع من حوله بدلا من الانسجام معهم! وسيضيف احتكاكا غير ضروري مما يعيق العمل أيضا! وهذا يسنجم مع ما ذكرناه سابقا: "القيود ليست مجرد تحديات نواجهها! بل هي محفزات معمارية أساسية، فهي تُجبرك على التفكير في الحلول التي تناسب الإطار الذي سيعيش أو سيبنى فيه المشروع، وسيؤثر هذا بشكل

مباشر على القرارات التقنية"...

- المبادئ: وهي مجموعة القواعد التي يتفق عليها الفريق، وهذه مهمة في رسم حدود العمل، مما يضمن أن الجميع يسير في نفس الاتجاه ويعمل من خلال منظومة واضحة، ومن الأمثلة على هذه المبادئ توجيه الفريق نحو رؤية مشتركة والوضوح والاتساق... مثل هل سنختار ال Monolith أم ال micro service لعملنا؟ وهذا ينسجم أيضا مع ما ذكرناه سابقا: "إن المبادئ تمثل مجموعة القواعد أو التوجيهات التي يختار الفريق تبنّيها من أجل إدخال أساليب قياسية تضمن الاتساق والجودة في الطريقة التي تُطور بها البرمجيات، وهناك العديد من المبادئ الشائعة في هذا العالم، منها ما يتعلق بعملية الـ Development ومنها ما هو متعلق بال Architecture، وكلاهما معا يمثلان العملية الصحية الكاملة لتطوير البرمجيات"

إذا بعد الحديث الجميل نستنتج ما وصلنا إليه سابقا: وهي أن المعمارية تمثل القرارات الصعبة والمكلفة في التغيير لاحقا...

ونستنتج أن الهدف من وجود معمارية تضع تصميما بشكل مسبق هو ليس الوصول للكمال! بل الوصول إلى رؤية تقنية تمنع الفريق من الوقوع ضحية أمام كرة الطين المتدحرجة ^^، وهنا يبرز دور التوازن الذكي، فالفرق الناجحة تميز بين المهم -مثل الهيكل العام والمخاطر العالية- وغير المهم حاليا -مثل التفاصيل الدقيقة للشيفرة البرمجية- وهنا نصل للقاعدة الذهبية:

ال Agile يرفض التصميم المسبق المبالغ فيه -BDUF-! لكنه يحتاج أيضا إلى الحد الكافي من التصميم ليعمل -Just Enough Up Front Design- ...

وهذا سيقودنا إلى سؤال آخر، ما هو مقدار ال Up Front Design الذي يجب علينا القيام به؟

مقدار التصميم المسبق -Up Front Design-: البحث عن التوازن المفقود

إن التساؤل السابق واحد من أهم الأسئلة التي يجب علينا أن نسألها عند التفكير بأي برمجية أو تطبيق نرغب ببناءه.. ويمكن الإجابة على هذا السؤال من خلال تقسيمه إلى عدة أجزاء:

1. طيف التصميم المسبق -The Design Spectrum-:

إن العالم التقني ينقسم إلى طرفين متطرفين، والحقيقة تكمن بينهما -في الوسط-!

- الطرف الأول BDUF: وعادة ما يرتبط هذا بأسلوب ال Waterfall، ويهدف لاتخاذ كل القرارات واعتمادها قبل كتابة الشيفرة البرمجية، ومشكلته الكبيرة في الجمود وعدم القدرة على التكيف مع التغيير وقد تحدثنا عن ذلك سابقا بمزيد من التفصيل.

- الطرف الثاني No Up Front Design: وهو ما يُنسب خطأ لـ Agile! ويعتمد على "الأحلام الوردية الجميلة" بأن المعمارية ستظهر لوحدها -Emergent Design-، ومشكلته الكبيرة هي الدخول في فوضى تقنية عارمة عند اصطدام الفريق بسمات الجودة المعقدة...

2. "قيمة الأعمال -business value-" مقابل "جودة المعمارية":

تعد هذه النقطة واحدة من الفخاخ الشائعة والتي تحدث مشكلة كبيرة خصوصا فيما يتعلق بال Technical Dept، واكتشاف المخاطر في وقت متأخر جدا مما يضطر الفريق للتوقف لإجراء ال Refactoring sprint... وهذه العملية مكلفة ومعطلة لدورة حياة المشروع!

إن هذه المشكلة تحدث عندما يركز الفريق التقني فقط على تلبية متطلبات ال Business Stakeholders فيما يتعلق بالميزات التي يحتاجها أو يراها المستخدم دون الاهتمام بجودة المعمارية أو إعطاء وقت لها أساسا! وبهذا يكون هدف الفريق فقط التحرك بسرعة لتنفيذ مزية ما.. وهذا وبدوره سينعكس على إهمال الهيكل العام بشكل مباشر!

والحل لهذه المشاكل التي في الأعلى ولإيجاد نقطة التوازن يمكننا العمل من خلال ال Risk Driven، ويقصد بذلك اعتماد نهج يضع المخاطر في مقدمة الاهتمام، بحيث يتم التخطيط والعمل على بناء أخطر الأمور أولا وليس فقط بناء على ترتيب المزايا التي يحتاجها المستخدم أو المهام! ومن هنا يمكننا أن نصل إلى نقطة التوازن ويمكن تلخيصها بما يلي:

- **التركيز على المخاطر:** وهنا لن نصمم كل شيء، بل سنصمم الأجزاء التي تشكل **خطرا كبيرا علينا** إذا أخطأنا فيها.
- **التصميم عالي المستوى:** سنجمع المتطلبات الأساسية ونفهم الصورة الكبيرة دون الدخول في تفاصيل الشيفرة البرمجية الصغيرة.

- **التنفيذ التكراري:** بعد وضع الأساس، ننتقل للعمل بأسلوب تكراري -**Iterative**- مما يسمح للمعمارية بأن تتطور بمرونة بمرور الوقت.

إذا جمعنا ما تحدثنا عنه حتى اللحظة، فيمكننا أن نصل إلى المقارنة التالية:

وجه المقارنة	ال **Big Up Front Design**	ال **No Up Front Design**	ال **Balanced / Just Enough Design**
الفلسفة	نقرر كل شيء قبل كتابة الشيفرة البرمجية	نبدأ فورًا ونترك التصميم يتشكل	نحدد الأساسيات ثم نتطور تدريجيًا
ال Architecture	مفصلة بالكامل من البداية	تنشأ تدريجيًا	يتم تحديد الهيكل عالي المستوى فقط
المخاطر	بطء وتعقيد زائد ووثائق كثيرة	فوضى وكرة الطين ^^ والدين التقني الذي سيتراكم عليك	هناك توازن بين السرعة والاستقرار
ال Agile Alignment	ضعيف	مرتفع ظاهريًا	متوافق فعليًا مع ال Agile
التعامل مع ال Risks	تُحل مبكرًا	قد تظهر متأخرًا بشكل مكلف وخطير	تُحدد المشاكل الخطيرة مبكرًا فقط...
سرعة التسليم	بطيئة في البداية	سريعة جدًا في البداية	سريعة لكن محسوبة
مناسب لـ	أنظمة حرجة وبيئات شديدة التنظيم مثل الأنظمة الحكومية *-*	منتجات تجريبية صغيرة جدًا	أغلب المشاريع الواقعية

الخلاصة:

إن الهدف الذي نسعى إليه ليس "الخطة الكاملة أو التصميم الكامل"، بل **الرؤية التقنية التي تحمي الفريق من الكوارث المستقبليّة**، فالمعمارية المتطورة -Evolutionary Architecture لا تعني البدء من "لا شيء"، بل تعني البدء من **أساس صلب** يسمح لها بالنمو والتكيف مع التغيرات المستقبلية دون الحاجة لإعادة كل شيء من الصفر!

حسنا، هل هناك قاعدة بصيغة رياضية يمكننا تطبيقها لأجل هذا الغرض؟

والجواب نعم، إليك هذه القاعدة البسيطة:

التصميم المسبق الذي نحتاجه = القرارات عالية الخطورة + القرارات صعبة التغيير + العوامل التي تؤثر على قراراتنا المعمارية => وبالإنجليزية:

Up Front Design = High Risk Decisions + Hard to Change Decisions + Architectural Drivers

هذه المعادلة العبقرية ببساطتها تلخص جوهر ما ذكرناه، فهي تحول عملية اتخاذ القرار من مجرد تخمين إلى منطق هندسي سليم ^^... فلا يمكن ترك القرارات عالية المخاطر للمصادفة لأنك ستقامر في المشروع، فعلى سبيل المثال لا يمكنك تجاهل أن النظام الذي تعمل عليه سيخدم ١٠٠٠٠ طلب في الثانية الواحدة! ولا يمكن تجاهل اختيار آلية تقسيم المشروع هل ستكون من خلال ال Microservices أم Monolith لأن تعديل مثل هذا القرار يعتبر أمرا صعبا جدا ولا يمكن إتمامه بسبرنت! ولا يمكن تجاهل المتطلبات التقنية الصارمة، فمثلا لا يمكن تجاهل أن النظام الطبي يجب أن يحفظ خصوصية المريض، لذلك يجب تطبيق نظام تشفير على بياناته الحساسة، وهذا ما ينبغي تصميمه منذ البداية لأن هذا العامل سيؤثر على شكل تدفق البيانات في كامل المشروع... وما بعد ذلك يمكن تركه للتطور التدريجي ^^.

ملاحظات عامة ومفاهيم قبل أن نتابع:

- لقد ذكرنا في حديثنا عدد من المفاهيم مثل ال TDD وال BDD وال DDD و ال RDD... هذه المفاهيم تمثل مختصرات تشير إلى منهجيات متعددة مستخدمة في تطوير البرمجيات، وتشترك جميعها بوجود دافع أو عامل معين لتوجيه عملية التصميم والعمل، وهذا هو ما يطلق عليه "Driven"...
  - ال TDD: هي عملية التطوير التي تتم بناء على كتابة الاختبار أولا -Test-، ودورة حياة هذا الأسلوب تبدأ بكتابة الاختبار للمهمة المطلوبة، وبكل تأكيد سيفشل الاختبار لعدم وجود الشيفرة البرمجية التي تخدم هذا الاختبار وهي ما يطلق عليها المرحلة الحمراء -red- ثم ننتقل للمرحلة الخضراء -green- والتي نكتب فيها أقل قدر ممكن من الشيفرة البرمجية حتى نجعل الاختبار ينجح، ومن ثم نصل للمرحلة الأخيرة وهي ال refactor والتي تمثل مرحلة التحسين والتنظيف والتأكد من أن الاختبار ما زال ناجحا... وأهم ما يرمي له هذا الأسلوب هو ضمان جودة الشيفرة البرمجية والتأكد من أن الشيفرة التي تمت كتابتها خالية من الأخطاء مع الحصول على شيفرة برمجية وتطبيق قابل للاختبار.
  - ال BDD: وهي عملية التطوير التي تنطلق من الجانب السلوكي، أي التطوير المدفوع بسلوك النظام من وجهة نظر المستخدم بدلا من التفاصيل التقنية، وتستخدم لتمثيل ذلك لغتنا الطبيعية لإيصال ما نريد بحيث يفهم المدير والمطور والفاحص السلوك المراد للنظام... مثل: "بفرض أن/عندما/إذا" = **بفرض** أن

رصيد الحساب هو صفر، فإننا عندما نحاول سحب النقود علينا إذا منعه من خلال طباعة تحذير تعلم المستخدم بفشل العملية لعدم وجود رصيد.

- ال DDD: وهي عملية التطوير التي تنطلق لتقسيم الأنظمة الكبيرة والمشاريع المعقدة إلى نطاقات متعددة؛ بحيث يخدم كل نطاق أو يمثل كل نطاق مجال عمل معين، وهذا يلزم وجود بعض المفاهيم المهمة مثل وجود لغة تفاهم أو تواصل مشتركة بين الجميع ضمن نطاق معين وهو ما يطلق عليه بال Ubiquitous Language، وهذا كله يرتبط ارتباطا وثيقا بال Bounded Context، وهي الحدود التي تجعل أجزاء النظام = مستقلة وذات معاني وقواعد خاصة بها، ودون الخوف من الاختلاط مع أجزاء النظام الأخرى! وأوضح مثال على هذا تخيل أن كلمة Order في نظام واحد قد تشير إلى معنيين مختلفين، مثل: قسم المبيعات = طلب شراء من عميل ما، وفي قسم المستودعات = أمر تجهيز أو شحن لبعض المواد! لو وضع قسم المبيعات وقسم المستودعات في نفس النطاق فسيحدث هناك التباس وتعقيد... وهنا تظهر أهمية الحدود التي ترسم اللغة المشتركة داخل كل نطاق!
- ال RDD وهو أسلوب يعتمد على تحديد مسؤوليات كل Object في النظام قبل التفكير في التفاصيل التقنية، وهو ما يستخدم بكثرة من خلال بال Object Oriented Programming ^^، وفكرة العمل هنا تبدأ من جواب السؤال التالي: "ما هي المسئوليات التي يجب أن يقوم بها كل object عنا في النظام؟" وبجواب هذا السؤال سنحصل على: "المسؤوليات الخاصة بكل object وعلى ال Collaborators والتي تشير إلى معرفتنا مع من سيتعاون كل object

لتنفيذ المسؤوليات التي وقعت عليه..."، والمسؤوليات نوعان إما Doing مثل حساب قيمة ما وإما Knowing مثل الاحتفاظ بحالة ال object...

- إن المعادلة التي تطرقنا إليها تمنع "شلل التحليل -Analysis Paralysis-"، لأنها تحميك من إضاعة وقتك في تصميم ألوان الواجهات وأسماء المتغيرات في المرحلة الحالية! بل تجبرك أو تقودك للتركيز على الأجزاء الثلاثة الكبار حسب المعادلة...
- ورد في جدول المقارنة مفهوم "الدين التقني" أو ال "Technical Debt"، هذا المفهوم مهم جدا في مجال تطوير البرمجيات، فهو يشير إلى وجود مشكلة سلبية، وسيترتب بوجودها دينا عليك سيتراكم إن لم تقم بإصلاحها في أسرع وقت، وهذا الأمر نتاج اتخاذ حلول سريعة أو غير مثالية أثناء كتابة الشيفرة البرمجية لتوفير الوقت أو الجهد في الوقت الحاضر... باختصار هو: تكلفة مستقبلية ناتجة عن حلول برمجية سريعة أو غير مثالية اتُّخذت في الماضي [٨٨]، ومن الأمثلة على ذلك عدم كتابة اختبارات أو تكرار كود معين في عدة أماكن بدل إعادة استخدامه... لكن يمكن أن يكون الدين التقني مفيدا في بعض الحالات مثل الحاجة لإطلاق منتج ما بشكل سريع مع خطة لسداد هذا الدين، وهذا أحد الأنواع الأربعة من أنواع الدين التقني [٨٨]، وأنواع الدين التقني يمكن تلخيصها بما يلي:
  - الدين المتعمد الحذر: وهذا أفضل نوع من أنواع الدين التقني كما ذكرنا في مثالنا أعلاه، فهنا الفريق يعرف أن الحل ليس مثاليا، لكن تم اتخاذ هذا القرار بشكل واع لخدمة هدف ما مع خطة لإصلاحه لاحقا.

- الدين المتعمد المتهور: وهنا يعرف الفريق أن هذه الشيفرة البرمجية سيئة لكن الفريق لم يهتم بذلك ولم يضع خطة لتحسينه وإنما اهتم فقط في سرعة إنجاز العمل، وغالبا ما يكون التعليق: "كود عجقة بتصلح بعدين" ^^
- الدين غير المتعمد الحذر: وهنا يكتب الفريق شيفرة برمجية جيدة حسب معرفته لكنه يكتشف لاحقا أن هناك طريقة أفضل لتصميم وكتابة هذه الشيفرة البرمجية، وهذا يظهر غالبا بعد اكتساب خبرة وفهم أعمق للمشكلة من خلال ما يعمل عليها المطورون خلال فترة حياتهم البرمجية وفي المشروع...
- الدين غير المتعمد المتهور: وهنا كتب المطور شيفرة برمجية سيئة مع سوء فهم للمشكلة ولم يكن يعرف أصلا أن هذا الحل سيء.

إن خلاصة موضوع الدين التقني هو أن الدين التقني لا يمكن منعه بشكل كامل؛ لكن الفرق الجيدة والمطورين الجيدين هم من يديرون هذه الديون التقنية بطريقة صحية وبشكل مستمر...

والآن، نعود للسؤال مجددا: ما هو مقدار التصميم المسبق -Up Front Design- الذي نحتاجه؟ بصيغة أخرى: ما هو القدر الكافي من التصميم اللازم لنتمكن من العمل بشكل صحيح؟ القدر الكافي من التصميم فقط!

الجواب ببساطة على هذا السؤال في هذا السطر: "إن القدر الكافي للتصميم هو مقدار ما تصممه بحيث يجعلك تعرف ما هو هدفك وكيف ستقوم بتحقيقه"!، وهذا هو جوهر ما ذكرناه في هذا الكتاب: "المعمارية تمثل القرارات المهمة، حيث تُقاس الأهمية بتكلفة التغيير" =

وهذا يعني: "مجموعة **القرارات التي يكون تعديلها مكلفًا**، وهي الأشياء التي **يجب أن يتم اتخاذها بشكل صحيح في أقرب وقت ممكن**"... ومن هنا لنشبك الماضي مع الحاضر ^^:

على سبيل المثال، لو قلنا أن النظام الخاص بنا يحتاج إلى خصائص مثل الأداء العالي والأمان العالي والقابلية للتوسع، وكذلك اختيار التقنيات البرمجية الأساسية والنمط المعماري المعتمد...إلخ = فهذه عادة أمور لا يمكن تجاوزها لأنها صعبة الإضافة لاحقا، وهي قرارات معمارية = وهي بذلك تمثل نقاط أساسية عند التصميم.

وهذا يقودنا لقاعدة مهمة: "إن تحديد العناصر المعمارية المهمة والمخاطر المرتبطة بها هو أمر يجب تطبيقه على جميع المشاريع البرمجية، وبغض النظر عن منهجية التسليم أو عملية ال process المتبعة"...

وإن سألت الآن مجددا، ما هو إذا المقدار الكافي من التصميم لمشروع مالي مثل بناء نظام تدقيق محاسبي، فالجواب سيكون: "هذا يعتمد على -It depends-" ^^، لأن كل فريق مختلف عن الآخر

- فبعض الفرق أكثر من خبرة من البعض الآخر، والخبرة الخاصة والعامة مؤثرة هنا أيضا... (خاصة في تطوير أنظمة مالية أو خبرة عامة في تطوير أنظمة متنوعة).
- وبعض الفرق تحتاج إلى توجيه كبير
- وبعضها تتغير بشكل مستمر أو متكرر
- وبعضها تعمل معا بشكل مستمر وإيجابي والفرق درجات في هذا...

ومن هنا يظهر لنا مفهوم ال: "Firm Foundations"، وترجمته الحرفية: "الأسس المتينة"! هذا المفهوم يروج له Simon لحل الصراع التقليدي بين BDUF والتطوير العشوائي دون تصميم أو دون تصميم كافي! وفلسفة هذا المفهوم: أن "الأساس" يجب أن يكون موجودا لإنه

يمثل الهيكل الذي ستبنى عليه المكونات -Component-، وعدم وجود الأساس = تكلفة تغيير هائلة ومخاطر هدم عالية! و"المتينة": أنها يجب أن تكون قوية بما يكفي لتحمل ضغط التطوير، لكنها في ذات الوقت مرنة بما يكفي لتسمح بالتفاصيل الصغيرة أن تتطور أثناء البرمجة... ولهذا الأسلوب ثلاثة ركائز وهي:

- الهيكل (Structure) - استخدام الـ C4 Model: وهنا يقصد بها فهم العناصر الهيكلية المهمة وكيفية ترابطها بناء على العوامل المعمارية، فبدلا من رسم مخططات عشوائية، يقترح Simon مستويات واضحة من ال Abstraction:

1. **ال System Context:** كيف يتفاعل نظامك مع العالم الخارجي (تحديد من هم المستخدمين والأنظمة الخارجية).
2. **ال Containers:** وهذه لا يقصد بها ال Docker Container :D بل يقصد بها أي وحدة موجودة يشترط وجودها قيد العمل "Running" ولديها مساحتها وعملياتها الخاصة... مثل: React CLient Side و PHP Server و Side و Mysql as a Data Store... إلخ
3. **ال Components:** المكونات البرمجية داخل كل **Container** مثل ال AuthService وال PaymentController... إلخ.

ملاحظة: ال C4 Model سيتم الحديث عنه لاحقا -بإذن الله-، وهو النموذج الذي قدمه Simon والذي تحدث عنه في الجزء الثاني من كتاب: Software Architecture for Developers، كما يمكنك الإطلاع عليه من خلال: c4model.com.

- الرؤية (Vision): توحيد الفهم: إن الهدف من وجود رؤية ليس "التحكم"، بل "المحاذاة" (Alignment)، فعندما يمتلك الفريق رؤية معمارية واضحة وخريطة ذهنية مشتركة:
  - سيعرف المطورون أين يضعون الشيفرة البرمجية الجديدة.
  - وتتوحد لغة الحوار بين أعضاء الفريق (مثلا: ما هو تعريفنا لـ Service؟).
  - ويقل الوقت الضائع في النقاشات الجانبية حول القرارات التقنية البديهية.

فمثلا لا يمكن تقبل أن يعمل جزء من الفريق باستخدام نمط معماري مثل ال Layered Architecture ويعمل مطور آخر على ميزة أخرى Hexagonal Architecture لينتهي بنا الأمر ب "فرانكشتاين برمجي - Software Frankenstein" يصعب صيانته، كما أن الرؤية لا قيمة لها إن بقيت في رأس المعماري، بل يجب أن تكون متاحة ومفهومة وواضحة لكل مطور... وهنا أيضا يأتي دور نموذج C4...

ملاحظة: ال Software Frankenstein هي استعارة مستوحاة من رواية لفرانكشتاين، حيث يقوم الطبيب فيها بتجميع أجزاء من جثث مختلفة ليصنع كائنا آخر، والنتيجة كانت مسخا مشوها -الحمد لله على نعمة الإسلام-، ومن هنا استخدم هذا المفهوم في الأنظمة البرمجية للدلالة على وجود نظام مشوه وقبيح تقنيا، بني وجمع من أجزاء لا تنتمي لبعضها البعض، فيصعب التعامل مع هذا النظام ويستحيل صيانته بسهولة! تخيل أن يستخدم شخص في نفس التطبيق MobX و Redux وReact Context... إنها غابة من التقنيات ^^.

- المخاطر -Risks-: الهجوم الاستباقي: بدلا من انتظار حدوث كارثة في الأداء أو الأمان عند الإطلاق، تدعوك هذه المنهجية للقيام بـ:
  - **ال Risk-Storming:** تحديد النقاط الضعيفة في التصميم مبكرا.
  - **ال Concrete Experiments (Spikes):** إذا كنت تشك في قدرة تكنولوجيا معينة على التعامل مع حجم بيانات ضخم، لا تنتظر؛ قم ببناء نموذج مصغر (Prototype) الآن للتأكد.

بناء على كل ما ذكرناه، ستكون خلاصة هذا الباب:

سنتوقف عن ال Up Front Design إذا حققنا الشروط التالية:

- فهمنا العوامل المعمارية الأساسية التي نحتاجها -Key Architectural Drivers-.
- فهمنا السياق -Context- والنطاق -Scope- لما تقوم ببنائه.
- فهمنا القرارات التصميمية المهمة -Significant Design Decisions- مثل Monolith ولا Microservice.
- لدينا طريقة واضحة لإيصال الرؤية -Communicate Vision-.
- لدينا ثقة أن التصميم يحقق المتطلبات المعمارية الأساسية.
- قمنا بتحديد المخاطر وأصبحنا مرتاحين للتعامل معها.

إن مفتاح النجاح الحقيقي في هذه العملية الوصول إلى مرحلة نستطيع فيها من وضع الحد الفاصل بين التصميم الضروري والتصميم التطوري.

فائدة

...ومن المفاهيم التي أنتجها مالك أيضا حول هذا السؤال مفهوم المعامل الاستعماري وكان بن نبي يرى أن الاستعمار لا ينتهي بالرحيل الحسي للمستعمر وإعلان الاستقلال، بل يبقى منه رواسب في العقول والثقافة يجب تطهيرها وكان يسمي هذا (تصفية الاستعمار).

- كتاب ماجريات الصفحة ١٣٧

خلاصة الجزء الأول من كتاب: من مبرمج إلى معماري

أولا: مفاهيم المعمارية الأساسية

- المعمارية بمفهومها الكامل تجمع بين بُعدين: البنية (Noun) وهي وصف هيكل النظام ومكوناته، والفعل (Verb) وهي عملية مستمرة تشمل اتخاذ القرارات وتوجيه الفريق طوال حياة المشروع.
- ال Software Architecture ليست عبارة عن رسم مخططات معقدة وقضاء شهور في التخطيط، بل هي تحديد المسار الصحيح وضمان أن الجميع يسير فيه، مع الحفاظ على المرونة وعدم إضاعة الوقت في تفاصيل لا تخدم نجاح المشروع.. باختصار: ال Software Architecture = قرارات مهمة مبكرة
- الفرق بين Architecture و Design: المعمارية هي مجموعة فرعية من التصميم، تتضمن القرارات التي يصعب تغييرها لاحقاً ولها تكلفة عالية عند التراجع عنها، لذلك كل معمارية تصميم، وليس كل تصميم معمارية.
- المعماريات الرئيسية هي:
  - ال Application Architecture: تصميم الشيفرة البرمجية داخل التطبيق الواحد.
  - ال System Architecture: ربط التطبيقات المختلفة ببعضها وبالبنية التحتية.
  - ال Software Architecture: مزيج بين النوعين السابقين مع رؤية شمولية للنظام.

- ال Enterprise Architecture: التخطيط الاستراتيجي على مستوى المؤسسة ككل، مع الابتعاد عن التفاصيل التقنية.

- كل مشروع يحتاج معمارية، والفرق يكمن في مقدار التفكير / التصميم المسبق المطلوب بحسب حجم المشروع وتعقيده وحجم الفريق وخبرته... دون إفراط في التصميم ولا تفريط فيه.

ثانيا: العوامل المعمارية (Architectural Drivers)

هي القوى التي تحدد شكل المعمارية النهائية، وتشمل أربعة محاور:

١. المتطلبات الوظيفية (Functional Requirements)

- تمثل ما يجب على النظام فعله من وجهة نظر المستخدم.
- لا يمكن البدء في التصميم دون فهم واضح لهذه المتطلبات ولو بشكل عالٍ المستوى.
- ال Agile لا يعني البدء دون فهم، بل يعني البدء مع وجود فهم كاف للمزايا الأساسية.

٢. سمات الجودة (Quality Attributes)

وهي خصائص لا يمكن تجاهلها ولا إضافتها لاحقا بسهولة، أبرزها:

- الأداء (Performance): سرعة استجابة النظام (Response Time) وزمن نقل البيانات (Latency).

- قابلية التوسع (Scalability): قدرة النظام على استيعاب عدد أكبر من المستخدمين والطلبات المتزامنة.
- التوفر (Availability): استمرار النظام في العمل دون انقطاع، وتقاس بالنسب المئوية "التسعات" = الفرق بين 99% و 99.99% هو الفرق بين التوقف 3.65 يوم والتوقف 52 دقيقة سنويا!
- الأمان (Security): يشمل التحقق من الهوية (Authentication) والصلاحيات (Authorisation) وحماية البيانات (Confidentiality).
- الخصوصية (Privacy): حماية البيانات الشخصية للمستخدمين والامتثال للتشريعات كـ GDPR.
- التعافي من الكوارث (Disaster Recovery): خطة استعادة البيانات والخدمات بعد أي انقطاع، وتشمل النسخ الاحتياطية والتكرار والتبديل التلقائي.
- إمكانية الوصول (Accessibility): دعم ذوي الاحتياجات الخاصة وفق معايير W3C.
- المراقبة (Monitoring): جمع بيانات أداء النظام وصحته في الوقت الفعلي وإرسال التنبيهات عند وجود مشكلة.
- الإدارة (Management): القدرة على التدخل وتعديل سلوك النظام أثناء التشغيل دون الحاجة لإعادة نشر (deploy).
- التدقيق (Audit): تسجيل تفصيلي لكل تغيير يحدث في النظام (من؟ متى؟ ماذا؟ لماذا؟).

- المرونة (Flexibility): قدرة النظام على التكيف مع تغييرات متطلبات العمل دون إعادة تصميم.
- قابلية التوسيع (Extensibility): إضافة وظائف جديدة دون تعديل الشيفرة الأساسية، كما في الـ APIs والـ Plugins.
- قابلية الصيانة (Maintainability): سهولة تعديل وتحديث الشيفرة مستقبلاً، وتتحقق من خلال فصل المكونات (Separation of Concerns).
- الامتثال القانوني (Legal & Regulatory): الالتزام باللوائح والتشريعات المحلية والدولية المتعلقة بالصناعة.
- دعم اللغات والثقافات (i18n): قدرة النظام على العمل بلغات ومناطق جغرافية متعددة، بما يشمل اتجاه النص والعملات والتقاويم.

القاعدة ذهبية: ليست كل السمات بنفس الأهمية في كل مشروع، والقرارات المعمارية يجب أن توجه بأهداف العمل المحددة لا بقائمة نظرية شاملة، كما أن المتطلبات الغامضة (نريد نظاما سريعا) يجب تحويلها إلى متطلبات قابلة للقياس.

٣. القيود (Constraints)

- قيود الوقت والميزانية: قد تجبرك على اختيار أسرع الحلول وأبسطها.
- قيود التقنية: التقنيات المعتمدة في الشركة، الأنظمة القديمة الموجودة، منصة النشر المستهدفة، نضج التقنية، قيود الشيفرة البرمجية مفتوحة المصدر، علاقات البائعين، الفشل السابق، الملكية الفكرية الداخلية.
- قيود الأشخاص: مهارات الفريق وحجمه وإمكانية التوظيف وفريق الصيانة المستقبلي.

- قيود تنظيمية: السياسات الداخلية للشركة، وطبيعة الحل (تكتيكي قصير الأجل أم استراتيجي طويل الأجل).

القيود ليست بالضرورة سيئة، فبعضها يحد من الفوضى ويوجه القرارات نحو الأفضل. ويمكن التنازل عن قيد أقل أهمية لصالح قيد أكثر أهمية.

٤. المبادئ (Principles)

وهي ما يختار الفريق اعتماده بإرادته (خلافا للقيود المفروضة عليه):

- مبادئ التطوير: مثل معايير كتابة الشيفرة البرمجية (Coding Standards)، والاختبارات التلقائية (Automated Testing)، وأدوات التحليل (Static Analysis Tools)...
- مبادئ المعمارية: مثل ال Layered Architecture، وال High Cohesion/Low Coupling ومبادئ ال SOLID، وال Stateless Components، ونماذج البيانات (Rich vs Anemic Domain Model)، وال Always vs Eventually Consistent...

تحذير مهم: احذر من نفخ "أفضل الممارسات"! كل مبدأ مناسب لسياقه، وما يصلح لمشروع قد يكون عبئا على مشروع آخر، فالسياق هو الأساس دائما، والتصميم المبالغ فيه لمشروع بسيط هدر صريح للوقت والجهد.

ثالثا: دور المعماري

- المعماري دور (Role) لا رتبة (Rank)، ولا يمكن الوصول إليه بمجرد سنوات الخبرة، بل بامتلاك المهارات المناسبة، إن الترقية بالأقدمية إلى هذا الدور هو ما يُعرف بـ"مبدأ بيتر".
- أهم مهام المعماري الخمس:
  - فهم العوامل المعمارية وإدارتها: جمع المتطلبات، وتحويل الغامض إلى محدد قابل للقياس، وتحدي المتطلبات غير الضرورية.
  - التصميم: تحويل المشكلات والقيود إلى حل معماري منظم قابل للتنفيذ.
  - إدارة المخاطر التقنية: التحقق من أن التقنيات المختارة تعمل فعلا في سياق المشروع! لا الاكتفاء بالوعود التسويقية!
  - القيادة التقنية: توجيه الفريق وضمان الاتساق طوال حياة المشروع.
  - ضمان الجودة: التأكد من أن التنفيذ يتطابق مع الرؤية المعمارية.
- المعماري الجيد لا يكتفي برسم المخططات ثم يغيب! إن نهج "Architecture as a Service" هو نهج فاشل يؤدي إلى فقدان السياق وتجاهل التصميم.
- معيار نجاح المعمارية تشمل تلبية العوامل المعمارية، وتوفير أساس متين لكل المطورين، وتحقيق القيمة التجارية للمشروع.

رابعاً: القيادة التقنية

- هدفها الأساسي التحكم في الفوضى، وذلك من خلال إطار عمل منظم يوجه الفريق دون الانزلاق إلى الإدارة التفصيلية المفرطة (Micromanagement).
- أدوات الاتساق الخمسة:
  - التحكم: اتخاذ قرارات معمارية واضحة (مثل: استخدام ال ORM من عدمه).
  - الانضباط: مقاومة إغراء تجربة كل تقنية جديدة.
  - الحدود: تحديد "من يملك ماذا" و"من يتحدث مع من" بين مكونات النظام.
  - الإرشادات: توصيات عملية قابلة للتعديل حسب السياق.
  - المبادئ: قواعد جوهرية تعبر عن فلسفة الفريق التصميمية وثابتة نسبيا.
- القيادة التقنية التعاونية ممكنة لكنها تحتاج وقتا ونضجا، والفرق تمر بثلاث مستويات فيها وهي: الفوضى ← التعلم ← التنظيم الذاتي، ولكل مستوى أسلوب قيادة يتناسب معها.
- فرق ال Agile تحتاج إلى معمارية، والعدو الحقيقي لـ Agile هو التصميم المبالغ فيه -BDUF- لا المعمارية بحد ذاتها، والمعماري في بيئة ال Agile يُبرز رؤية تتطور تدريجيا لا مخططا جامدا.

خامساً: أبرز القواعد الجامعة

- لا يوجد حل مثالي، بل يوجد أفضل المقايضات الممكنة في السياق المعطى.

- فهم العوامل المعمارية مبكرا يجنبك إعادة عمل مكلف لاحقا، والتصميم الناقص والتصميم المبالغ فيه كلاهما أمر سيء!
- كل قرار له ثمن؛ كن واعيا بتكلفة كل اختيار قبل اتخاذه، وأوضح للعميل هذه المقايضات.
- لا تحاكم تصاميم الماضي بأدوات الحاضر؛ فالسياق والظروف والتقنيات تتغير.. وهذب نفسك واحترم جهد الآخرين!
- الوسطية شعار المعماري الناجح: لا إفراط في التصميم المسبق ولا تفريط في التفكير المعماري.
- وثّق دور المعماري في مؤسستك لتجنب الغموض والتضارب في المسؤوليات.
- أي ممارسة برمجية لا تقدم قيمة حقيقية ملموسة للفريق تعتبر هدرا للوقت والجهد، فالهدف ليس "التعقيد" بل "الفائدة"!
- أنت كقائد تقني أو معماري من يحدد كمية التصميم المطلوبة لتكون "كافية فقط" دون مبالغة قد تعيق السرعة.
- التصميم المسبق يهدف إلى "رفع احتمالات النجاح" وتقليل المخاطر.
- الهدف المرجو من كل هذا هو = زيادة فرص النجاح وتقليل نسب الفشل^^.

والحمد لله رب العالمين

الخاتمة

وصلنا الآن إلى نهاية الجزء الأول من هذه الرحلة، والتي بدأت بسؤال بسيط: ما هي معمارية البرمجيات؟ ثم ما لبثنا حتى تعمقنا شيئا فشيئا حتى وجدنا أنفسنا أمام عالم واسع من القرارات والمقايضات والمسؤوليات.

لو أردنا أن نُلخّص ما تعلمناه في كلمات قليلة، لقلنا: أن **المعمارية ليست رسما على ورقة، بل هي طريقة تفكير،** فهي القدرة على النظر إلى النظام من أعلى دون أن نفقد التواصل مع ما يحدث في الأسفل، والتوازن بين ما يجب أن يُبنى وما يمكن أن يُبنى في ضوء القيود والسياق المحيطَين بك.

إن الذي يجب أن يبقى معك بعد إغلاق صفحات هذا الكتاب، ليس مجموعة أسماء ومصطلحات! بل هو هذا السؤال الذي يجب أن تحمله معك في كل مشروع، وأن تسأله لنفسك في كل مرة: "**ما الذي يهمّ هنا، ولماذا؟**" فإن تمكنت من الإجابة على هذا السؤال فقد أصبت كبد الحقيقة، ونلت المراد بإذن الله ^^.

أخيرًا،

اللهم اغفر لي ولوالدي، ربي ارحمهما كما ربياني صغيرا.

اللهم اغفر لي وللمؤمنين، واغفر اللهم ربنا لإخواننا الذين سبقونا بالإيمان.

اللهم فرج عن إخواننا في فلسطين والشيشان وأفغانستان والسودان والصين والهند والبوسنة والسويد وفرنسا وفي كل مكان تنتهك به دماء المسلمين وأعراضهم، وفرج اللهم عن إخواننا المعتقلين في مشارق الأرض ومغربها، ولا حول ولا قوة إلا بالله...

مع خالص التمنيات بالتوفيق والنجاح في كل ما تقومون به.
أخوكم: أنيس حكمت أبوحميد

وآخر دعوانا أن الحمد لله ربّ العالمين