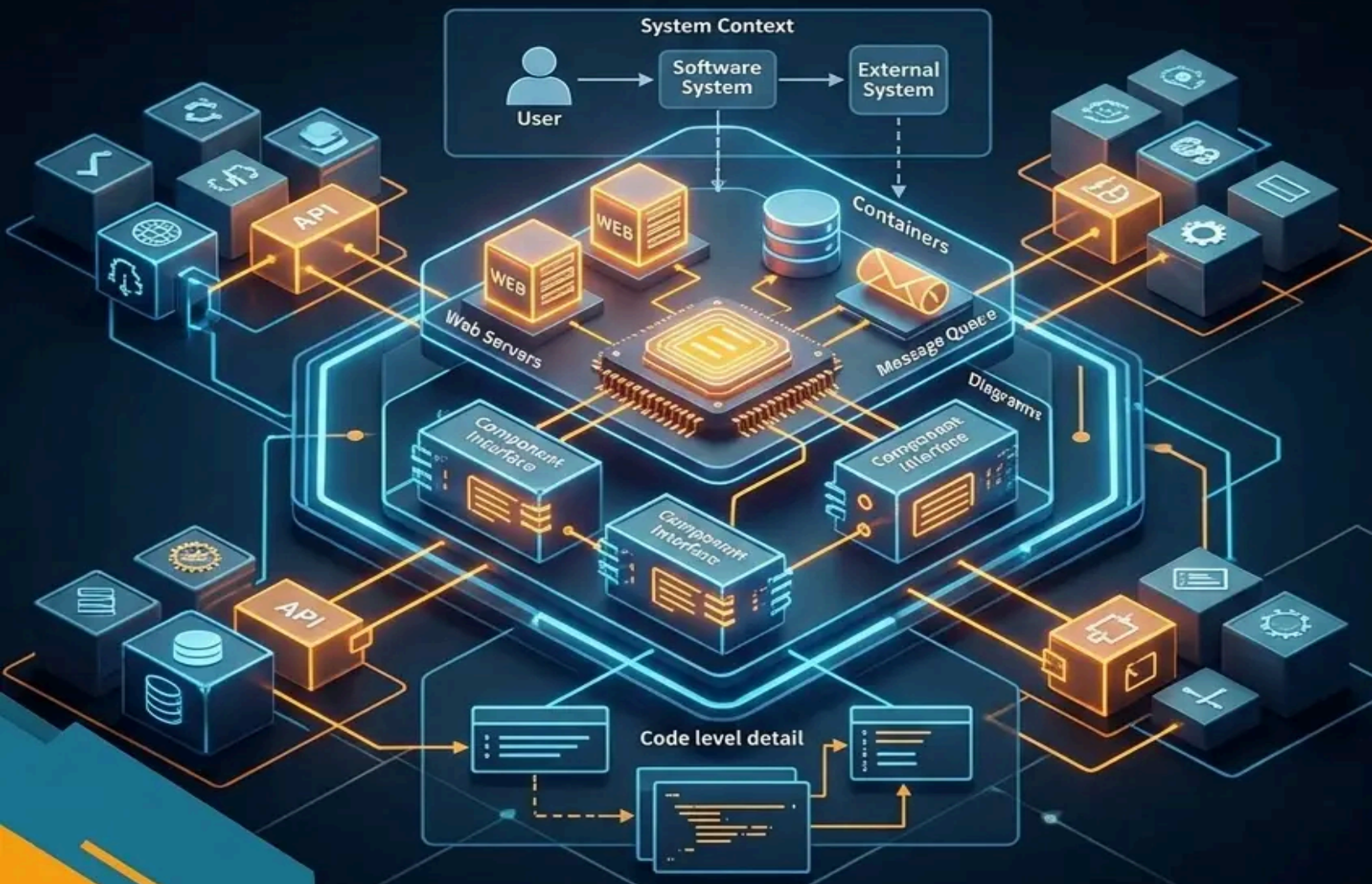
# Software Architecture

## for Developers

Volume 2

Anees Hikmat



بسم الله الرحمن الرحيم

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

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 الصامدين المرابطين في غزة والسودان وبورما وفي كل بقاع الأرض...

إليكم نُهدي هذا الكتاب

نهديكم هذه الكلمات عهدًا منا ووفاءً لكم؛ بأنكم في القلب باقون، وأننا لم ولن ننساكم مهما طال الزمان أو قصر...

نهديكم هذه الكلمات لتبقى خالدة تحيي ذكراكم، وتذكر العالم أجمع بمعاناتكم...

نهديكم هذه الكلمات لتكون شاهدا حيا يخلّد تضحياتكم، وصوتاً يصدح بمعاناتكم، حتى وإن خرس صوت الإعلام، وأصاب عدساته العمى...

نهديكم هذه الكلمات لأننا لا نرقى لصمودكم، ولم نرقى لأن نكون معكم... فلا شموخكم بلغنا، ولا في خنادقكم وقفنا؛ فتقبلوا منا هذا القليل، نبضا من قلوبنا، ومدادا من أقلامنا..

اللهم نصرك المؤزر المبين... ولا حول ولا قوة إلا بك

# 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 قد وصلت إلى هنا، فقد أدركت يقيناً في الجزء الأول أن المعمارية ليست مجرد أسطر من الشيفرة البرمجية، بل هي رؤية شاملة وحلول لمقايضات صعبة، وأنت الآن تقف أمام تحدٍ جديد، وشعورٌ مألوف بالضياع حين تقف أمام لوح أبيض محاولا شرح فكرتك لفريقك، أو حين تقرأ توثيقا فنيا لا يمت بصلة للشيفرة البرمجية التي بين يديك!

إن الفكرة العظيمة تموت إذا لم نستطع إيصالها، وإن المعمارية التي لا تُوثق وتُشرح بشكل صحيح هي مجرد وهم يعيش في رأس صاحبها فقط! وهذا الجزء كتب خصيصا ليأخذ بيدك من عالم التنظير إلى عالم العمل، ومن الخيال النظري إلى الواقع العملي...

**لمن هذا الكتاب؟**

هذا الكتاب لكل مطور سئم من قضاء الساعات في محاذاة المربعات والأسهم في أدوات الرسم التقليدية ليكتشف في اليوم التالي أن الشيفرة البرمجية قد تغيرت وأصبحت رسمته كاذبة!

وهذا الكتاب لكل مطور يرى في التوثيق الفني عبئا ثقيلا يتمنى الهرب منه؛ لأنه لم يجرب يوما متعة تحويل المعمارية إلى شيفرة برمجية -Architecture as Code-.

وهذا الكتاب لكل من يريد أن يجعل نظامه البرمجي كتابا مفتوحا، يرحب بأي مبرمج جديد، ويروي قصة النظام بوضوح واتساق لا لبس فيه.

**ماذا ستجد في هذا الكتاب؟**

لن تجد في هذا الكتاب قوالب توثيق جامدة ومملة من مئات الصفحات التي لا يقرأها أحد! بل ستجد دليلا عمليا يعلمك كيف تبني دليلا سياحيا حقيقيا وعمليا لنظامك، وسنتعلم كيف نتحدث بلغة بصرية موحدة عبر نموذج C4، وكيف سننتقل من عقلية "رسم المخططات الميتة" إلى عقلية "بناء النماذج الحية -Modeling-"، وستكتشف أن التوثيق يمكن أن يكون ممتعا، وأن المخططات يمكن أن تتحدث تلقائيا مع كل تعديل...

**كيف تستفيد منه؟**

اقرأ هذا الجزء وعينك على مشروعك الحالي، ولا تكتفِ بهز رأسك إعجابا بالأدوات والمنهجيات؛ بل افتح محرر الشيفرة البرمجية ثم حلل، وجرب، ووثق بعض القرارات المعمارية.. انظر لمشروعك، ارسمه وارسم حاوياته ومكوناته واستمتع بما تشاهد ^^

**المرجع الرئيسي لهذا الكتاب؟**

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

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

القرآن اعتبر العلم بثمرته لا بآلته، وثمرة العلم العبودية لله، فمن ضيع الثمرة لم تنفعه الآلة...

- كتاب رقائق القرآن، إبراهيم السكران -فرج الله عنه-

# الفهرس


**إهداء..... 3**
**المقدمة ..... 4**
**الفهرس..... 7**
**الوحدة الأولى: هناك عجز عن التواصل الفعال ..... 11**
الفصل الأول: ثمة شيء ما لا يزال مفقوداً ..... 11
ماذا حدث لـ SSADM و RUP و UML...إلخ؟ ..... 14
1. أول مثال عملي = نظام المخاطر المالية: ..... 23
قائمة التسوق -:-The Shopping List ..... 24
صناديق بلا خطوط -:-Boxes and No Lines ..... 27
المنظور / العرض الوظيفي -:-The Functional View ..... 31
خريطة الطيران أو خريطة مسارات الخطوط الجوية -:-The Airline Route Map ..... 33
الصحيح عموماً -:-Generically True ..... 37
المنظور المنطقي -:-The logical view ..... 39
غياب التفاصيل التقنية -:-Missing Technology Details ..... 41
آلية النشر مقابل بيئة التشغيل -:-Deployment vs Execution Context ..... 44
كائنات سي شارب المشرّدة -:-Homeless Old C# Object (HOCO) ..... 48
اختر مغامرتك الخاصة -:-Choose your own adventure ..... 52
المقاتلون النمطيون، جنود بلا ملامح -:-Stormtroopers ..... 55
يعد هذا الخطأ واحد من أكثر الأخطاء التي يقع فيها المطورون في أول أيام تعلم الرسم المعماري، هذا الخطأ هو رسم صور أشخاص دون تفاصيل مفيدة؛ كما في الصورة أدناه: ..... 55
كان ينبغي علينا استخدام اللوح الأبيض -:-Should have used a whiteboard! ..... 57
المشكلات الشائعة - خلاصة المشاكل التي تعلمناها من الأمثلة ..... 61
الافتراضات الخفية - فخ: سأشرحها لكم في الاجتماع! ..... 63
الفصل الثاني: أزمة المصطلحات، نحن نحتاج إلى مفاهيم مشتركة أكثر من حاجتنا لرموز موحدة! ..... 65
المفاهيم المشتركة أهم من الرموز --Common abstractions over a common notation ..... 66
البنية الثابتة -Static Structure-: الهيكل العظمي للنظام (ولادة نموذج C4) ..... 67
مثال عملي ٢: عيادة الحيوانات الأليفة -Spring PetClinic-، تمثيل الانتقال من مستوى الشيفرة البرمجية إلى المكونات --components ..... 76
من الشيفرة البرمجية إلى المُكوّن -Component-: التصفية والتجميع! ..... 77
أين اختفت الوحدات -Modules- والأنظمة الفرعية -Subsystems-؟ ..... 86
ما هو موقع ال Microservices في هرم ال C4؟ ..... 88
ماذا عن تقنيات ال Serverless؟ ..... 91
أين نضع المنصات -Platforms- وأطر العمل -Frameworks- والمكتبات -Libraries- في نموذجنا المعماري؟ ..... 91
اصنع قاموسك الخاص: الجوهر أهم من المسميات! ..... 95
الفصل الثالث: نموذج C4! ..... 96
الخرائط الهرمية ومبدأ شنايدرمان --Shneiderman ..... 99
التنقيب البرمجي --Software Archaeology ..... 103
الفصل الرابع: Level 1: System Context diagram ..... 106
أولا: الهدف المعماري - الأسئلة الثلاث ..... 107

ثانيا: رفع الأساسات وضبط قواعد الرسم؛ بطل قصتنا يجب أن يكون في قلب البناء! ..... 108
الفصل الخامس: Level 2: Container diagram ..... 118
الفصل السادس: Level 3: Component diagram ..... 131
لغة الأسهم ..... 136
معضلة البنية التحتية والاهتمامات المتقاطعة --Cross-cutting Concerns ..... 137
معضلة المكتبات والمكونات المشتركة -:-Shared components and libraries ..... 139
تحدي تعدد أشكال التنفيذ للمكون الواحد --Multiple component implementations ..... 140
الحفاظ على السياق (البشر، الأنظمة، والحاويات) ..... 142
حدود الحاوية --Container boundary ..... 142
إليك هذه الصدمة المعمارية: هل تعتقد أن هذا المخطط إلزامي أم اختياري؟ ..... 144
الفصل السابع: Level 4: Code-level diagram ..... 149
فخ التفاصيل: كم مقدار التفاصيل المطلوب في هذا المستوى؟ ..... 150
إليك هذه الصدمة المعمارية: هل تعتقد أن هذا المخطط إلزامي أم اختياري؟ ..... 151
الفصل الثامن: Notation ..... 153
1. العناوين --Titles ..... 154
2. مفاتيح الخريطة -Keys and legends-: لا تفترض أن القارئ هو المحقق كونان! ..... 155
الأشكال الهندسية وهويتها ..... 157
الألوان: ليس للزينة فقط، بل تنقل تصورات ورؤى بصرية! ..... 162
الأشكال -:-Shapes ..... 164
الحدود -Borders-: فن التجميع والتأكيد ..... 165
الأحجام ليست عشوائية وذات معنى: الفخ النفسي! ..... 166
احذر الرسائل البصرية الخفية في مخططاتك! ..... 167
الخطوط! ..... 167
الشاشات مقابل السبورات --Electronic tools vs Whiteboards ..... 171
المختصرات --Acronyms ..... 173
سمات الجودة --Quality attributes ..... 174
نطاق المخطط --Diagram scope ..... 176
الفصل التاسع: المخططات يجب أن تعكس الواقع --Diagrams must reflect reality ..... 184
المعمارية دون تقنية ليست سوى أحلام يقظة ..... 188
الخيال المعماري مقابل الواقع البرمجي ..... 190
الفصل العاشر: مخططات أخرى ..... 192
العمليات التجارية وسير العمل -:-Business process and workflow ..... 205
نموذج النطاق والمفاهيم --Domain Model ..... 206
السلوك الديناميكي ووقت التشغيل --Runtime & Behaviour ..... 209
البنية التحتية --Infrastructure ..... 216
مخطط النشر --Deployment Diagram ..... 217
نصائح خطيرة؛ لكنها ذهبية ..... 222
**الوحدة الثانية: التوثيق ..... 225**
الفصل الحادي عشر: توثيق البرمجيات كدليل إرشادي ..... 225
التوثيق كواجب أخلاقي ومهني --Our duty to deliver documentation ..... 226
عقدة التسعينيات: التوثيق المرعب --The 1990's template ..... 227

العنصر الأول: الخرائط --Maps ..... 228
العنصر الثاني: المعالم --Sights ..... 229
العنصر الثالث: التاريخ والثقافة --History and culture ..... 230
العنصر الرابع: المعلومات العملية --Practical information ..... 232
وثّق ما تعجز الشيفرة البرمجية عن وصفه وقوله! ..... 232
المشروع -Project- لا يساوي المنتج -!-Product ..... 234
كيف يمكننا أن نمنع موت التوثيق بسبب الإهمال والتجاهل؟ وكم يجب أن يكون طول التوثيق حتى لا يموت بهذا السبب؟ . 237
الفصل الثاني عشر: القسم الأول من الدليل = السياق --Context ..... 239
الفصل الثالث عشر: القسم الثاني من الدليل = Functional Overview ..... 240
الفصل الرابع عشر: القسم الثالث من الدليل = Quality Attributes ..... 241
الفصل الخامس عشر: القسم الرابع من الدليل = Constraints ..... 243
الفصل السادس عشر: القسم الخامس من الدليل = Principles ..... 244
الفصل السابع عشر: القسم السادس من الدليل = Software Architecture ..... 247
الفصل الثامن عشر: القسم السابع من الدليل = Code ..... 248
الفصل التاسع عشر: القسم الثامن من الدليل = Data ..... 249
الفصل العشرون: القسم التاسع من الدليل = Infrastructure Architecture ..... 251
الفصل الواحد والعشرون: القسم العاشر من الدليل = Deployment ..... 252
الفصل الثاني والعشرون: القسم الحادي العشر من الدليل = Operation and Support ..... 254
الفصل الثالث والعشرون: القسم الثاني العشر من الدليل = Development Environment ..... 256
الفصل الرابع والعشرون: القسم الثالث عشر من الدليل والأخير = Decision Log ..... 257
**الوحدة الثالثة: الأدوات ..... 261**
الفصل الخامس والعشرون: المسودات الأولية، والمخططات، والنماذج، والأدوات -Sketches, diagrams, models and tooling- ..... 261
المسودات الأولية / الرسومات اليدوية --Sketches ..... 261
المخططات -Diagrams- وفخ الصور الفوتوغرافية ..... 262
أدوات المخططات النصية --Text-based diagramming tools ..... 264
النماذج --Models ..... 265
أدوات النمذجة --Modeling tools ..... 266
لغات وصف المعمارية --Architecture Description Languages ADLs ..... 269
ال Structurizr ..... 270
**الخاتمة ..... 273**

# الوحدة الأولى: هناك عجز عن التواصل الفعال

قبل أن نبدأ في وضع الحلول ورسم المخططات وعرض الآليات الممكنة للوصول لنماذج تعبر عن المعمارية بشكل صحيح، وتبني طرق تواصل فعالة... يجب علينا أن نفهم المشكلة من خلال عملية استقراء لما هو موجود وما هو مفقود... ومن هنا نبدأ، ما هو الشيء الذي ينقصنا بالرغم مما وصلنا إليه اليوم؟!

## الفصل الأول: ثمة شيء ما لا يزال مفقوداً

في عالمنا اليوم لدينا العديد من القفزات المبهرة، والتقنيات المتعددة والمتنوعة، فتجد الأنظمة الموزعة عبر العالم، وتجد أنواع مختلفة من المعماريات المنتشرة، وتجد آليات حوسبة سحابية رائعة، وتجد آليات اختبار آلي ممتازة، وتجد آليات وطرق لإدارة المشاريع تتناسب مع احتياجات التطبيقات والمؤسسات المختلفة... لكن، وبالرغم من كل هذا التقدم؛ ثمة شيء جوهري قد ضاع منا، ولم ينل حقه!

لقد ذكرنا في الجزء الأول من هذا الكتاب، الترابط العجيب بين معمارية البرمجيات والهندسة المعمارية والتشييد -وتحدثنا عن مفهوم البناء الرئيسي تاريخيا-، وتحدثنا عن أهمية استنساخ تجارب العلوم الأخرى لنستفيد منها ونتعلم، ورأينا كيف أثر ذلك على العلوم المتعلقة بمعمارية البرمجيات، لكن، لو قمنا بطرح أسئلة بريئة لكل من معماري البرمجيات ومعماري البناء حول

النماذج المعمارية -تجاهل أن هذه العلوم موجودة حاليا وهي ما سنقوم بتعمله في هذا الكتاب ^^-، فما هي النتيجة المتوقعة حسب ظنك؟ لن نعلم حتى نجرب... لنصنع جدولا:

| السؤال | سؤال موجه لمعماريّ البناء | سؤال موجه لمعماريّ البرمجيات |
|---|---|---|
| هل هناك خرائط موقع واضحة لما سيتم بناؤه؟ | سنحصل على خرائط واضحة لكل ما نحتاجه. | سنحصل على فوضى مركبة وكثير من الصناديق السوداء -المجهولة- والمترابطة فيما بينها بخطوط لا تعبر عن شيء بشكل واضح! |
| هل هناك مخططات لكل طابق من طوابق البناء؟ | نعم | لا يوجد شيء يعبر عن هذا الشيء في صناعة البرمجيات بشكل حقيقي أو فوضى من صناديق وخطوط غير مفهومة. |
| هل هناك إمكانية لاستعراض المعمارية بشكل رأسي -Elevation Views- | نعم، رسم ثنائي البعد يمكنك من خلاله النظر للمبنى من الجانب دون الحاجة للقطع ودون منظور أو عمق، مثل | لا يوجد شيء يعبر عن هذا الشيء في صناعة البرمجيات بشكل حقيقي |

| | | |
|---|---|---|
| | النظر للشكل الخارجي للبناء وتحديد الارتفاعات وأماكن البلكونات ونحو ذلك.. | |
| العروض المقطعية -Cross-section Views- | نعم، رسم ثنائي البعد يمكنك من النظر داخل المبنى وكأنك قمت بتطيعه | لا يوجد شيء يعبر عن هذا الشيء في صناعة البرمجيات بشكل حقيقي |
| رسومات تفصيلية -Detail Drawings- | نعم، فيتم رسم رسومات هندسية بمقياس رسم كبير -تكبير عالي كإنك عامل زووم-، تُركز على جزء صغير ومحدد من المبنى، ولا يمكن لنا رؤية هذه التفاصيل في المخططات العامة. | لا يوجد شيء يعبر عن هذا الشيء في صناعة البرمجيات بشكل حقيقي |

بعد النظر لهذا الجدول، سأسألك سؤالا، لو عرض عليك مهندس معماري شقة للبيع وأراك فقط موقع البناء وعليها صندوق مكتوب عليه "مطبخ" وصناديق مكتوب عليها "غرف نوم"، دون أبعاد واضحة أو اتجاهات أو أبواب أو مواد أو طابق... إلخ، فهل ستقوم بشراء هذه الشقة؟! الجواب بكل تأكيد لا! لن يشتري أحد منا السمك في الماء!

لكن، هذا هو الحال الذي تبدو عليه معظم المخططات البرمجية الموجودة، وهذه النظرة هي نظرة Simon للوضع المعماري الذي كتب كتابه في محاولة لحل هذه المشكلة... فيقول: "لقد

**نسينا** كيف نُصوِّر -نعرض visualise- تصميم البرمجيات، سواء كتوثيق بعد المشروع أو خلال عملية التطوير نفسها"، وهذه النتيجة وصل إليها بعد جمع آلاف الصور من ورش العمل المختلفة، وسؤال الكثير من المطورين من حول العالم، لذلك وصل إلى استقراء مفاده: إن النتائج "لا تزال تفاجئه"، وهذا فيه إشارة إلى أن المشكلة ليست في فرد أو فريق، بل في بنيوية صناعة البرمجيات نفسها، والتي تحتاج لتحسين وضبط حتى تصير النتائج معبرة كما هو حال النماذج المعمارية للأبنية...

ولأننا عرفنا سبب المشكلة الآن، فيمكننا الانتقال للقسم التالي...

## ماذا حدث لـ SSADM و RUP و UML...إلخ؟

هذه المصطلحات يمكن أنك سمعت بها أثناء دراستك الجامعة، أو أثناء عملك قديما -إذا كنت من المحاربين القدامى ^^-، هذه المصطلحات غالبا لم نعمل عليها أو تجاهلناها حتى ضاعت، لذلك قال Simon في الفقرة السابقة: "**نسينا**"، لإن السؤال الحقيقي متى ضاعت هذه المهارة المهمة! أي أنها كانت موجودة بطريقة ما ثم ضاعت! وحتى تفهم سبب الضياع دعونا نلقي نظرة خاطفة على هذه المصطلحات:

- ال RUP: هي اختصار ل Rational Unified Process، وترجمتها الحرفية: العملية العقلانية الموحدة، وهو إطار عمل لتطوير البرمجيات أنتجته شركة Rational، ويقسم عملية تطوير النظام إلى أربعة مراحل وهي البدء (تحديد فكرة المشروع ونطاقه) والإعداد (تحليل المتطلبات بدقة وتصميم بنية النظام والتحقق من المخاوف الكبرى) والبناء (مرحلة كتابة الشيفرة البرمجية وبناء النظام) والانتقال (تسليم النتائج للمستخدمين، مع وجود التدريب والصيانة ونحو ذلك)... كما أن هذا الأسلوب فيه

عشرات الأدوار (مثل مصمم ومعماري ومحلل نظم ومسؤول أداء...إلخ) وعشرات الوثائق، وهو من النوع التكراري (لا ينتظر حتى نهاية المشروع، بل يتم تقسيم العمل لوحدات صغيرة يمكن إنتاجها بشكل مستقل وتسليمها والحصول على تغذية راجعة خاصة بها)... ومع ذلك فهذا النموذج ثقيل جدا ومليئ بالوثائق! لكنه مع ذلك كان يعد قفزة قوية ومثيرة مقارنة بالنموذج التقليدي لل Waterfall، خصوصا مع وجود فكرة التكرار... وبالمناسبة، كان هذا الأسلوب الثقيل أحد الأسباب التمهيدية التي ساهمت في قبول ال Agile عند ظهوره ^^.

- ال SSADM: هي اختصار ل Structured Systems Analysis And Design Method، وترجمتها: منهجية التحليل والتصميم الهيكلي للنظم، وهي منهجية بريطانية رسمية طورتها الحكومة لاستخدامها في المشاريع الحكومية الكبرى، وتستخدم مخططات ال DFDs ونماذج ال ERDs، وتمثل هذه المنهجية المثال المباشر والنموذجي لأسلوب ال Waterfall الصارم... والآن تعتبر هذه المنهجية شبه منقرضة ^^.

ملاحظة: ال DFD اختصار ل Data Flow Diagrams، ويقصد بها المخططات التي تركز على حركة وسير العمليات والبيانات داخل النظام، وهي تجيب أو تمثل: من أين تأتي؟ كيف تتغير؟ وأين تذهب؟ أما ال ERDs فهي اختصار ل Entity Relationship Diagrams، وهي تمثل المخططات الهندسية لقاعدة البيانات، بحيث يتم تحديد العلاقة بين الجداول فيما بينها وهيكلها...

- ال UML: وهي اختصار ل Unified Modeling Language، وترجمتها الحرفية: لغة النمذجة الموحدة، وهي لغة قياسية لرسم مخططات النظم البرمجية، وتتضمن ١٤ نوعا مختلفا من المخططات مثل مخططات الفئات ومخططات حالات الاستخدام ومخططات التسلسل... إلخ، وتم اعتمادها كمعيار من منظمة OMG...، وسأسلك سؤالا، هل تظن أن ال UML ما زالت موجودة ومستخدمة بقوة؟... الجواب، نعم، ال UML لم تمت بعد! ولاحظ أنني قلت أنها لم تمت، والسبب في ذلك أن الجواب مرتبط بسؤال: "هل ما زالت نماذج ال UML فعالة؟"... والحقيقة أن كثير من الفرق طردوها خارج أعمالهم وكثير من الفرق لا يعرفونها أساسا ولا يرغبون في العمل عليها إن عرفوها!

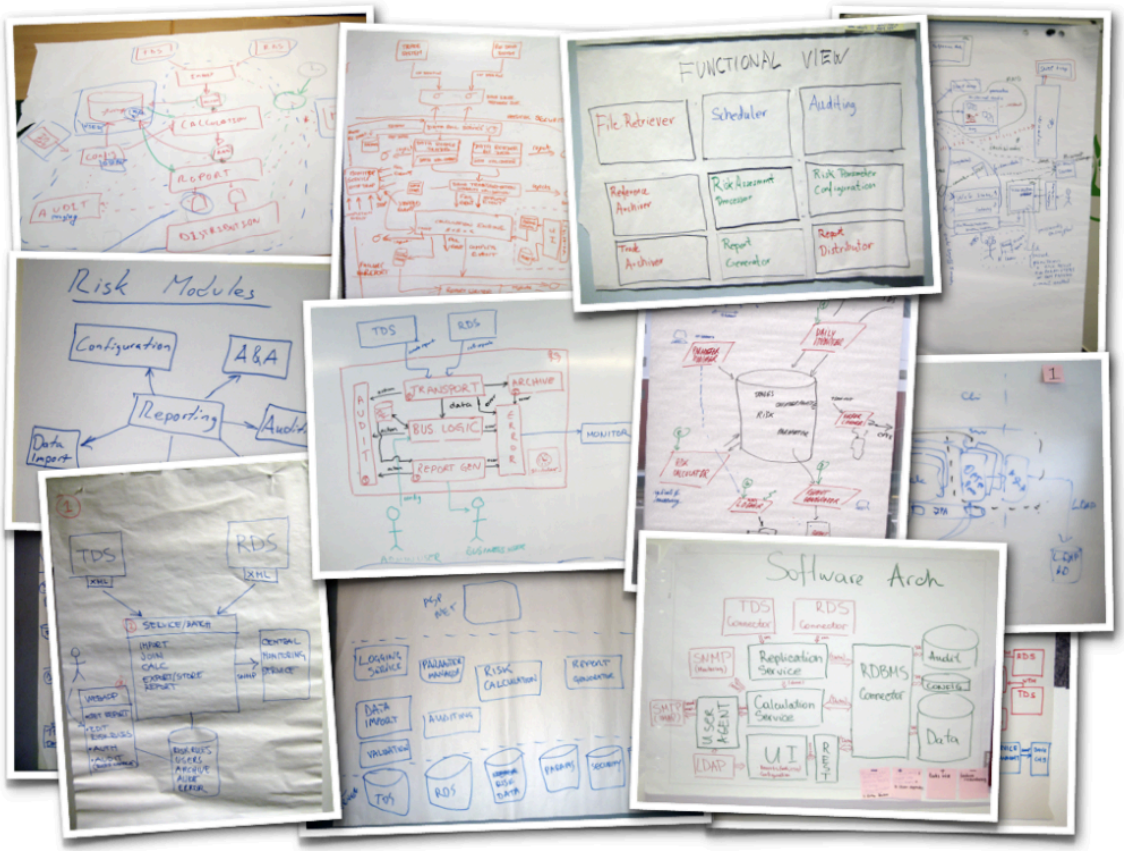
ملاحظة: منظمة OMG هي اختصار لـ (Object Management Group)، وهي الجهة العالمية المسؤولة عن وضع معايير هندسة البرمجيات، وقد اعتمدت الـ UML رسمياً عام 1997.

بعد هذا الاستعراض الجميل للتراث القديم، نجد أن الحاضر يعمل بإسلوب مختلف تماما!، وذلك من خلال فوضى عارمة -بدل الوثائق الكثيرة والكبيرة- وذلك من خلال "رسومات غير رسمية تعرض من خلال مجموعة فوضوية من الصناديق والخطوط" التي لا معنى لها دون سرد أو شرح مفصل! لذلك، يمكننا القول: "رغم أن صناعة البرمجيات تقدمت في كثير من الأمور، إلا أننا نسينا بعض الأشياء الجميلة والمهمة التي قدمتها لنا المنهجيات السابقة..."، وهنا

يجب أن نسأل أنفسنا، لماذا لم نأخذ موقفا وسطيا بين النماذج والمخططات الثقيلة التي استخدمت في السابق وبين الفوضى الحالية في تصميم البرمجيات؟!

فكرة: "المخطط الذي لا يقف بذاته ليس مخططا! بل هو مجموعة من الملاحظات لا أكثر... فإذا احتجت لشخص حتى يشرحه لك، فأنت تعتمد على **ذاكرة بشرية** قابلة للضياع، لا على **توثيق دائم**"!

شاهد هذه الصورة ثم اتبعني ^^:




**A selection of typical "boxes and lines" diagrams**

كما تلاحظ في الصورة السابقة، فهي نموذج عملي وواقعي شاركه Simon لمجموعة من النماذج التي قام برسمها المطورين خلال ورش العمل... وكما ترى، فإن هذه الصور لا تعبر عن مخطط أو نموذج معماري حقيقي! بل هي مجرد خطوط وصناديق تحتاج إلى شرح!

ومن هنا، ينطلق Simon ليضع حدودا واضحة لما سنتعلمه، وهو الأسلوب الذي الذي يراه وسطيا للتخلص من التعقيدات السابقة، لذلك فهو يصف آلية العمل هذه بكونها: "Lightweight"، فلن نحتاج إلى النمذجة التفصيلية ولا إلى نماذج ال UML الشاملة ولا إلى ال Model-Driven Development... بل إن ما سنقوم بفعله هو بناء آلية تواصل فعال للمعمارية التي سنقوم ببنائها، فالهدف من التوثيق والرسم ليس مجرد رسم أشكال هندسية معقدة، بل هو تواصل يجعل فريق العمل المبرمجين والمدراء والعملاء يفهمون الهيكل العام للنظام -المعمارية- بسرعة وكفاءة، ودون الغرق في تفاصيل لا داعي لها ^^.

والسؤال الآن، ما هي الفوائد والأهداف التي ستقودنا لاستخدام نهج ال Lightweight أو سنجنيها من وراء استخدامه؟ والجواب يمكن تلخيصه في هذه النقاط:

1. رؤية الصورة الكبرى -The Big Picture-: ويقصد بها مساعدة الجميع على فهم "الصورة الكبرى"، فمثلا من خلال الصورة الكبرى يمكننا جعل المبرمج يرى النظام كاملا من الأعلى، وكيف يرتبط هذا النظام بالأنظمة الأخرى في الشركة، بدلا من أن يكون تركيزه محصوراً فقط على السطور البرمجية الذي يكتبها!
2. بناء رؤية مشتركة -Shared Vision-: خلق رؤية مشتركة لما نقوم ببنائه داخل الفريق، وذلك يضمن أن جميع أفراد الفريق من مبرمجين ومصممي قواعد بيانات

ومختبري جودة؛ لديهم نفس الفهم المعماري للنظام، مما يمنع التشتت والعمل في جزر منعزلة...

3. الحفاظ على التركيز -Focal Point-: وجود مخطط واضح ومبسط يعمل كمرجع دائم يعود إليه الفريق للتأكد من أن الفريق لم ينحرف عن التصميم الأساسي أثناء فترات العمل وطوال فترة حياة المشروع! -أي أن هذا الرسم سيوفر نقطة ارتكاز لفريق التطوير حتى يحافظوا على تركيزهم في ماهية البرنامج وكيفية بنائه.
4. توجيه النقاشات التقنية -Technical Conversations-: يسهل ويمحور النقاشات التقنية حول كيفية تنفيذ الميزات الجديدة، فعندما يطلب العميل ميزة جديدة؛ سيستخدم الفريق هذه المخططات للنقاش مثل: أين سنضع هذه الشيفرة البرمجية الجديدة؟ وهل سيؤثر هذا التعديل على قاعدة البيانات؟ ... إلخ، إن المخطط يسهل هذه النقاشات بدلاً من القيام بضربات تخمين عشوائية هنا وهناك!
5. خريطة للشيفرة البرمجية -A Map for Source Code-: توفير "خريطة" يمكن أن يستخدمها المبرمجون للتنقل داخل الشيفرة البرمجية للمشروع، فعندما تنظر إلى المخطط؛ يجب أن تعرف تقريبا في أي مجلد أو ملف ستجد الشفرة البرمجية الخاصة بهذا الجزء من النظام.
6. الشرح لغير التقنيين -Communication outside the team-: مساعدتك في شرح ما تقوم ببنائه للأشخاص خارج فريق التطوير، سواء كانوا تقنيين أو غير تقنيين!
فالمخططات الخفيفة مفيدة لتبسيط وإيضاح فكرة النظام = للمدراء، أو المستثمرين، أو قسم المبيعات... إلخ، وفي نفس الوقت تكون دقيقة بما يكفي ليفهمها مدراء الخوادم والشبكات باعتبارهم تقنيين من خارج الفريق...

7. تسريع اندماج الموظفين الجدد -Fast-track On-boarding-: تسريع عملية دمج المبرمجين الجدد في الفريق، فعندما ينضم مبرمج جديد للشركة نقوم بإعطاؤه المخططات ليفهم معمارية النظام ويبدأ العمل والإنتاج في وقت قياسي! بدلاً من إجباره على قراءة آلاف الأسطر من الشيفرة البرمجية ليفهم النظام -وهو أمر مرهق وبطيء-.

الخلاصة: إن أي مخطط ستقوم بإنتاجه = يجب أن يعكس الواقع! فإذا عكس الواقع رأيت الثمرات التي بالأعلى أمام عينيك، وذقت حلاوتها... أما إن كان المخطط لا يعكس الواقع فهو في هذه الحالة **تضليل لا توثيق** ^^... وهذا ما سنتحدث عنه لاحقا تحت عنوان: فجوة النموذج والشيفرة البرمجية -Model-Code Gap-... لكن كتشبيه سريع لهذه الخلاصة، تخيل أنك تستخدم تطبيق للخرائط لم تقم بتحديثه منذ ١٠ سنوات... الشوارع تغيرت وأحياء جديدة ظهرت وأخرى هدمت، فهل يمكنك استخدام هذه الخرائط القديمة للسفر والتجوال في هذه المنطقة؟! الجواب بكل تأكيد لا، لأنك ستضل، وستضيع الكثير من الوقت والمال وأنت في تيه، أما إن كانت الخرائط محدثة فأنت تطبق ما وصلنا إليه في هذه الخلاصة، وسنتحدث عن ذلك في مزيد من التفصيل لاحقا بإذن الله تعالى...

ملاحظة: يستحضر Simon في كتابه نوعين من ورش العمل التي يستخدمها وهي:

1. ورشة كاتا المعمارية -Architecture Kata-: ال Kata هي مجموعة من الحركات القتالي في فن الكاراتيه والتي يتم إعادتها الكثير من المرات حتى تصبح هذه الحركات جزء من طبيعة اللاعب وتخرج منه كرد فعل طبيعي، وعلى نفس المنوال فإن ذلك في سياق البرمجيات يعني أن يقوم المبرمجون بنفس عملية التكرار والتدريب حتى يصبح

ذلك سلوكا طبيعيا عندهم، لكن هنا على مستوى "العضلات الذهنية ^^"، ومن الأمثلة على ذلك: أن يُعطى الفريق مشكلة وهمية أو افتراضية مثل صمم تطبيق بنكي أو نظام حجوزات طيران...

2. ورشة المخططات -Diagramming Workshop-: هنا يطلب من المطورين رسم النظام الذي يعملون عليه فعلياً في الوقت الحالي، وهذا أكثر صعوبة من Kata لأنه يكشف فجوة الفهم الخاصة بهم لنظامهم! وعادة ما تنتج هذه الورشة صدمة إيجابية للمطورين، لإنها تبرز لهم جوانب كثيرة لم يفهموها بشكل جيد، أو نُسيت أو هناك فجوة بين الموجود والذي في أذهانهم، وهذا سيعطيهم دفعة قوية لتعلم آلية الرسم الصحيح لمعماريتهم.

والآن، بعد هذا الكلام اللطيف، دعونا نبدأ ^^

إن التواصل البصري للمعمارية لا يُعلم / يُدرس بشكل استباقي؛ أكان ذلك في المؤسسات التعليمية أو في بيئات العمل... مما خلق لدينا حلقة تكرارية مفرغة من معلمين لم يدرسوا هذا الباب، وبين متدربين لم يتعلموها بشكل صحيح فصاروا مدربين غير قادرين على التدريب... وهكذا، وهذا يقودنا إلى صدمة غياب التوجيه (الأسئلة الأربعة)، هذه الأسئلة تكشف لنا مشاكل الجهل البنيوية التي لدينا، وهي:

1. ما هي أنواع المخططات التي يجب أن نرسمها؟

وهذا السؤال يكشف الجهل البنيوي حول وجود أنواع متعددة من المخططات، منها ما هو على مستوى المكونات -component-، ومنها ما هو على مستوى الحاويات -container-...إلخ

2. ما هي آلية التدوين المستخدمة أو المعتمدة؟ ما هي الرموز والأشكال التي يجب استخدامها؟

وهذا يكشف الجهل المتعلق بعدم وجود لغة بصرية موحدة... فهل استخدم مربعات؟ أم دوائر؟ أم أسهم متقطعة؟

3. ما هو مستوى التفصيل الذي يجب تقديمه؟

وهذا يكشف الجهل المتعلق بمستويات التجريد -abstraction- الموجودة... فهل نرسم النظام من الخارج فقط، أم ندخل في تفاصيل كل شيفرة برمجية؟...إلخ

4. من هو الجمهور المستهدف من هذه المخططات؟

هل نرسمها للمدير، أم للعميل، أم للمبرمجين الجدد؟

لو أردنا أن نسقط هذه الأسئلة على الواقع على سبيل التشبيه، فإن هذه الأسئلة الأربعة تشبه أن يُسأل المهندس المعماري الذي يعمل في تشييد الأبنية: "هل أرسم مخططاً أفقياً أم رأسياً؟ وبأي مقياس؟ وبأي اصطلاحات؟ ولمن؟"... في تشييد الأبنية هذه الأسئلة لها إجابات معيارية تُدرَّس في الجامعات، لكن في البرمجيات لا توجد هذه الرفاهية عندنا :P.

الخلاصة: "المبرمجون لا يعرفون كيف يرسمون أنظمتهم لأن الجامعات لم تعلمهم ذلك، ولأنهم تخلوا عن الـ UML لتعقيده، ولأنهم فهموا الأجايل Agile بشكل خاطئ؛ فتركوا التوثيق تماماً، ولهذا نحن بحاجة إلى طريقة جديدة وخفيفة وموحدة لحل هذه الفوضى"...

والآن، أعتقد أنك سئمت من كثرة التنظير ^^، فهيا بنا للنتقل من مرحلة التنظير وذكر المشاكل إلى **التطبيق العملي** ^^، لذلك سنقوم باستعراض أمثلة حقيقية من ورش العمل التي أدارها Simon، ليرينا كيف يقوم المبرمجون برسم الأنظمة بشكل خاطئ أو غير مكتمل ^^... هيا بنا لنشاهد هذه الأمثلة ^^:

## 1. أول مثال عملي = نظام المخاطر المالية:

في ورش العمل، يطلب Simon من المتدربين -المبرمجين- تصميم نظام وهمي لبنك يُسمى **نظام المخاطر المالية -Financial Risk System-**، وقد حدد لهم المتطلبات التالية للنظام لكي يقوموا برسمه:

- **المدخلات:** النظام يجب أن يسحب البيانات من مصدرين (نظام بيانات التداول -TDS-، ونظام البيانات المرجعية -RDS-).
- **المعالجة:** يقوم النظام بدمج هذه البيانات معاً وإجراء حسابات معينة لقياس المخاطر.
- **المخرجات:** يُصدر النظام تقارير بصيغة Microsoft Excel ليقرأها ال Business users.
- **التفاعل:** مجموعة محددة من هؤلاء الموظفين لديهم صلاحية لتعديل بعض المُدخلات أو المعايير -Parameters- التي تُستخدم في الحسابات.

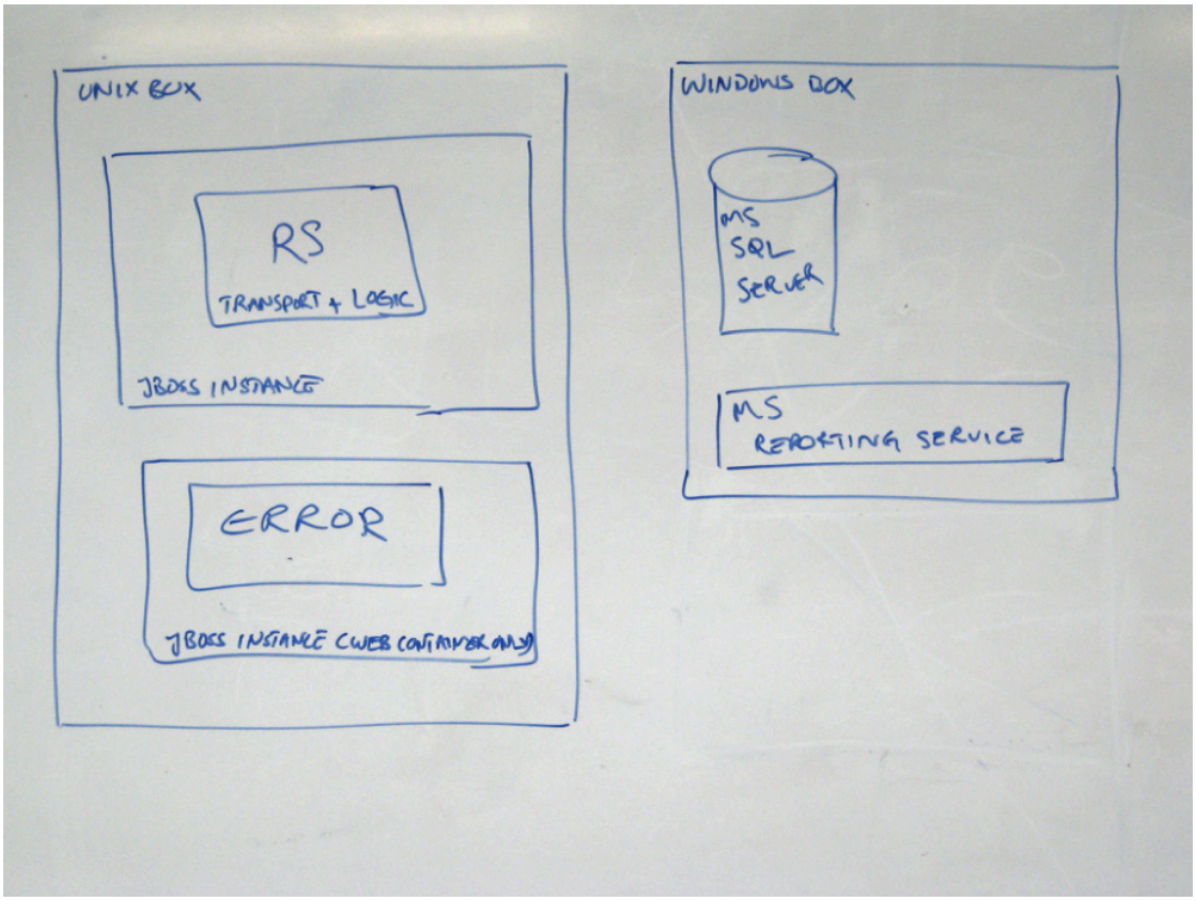
إن متطلبات هذا المثال لهذا النظام واضحة جداً، وذلك كي نتمكن من الحكم على الرسومات والمخططات التي سيتم عرضها لاحقاً لمعرفة ما إذا كانت تلبي هذه المتطلبات أم لا...

ملاحظة مهمة: قبل أن نبدأ في طرح الأمثلة، عليك أن تعلم أن وقوعك في مثل هذه الأخطاء لا يعني أبداً أنك سيئ أو غبي أو قليل الكفاءة؛ فمن منا لم يقع يوماً فيها! إن أي نقدٍ ستراه هنا هو نقدٌ لنتيجة المخطط، وليس لشخصك كمطور برمجيات! ولو كان الهدف هو الانتقاص، لما كان هناك داعٍ لكتابة هذه الكلمات أو عرض هذا الكتاب، ولكن؛ لأننا نؤمن أن أولى مراحل العلم هي أن تدرك أنك لا تعلم؛ فإن هذه الكلمات بإذن الله تعالى ستقودك لتتعلم بعض ما كنت تجهله، والله المستعان.

ملاحظة ٢: ستجد كثير من المخططات التي يمكن أن نستعرضها، إن استعراضها لا يعني أنها سيئة بالكلية، أو أنها جيدة بالكلية، بل سنمر ونأخذ الجيد من كل مخطط حسب حاجتنا، وسنشير للسيء في كل مخطط حتى نتعلم من الأخطاء التي وجدت فيه... وبهذا كله = سنبني في عقولنا فهما متينا بإذن الله تعالى.

**قائمة التسوق -The Shopping List-:**

إن أول مثال لمخطط سيء يُعرض ويقوم به المبرمجون هو ما أطلق عليه Simon قائمة التسوق! وهو عنوان ساخر -كما هي عادته ^^- يشير فيه إلى نمط الفشل في هذا الأسلوب... شاهد صورة المخطط ثم تابع معي:




بناء على الصورة التي بالأعلى، هل أدركت لماذا أطلق عليها اسم قائمة التسوق؟

نعم يا صديقي، السبب في ذلك يعود لكون هذا المخطط لا يخبرك الكثير عن الحل، بل هو مجرد عرض لقائمة من التقنيات التي جلبها المبرمج من السوق ووضعها داخل صناديق! إن هذا المخطط يشبه تماما أن يرسل لك صديقك مكونات صنع العوامة -يا سلام على صحن- أو الهريسة دون أن يخبرك كيف ستستخدم هذه المكونات... أنت حينها قمت فقط بالتسوق! لأنك لا تملك الوصفة!

حسنا، لو قمنا في النظر إلى الصورة مجددا وحاولنا الحصول على معلومات، ما هي المعلومات التي يمكن تحصيلها؟

- صندوق ال UNIX والذي يمثل نظام تشغيل
- صندوق ال Windows والذي يمثل نظام تشغيل
- صندوق ال JBoss وهو عبارة عن خادم لتشغيل تطبيقات الجافا، وهناك نسخة أخرى مستخدمة لمعالجة طلبات الويب فقط
- صندوق لقاعدة البيانات
- صندوق ال MS Reporting Service ويمثل خدمة لتوليد التقارير من قاعدة البيانات.

حسنا، بناء على هذه التفاصيل، ما هي المشاكل التي جعلت هذا المخطط يظهر بهذا الشكل الفاسد أو الرديء؟

الجواب باختصار، بسبب:

- **غياب الوظيفة:** "إننا لا نعرف ماذا تفعل هذه المنتجات"، فالمخطط يخبرنا بوجود قاعدة بيانات -SQL Server! لكنه لا يخبرنا **ما هي البيانات** المخزنة فيها؟ هل هي بيانات التداول أم تقارير المخاطر؟
- **غياب الروابط -Connections:** لاحظ أنه لا يوجد أي خط أو سهم يربط بين خادم الـ Unix وخادم الـ Windows، فإذا كان النظام يُفترض أن يدمج

بيانات من مصادر مختلفة، فكيف ستنتقل البيانات إذا لم تكن هناك آلية تواصل واضحة بين الخوادم في المخطط؟

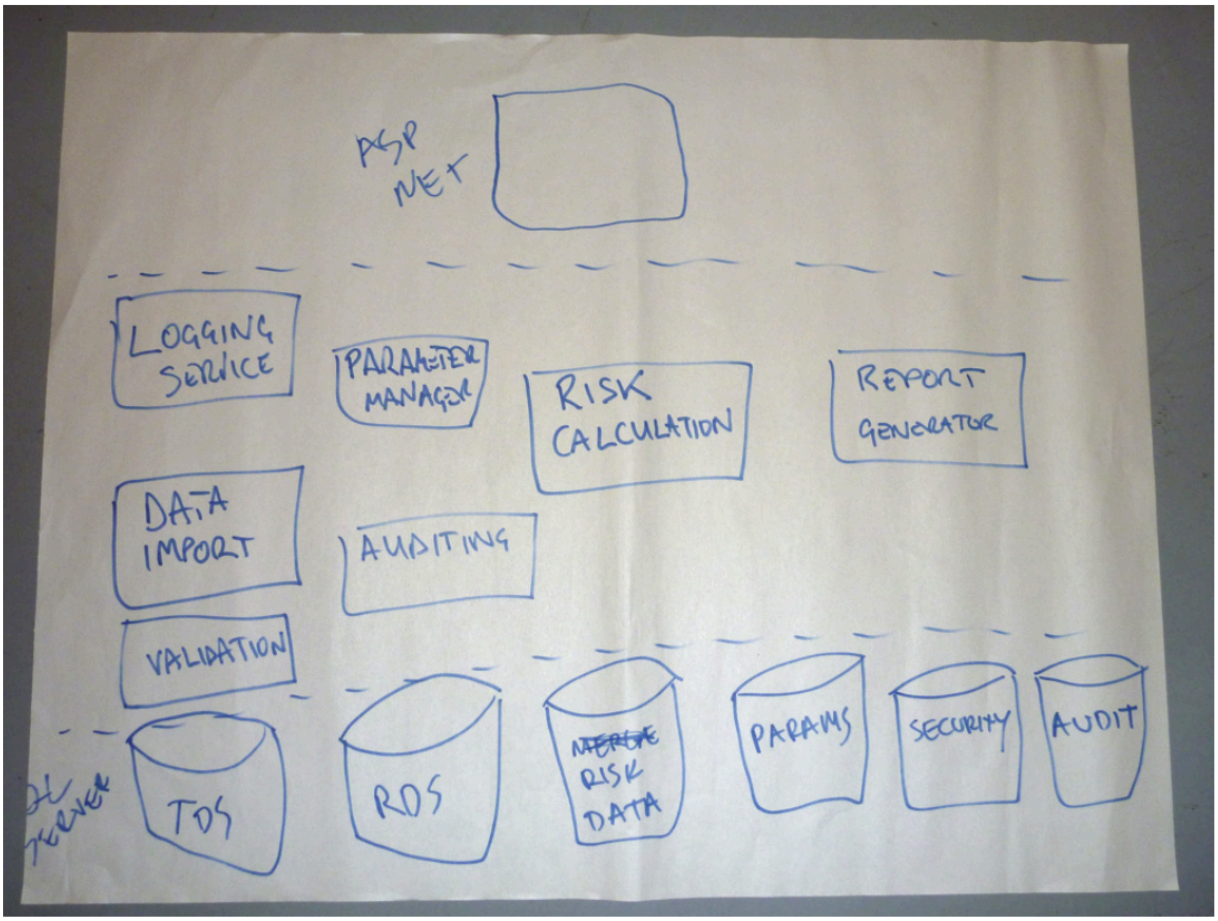
ما فعله هذا المبرمج هنا: هو مجرد وضع "للنقاط النصية" داخل صناديق لتبدو وكأنها رسم هندسي! لكنها في الحقيقة خالية من أي قيمة معمارية توضح كيف سيعمل النظام وكيف ستتدفق فيه البيانات!

الخلاصة التي نخرج منها من هذا المثال:

إن هذا المخطط في جوهره لا يمثل سوىٍ قائمة على هيئة مخطط، لكنك قمت بزخرفتها فقط من خلال الرسم بدلا من سردها على شكل نقاط! أما **المخطط** الحقيقي فهو يُظهر **العلاقات** بين المكونات المختلفة... إذا أول مشكلة انتهينا منها: "عشوائية التقنيات - قائمة التقنيات المجردة"

**صناديق بلا خطوط -Boxes and No Lines-:**

إن المطورون يتفقون على أن المعمارية = صناديق + خطوط -boxes and lines-، لكن واقع كثير من المطورين هو = رسم الصناديق دون خطوط ^^... شاهد الصورة أدناه:




لاحظ أن هذه الصورة مقارنة بالصورة الأولى تعد أوضح، وفيها يظهر تطور إيجابي، فقد فعل الفريق هنا شيئا صحيحا، وهو تقسيم المعمارية لطبقات، وتقسيم الطبقة الوسطى إلى مكونات صغيرة، وهذا يعد تقدما ممتازا بدلا من الفوضى التامة!

إن المشكلة الأساسية في هذا المخطط يمكن تقسيمها إلى:

- الغموض في دلالة الاسطوانات المرسومة، فهي في تقاليد رسمنا لقواعد البيانات أو تمثيلها يمكن أن تستخدم للإشارة لقاعدة البيانات على server منفصل، أو schema داخل قاعدة بيانات أو جدول داخل مخطط! وكل واحدة من هذه الخيارات لها انعكاسات خطيرة على المعمارية متعلقة بالأداء أو النسخ الاحتياطي أو الأمان أو

القدرة على التوسع... تركها غامضة هكذا = ترك قرار معماري جوهري ليكون دون إجابة واضحة.

- غياب التفاعلات والمسؤوليات بين الصناديق المختلفة، فمن هو المسؤول هنا عن إرسال النتائج ومن هو المسؤول عن طلبها، وماذا تفعل كل خدمة موجودة؟ ومن يستدعي من؟ أي ما هو تسلسل العمل؟...إلخ

الخلاصة التي نخرج منها من هذا المثال هي:
إن جوهر المعمارية البرمجية يكمن في بنيتها الهيكلية المعتمدة على عنصرين: "الأجزاء" (الصناديق) و"التفاعلات" (الخطوط)، وغياب أحدهما يعني انهيار هذا الهيكل؛ وهذا كمثل الهيكل العظمي للإنسان، حتى يظهر كاملا فيجب أن تتصل العظام -الصناديق- ببعضها البعض بطريقة صحيحة من خلال المفاصل -الخطوط-، ولن يبقى الهيكل هيكلا إذا نُزعت مفاصله، أو بعثرت عظامه وجهلنا كيف نربطها معاً!

# فائدة

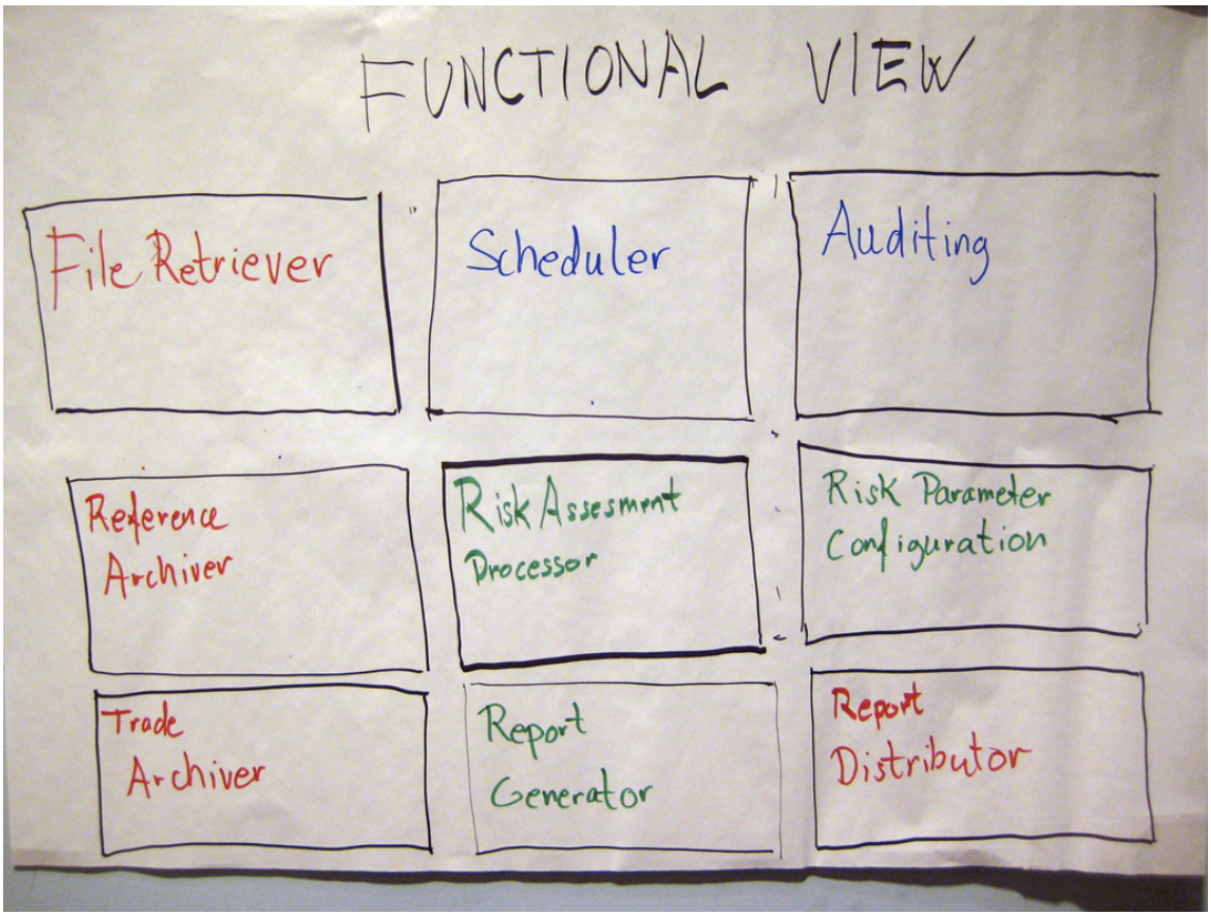
ساعات كانت لنا ذهبت، نعم ذهبت ولن تعود، انتهت الفرصة..! كلما تأمّلت في هذا المعنى تغشّاني الذهول من برودنا أمام دقّات الساعة التي لا تتوقف...

- كتاب رقائق القرآن، إبراهيم السكران -فرج الله عنه-

## المنظور / العرض الوظيفي -The Functional View:-

إن بعض المطورين يقومون بتقديم مخططهم المعماري من منظور وظيفي، أي بناء على "**المهام والأعمال**" التي يقوم بها هذا النظام، بدلاً من التركيز على هيكل النظام التقني ومكوناته...

شاهد الصورة أدناه:



في هذه الصورة تم تقسيم النظام من قبل الفريق إلى تسعة وظائف، وهو ما يطلق عليه التفكيك الوظيفي -Functional Decomposition-، وهي ممارسة تستخدم لتقسيم نظام كبير إلى وحدات أصغر بحسب الوظائف والمهام الموكلة لهذه الوحدات...

وهذا المخطط أيضا فيه المشاكل السابقة مضافا لها مشكلة جديدة ^^... فهذا النظام لا يوضح التفاعلات بين الوحدات -فقط صناديق دون خطوط-، لكن المشكلة الجديدة هنا هي الدلالات البصرية في هذا المخطط ^^...

برأيك، إلى ماذا يرمز اللون الأحمر؟ و إلى ماذا يرمز اللون الأزرق؟ إلى ماذا يرمز اللون الأخضر؟ وإلى ماذا ترمز الألوان المستخدمة عموما؟

هل ترمز إلى للمدخلات مقابل المخرجات؟

هل ترمز للتقنيات القديمة مقابل الجديدة؟

هل ترمز للأشياء التي سنبرمجها بأنفسنا مقابل التي سنشتريها جاهزة -Buy vs build-؟

أم أنها كانت مصادفة لأن كل شخص في المجموعة كان يمسك قلماً بلون مختلف فحسب؟!

إن هذه الأسئلة تجعلك توقن أن جميع ما ذكرناه ما هو إلا مجرد افتراضات، وأننا لم نحصل على فهم صحيح من الصورة التي بالأعلى، ولم تزدنا الألوان هنا إلا ضياعا وشتاتا! ثم خذ المفاجأة الكبرى ^^، ما رأيك في صندوق "Risk Assessment Processor"؟ ألا يملك border أسمك من غيره؟ ما السبب برأيك؟ يجيب Simon على هذه "ساخرا": "بصراحة لا أعرف! لكنني أظن أن الشخص الذي كان يرسم على السبورة قام بإمالة القلم قليلاً فتغير سمك الخط!" ^^... وحقيقة عندما قرأت الكتاب تفاجأت من هذه الإجابة الجميلة، لأنني

كنت أبحث حقيقة عن دلالات الألوان ودلالات الخطوط العريضة داخل هذه الصورة المشوهة ^^، هل وقعت في نفس هذا الفخ عند قرائتك لهذا الكتاب أو مشاهدة الصورة؟

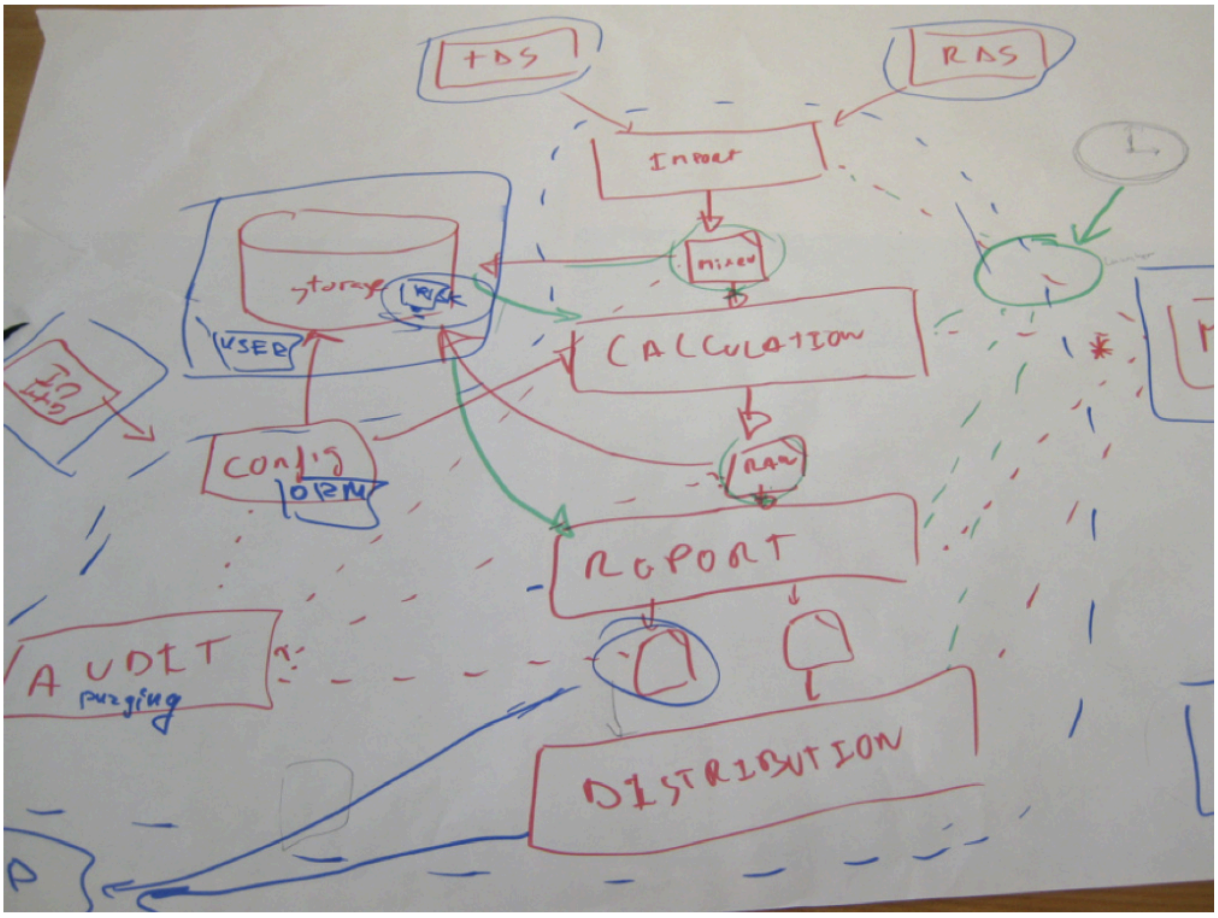
بناء على ما ذكرناه، فيمكننا القول إن الدلالات البصرية أو أي اختلاف بصري في مخطط احترافي سيفسر كرسالة للقارئ المتمعن الذي يبحث عن المفاتيح، ويبحث عن أي اختلاف! لذلك، إذا لم تقصد رسالة بصرية ما في المخطط، فلا تتركها مرئية في مخططك!

الخلاصة التي نخرج منها من هذا المثال هي:
في رسم المخططات المعمارية؛ **كل شكل، ولون، وخط يحمل دلالة ومعنى،** فإذا استخدمت ألواناً أو خطوطاً عشوائية لمجرد "الزينة" أو دون قصد؛ فإنك ستجعل المبرمجين الآخرين يضيعون ساعات في محاولة تحليل أسرار وطلاسم هذا المخطط ^^، بينما في الحقيقة.. أنت كنت فقط تمسك القلم بشكل مائل!

**خريطة الطيران أو خريطة مسارات الخطوط الجوية -The Airline Route Map:-**

هل أنت مستعد للمغامرة والذهاب في رحلة جوية؟ أظنك ستقول نعم -حتى لو قلت لا، فستصير نعم ^^- حسنا، أين خريطة المسارات التي سنستخدمها؟ ممم لم تجدها، لا عليك انظر للصورة التي في الأسفل، ثم دعنا نسافر سويا لحل ومعالجة هذه الطلاسم ^^

ملاحظة: يمثل هذا العنوان أحد أكثر الحلول فوضوية، لأن الخطوط فيه تتقاطع وتتداخل بعشوائية شديدة وكأنها خريطة معقدة لرحلات طيران تجوب العالم ^^




لنبدأ بتحليل الصورة، كما تلاحظ يظهر من المخطط أن تدفق البيانات من الأعلى (TDS, RDS) عبر سلسلة من الخطوات: استيراد ثم حساب ثم توليد التقارير ثم التوزيع كانت عملية جيدة وواضحة، وهذه نقطة إيجابية تحسب لهذا الفريق، وهذا الشكل أحد أشكال نماذج ال UML ال ١٤، وهو مخطط ال Activity Diagram، وفكرته هي إظهار ال Workflow عبر سلسلة من النشاطات -Activities-.

لكن ماذا حصل بعد ذلك؟! لقد تم تدمير كل شيء ^^... تعال معي لنحلل الكوارث التي دمرتنا:

- الكارثة الأولى: نقطة الانهيار والغموض (الجانب الأيمن من الصورة):
  - **الدائرة الخضراء الغامضة:** ستجد دائرة خضراء غامقة على يمين الصورة، وكل أسهم النظام تتجه نحو هذه الدائرة وتشير إليها، وهذا يوحي بأنها "شيء في غاية الأهمية"! ومع ذلك فلا أحد يعرف ما هي هذه الدائرة! فلا يوجد عنوان لها، ولا تسمية، ولا شرح... وهذا تكرار لخطأ الدلالات البصرية - غياب مفتاح المخطط -Legend-.

  - **الساعة الغامضة:** هناك ساعة مرسومة، مما يعني أن هناك مهاما مجدولة -Scheduled Tasks- تعمل في وقت محدد، ولكن تم تركها هكذا دون توضيح، مما يجعل كلامنا فيها مجرد استنتاجات مبنية على التخمين لا اليقين...

- الكارثة الثانية: الفوضى البصرية للجانب الأيسر:

  هنا يتجلى سبب تسمية المخطط بـ "مسارات الخطوط الجوية"، فالجانب الأيسر ليس سوى فوضى عارمة من الخطوط ذات الألوان والأنماط المختلفة (متقطعة، متصلة، متعرجة) تتقاطع وتتطاير فوق بعضها البعض بشكل يربك العين ويصعب من تتبعه...

- الكارثة الثالثة: طبخة المقلوبة اللذيذة، عذرا، أقصد ال UI المقلوبة:

إذا دققت النظر في المخطط، ستجد حروف "UI" مكتوبة بالمقلوب -رأسا على عقب-!، والسبب في ذلك أن الفريق كان يجلس على طاولة وكل واحد منهم يكتب على المخطط من زاويته أو مكانه هو! وهذا يعكس فوضى العمل لدى هذا الفريق! وهذا يدلنا على أن الفريق لم يكن لديه قائد يقود العملية أو مالك للمخطط، ولم يفكر أحد بالقارئ النهائي...

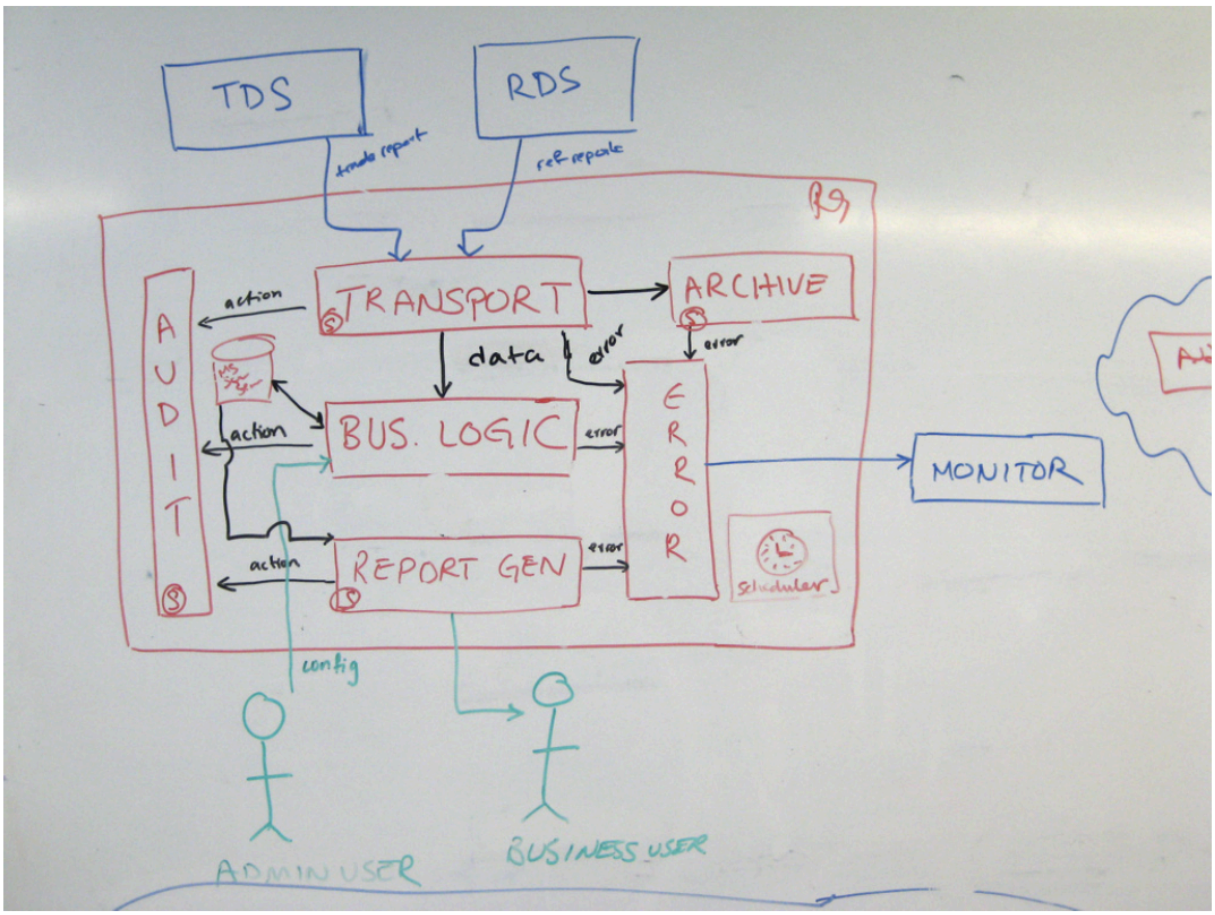
إن المخطط المعماري يجب أن يكون له "اتجاه قراءة موحد" ووجهة نظر واحدة، فكتابة الكلمات بالمقلوب تجعل قراءة المخطط عذاباً لمن لم يكن حاضراً في تلك الجلسة! ومع أن هذا الخطأ ليس بخطأ تقني، إلا أن هذا الخطأ أهمل الجانب اجتماعي، وكما ذكرنا سابقا، فالمخطط هو منتج تواصل، والمنتج بدون قائد ينتج فوضى...

الخلاصة التي نخرج منها من هذا المثال هي:

في رسم المخططات المعمارية، لا تحاول دمج "كل شيء" -تدفق البيانات، الواجهات، المهام المجدولة، والمكونات- في رسمة واحدة، فتتقاطع الخطوط وتصبح معقدة وكأنها خريطة مسارات طيران عشوائية، والأهم من ذلك: المخطط المعماري يُرسم ليقرأه شخص غائب، وليس توثيقاً لجلسة حول الطاولة، فلا تجبر المبرمجين على شقلبة شاشاتهم أو أن يلوو أعناقهم ليقرؤوا كلمة UI التي كتبتها بالمقلوب ^^، فقط لأنك كنت تجلس في الجهة المقابلة من الطاولة!

## الصحيح عموماً -Generically True:-

هناك مثل مشهور في ثقافتنا: "كلامك صح، بس شو دخلو في الموضوع!"... هذا المثل يمثل تماما خطأ يقوم به بعض المبرمجين عند محاولتهم الهروب من فخ التعقيدات والمخططات، فيرسمون مخططا معماريا عاما ومطاطيا بشكل كبير؛ بحيث يصلح لكل شيء! وهو بذلك من الناحية النظرية مخطط صحيح! لكنه لن يقدم لي أي شيء مفيد! انظر إلى الصورة التالية:




إن هذا المخطط الذي تراه أمام عينيك، يصلح لكل زمان ومكان -مجازا-! لذلك يسخر Simon من هذا النموذج بقوله: "في المرة القادمة التي يطلب منك أحدهم رسم معمارية

لنظام، أعطه هذه الصورة وتكون قد انتهيت!"... وما يقصده Simon من هذه السخرية هي الإشارة لحقيقة مهمة، وهي: **أن المخطط الذي يصلح لكل نظام لا يصف أي نظام!**

أظنك تتسائل الآن، هذا المخطط يبدو جميلا، ويحتوي في داخله صناديق وأسهم! ومع ذلك تقول أن هذا المخطط ليس مخططا جيدا!!!
أحسنت يا صديقي، استنكارك هذا منطقي جدا، لأنني في أول مرة شاهدت فيها الصورة قلت، الصورة جميلة؛ فأين المشكلة فيها!
وسأخبرك بجواب شاف لإشكالك بإذن الله،.. أولا علينا أن نتفق أن هذا النموذج من الناحية البصرية فعلا جميل وسهل، لكنه لا يتجاوز أن يكون مقدمة لمادة تدرسها في الجامعة تتحدث عن معمارية الأنظمة في أولى صفحات الكتاب ^^... وهذا سيقودنا إلى الجزء الثاني، وهو العيب الجوهري فيه، ويمكن القول أن عيوب هذا المخطط هي:

- الصناديق التي تم ذكرها داخل الإطار الأحمر تحتوي عدة مكونات مثل ال Transport وال Archive وال Audit وال Error handling، ومع أن وجود هذه الأشياء صحيح وبديهي، لكن هل يمكنك أن تخبرني عن نظام يعمل دون Error Handling؟ وهل يمكنك أن تخبرني كيف تعمل الأرشفة؟ وهل يمكنك أن تخبرني أين يتم التدقيق؟ ألا ترى أن هذه المكونات يمكن نسخها كما هي واستخدامها في نظام طبي أو تعليمي أو موقع تجارة إلكترونية؟ ما الذي يمنع ذلك؟

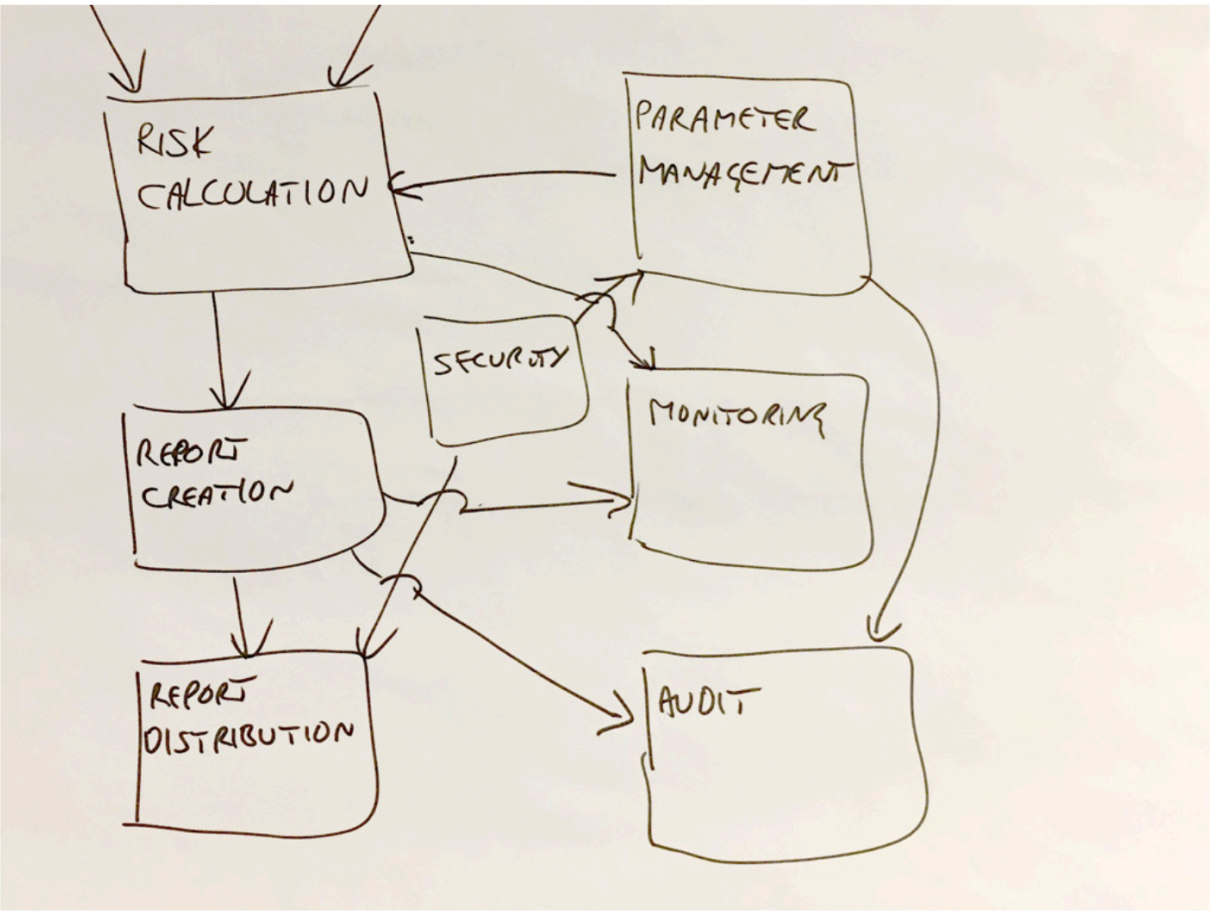
- صندوق الطامة الكبرى، هل رأيته؟ هذا الصندوق الذي كتب بداخله "Business Logic"، وهو الصندوق الذي يفترض أن يمثل المنطق الذي نحتاجه، لكنه هنا يخبرنا

فقط باسمه، وكمثال تشبيهي حتى تفهم الطامة التي في هذا الصندوق، تخيل أن أسأل طباخا عن آلية تحضيره لهذه العوامة الشهية = فيرسم صندوق بداخله نص:
"استخدمت مهاراتي في الطبخ!" يا سلام عليك يا بطل ^^... جواب هذا الطباخ لم يخبرك بشيء! وكذلك هذا الصندوق لم يخبرنا بشيء!

الخلاصة التي نخرج منها من هذا المثال هي:
المخطط المعماري الناجح يجب أن يكون "مُفصّلاً على مقاس المشروع"، ولا يجب أن يكون قالباً عاماً يصلح لأي نظام! هروبك من التفاصيل التقنية باستخدام مصطلحات فضفاضة مثل منطق العمل أو معالجة الأخطاء أو قاعدة بيانات دون توضيح ماهيتها وكيفية عملها الداخلي = يحول مخططك من "خريطة هندسية للنظام" إلى "كلام بديهي فارغ من أي قيمة عملية" ، لذلك: كلما كان عنوان الصندوق أكثر **عمومية وتجريداً**، كان أقل **فائدة** ^^.

## المنظور المنطقي -The logical view-:

إن هذا النوع من المخططات شائع بشكل كبير بين المطورين، وفكرته ببساطة هي رسم مجموعة من "المكعبات أو اللبنات الأساسية" للنظام بشكل منطقي أو مفاهيمي -Conceptual-... والمشكلة هنا أن هذا المخطط سيعرض هذه المفاهيم بشكل نظري بحت! ولن يقدم أي معلومات حقيقية أو تفاصيل تفيد المبرمج في بناء النظام فعلياً على أرض الواقع... شاهد الصورة أدناه:




يقول Simon في تعليقه على هذه الصورة -بتصرف-: "إن هذه الصورة نتاج إحدى الخرافات الكبرى، وهي إحدى المفاهيم الخاطئة والشائعة -The Common Misconception-، والسبب في انتشار هذه الخرافة هو: الخوف من التكنولوجيا!، فيعتقد المبرمجين أن المخطط المعماري يجب أن يكون منطقياً ومجردا تماما! وأنه **يُمنع منعاً باتاً** كتابة أسماء التقنيات (مثل: SQL Server، Amazon AWS) في المخطط، خاصة في المراحل الأولى وقبل كتابة الشيفرة البرمجية، فيكتفون بكتابة كلمات مجردة مثل: قاعدة بيانات، وخادم تطبيقات..." ، لذلك يرفض Simon هذه الفكرة ويطلق عليها اسم: "Misconception" والتي تعني: "فهم

خاطئ"، فالمعمارية الجيدة في نظره لا يمكن أن تحلق في سماء النظريات بعيداً عن أرض الواقع التقني.. إن المبرمج يحتاج إلى معرفة التقنيات التي سيستخدمها لكي يتخذ قرارات صحيحة!

ملاحظة: سيتم مناقشة هذه الأفكار بالتفصيل في الفصول قادمة من الكتاب بإذن الله تعالى.

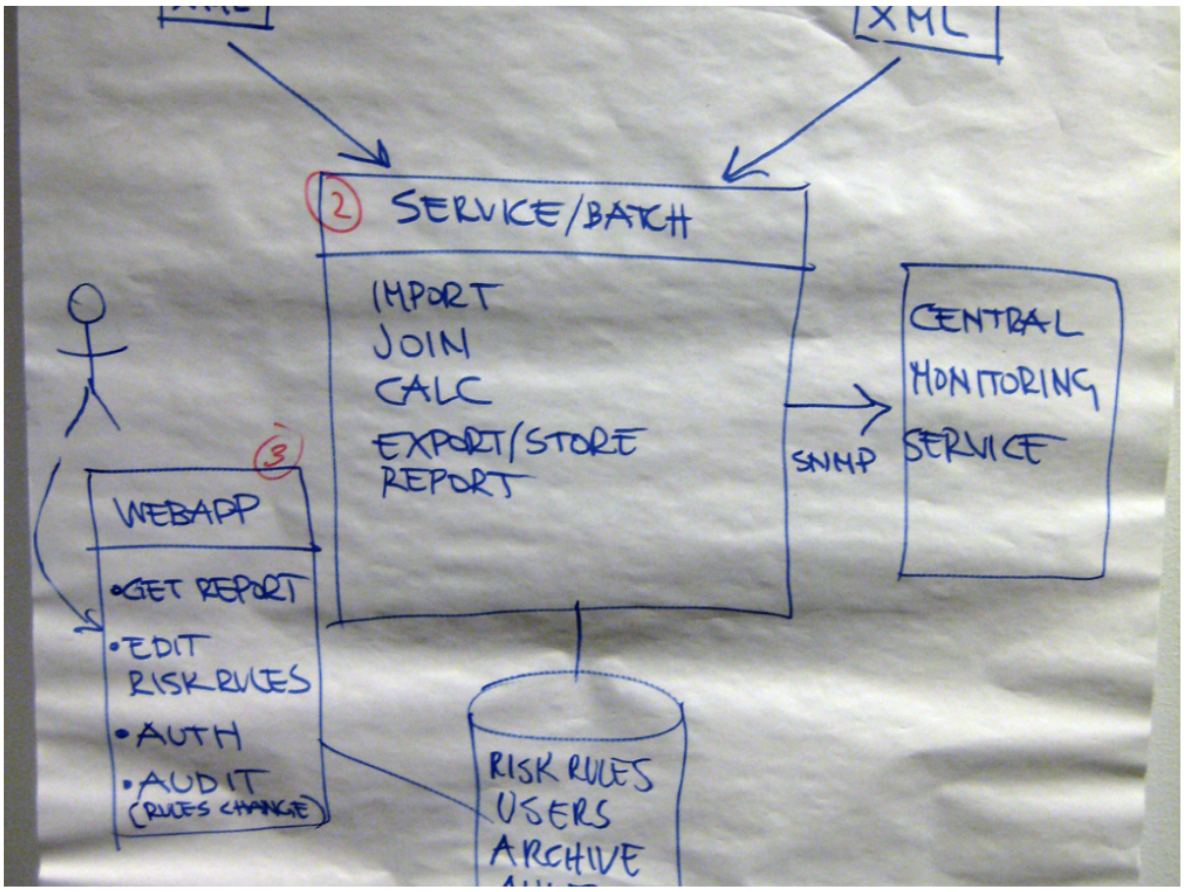
الخلاصة التي نخرج منها من هذا المثال هي:
هناك خرافة تقول بأن المعمارية البرمجية يجب أن تكون نظريّة ولا تتدخل في تفاصيل التقنيات المستخدمة، وهذا الفهم الخاطئ يُنتج لنا مخططات "منطقية" عائمة تشبه أحلام اليقظة؛ جميلة وبسيطة، لكنها لا تُنفّذ! المخطط المعماري الناجح هو الذي يدمج بين "المنطق" وبين "الواقع التقني"، فلا تخف من تسمية التقنيات وأدوات البرمجة في مخططك إذا كانت ستساعدك وتساعد فريقك على فهم كيفية بناء النظام على أرض الواقع.

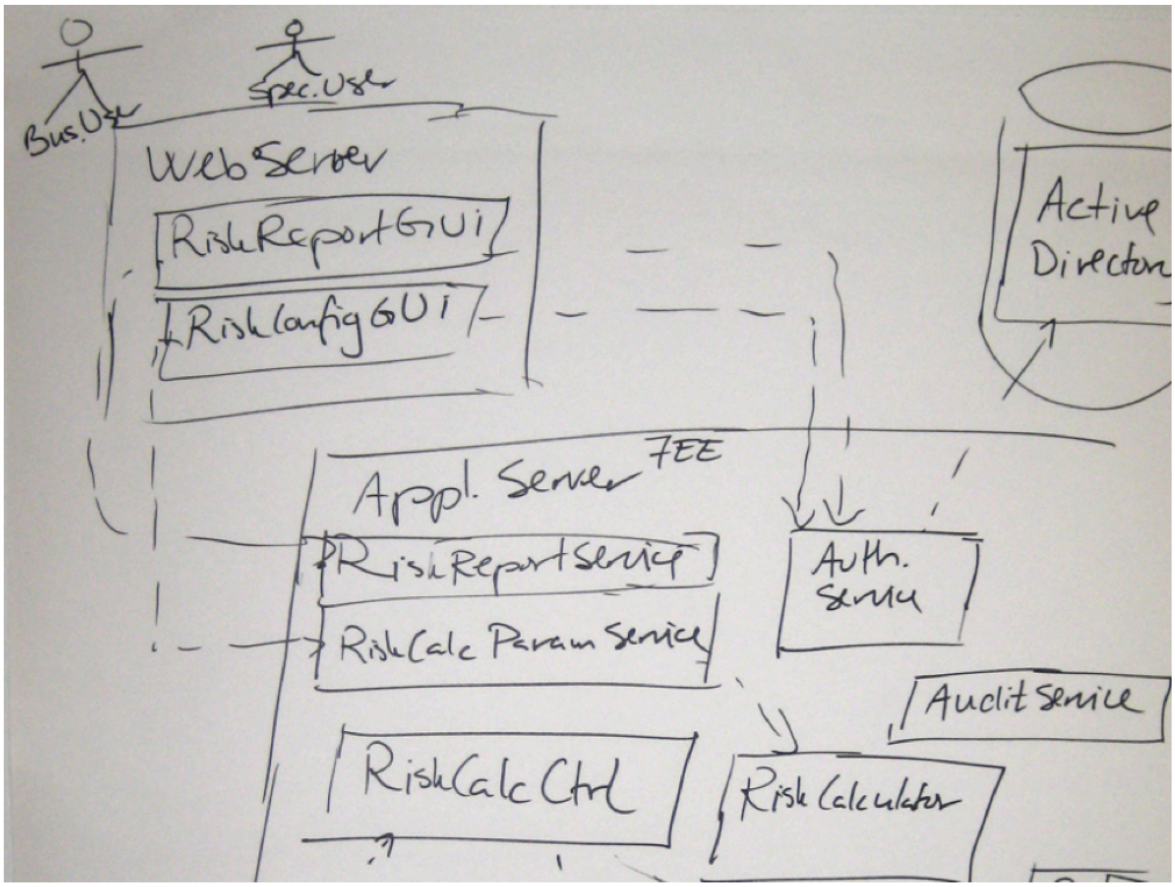
**غياب التفاصيل التقنية -Missing Technology Details-:**

لقد تكلمنا في المثال السابق عن مشكلة رسم المخطط من منظور منطقي، وعطفا على المثال السابق هناك مشكلة أخرى تظهر بعد معالجة المنظور المنطقي، وهي مشكلة غياب التفاصيل التقنية، والتي غالبا ما تكون مرتبطة "بالخطوط" التي تربط الصناديق...

ملاحظة: في المشكلة السابقة كانت تتمحور المشكلة حول التهرب من كتابة أسماء التقنيات التي سيتم استخدامها في الصناديق واستخدام الأسامي العامة بدلا من ذلك... في حين أن هذه الفقرة تتحدث غياب التفاصيل التقنية المرتبطة بالخطوط التي تربط الصناديق...

والآن شاهد الصورتين التاليتين:






في الصورة الأولى يتميز المخطط بالوضوح وتحديد المسؤوليات المُناطة لكل جزء، وهذه نقطة جيدة في المخطط... إلا أن الفريق ارتكب خطأ عندما لم يحدد اللغات التي يجب استخدامها أو التقنيات التي ينبغي علينا استخدامها... وترك هذا لخيال القارئ، وهذه مشكلة مهمة سنتحدث عنها بالتفصيل في الفصول القادمة بإذن الله.

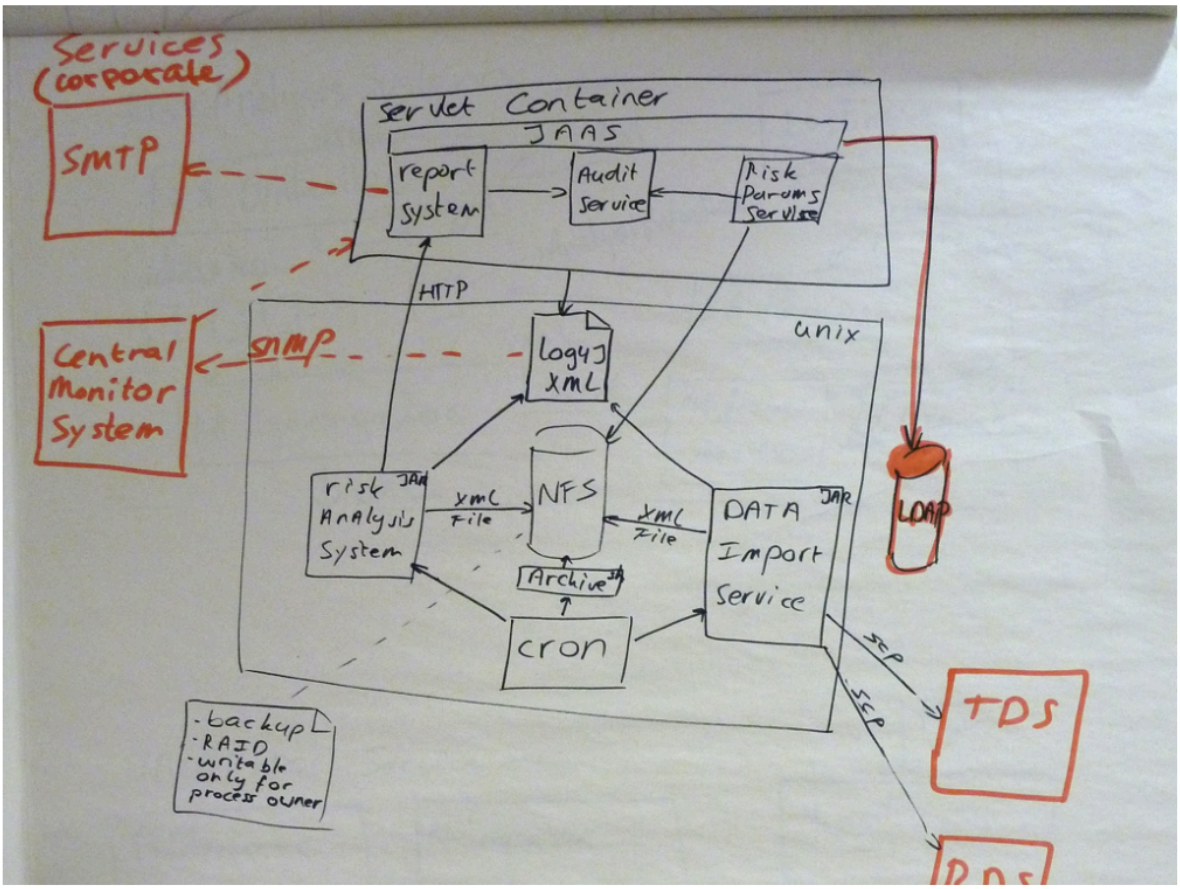
أما الصورة الثانية فقد حاول الفريق تحسين المخطط ووضعوا مجموعة من الطبقات، وتم كتابة أن اللغة المستخدمة هي JEE اختصار ل Java EE، لكن هذا المخطط وقع في مشكلة شائعة وهي أنه رسم الخطوط بين الطبقات دون شرح كيف سيتم التواصل فيما بينها... فمثلا:

المخطط رسم خطاً يربط بين ال Web Server و ال Application Server، لكن هل سيتواصلان عبر بروتوكول SOAP؟ أم عبر واجهة JSON؟ أم من خلال رسائل غير متزامنة -Asynchronous messaging-؟

الخلاصة التي نخرج منها من هذا المثال هي:
إن رسم خط بين صندوقين في المخطط المعماري هو مجرد بداية، فالخط العاري الذي لا يحمل أي تفاصيل -مثل كيفية التواصل ونوع البروتوكول- هو خط أعمى! لذلك، لا تصدق الحجة الشائعة: "النظام بسيط ويمكن بناؤه بأي تقنية" = لأن المعمارية هي فن اتخاذ القرارات، وتحديد التقنيات -مثل JSON أو XML-، بل إن تحديد طريقة نقل البيانات يعد من صلب القرارات المعمارية، وترك مثل هذه التفاصيل التقنية لخيال المبرمجين سيؤدي إلى كارثة عند التنفيذ بكل تأكيد!

## آلية النشر مقابل بيئة التشغيل -Deployment vs Execution Context-:

قد يبدو عنوان هذه المشكلة مبهما، لكن لا عليك، سنحاول تبسيطه بإذن الله ^^...
إن المراد من هذا المثال هو التركيز على أن المخطط السليم يجب أن يركز على "شكل البرنامج وهو يعمل" بدلاً من "شكل الملفات"... بينما الخطأ الشائع الذي يقع فيه المبرمجون هو: التركيز على "شكل الملفات" بدلاً من "شكل البرنامج وهو يعمل"، وأظنك تتساءل ما الفرق؟ وهل هذا الاختلاف فعلا مهم ويسبب مشكلة للمخطط؟
حسنا يا صديقي، انظر للصورة أدناه:




قبل الخوض في الصورة ستلاحظ أن هذا المخطط نوعا ما رسم بطريقة جيدة، فيه العديد السمات الجيدة، وكثير من المشاكل السابقة ليس موجودة هنا... لكن مع ذلك نجد هنا مشكلة مهمة وهي: "educated guesses" وترجمتها الحرفية: "تخمينات مدروسة أو مستنيرة"، ويقصد بهذه المشكلة أن القارئ الخبير قد يصل فعلا لتخمين صحيح للمراد، لكنه أجبر على وضع افتراضات قد تقوده لخيارات خاطئة، كما أن قارئان مختلفان قد يصلان لنتيجتين مختلفتين، وسيهدر كثير من الوقت في حل الألغاز الموجود في المخطط بدل من الاستفادة منه، وهنا فقدنا واحدة من أهم النقاط التي تشير لمخطط سليم وهي قدرة المخطط على

"التواصل"... إن المشكلة الأساسية لدينا في هذا المخطط والتي دعتنا للتخمين هي وجود كلمة JAR في المخطط، وجود هذه الكلمة أدى لظهور المشكلة الشائعة التي وضع هذا المثال لأجلها: وهي مشكلة الالتباس بين "النشر" و"التنفيذ"...

دعونا نفكك المشكلة خطوة بخطوة:

إذا نظرنا للصورة سنجد أن المبرمجين قاموا برسم خادم Unix، ووضعوا بداخله صندوقين، وكتبوا فوق كل صندوق كلمة **(JAR)**، وذلك هم أعلنوا عن "آلية النشر -Deployment"؛ أي أنهم أخبرونا أنهم سيقومون بضغط الشيفرة البرمجية المكتوبة بلغة الجافا في ملف اسمه (JAR) وسيرسلونه إلى الخادم... لكن، هنا تكمن المشكلة! فالمعمارية يجب أن تصف النظام وهو **"يعمل وينبض بالحياة"**، وليس وهو مجرد ملف مخزن لا حياة فيه!

إن ملف الـ JAR أو ملف الـ DLL في ال .NET...إلخ ليسو سوى مخازن أو صناديق موجودة على الخادم، هذه المخازن أو الصناديق لن تعمل هكذا من تلقاء نفسها! جرب أن ترمي ملف JAR على جهازك، هل سيتغير شيء؟ أم سيظل موجودا عندك كقطعة جماد تشغل مساحة من ذاكرة التخزين فقط؟

حسنا، بناء على التحليل السابق يستحيل أن يكون ملف ال JAR الموجود بالمخطط هو مجرد ملف مخزن على الخادم، فإننا سننتقل إلى مرحلة ال Execution Context -سياق بيئة التشغيل / ما يحدث في بيئة التشغيل-، أي أننا سنبدأ في مرحلة التخمين حتى نصل للمراد منها، وهنا يمكننا القول بأن هذا الملف حسب السياق يستحيل أن يكون مجرد ملف مخزن، ويستحيل أن يقوم الخادم بتشغيل ال JAR بشكل تلقائي؛ دون أن يكون هذا الملف قد

كتب وبرمج ليكون Standalone Application -تطبيقا مستقلا- يحتوي على أمر تشغيل ذاتي، ويتم جدولة هذه التطبيقات لتعمل في وقت محدد عبر ال cron الموجود على الخادم...

ممتاز، أعتقد الآن أن الفكرة صارت واضحة ومباشرة ^^... إن النتيجة التي وصلنا لها بعد التخمين نتيجة منطقية، لكن المشكلة التي نريد حلها من الأساس هي هذه:
لماذا جعلتني أُخمّن؟ لماذا رسمت لي ملف JAR بدلا من رسم صندوق مكتوب عليه "تطبيق مستقل يعمل في الخلفية"؟!
لذلك، في معمارية البرمجيات: لا ترسم صناديق ميتة لا حياة فيها مثل: ال (/ JAR / DLL ZIP)، بل ارسم "البرنامج بعد أن يتم إخراجه من الصناديق الميتة ومن ثم تشغيلها" مثل: تطبيق ويب أو واجهة مستخدم أو خدمة تعمل بشكل خفي...

ملاحظة: أظنك رأيت أنني أسهبت في شرح هذا المثال وهذه المشكلة، وتعمدت تكرار بعض الفقرات والأمثلة بأشكال مختلفة حتى تصل الفكرة بأفضل صورة... إن المشكلة الكبيرة هنا تكمن في ضبابية أو صعوبة اكتشاف هذه المشكلة من الوهلة الأولى، ويظن المبرمجون عادة أنهم كتبوا كل شيء بشكل صحيح... لذلك، ولننزع عنا هذه الضبابية، وحتى لا نقع في هذا الخطأ، فمنا تبسيطها وتفكيكها قدر الإمكان... فالحمد لله.

الخلاصة التي نخرج منها من هذا المثال هي:
المخطط المعماري ليس فاتورة استلام! لا تكتب لي الامتدادات وطريقة ضغط الملفات (JAR, ZIP, DLL) ثم تظن أنك رسمت معمارية! بل يجب أن تكتب لي **وظيفتها وحالتها**

**أثناء التشغيل،** مثل: تطبيق console مستقل، أو خدمة تعمل في ال Background Service، أو تطبيق ويب...إلخ! إن المبرمج الذي يقرأ المخطط يريد أن يرى النظام "وهو ينبض ويعمل"، وليس وهو "مضغوط ومخزن" داخل الخادم!

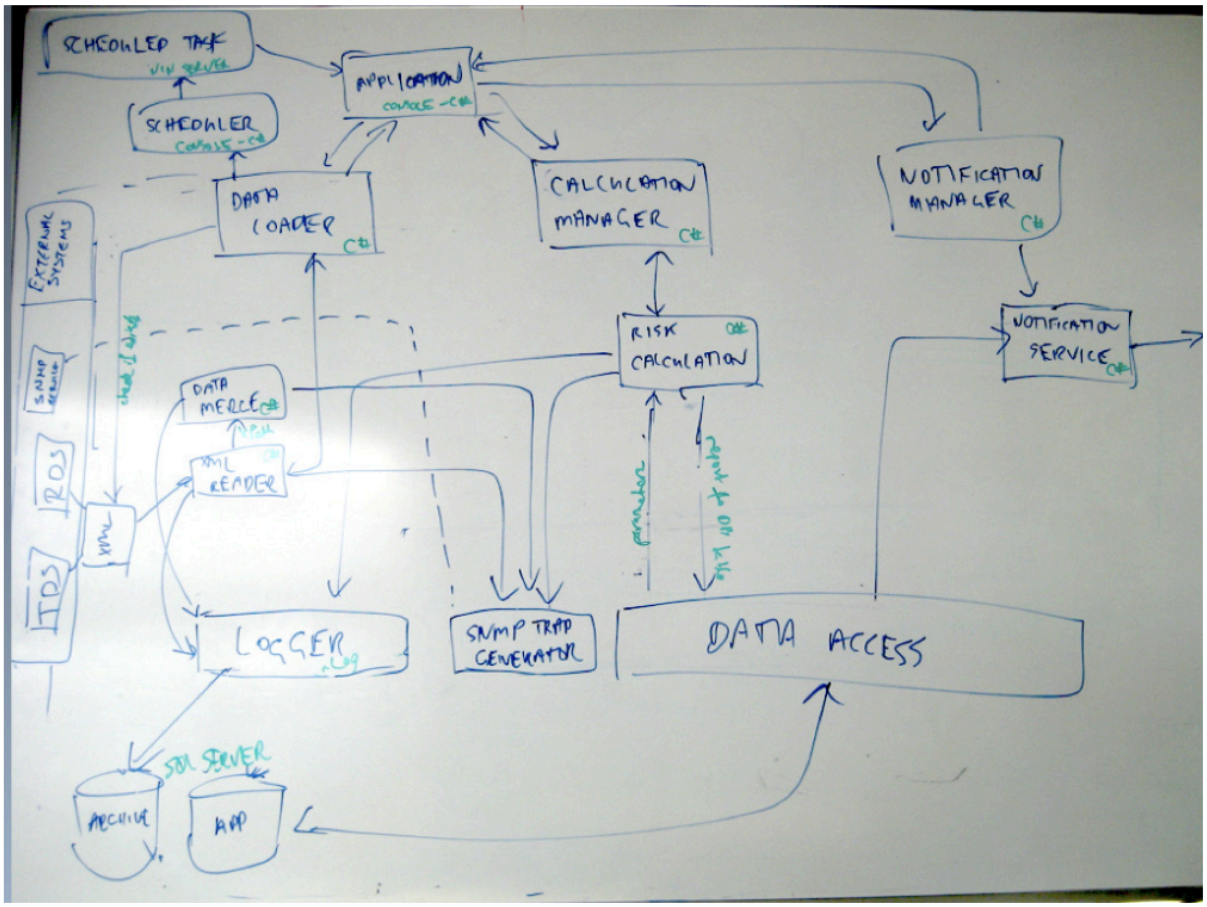
## كائنات سي شارب المشرّدة -Homeless Old C# Object (HOCO)-:

في عالم البرمجة الكينونية -OOP- هناك مصطلحات مشهورة مثل POCO و POJO، وهذه المصطلحات تشير إلى كائن من Class برمجي عادي، يُستخدم غالباً لتمثيل البيانات، ويشترط ليكون POCO أن لا يرتبط هذا ال Class مع أي مكتبات خارجية -Frameworks- وألا يرث -Inherit- من كائنات معقدة تفرضها بيئة العمل... وستجده يحتوي فقط على المتغيرات والخصائص الأساسية مثل الاسم، العمر، الوزن...إلخ، وربما بعض العمليات المنطقية البسيطة جداً... ومن أهم مزايا هذا النوع هو الاستقلالية، فهو يمثل شيفرة برمجية نظيفة لا تعتمد على مكتبات ضخمة، فلو غيرت التقنية المستخدمة لقاعدة البيانات، فلن تضطر لتغيير كائنات الـ POCO الخاصة بك... ببساطة: إذا قال لك مبرمج ما: "هذا مجرد POCO"، فهو يقصد أن يقول لك: "لا تقلق، هذا مجرد كائن برمجي بسيط ونظيف لتخزين البيانات، وليس مرتبطا بأي تعقيدات أو تقنيات خارجية"، وحتى لا تصدم، فإن هذا ال POCO يعد أحد أهم المبادئ الهندسية التي تقيس كفاءة الشيفرة البرمجية والمعمارية -Clean Architecture-... وال POJO هو المصطلح المقابل في لغة جافا.

وحتى يكون الأمر ممتعا، لنعود في التاريخ قليلا للوراء: في حوالي ال ٢٠٠٠م لاحظ كبار المبرمجين ازدياد التعقيد الخاص بالكائنات وازدياد ارتباطاتها، وحاولوا توعية المبرمجين

لضرورة جعل الكائنات بسيطة، فقام مارتن فاولر ورفقاه بإطلاق اسم POJO ليكون اسما جذابا للمبرمجين، وفعلا نجحت هذه الحركة في جذب المبرمجين لتطبيقها -بالمناسبة واحدة من أكثر مهارات مارتن هي إطلاق المسميات على الأشياء-... وبعد ال POJO أصبحت تشاهد POCO إشارة لنفس المفهوم لكن في لغة C# وممكن POPO إذا كانت PHP or Python ^^...

بعذ هذه المقدمة المملة -.-، لنعد لموضوعنا ^^، لقد وضع Simon عنوانا ساخرا لهذا المثال فاستبدل ال P ب H، وذلك لتصبح العبارة تدل على الكائنات المشردة بدلا من الكائنات البسيطة ^^، وهو بذلك يرمي بطريقة فكاهية لوجود شيء مشرد في المخطط بلا component أو context أو تموضع حقيقي في النظام ^^... أي أن المبرمجين الذين رسموا هذا المخطط الذي في الأسفل تركوا بعض المكونات في العراء دون أن يضعوها داخل إطار يوضح أين تعيش وكيف تعمل!



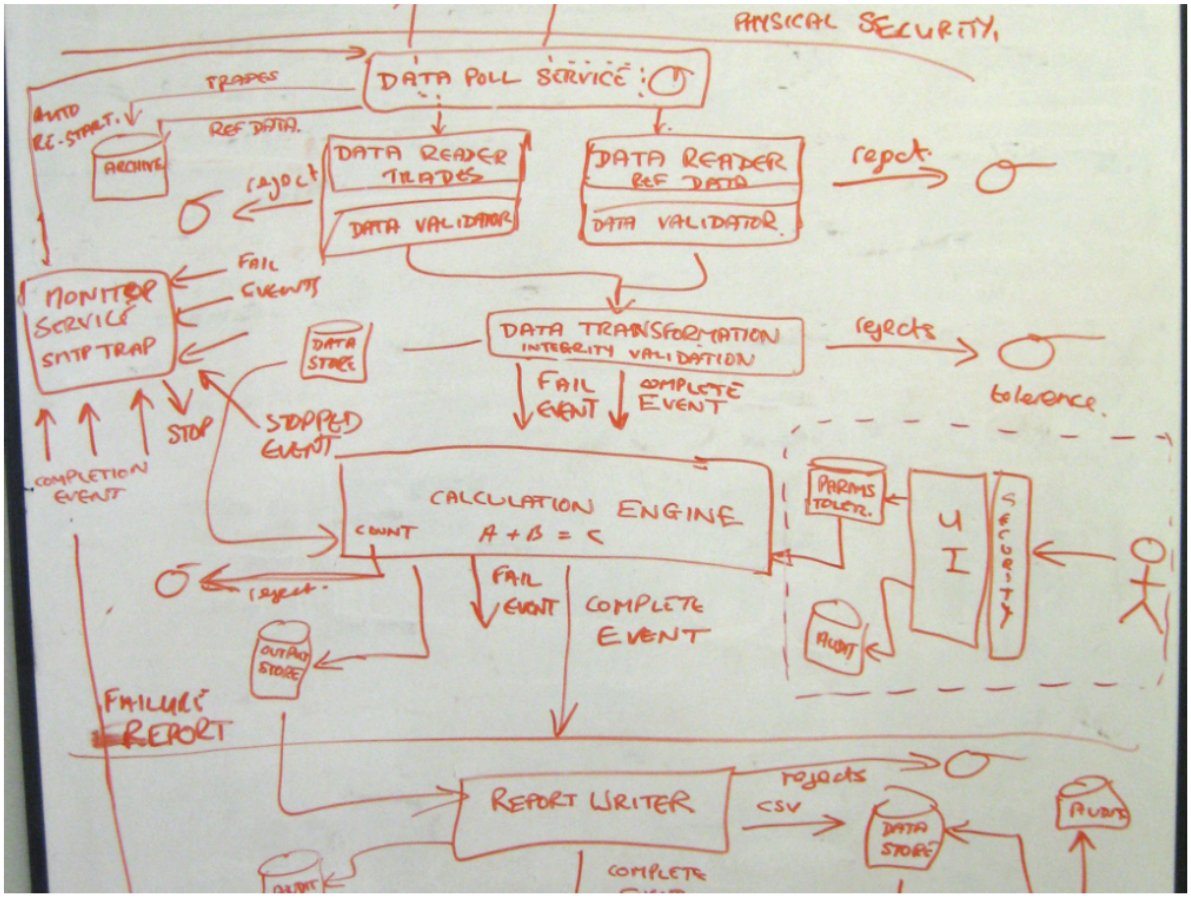

إذا أمعنت النظر في الصورة ستجد هناك مستويات مختلفة من التفصيل:

- فلدينا مستوى يعرض Console-C# Application وقاعدة البيانات ضخمة على خادم -SQL SERVER-... وهذا تصرف منطقي وممتاز.
- وفي نفس الرسمة هناك مربعات صغيرة تمثل ال Data Access، وال Logger...، وهنا تظهر لدينا أزمة حدودية ^^، أي أزمة في معرفة حدود هذه المربعات، فمثلا هل ال Logger تطبيق مستقل بذاته يقف فعلا جنبا إلى جنب مع ال Console App وقاعدة البيانات؟ أم أنها مجرد أجزاء صغيرة من ال Console App -تتبع أو تقع داخل ال console app-؟ بالتحليل العميق للصورة ستكتشف أنها أجزاء صغيرة

داخل ال console app! وكان يمكن حل هذه المشكلة ببساطة من خلال رسم إطار يضع كل هذه المكونات داخل صندوق مكتوب على رأسه Console App ^^...
هل أدركت الآن لماذا سميت مشردة؟ لأنها رميت في المخطط دون بيت أو مأوى ودون أن تتبع لحاوية -Container-، وهذه الفكرة ستكون إحدى الأفكار المهمة التي سنعتمد عليها في ال C4...

الخلاصة التي نخرج منها من هذا المثال هي:
في المخطط المعماري، لا تخلط مكونات النظام الداخلية مع التطبيقات الخارجية والأنظمة في نفس المستوى، ودون استخدام "**للحدود -Boundaries-**"، لذلك، إذا قمت بتفكيك تطبيق ما إلى مكونات صغيرة حتى تشرحها، فارسم صندوقا كبيرا يحيط بها لتعطي لهذه المكونات منزلا تعيش فيه، بدلا من تركها تتسكع كمكونات مشردة تربك من يقرأ المخطط ^^

**اختر مغامرتك الخاصة -Choose your own adventure:-**




يعد هذا المثال أحد المخططات المزعجة والمعقدة والتي يصعب تحليلها وقراءتها! مع أن هذه الصورة تظهر فقط منتصف المخطط فقط وهناك أجزاء خارجية أخرى! وسبب التعقيد هنا هو محاولة وضع كل المسارات والخطوات والخيارات وال events في مكان واحد حتى تصبح وكأنها متاهة... هذه المتاهة هي سبب تسمية هذه المشكلة باختر مغامرتك الخاصة، وهذا الاسم مستوحى من نوع من أنواع كتب قصص الأطفال التي كانت تمثل مغامرة يسير فيها الطفل، فالطفل لا يقرأ القصة بتسلسل عادي، بل يصل إلى نقطة يخيره فيها الكتاب -مثلا-: "إذا أردت مهاجمة الوحش المخيف، اذهب إلى الصفحة 47 أما إذا أردت الهرب

كالجبناء فاذهب إلى الصفحة 205"... وهكذا تستمر القرارات والقفزات حتى الفوز أو أن يموت البطل وتضطر للبدء من جديد ^^

إن هذا المخطط يشبه قصص الأطفال التي ذكرناها، فأنت ستعيش في متاهة في داخلها، وستستمر بالقفز هنا وهناك، ستبدأ من أعلى المخطط، ثم ستسير في التعرج نزولا عبر معمارية معقدة جدا تعتمد على مجموعة من ال Asynchronous events، وطوال فترة التتبع ستقع أمام مفترق طرق لاختيار لمغامرتك، أستذهب مع Fail event أم مع Complete event؟ ثم بعد هذا الجهد الجهيد، ستقع بالفخ! وسيموت البطل ^^... والفخ الذي ستقع فيه هو: "فخ الـ SNMP"، فكما ترى في المخطط، كل الطرق تؤدي إلى ال SNMP.

ملاحظة: لاحظ في المخطط تكرار كلمة Event على الخطوط، مثل: Fail Event, Complete Event, Stop Event... إلخ، وهذا يكشف لنا أن معمارية هذا النظام تعتمد على منهجية مبنية على الأحداث -Event-Driven Architecture-، وهذا المصطلح يشير إلى المعمارية التي تتواصل فيها المكونات مع بعضها من خلال إرسال واستقبال الأحداث بدلاً من الاستدعاء المباشر، وآلية عملها بسيطة ^^: يُرسل مُكون ما حدثا عبر ناقل الأحداث -Event Bus-، وتقوم المكونات الأخرى المهتمة بهذا الحدث بالتقاطه والتعامل معه بحسب وظيفتها.

بناء على ما ذكرناه فيمكن تلخيص مشكلة هذا المخطط بنقطتين:

- المخطط يحاول إظهار -حشر- كل التفاصيل والمسارات في رسمة واحدة، مما جعله معقدا جدا للعين، لذلك اعلم أن المخطط الجيد ليس الذي يحتوي على كل شيء، بل الذي **يختار** ما يُظهره **بحكمة**، ويمكن حل هذه المشكلة بأكثر من طريقة من إزالة بعض المعلومات غير الضرورية والاقتصار على المهم منها، ومنها تجزئة المخطط إلى مخططات أصغر، وهذه العملية ما يطلق عليها بالتصفية -Filter-.
- من الناحية البصرية فاستخدام لون واحد -البرتقالي- لجميع المسارات والخطوط يجعل من عملية التتبع صعبة، فلا يمكن معرفة أو تتبع مسارات الفشل بسهولة... وحل هذه المشكلة سهل أيضا، فكل ما عليك القيام به هو استخدام الألوان بذكاء؛ لتسليط الضوء على المسارات المختلفة (مثلا: خطوط حمراء لمسار الأخطاء)، وهذا سيجعل القارئ يتتبع المغامرة بوضوح دون أن يقع في الفخ!

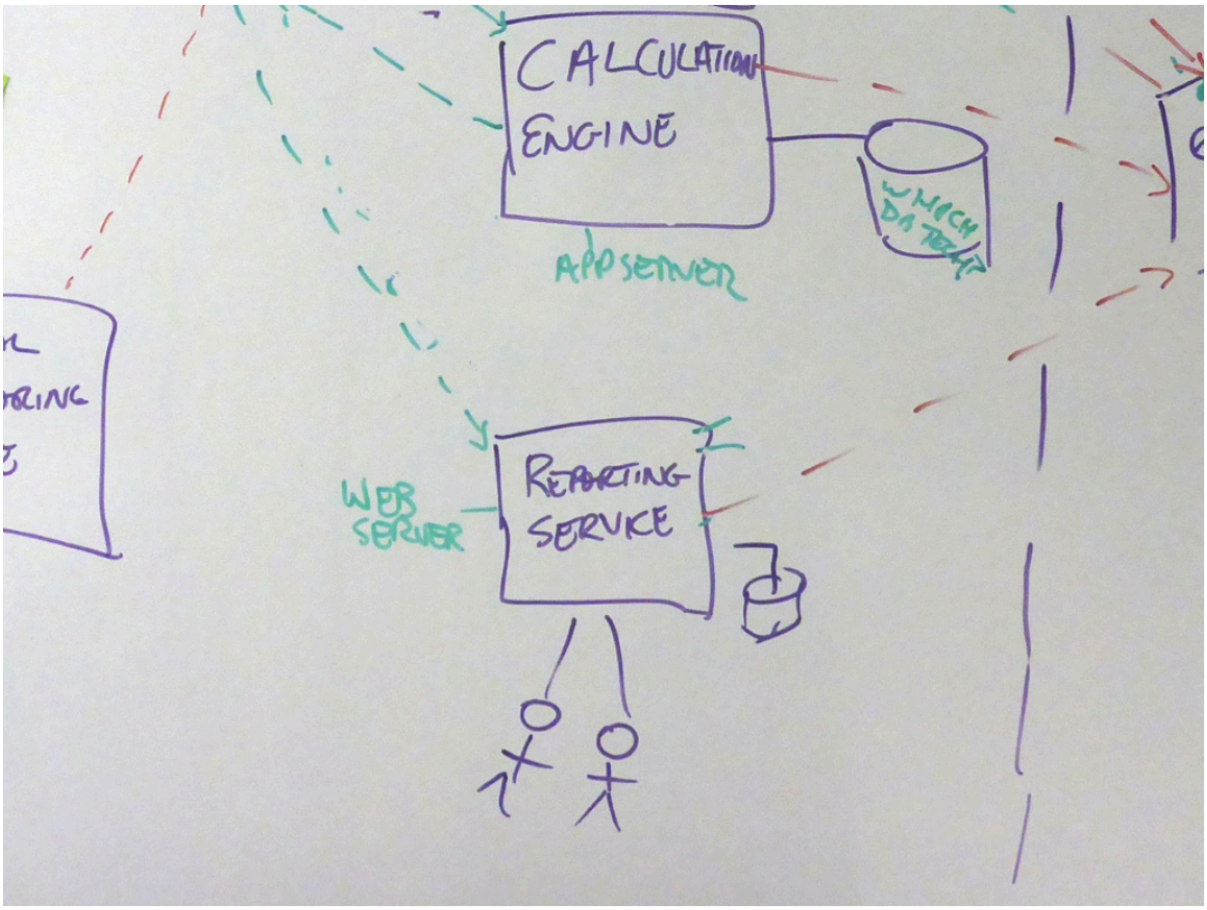
الخلاصة التي نخرج منها من هذا المثال هي:

إن **المخطط المعماري وُجد ليختصر المسافات، لا ليصنع المتاهات!** لذلك، إذا حاولت حشر كل الأحداث، والمسارات، والاحتمالات المعقدة -في الأنظمة المعتمدة على الأحداث Event-Driven- داخل رسمة واحدة وبلون موحد = فإن النتيجة هي "لغزاً بصرياً" يشبه ألعاب المتاهات! لذلك، يمكن استخدام **القاعدة البغيضة التي استعملها العدو المحتل ضد أمتنا الإسلامية: (فَرّق تَسُد) ولوّن تُرشِد**، وبهذا إذا كان نظامك متشعبا؛ فلا تضع كل الخيارات في سلة واحدة، بل قم بتفكيك المخطط الضخم إلى عدة مخططات مبسطة (مثلاً: مخطط مستقل لمسار العمليات الناجحة، ومخطط آخر لمسار معالجة الأخطاء)، والأهم من ذلك:

وظّف "الترميز اللوني -Color Coding-" بذكاء لتوجيه عين القارئ بسلاسة، **فالمخطط الناجح هو الذي يمسك بيد المبرمج ليرشده للحل، وليس المخطط الذي يتركه يتخبط في مسارات لا نهائية حتى يقع في الفخ!**

**المقاتلون النمطيون، جنود بلا ملامح -Stormtroopers-:**

**يعد هذا الخطأ واحد من أكثر الأخطاء التي يقع فيها المطورون في أول أيام تعلم الرسم المعماري، هذا الخطأ هو رسم صور أشخاص دون تفاصيل مفيدة؛ كما في الصورة أدناه:**



إن المقاتلين الذين يرتدون ملابس موحدة، ودروع موحدة، وأقنعة موحدة، سيكونون نسخ متطابقة مجهولة الهوية، بلا ملامح! وهذه المشكلة ببساطة تترجم إلى: "من أنت وماذا تريد؟" ^^، فكما في الصورة أعلاه، ماذا يعني وجود شخصين فيها؟ وإلى ماذا يشير كل واحد منهما؟ ماذا لو قمنا بوضع كلمة user أسفل كل شخص، فهل سيحدث فرقا؟
إن إجابة هذه الأسئلة هي: لا أعلم ولا أعلم ولن يحدث فرقا فال user لم يعبر عن شيء!

إن وجود شخصية مجهولة الهوية في المخطط المعماري يعادل وجود "شبح" مخيف بداخله ^^، لذلك، فإن المخطط الجيد يجب أن يجيب على سؤالين أساسيين حول أي Actor يتفاعل مع النظام:

- من هو هذا ال Actor؟ هل هو مدير النظام؟ أم عميل مسجل لدينا؟ أم زائر مجهول؟ أم موظف خدمة العملاء؟
- لماذا يستخدم هذا ال Actor النظام؟ هل دخل ليقرأ تقريراً؟ أم ليعدّل بيانات حساسة؟ أم ليشتري منتجاً؟

وهذا الكلام يمهد لمفهوم جوهري في نموذج ال C4: = كل شخص -Person- في المخطط يجب أن يكون له دور -مثل Customer- ووصف -A customer of the bank-.

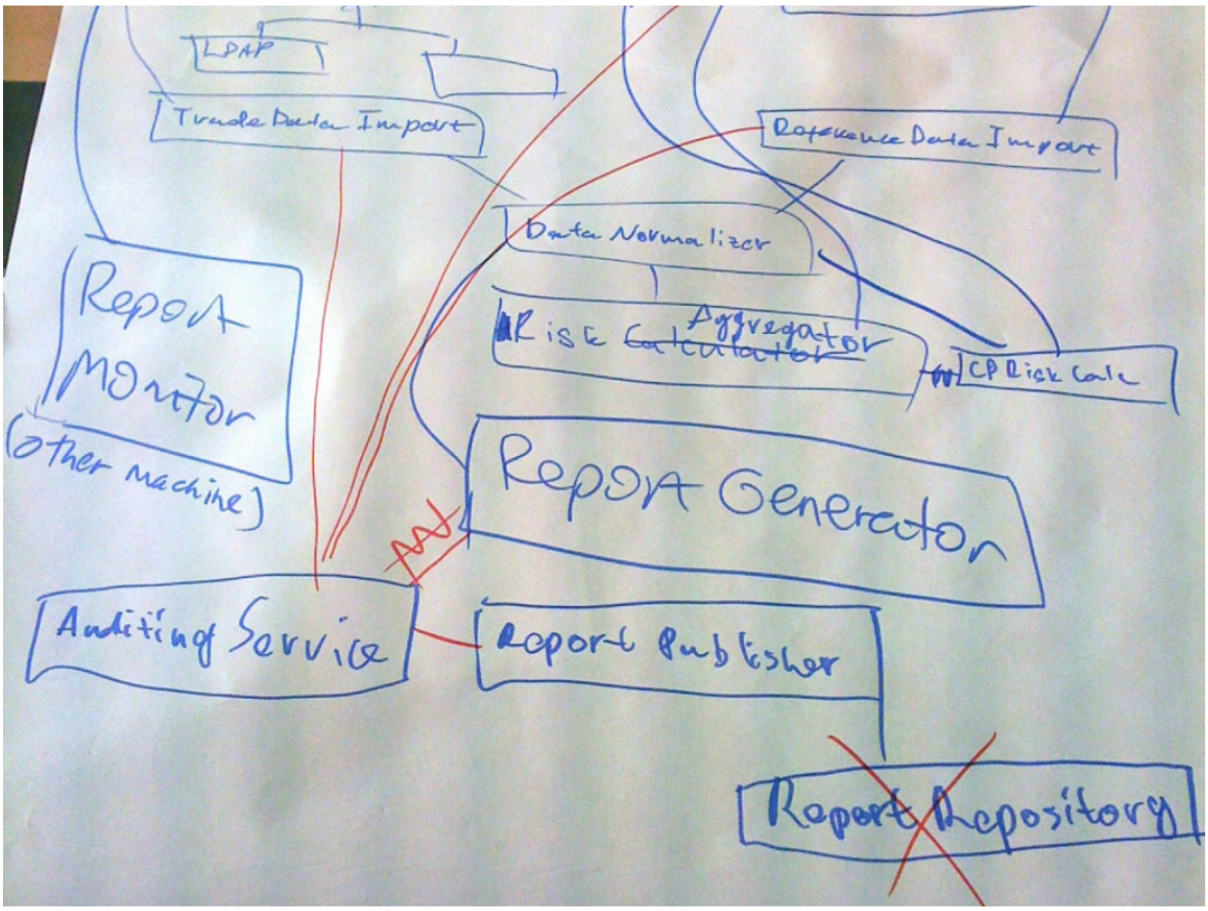
الخلاصة التي نخرج منها من هذا المثال هي:
إن المستخدمون في نظامك ليسوا مجرد "جنود نمطيون" مجهولو الهوية، ورسم أيقونة شخص فارغة وكتابة كلمة "مستخدم" تحتها هو هروب من تحديد المتطلبات الحقيقية للمعمارية! فكل مستخدم في المخطط يجب أن يحمل "اسماً وصفة"، مثل: عميل، ومحاسب، ومدير نظام، ويجب

أن يُرافق هذا الاسم وصف موجز لـ "هدفه" من استخدام النظام (لماذا هو هنا؟)، فإذا لم تكن تعرف من هو المستخدم وماذا يريد، فمن الأفضل ألا ترسمه من الأساس حتى تعلم!

**Should have used a- كان ينبغي علينا استخدام اللوح الأبيض whiteboard!-:**

هنا سنوجه الانتقاد لمكان رسم المخطط متجاوزين المخطط نفسه ^^، والمشكلة باختصار هي: "الرسم على الورق" إذا توفر لديك لوح أبيض! ويمكن صياغة العنوان بطريقة أخرى: "نخ الأدوات الورقية والرقمية.. وتميز اللوح الأبيض!"

شاهد الصورة: التي تمثل رسما رُسم على ورق أزرق:




إن استخدام اللوح يهب لنا ميزات مهمة، مثل: سهولة التعديل عند حدوث أي خطأ أثناء الرسم، كما يمكننا من التجريب والإصلاح، والعمل مع الفريق؛ مما يعطي مرونة في الإصلاح والرغبة في ذلك عند اكتشاف الخطأ، ويمنع تراكم الأخطاء التي تم شطبها على الورقة... إلخ

وهذا الكلام لا ينطبق فقط على الورق، بل ينطبق أيضا على بعض الأدوات الرقمية التي يمكننا أن نستخدمها لرسم المعمارية، فبعض الأدوات بسبب إمكانياتها العالية ستتيح لنا رسم الكثير من المربعات، وستسمح لنا بإضافة الكثير من التفاصيل؛ مما قد يجعل المخطط معقدا ومليء بأمور لا حاجة لنا لها... لذلك يفضل كثير من المعماريون استخدام اللوح للمزايا التي

تحدثنا عنها سابقا، مضافا لها مزية مهمة وهي محدودية مساحة اللوح -مساحة فيزيائية ثابتة-، مما سيجبرك على التجريد -Abstraction- والتركيز على الصورة الكبيرة والمكونات الأساسية، وسيمنعك من الغوص في التفاصيل الدقيقة التي ستعقد المخطط دون داع...

الخلاصة التي نخرج منها من هذا المثال هي:

إن من أكثر الفخاخ التي نقع فيها كثيراً **كمطورين** للبرمجيات؛ **هو** الاستخدام المفرط لبرامج الرسم الرقمية في المراحل الأولى من تصميم المعماري، لأن هذه البرامج تمنحك مساحة لا نهائية تغريك بحشر كل التفاصيل الممكنة أثناء الرسم؛ حتى يتحول مخططك إلى "شبكة عنكبوتية" غير مقروءة! في المقابل، فإن استخدام "الورق" مع الفريق في أولى مراحل التصميم سيصعب من مهمتنا في التعديل، والتنقيح، والإصلاح، والتجريب.

**ولهذا السبب**، فإن أفضل صديق للمعماري في بدايات التصميم هو "اللوح الأبيض"، فمساحة اللوح المحدودة ستجبرك على تجاهل التفاصيل الدقيقة والتركيز على الصورة الكبيرة والمكونات الأساسية للنظام، **ونظراً لطبيعة اللوح**، سيسهل علينا التعديل، والإصلاح، والمشاركة الفعالة مع الفريق.. وخلاصة القول هي: استخدم اللوح للتفكير والنقاش، واترك برامج الرسم الرقمية لمرحلة "التوثيق النهائي"!

# فائدة

لازلنا الآن في الساعات الأخيرة التي تسبق إغلاق باب التوبة، والتوبة إلى الله تحتاج قراراً فورياً عاجلاً، قراراً لا يحتمل التأجيل ثانية واحدة، قراراً يجب أن يدشن الآن قبل أن تفوت الفرصة...

- كتاب رقائق القرآن، إبراهيم السكران -فرج الله عنه-

## المشكلات الشائعة - خلاصة المشاكل التي تعلمناها من الأمثلة

ستقوم بتقسيم المشكلات التي تعلمناها وسنقسمها إلى جزئين:

أولا: المشكلات المتعلقة بمخطط واحد

- **الرموز المبهمة -Notation-:** استخدام الألوان والأشكال هندسية دون وجود مفتاح -Legend- يشرح دلالات هذه الرموز أو الألوان، أو استخدام الرموز والألوان بشكل عشوائي غير متسق.
- **الغموض -Ambiguity-:** غموض بعض المربعات والمكونات، أي أجزاء غير مفهومة.
- **الروابط الضائعة -Relationships-:** الخطوط التي تربط بين المكونات إما غائبة تماماً، أو موجودة لكنها لا تشرح "كيف" و"لماذا" تتواصل هذه المكونات فيما بينها.
- **المصطلحات الفضفاضة -Generic terms-:** استخدام كلمات عامة جداً مثل "منطق العمل -Business logic-" بدلاً من تسمية الأشياء بمسمياتها الدقيقة التي تعكس وظيفتها.
- **غياب الهوية التقنية -Technology choices-:** إخفاء أسماء التقنيات المستخدمة مثل إخفاء نوع قاعدة البيانات أو لغة البرمجة.
- **الخلط بين مستويات التجريد -Mixed levels of abstraction-:** وضع أنظمة ضخمة جداً جنباً إلى جنب مع خدمات صغيرة في نفس الرسمة ودون إطار يضمها -كما رأينا في مثال ال HOCO-.

- **التفاصيل المفرطة أو الشحيحة:** إما تفاصيل كثيرة تصنع "متاهة"، أو تفاصيل قليلة جداً لا تفيد القارئ.
- **غياب نقطة الانطلاق -No context-:** المخطط يجعلك تائهاً لا تعرف من أين تبدأ القراءة، ولا تفهم السياق العام للنظام.

ثانيا: المشكلات المتعلقة بوجود أكثر من مخطط

لقد تحدثنا عن المشاكل التي صادفتها وكانت جميعها لعرض مشاكل المخطط الواحد، لذلك، إذا كان وجود مخطط واحد سيئ سيسبب مشكلة من التي ذكرناها، فإن **مجموعة من المخططات السيئة ستضاعف الأخطاء الممكنة مما سيضاعف الكارثة!** وعندما يقوم المبرمج برسم عدة مخططات لنفس النظام، فهنا ستظهر بعض المشاكل الإضافية التي يجب علينا مراعاتها:

- **غياب الترميز القياسي:** تكمن المشكلة هنا بتغيير الرموز والألوان من مخطط لآخر، مثل أن يتم اعتماد اللون الأحمر في المخطط الأول ليمثل "قاعدة بيانات"، وفي المخطط الثاني ليمثل "الأخطاء"!
- **تسميات غير متسقة:** تسمية نفس المكون بأسماء مختلفة بين المخططات، مما يشتت ذهن القارئ.. مثل: Risk Calculator و Risk Calc و Calculation Engine
- **التسلسل المفقود:** غياب الترتيب المنطقي لقراءة المخططات مثل أي مخطط سأقرأ أولاً؟ هل أبدأ بالصورة العامة أم بالتفاصيل؟
- **الانتقال غير المفهوم:** لا يوجد آلية واضحة للتنقل بين مخطط وآخر، ولن نفهم كيف يرتبط هذا المخطط بالمخطط الذي يليه.

ملاحظة: نعيد التذكير بأن المشاكل التي ذكرناها لا تتعلق بك، فلا تقلق! فهذه ليست مجرد أخطاء فردية، بل أخطاء شائعة؛ فبحث سريع في صور جوجل سيكشف لك أن هذا الوضع الكارثي هو الوضع السائد في عالمنا اليوم! ونحن جئنا هنا للحصول على الترياق ^^.

والآن، وقبل أن ننتقل إلى الحلول، علينا أن نحذر من أحد أهم وأشهر الأفخاخ التي يستعملها المبرمجون لتبرير المشاكل التي ذكرناها... فانطلق معي لفخ: "سأشرحها لكم في الاجتماع" ^^

## الافتراضات الخفية - فخ: سأشرحها لكم في الاجتماع!

إن من أسهل الطرق وأكثرها فاعلية لاختبار جودة مخططك المعماري هي "اختبار الصمت"؛ أعطِ مخططك لأحد أصدقائك أو زملائك واطلب منه فهمه دون أن تنطق بكلمة واحدة! فإذا تعثر وبدأ في طرح الأسئلة أو افترض افتراضات خاطئة فذلك إشعار بوجود خلل ما في المخطط... لا نريد أن نصير كمن أرسل بريدا إلكترونيا لشخص ما ثم اتصل به هاتفيا ليشرح له ما فيه *_*، فإن حصل هذا = فقد البريد الإلكتروني قيمته الأساسية.

إن المخطط المعماري الجيد يجب أن يكون قادراً على الوقوف بمفرده ليروي قصته ^^، فمن يشاهده سيستمتع بالصناديق الخلابة والخطوط الرائعة، وسيكون الشرح الكلامي عاملاً **مُكمّلا** يضيف عُمقاً للنقاش، لا **مُفسّراً** لمجموعة من الطلاسم المبهمة!

وللأسف، كثيراً ما نسمع في الاجتماعات أعذارا من نوع: "سنتحدث بالتفصيل عن هذا المخطط لاحقاً"، أو "هذا الجزء غير مفهوم على الورق، لكننا سنشرحه أثناء العرض" أو "ستفهمون المخطط بعد أن أقوم بشرحه لكم، لا عليكم"... إلخ

يمكننا تلخيص ما ذكرناه هنا:

- اختبار الصمت وسيلة وأداة رائعة تكشف زيف الافتراضات وتبرز جودة المخطط المعماري.
- مجرد سماعك لأعذار مثل "ستفهمون المخطط بعد أن أقوم بشرحه لكم" ؛ فهذا يعني = مشكلة الأعذار الواهية في غرف الاجتماعات ^^
- إذا كان المخطط يحتاج دائماً إلى متحدث يشرحه؛ فإن هذه كارثة اسمها "الفجوة -The Gap-"، وهي تمثل الفجوة بين "المعلومات المكتوبة على الورق" وبين "المعلومات المحبوسة في رأس المبرمج"، لذلك ستظهر المشاكل مباشرة عندما ينتهي الاجتماع، أو عندما يستقيل هذا المبرمج من الشركة! وهنا وفي هذه اللحظة، سيصبح هذا المخطط الذي تم حفظه كتوثيق طويل الأمد مجرد رسمة لا قيمة لها؛ لأن "الصوت" الذي كان يشرحها قد رحل!

الخلاصة التي نخرج منها من هذا الموضوع المهم:

افتراضك بأن مخططك سيرافقه دائماً "شرح صوتي" يخلق فجوة خطيرة بين ما هو مرسوم على الورق، وبين ما هو محبوس في رأسك!
إن المخطط الذي يحتاج إلى "محام ليدافع عنه ويشرحه" هو مخطط عديم القيمة، خصوصا عندما يُحفظ كمرجع وتوثيق طويل الأمد للأجيال القادمة من المبرمجين الذين لن يجدوك بجانبهم لتشرح لهم ما كنت تقصده! وهذا معيار يمكنك استخدامه لقياس جودة رسوماتك ^^.

والآن، نكون قد انتهينا بفضل الله -سبحانه وتعالى- من تشخيص الأمراض الشائعة -مشاكل المخططات والرسوم المعمارية- ووصفها، والآن سننتقل بدورنا لعلاج هذه الأمراض وسنبدأ بتقديم الحلول شيئا فشيئا حتى نصل لما نحب رؤيته وسماعه -بإذن الله-.

## الفصل الثاني: أزمة المصطلحات، نحن نحتاج إلى مفاهيم مشتركة أكثر من حاجتنا لرموز موحدة!

قد تتعجب من هذه البداية، لكنها بداية موفقة حقيقة... فهي تبدأ بمعالجة المشاكل من جذورها، فإن سلم الجذر سلم ما فوقه!
إن مشكلتنا الحقيقية ليست فقط في رسم مجموعة من "المربعات والخطوط" بعشوائية! بل في كوننا كمبرمجين لا نمتلك "قاموساً مشتركاً" لوصف ما نبنيه... ولعل من أشهر الأمثلة على ذلك عند حديث أحدنا عن مصطلحات مثل Component أو Module ماذا يقصد بها؟ لإن سياق الكلام في نفس المكان لنفس الكلمات قد يختلف! فمثلا قد تشير أنت إلى الموقع

الإلكتروني باعتباره Component داخل نظام المؤسسة، في حين قد يتحدث زميلك على Component مشيرا لأحد الأجزاء داخل الموقع نفسه... ومع أن كلاهما صحيح لغويا، إلا أن هذا سيؤدي لمشكلة كبيرة لإن كل واحد منهم يقصد مستوى مختلف من ال abstraction.

ملاحظة: هذه المشكلة -مشكلة المصطلحات- أعدها واحدة من أكثر المشاكل التي نحتاج لحلها، فأمتنا الإسلامية أمة عظيمة عريقة، ولغتها لغة عذبة غنية، فلا أقل من أن تُدرس العلوم بلغتنا، وأن يوضع قاموس علمي -تقني وغيره- عربي تتفق عليه الأمة، ويتم استخدامه من قبل الأفراد... وهذا يحتاج لمجهود عظيمة من الدول والأفراد، مجهود لزرع العزة في نفوس أفراد الأمة، ومجهود لنقل العلوم وتطويرها وتعليمها ونحو ذلك، ومجهود للتخلص من آفات الماضي، وآفات الذلة والخضوع لثقافة الغالب، والله المستعان.

بناء على ما سبق، تبرز لنا أول نقطة مهمة، وهي:

## المفاهيم المشتركة أهم من الرموز -Common abstractions over a common notation-

ويقصد في هذه النقطة أن الاتفاق على "المفاهيم ومستويات ال abstractions" أهم بكثير من الاتفاق على الألوان والأشكال -الرموز- التي سيتم استخدامها، فكما أشرنا في هذا الكتاب، فإن فن معمارية الأنظمة لم يصبح ناضجا بما يكفي كما نضجت العلوم الأخرى مثل الهندسة الكهربائية -أي مهندس كهربائي في الأردن أو مصر أو الجزائر... سيفهم رمز المقاومة أو

شدة التيار ونحو ذلك بمجرد النظر لأنها رموز مشتركة، ولا تتواجد مثل هذه المزية حاليا في معمارية الأنظمة-... وحتى تتضح هذه الفكرة لنأخذ مثالا، تخيل أن لديك أكثر من تطبيق للخرائط مثل خرائط جوجل وخرائط أبل، ستجد أن بعض الرموز أو الخطوط أو الألوان مختلفة، لكنها مع ذلك تتفق وتتوحد في التجريد -abstractions- مثل الطرق والأنهار والغابات والمدن ونحو ذلك -اتفاق في المفاهيم-، ويبرز هنا دور مفتاح الخريطة -Legend-، وبناء على هذا التجريد ودليل الرموز، ستتمكن من قراءة أي خريطة بسهولة ومهما اختلفت ألوانها ورموزها؛ لأن "المفاهيم" واضحة وموجودة في ذهنك.

الخلاصة التي نخرج منها من هذا الموضوع المهم:
كما رأينا مثال الخرائط الجغرافية؛ فإن **"المخططات المعمارية هي الخرائط التي تُمكن المبرمجين من الاستكشاف والتنقل داخل غابات الشيفرة البرمجية المعقدة"**، وكما لا يمكنك التنقل في مدينة كبيرة دون خريطة، فلا يمكنك التنقل في نظام برمجي معقد دون مخطط معماري! لذلك، إن الاتفاق على التضاريس والمفاهيم ومستويات التجريد هو أول وأهم خطوة، ثم ارسم مخططك بعد ذلك بأي شكل أحببت، وبأي ألوان رأيت، بشرط ألا تنسى وضع مفتاح -Legend- واضح للخريطة!

## البنية الثابتة -Static Structure-: الهيكل العظمي للنظام (ولادة نموذج C4)

والآن، سنقوم بتجهيز المهد ليستقبل المولود الجديد ^^، ومن هنا سيبدأ Simon بوضع حجر الأساس لنموذجه المقترح C4، وسيبدأ بطريقة رائعة من خلال تعليمنا البناء من الأساس

وصولا للتشطيبات النهائية، وبهذا سنفهم كيف سنبني مستقبلا أي مخطط، وكيف سنتمكن من الرسم!

وحتى نصل لهذه المرحلة علينا ألا ننسى أن أول خطوة مهمة هي "الاتفاق على قاموس المصطلحات والمفاهيم" -تذكر مشكلة ال Component التي تحدثنا عنها-، والآن، ماذا نقصد بالبنية الثابتة -Static Structure-؟

إن البنية الثابتة -Static Structure- تعني = ما هي مكونات النظام وهو ساكن لا يعمل؟ أو ما هي القطع التي تشكل هذا النظام؟ بمجرد أن تفهم البنية الثابتة وتتفق عليها، سيكون من السهل علينا إضافة مخططات أخرى تشرح "البنية الحركية / الديناميكية -Dynamic Behaviour-"، مثل كيف يعمل النظام وقت التشغيل -Runtime-؟، أو كيف تعمل "بنية النشر" مثل أين سيتم استضافة ال Deployment؟

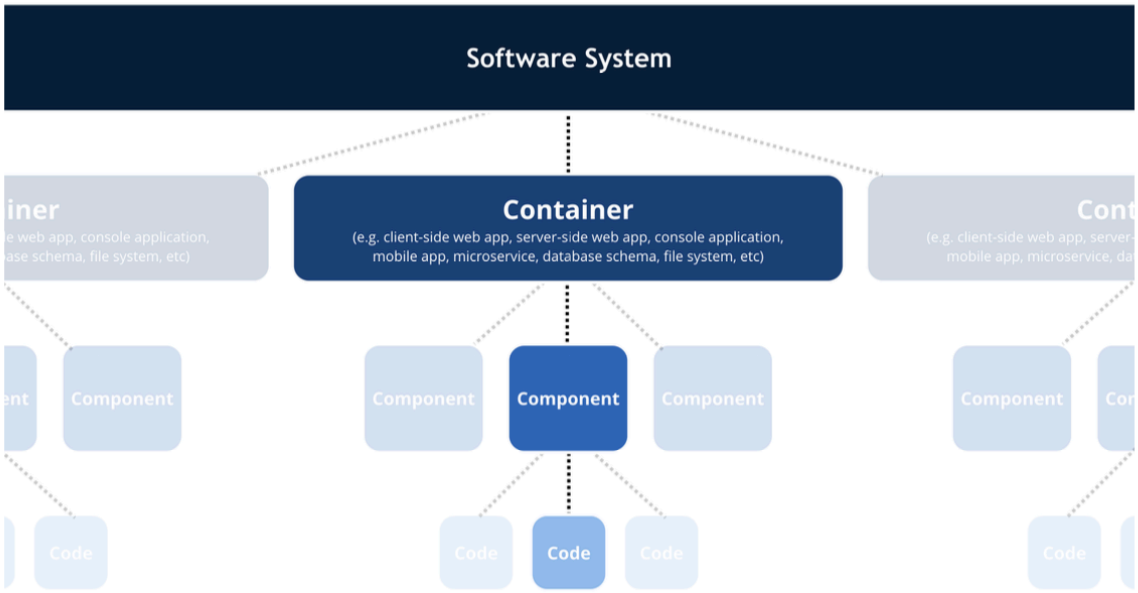
إن البنية الثابتة تشبه "الهيكل العظمي" للبرمجيات، فبمجرد أن نتفق على شكل هذا الهيكل ومستوياته، سيصبح من السهل علينا أن نكسوه بالعضلات (من خلال توضيح سلوكه وقت التشغيل-Runtime-)، أو أن نُلبسه الثياب (من خلال توضيح آلية نشره واستضافته Deployment)... وكما أنه لا يمكنك فهم كيف يجري الإنسان قبل أن تعرف أن لديه ساقَين، فلن تتمكن من بناء مخطط معماري لنظام معقد دون أن تبني هيكله العظمي [٨٨].

ملاحظة: لو شبهنا ما نتحدث عنه بالإنسان، فإن العظام والعضلات والأعضاء هي البنية الثابتة -Static Structure-، أما البنية الديناميكية -Dynamic Behaviour- فستكون الإنسان وهو يركض أو يتحرك أو يتنفس أو يتحدث... إلخ، هذا التشبيه مهم ومفيد من وجهة

نظري حتى تتمكن من الحفاظ على مفاهيمك دون شوائب -بإذن الله-... على الأقل هذا ما أتأثر به شخصيا ^^.

حسنا، هذا الكلام جميل جدا حتى هذه اللحظة، ولكن لم نشاهد البنية الثابتة هذه حتى الآن، فأين هي؟!

أشكرك يا صديقي على حماسك هذا، إذا انظر للصورة أدناه ثم اتبعني:




A **software system** is made up of one or more **containers** (web applications, mobile apps, desktop applications, databases, file systems, etc), each of which contains one or more **components**, which in turn are implemented by one or more **code** elements (e.g. classes, interfaces, objects, functions, etc).

تعد هذه الصورة أول صورة لمخطط قام Simon بنفسه بتصميمه -إذا هو ليس من ورشات العمل السابقة-، وهو نموذج نظري يقدم به أول خطوات الحل، وينقسم لعدة أجزاء رئيسية وهي:

- الطبقة الأولى أو العليا أو الكبرى: وهي الطبقة المعنونة ب Software System، وهي ذات لون أزرق غامق، وتمثل أعلى مستوى من التجريد -abstraction-، هذه الطبقة ليست مجرد شيفرة برمجية، بل هي كيان قائم بذاته يقدم خدمة محددة للمستخدمين، والمستخدمين هنا قد يكونون أشخاصا وقد يكونون أنظمة خارجية أخرى تستخدم هذه الخدمة ^^، لذلك، في هذا المستوى نحن لا نهتم بالتقنية المستخدمة، بل ننظر إليه كصندوق أسود كبير يحتوى بداخله خدمة محددة يريد تقديمها...

- الطبقة الثانية: وهي طبقة الحاويات -Containers-، وهذه الطبقة يمكن أن تحتوي واحد أو أكثر من الحاويات، ومن الأمثلة على الحاويات الممكنة ال client-side app, server-side app, database...إلخ، وربطت هذه الحاويات بخط متقطع مع الطبقة الأولى للدلالة على أن الحاوية هي جزء من النظام.. وهي ذات لون أزرق أفتح درجة من أزرق النظام، وهذه الحاويات -Containers- ليست ال Docker Container، لذلك، في سياق حديثنا عن مفهوم ال الحاويات -Containers- هنا، فاعلم أننا نتحدث عن C4 Container وليس Docker Container ^^... والحاوية هنا تشير إلى أي وعاء أو بيئة تستضيف شيفرة برمجية ليتم تطبيقها أو بيانات ليتم تخزينها... وهو بذلك يعني شيء يجب أن يكون قيد التشغيل حتى يعمل النظام.
وحتى تتضح الفكرة دعونا نلقي نظرة سريعة على أشهر الحاويات -Containers- التي نستخدمها:

- **تطبيقات الويب (Web Apps)**: سواء كانت من جهة الخادم (مثل تطبيقات Node.js، ASP.NET، Java) أو من جهة العميل (مثل تطبيقات Angular، React).

- **تطبيقات سطح المكتب والهواتف (Desktop & Mobile Apps)**: تطبيق ويندوز، تطبيق أندرويد، أو iOS.

- **قواعد البيانات (Databases)**: أي مخطط (Schema) في قاعدة بيانات سواء كانت SQL Server أو MongoDB وغيرها.

- **ال Microservices & Serverless**: كل Microservices موجودة عندنا تمثل حاوية، وكل دالة Serverless تمثل عادة حاوية مثل AWS Lambda.

- **أماكن تخزين الملفات (File Systems & Blob Storage)**: مثل (Amazon S3) أو أي نظام ملفات محلي.

بناء على هذه الأمثلة، نستنتج أن الشرط الأساسي لنعتبر جزء ما على أنه حاوية -Container- هو: أن يكون هذا الجزء قابل للنشر بشكل مستقل عن بقية النظام، وهذا يعني أن كل حاوية يمكن أخذها ووضعها على خادم -Server- منفصل تماما

-سواء كان خادماً فيزيائياً، أو وهمياً، أو حاوية Docker- فذلك يشير لكونها حاوية -Container-... وبناء على هذه المعطيات سيترتب على ذلك أمر حتمي، وهو: بما أن كل حاوية -Container- تعيش في بيئة مستقلة، فإن اتصال حاوية بأخرى لا يتم عبر استدعاء شيفرة برمجية داخلية بسيطة ومباشرة، بل يتطلب الوصول أو الاتصال عبور حدود الشبكة أو المعالج -Network or Process boundary-، أي أن تواصل الحاويات مع بعضها البعض يتطلب نداءات خارجية (مثل API Calls، أو استعلامات شبكة لقاعدة البيانات)... وهذا سيجعلنا ندرك أن الخط الذي سربط حاويتين معا ليس مجرد خط عادي في النموذج، بل هي رحلة طويلة للبيانات والعمليات عبر الشبكات وأنظمة التشغيل.

ملاحظة: Out-of-process Call && Remote Procedure Call (RPC) هي مفاتيح أساسية لفهم السياق الذي نتحدث عنه، ولفهم المقصود بهذا فاعلم أن كل برنامج تقوم بتشغيله يمثل غرفة محصنة ومغلقة، وله ذاكرته المخصصة -Memory-، ولا يُسمح لأي برنامج آخر بالتلصص على هذه الغرفة، وهذه تسمى ال Process، ومن هنا يظهر لنا مفهومان، الأول: In-process والذي يشير إلى العمليات التي تجري داخل هذه الغرفة المحصنة، مثل لديك Class Car و Class Engine وأحدهما يستخدم دالة من الآخر، هنا يكون الاتصال بين هذه ال classes سريع جدا لأنهما يتشاركان نفس الذاكرة، أما المفهوم الثاني هو Out-of-process، وهو يشير إلى التواصل الذي سيتم بين غرفتين محصنتين، مثل حاجة ال Class Car بالاتصال بقاعدة البيانات من نوع Mysql، وذلك يعني أن الاتصال سيخرج من غرفتي المحصنة

متخطيا الحدود الداخلية الآمنة لينتقل للخارج، وهذا سريع جدا لكنه أبطأ من ال in-process... لكن ماذا لو كان الاتصال سيتم بين servers مختلفة؟ مثل تشغيل قاعدة البيانات على server A وتشغيل البرنامج على server B؟ هنا تصبح الحدود الفيزيائية موجودة وليست حدود على مستوى الذاكرة، وهذه ما يطلق عليها ال Network Boundary، والتواصل هنا يتم عبر وسائل متنوعة و بروتوكولات لهذا الغرض مثل ال HTTP/TCP... ومن هنا ظهرت طريقة ال RPC والتي هي اختصار ل Remote Procedure Call، ولعل من أشهر الأمثلة عليها ال gRPC، كما يمكن استخدام ال REST API كخيار آخر وشائع للتواصل خارج حدود الشبكة...

بناء على المعلومات التي ذكرناها، لو كان لدينا تطبيق client side مصمم من خلال ال react ويعمل لوحده ويتصل مع الباك اند من خلال api، وتطبيق باك اند كتب بلغة nodejs وتم وضع كلامها على نفس السيرفر، فكم حاوية ستقوم برسمها في مخططك؟ ماذا لو قمنا بدمجهما معا؟ الجواب ببساطة: حاويتين، والثاني حاوية.

- الطبقة الثالثة: وهي طبقة المكونات -Components-، وهذه الطبقة يمكن أن تحتوي واحد أو أكثر من المكونات، مثل: مكون ال PaymentGateway ومكون ال OrderProcessor، وتم ربط هذه المكونات بخط متقطع مع الطبقة الثانية للدلالة على أن هذه المكونات هي جزء من الحاويات.. وهي ذات لون أزرق أفتح درجتين من أزرق النظام... وحتى يتم صياغة تعريف يُمكننا من فهم ماهية المكون والتفريق بينه وبين الأجزاء الأخرى من النموذج، علينا تعريفه بشكل دقيق: ويمكن القول أن

المكون -component-هي تجميع لوظائف مترابطة، يتم تغليفها -Encapsulated- وإخفاؤها خلف Well-defined Interface، ويقصد بال Well-defined Interface البوابة التي يستطيع من خلالها هذا المكون من التواصل مع العالم الخارجي، وبذلك تكون ال Interface البوابة أو العقد الذي سيمكننا من التواصل وفهم ما سنأخذه وما سنرسله -ال Interface هنا ليست ال UI-، وحتى تكون هذه البوابة Well-defined فعليها أن تحقق مجموعة من الشروط أهمها ال Clear Contract، بحيث إذا أعطيتني المدخلات س فستحصل على المخرجات ص، وأيضا تم تغليفها وإخفاء التعقيدات بداخلها -Encapsulation-، فلا يوجد هناك داع لأن تعرف أو ترى المكونات الأخرى ما بداخل المكون الآخر... مثل ال Payment Component، حتى تكون جيدة يكفي أن تتعامل المكونات الأخرى التي تحتاج هذا المكون معها من خلال معرفة ما سيتم ارساله وما سيتم استلام، فلو تغيرت طريقة الدفع لن يتوجب على المكونات الأخرى تغيير مئات الأسطر حتى تعمل، لإن كل شيء سيبقى يعمل ^^...

بناء على ما ذكرناه، فإن المكون -Component- يختلف عن الحاوية -container- بكونه ليس قابلا للنشر بشكل مستقل! فجميع العناصر الموجودة داخل المكون -Component- ستعمل in-process داخل الغرفة المحصنة ^^.

- الطبقة الرابعة: وهي طبقة الشيفرة البرمجية -Code-، وهذه الطبقة يمكن أن تحتوي واحد أو أكثر من عناصر الشيفرة البرمجية مثل ال Class, Methods...إلخ... تم ربط هذه المكونات بخط متقطع مع الطبقة الثالثة للدلالة على أن هذه الشيفرة

البرمجية هي جزء من المكونات.. وهي ذات لون أزرق أفتح بثلاث درجات من أزرق النظام.. هذا الجزء يمثل آخر مستوى أو أصغر مستوى للتفصيل، فلا أصغر منه في نموذج C4.

إن الصورة التي بالأعلى تمثل حجر أحجار البناء التي تم رصها بشكل متسلسل من الأكبر وصولا للأصغر، فيبدأ الأمر بمستخدمين يتفاعلون مع النظام البرمجي الذي يحتوي بداخله مجموعة من الحاويات والتي بدورها تحتوي بداخلها مجموعة من المكونات والتي بداخلها تحتوي مجموعة من الشيفرات البرمجية... فقط وببساطة ^^

لحظة لحظة... أظن أن لديك سؤالا ذكيا وخطيرا ^^، وحقيقة هذا السؤال أشكل علي شخصيا عندما قرأت الكتاب، وهو: أين الأشخاص -People-؟ لماذا لا أراهم في الصورة؟ ألم نتحدث عن شيء مشابه لهذا في المشاكل الشائعة؟ وانطلقت علامات الاستفهام والتعجب في عقلي حتى فقدت توازني هههههه... حسنا، فكر قليلا وستجد الجواب في ثنايا السطور ^^

إن إجابة هذا السؤال ببساطة ستجدها في كلمة "يتفاعلون" في الفقرة السابقة، وهذا هو الشرح التفصيلي للإجابة ^^:

إن هذه الصورة أو المخطط الهرمي يمثل أحجار البناء، وهو يخبرك أنا "أتكون من" و "أحتوي على"، فلو قمنا بوضع "الأشخاص" على رأس التسلسل للهرم، فهذا وكأنك تقول عند قرائتك للمخطط: "الأشخاص يتكونون من عدة أنظمة برمجية -Software Systems-"، وطبعا هذا شيء غير منطقي ومضحك، ومع ذلك، فهذا ليس السبب الوحيد، إن السبب المهم هنا هو أن "الأشخاص" يمثلون ال Context الذي يتفاعل مع النظام لا البناء ذاته، أما موضوعنا الذي نتحدث عنه هنا هو **البنية الثابتة وأحجار البناء، لا عن التفاعلات التي ستحدث مع هذه**

**البنية**... وهذا ما سيتم شرحه لاحقا -بإذن الله- عند ال System Context Diagram في نموذج C4.

## مثال عملي ٢: عيادة الحيوانات الأليفة -Spring PetClinic-، تمثيل الانتقال من مستوى الشيفرة البرمجية إلى المكونات -components-

يعد هذا المثال واحد من أشهر الأمثلة التي يتم استخدامها لتعلم جافا spring... وهو يحتوي بداخله على تطبيق ويب مبني بلغة جافا، وفيه قاعدة بيانات، ومن هذه المعطيات يمكننا أن نستنتج مباشرة أن هذا المثال يتكون من حاويتين -Two Containers- ^^

لكن قبل أن نبدأ في المثال وجب علينا أن ننوه إلى نقطتين مهمتين:

1. لا تخف من أن المثال كتب بلغة جافا أو غيرها... فنحن لن نتعلم جافا سبرنج هنا، بل سنتعلم كيف نرسم مخطط معماري لهذا التطبيق، وما نحتاجه من معطيات سنجدها ^^
2. المكون -Component- ليس مفهوما محتكرا لا يمكن استخدامه إلا مع البرمجة الكينونية OOP.. بل هي مفهوم يمكن أن يستخدم ب:
   a. **في لغات البرمجة كائنية التوجه -OOP-** (مثل C++ أو جافا أو C#:): وهنا يتشكل المكون من تجميع عدد من الفئات -Classes- والواجهات -Interfaces- المترابطة.

b. **في لغات البرمجة الإجرائية -Procedural-** (مثل: *C*): يمكن أن يكون المكون عبارة عن تجميع لعدد من ملفات الكود -C files- المحفوظة معاً داخل مجلد -Directory- معين.
c. **في بيئة JavaScript**: يُمثل المكون هنا "وحدة برمجية" -Module-، والتي تتكون بداخلها من مجموعة من الكائنات -Objects- والدوال -Functions- المترابطة.
d. **في لغات البرمجة الوظيفية -Functional-** (مثل: *F#, Haskell*): يظهر المكون أيضاً على شكل "وحدة برمجية" -Module-، وهو يمثل تجميعاً منطقياً لدوال -Functions- وأنواع بيانات -Types- تخدم غرضاً واحداً.
e. **في قواعد البيانات العلائقية -Relational Databases-**: المكون هنا ليس شيفرة برمجية تقليدية، بل هي "تجميع منطقي للوظائف" تتشكل من مجموعة مترابطة من الجداول -Tables-، وال Views، والإجراءات المخزنة -Stored Procedures-، والدوال -Functions-، وال Triggers.

حسنا، والآن لنبدأ على بركة الله:

## من الشيفرة البرمجية إلى المُكوّن -Component-: التصفية والتجميع!

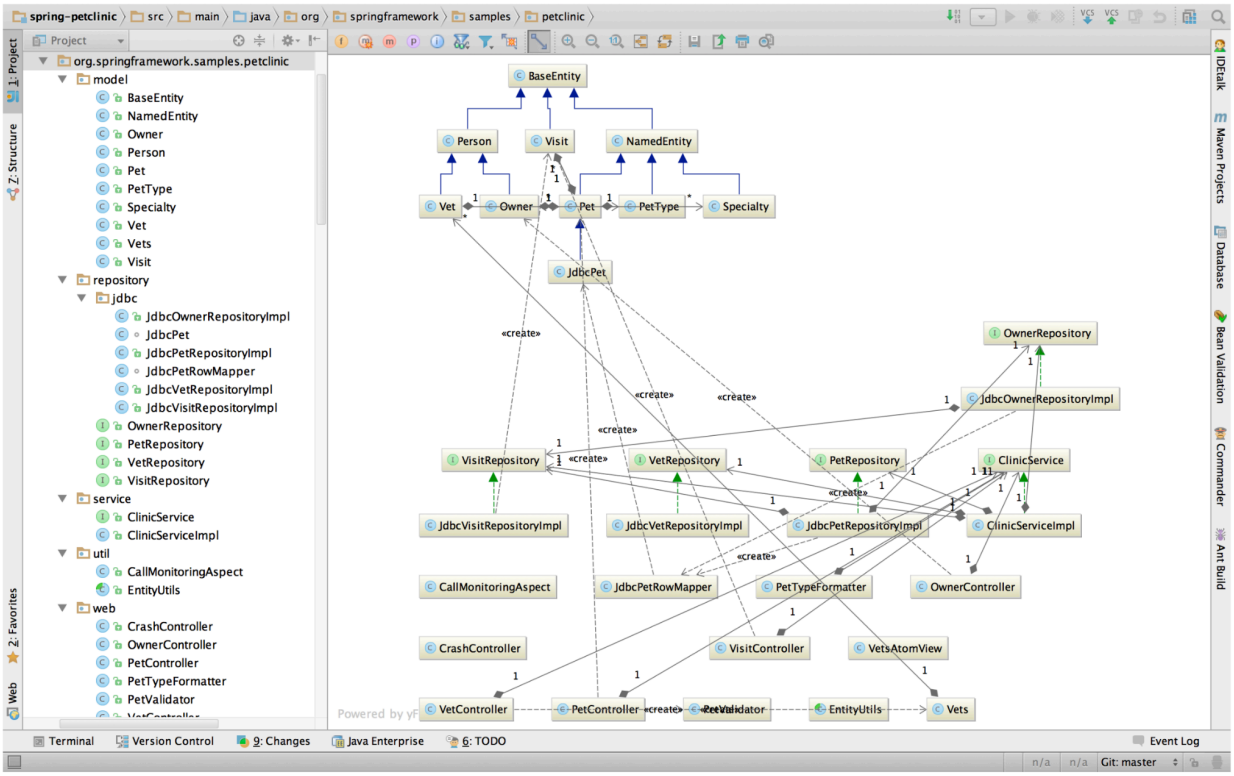
سنبدأ الآن في رحلة بصرية متدرجة حتى نصل إلى النتيجة الصحيحة والمتوقعة بناء على ما تعلمناه، وسنرى خلال هذه الرحلة الفرق الكبير والجوهري بين المخططات مثل ال UML

وبين ما سنصل إليه في طبقة ال component بعد التصفية والتجميع وهي ما يطلق عليها بالتجريد ^^.

رابط الريبو الخاصة بمثالنا تجده هنا.

حسنا، في أول خطوة سنقوم بالهندسة العكسية، أي أننا سنقوم بتحميل الشيفرة البرمجية من الريبو ثم سنقوم بإنشاء مخطط UML من الشيفرة البرمجية الموجودة، ويمكنك القيام بذلك بسهولة من خلال ال IDE الذي تستخدمه.

شاهد الصورة:



من الصورة نلاحظة أن المخطط مليء بعشرات الصناديق التي ترتبط فيما بينها بالكثير من الخطوط المتشابكة والتي تمثل العلاقات مثل الوراثة أو الاستخدام أو الإنشاء ونحو ذلك، وكل هذا التعقيد الموجود في الصورة وهو لم يعرض سوى ال Classes وال Interfaces والعلاقات التي بينهم، فكيف لو قمنا بعرض ال Properties وال Methods داخل هذا ال UML؟!

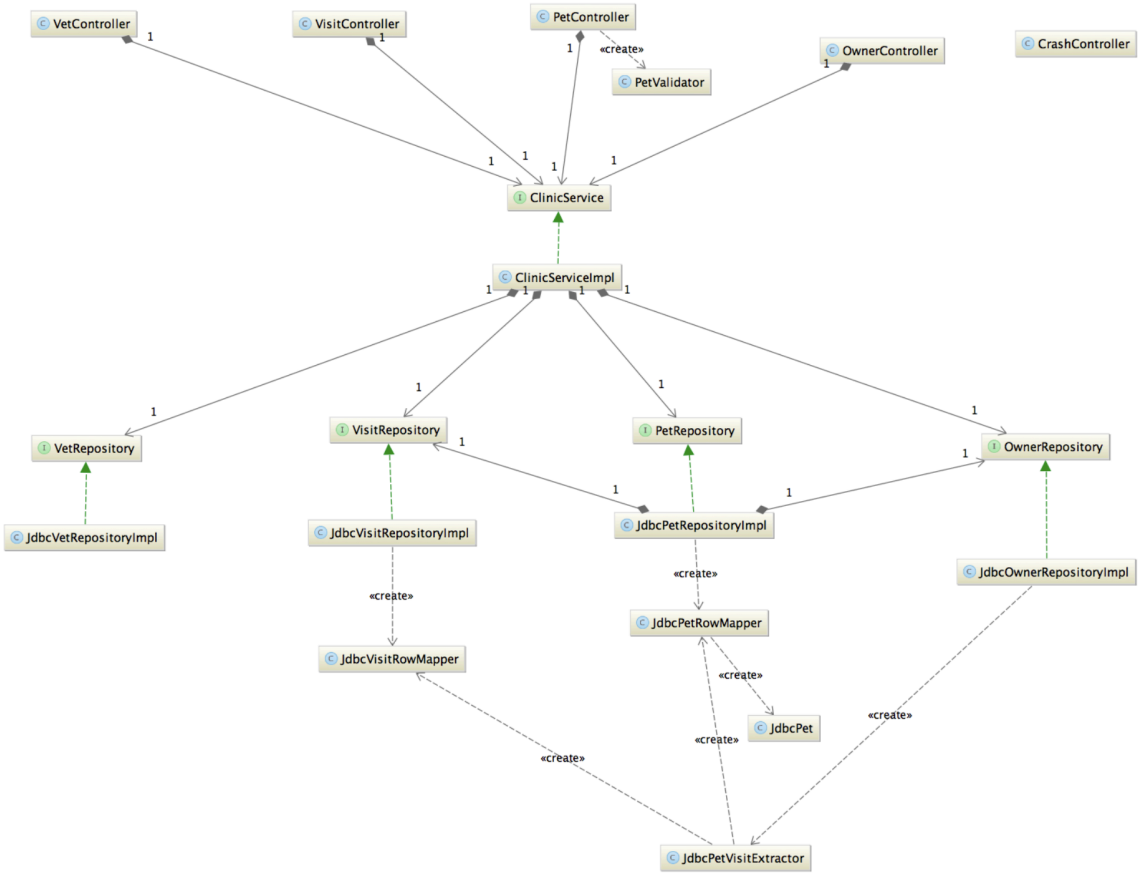
إذا الخلاصة الأولية هنا: هي أن هذا المشروع فعليا بسيط جدا، ويمثل نظام عيادة بيطرية صغير كما نرى في الصورة، ومع ذلك فإننا إذا نظرنا إلى المخطط سنجد أنه مزعج ومربك! ومع أن هذا المخطط صحيح؛ إلا أننا لا يمكن اعتماده في مشاريعنا الكبرى أو المعقدة.. لأن المشروع البسيط أتى بمثل هذا المخطط، فكيف بالكبير أو المعقد؟!

إن هذا المخطط يشبه خريطة تُظهر كل منزل وكل شجرة وكل سيارة في المدينة، ومع أن الخريطة صحيحة، لكنها ستكون مزعجة لنا وبلا فائدة للتنقل ^^، لأن ما نحتاجه للتنقل هو خريطة تُظهر الأحياء والطرق الرئيسية.

والآن إلى الخطوة الثانية، وهي القيام ببعض التصفية من خلال حذف ال classes التي لا تُهمنا في النقاش المعماري بما يتناسب مع ال static structure التي تحدثنا عنها سابقا، ويمكن أن يتم ذلك من خلال حذف ال Model -ال domain- وال Utils classes ^^.. والسبب في ذلك أن ال model تمثل وظيفة لا بنية معمارية، فهي مجرد بيانات تنتقل بين الطبقات المختلفة ولا تمثل أي بنية أو مسؤولية معمارية، وال util تمثل تفاصيل تنفيذية لا معمارية.

إن هذه الخطوة تشبه الخريطة التي لا تظهر السيارات -model- التي تسير فيها -فهي تتغير باستمرار-، بل تظهر الطرق ذاتها والتي تمثل بنية ثابتة -static structure-.

شاهد الصورة التالية:




نلاحظ الآن بعد الاطلاع على الصورة أن المخطط أنظف بكثير، وأسهل للقراءة ^^، فبعد حذف ال model وال utils تبقى لدينا الأشياء المهمة في المعمارية وهي ال Controllers والتي تمثل الطبقة العليا الآن، وال Services والتي تمثل الطبقة الوسطى، وال

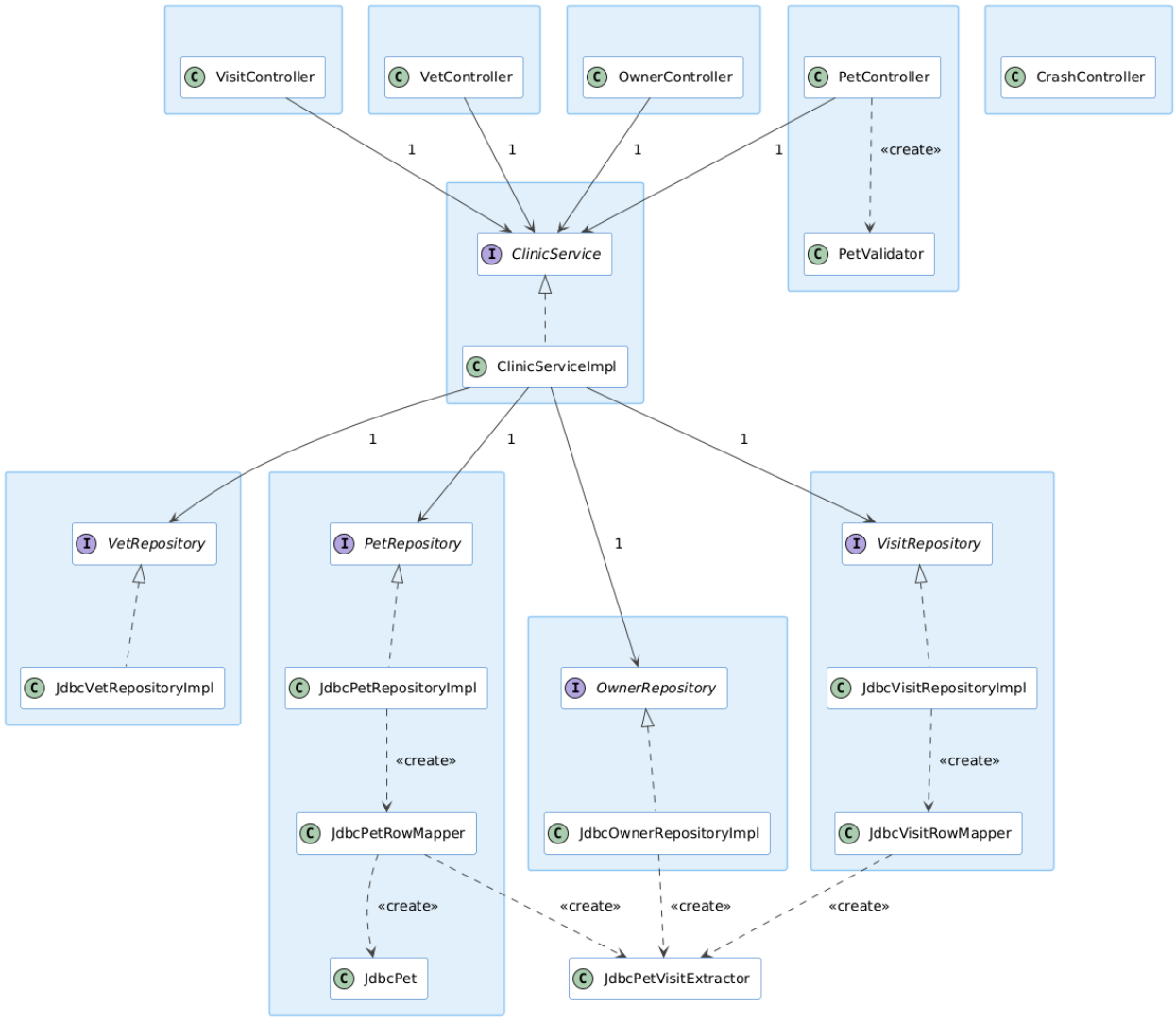
Repositories -طبقة الوصول للبيانات- والتي تمثل الطبقة الأخيرة... وهذا النمط المعماري هو: ال Three-Tier / Layered Architecture.

لكن، ما زالت هناك مشكلة موجودة وإن صار المخطط أفضل، فما زلنا ننظر إلى المخطط باعتباره شيفرة برمجية، أي أننا ما زلنا بطبقة ال code في النموذج المعماري الذي نرغب في بنائه... لذلك، علينا أن نقوم بعمل Zoom Up -تذكر الخريطة دائما- حتى نخرج من مستوى الشيفرة البرمجية ونرى مستوى ال component، وذلك سيتم من خلال تحديد عناصر الشيفرة البرمجية التي يمكننا جمعها معا...

وهنا سيظهر لنا سؤال مهم، كيف سنقوم بتجميع الشيفرة البرمجية في مكونات؟ وهل هناك قاعدة واحدة للقيام بذلك؟

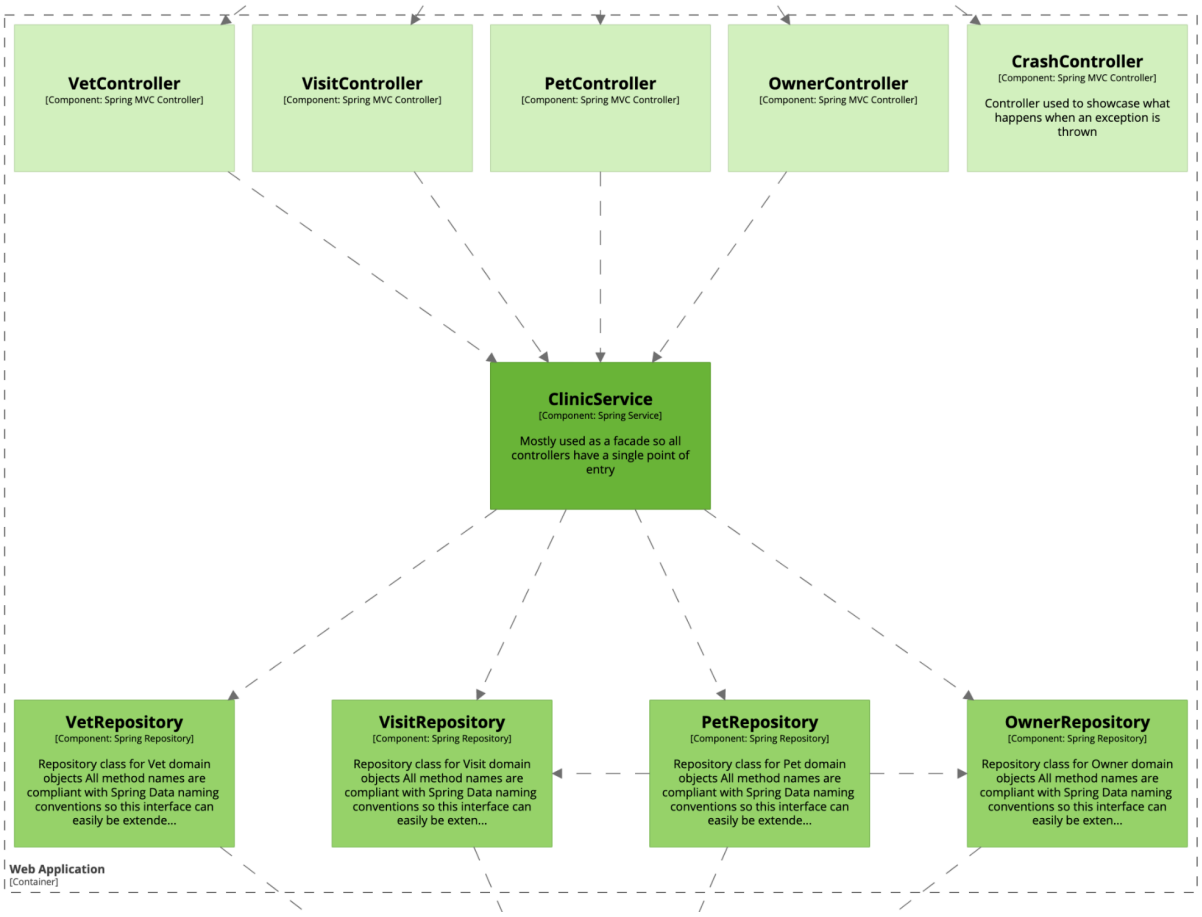
والجواب يا صديقي أنه لا توجد قاعدة واحدة للقيام بذلك، بل يمكن القيام بذلك بأكثر من أسلوب أو بأكثر من استراتيجية، ولهذا فإن الاستراتيجيات يمكن أن تختلف من مشروع لآخر، وسنتحدث عن هذا بمزيد من التفصيل لاحقا -بإذن الله تعالى-.. أما في مشروعنا الصغير هذا، فسنقوم بتجميع كل واجهة -interface- مع ال Implementation class الخاص بها داخل مكون -component- واحد ^^

شاهد الصورة:




بناء على الصورة نرى كم صار الموضوع أسهل وأوضح، فقد تم تجميع الشيفرة البرمجية داخل مكونات -components- فعلا، والمكون الواحد قد يحتوي بداخله class أو أكثر مثل ال VetController وال VetRepository.. أما استراتيجة التجميع فكانت كما قلنا، كل واجهة وتطبيقاتها = مكون -component-، وكل controller يمثل مكون...، والآن بعد أن حددنا المكونات، يمكننا إزالة التفاصيل الداخلية الموجودة داخل كل صندوق أزرق بسهولة،

وبهذا سنزيل أي رواسب متعلقة بالشيفرة البرمجية نفسها، ونبقي فقط على المكونات نفسها، وهذا ما سيقودنا للمخطط النهائي للمستوى الثالث -component level-، شاهد الصورة:




كما تلاحظ في الصورة، فإن المخطط المعماري يبرز المكونات بشكل واضح تماما، ولا توجد ضوضاء بصرية تعيقنا عن فهم هذه المكونات...

لكن، أظن أن هناك سؤال في رأسك الآن، هناك عنصر غريب وهو ال JdbcPetVisitExtractor .. فهذا العنصر في الصورة قبل الأخيرة وجد لوحده دون أن يكون مكونا -component- بذاته! وكذلك لم ينقل معنا للمخطط الأخير، فما هو هذا العنصر؟ ببساطة هذا class يتم استخدامه من أكثر من component، هذا ال class يمثل

شيفرة برمجية مجردة أو مساعدة ترتبط في هذه ال repos، لذلك تم وضعه في المخطط قبل الأخير، وتم حذفه في المخطط النهائي الذي يمثل المكونات فقط.. وهذا ما سيقودنا إلى فكرة مهمة، وهي:

**تحدي أو مشكلة الشيفرات البرمجية المشتركة -Shared Code-:**

فأثناء عملية التجميع التي ستقوم بها، ستواجه حتماً تحدي "الشيفرات البرمجية المشتركة"، مثل فئات -class- التي تقدم مساعدة ما مثل ال Helpers وال Utils، أو الفئات -classes- الأساسية المجردة التي تُستخدم في أكثر من مكان، لذلك، فالقاعدة هنا هي: حاول إعادة هيكلة الشيفرة البرمجية -Refactoring- لإدخال وإخفاء أكبر قدر ممكن من هذه الفئات -classes- "داخل" المكونات -components- التي تستخدمها حصرا، ولكن مع ذلك كن واقعياً! فبعض الشيفرات البرمجية المشتركة هي "أمر لا بد منه" وستبقى متقاطعة مع عدة مكونات إلى الأبد ^^.

ملاحظة مهمة: تجميع الشيفرة البرمجية وتحويلها لمكونات ينطبق على مختلف المدارس المعمارية مثل ال Hexagonal أو من خلال Package by Feature... الفكرة واحدة وإن اختلفت الأساليب.

ملاحظة مهمة ٢: كما تلاحظ، فإن الأدوات التي تستخرج المخططات من الشيفرة البرمجية (Static Analysis Tools) تنتج مخططات تخص الشيفرة البرمجية لا مخططات معمارية! والسبب في ذلك يعود لأن المعلومات المعمارية غالبا غير موجودة في الشيفرة البرمجية بشكل صريح... وسيتم معالجة هذه النقطة لاحظا بإذن الله.

ملاحظة مهمة ٣: هل لاحظت في الصورة أن المخطط الناتج أحيط بخطوط متقطعة وفي أسفل الزاوية اليسرى كتب: [Container Web Application]؟ وهذه تشير إلى ما تحدثنا عنه سابقا، فهذه المكونات لم تعد مشردة -مشكلة HOCO-، بل لها بيت واضح... وكما تلاحظ في الصورة أيضا فالمشاكل مثل مشكلة الغموض تم حلها من خلال استخدام اسم واضح مثل VetController، ومشكلة إغفال التقنية تم حلها من خلال [:Component Spring MVC Controller]، ومشكلة المصطلحات العامة تم تحلها من خلال وصف يحدد المسؤولية لكل مكون، والمخطط البصري واضح واتجاه الأسهم واضح ويظهر الاعتماديات... وهذا يعني أن ما درسناه سابقا من مشاكل لم يكن شيئا عبثيا، وأننا ما أن وضعنا يدنا على المشكلة حتى ظهر لنا الحل، وأن ما درسناه من معلومات نظرية ما هو إلا مفتاح لفهم الحلول ومعرفة آلية وكيفية تطبيقها، والقدرة على إيجاد المشكلة لحلها ^^

خلاصة مهمة بعد كل هذه الرسومات والخطوات:
إن الهدف النهائي والأسمى من كل هذا هو معالجة "الانفصام المعماري" الشائع في مخططاتنا الآن؛ وهي الفجوة بين المنظور المنطقي -كيف يفكر المبرمجون في المكونات ويرسمونها كمربعات على اللوح-، وبين المنظور الفيزيائي -كيف تبدو الشيفرة البرمجية الحقيقية داخل بيئة التطوير-.. لذلك، يجب أن تكون المكونات -component- أشياءً حقيقية، وملموسة، وظاهرة بوضوح -Evident- في هيكل الشيفرة البرمجية، بحيث إذا فتحت الشيفرة البرمجية ونظرت إليها شعرت وكأنني بطريقة ما أنظر إلى المخطط، ولا يجب أن تكون المكونات مجرد مفاهيم منطقية خيالية؛ نستخدمها للتباهي في غرف الاجتماعات ثم نكتب شيفرة برمجية لا تمت لها بصلة! فالمعمارية الحقيقية لا تكذب على الشيفرة البرمجية!

خلاصة الخطوات التي قمنا بها للانتقال من المستوى الرابع -مستوى الشيفرة البرمجية- إلى المستوى الثالث -مستوى المكونات- هي:

1. لدينا مخطط UML كامل، يحتوى كل ال classes وال Interfaces، فيه فوضى بصرية كبيرة ومستوى عال من التعقيد
2. نقوم بحذف ال classes المتعلقة بال utils وال model/domain من هذا المخطط، فتصير المخططات المعمارية مرئية، ولا يزال لدينا شيفرة برمجية، وما زلنا في المستوى الرابع.
3. نبدأ مرحلة التجميع بحيث نضع كل Implementation class وال interface معا في مكون واحد... وهنا نكون انتقلنا من المستوى الرابع إلى المستوى الثالث... لكن لدينا ضوضاء بصرية مرتبطة بالشيفرة البرمجية ما زالت موجودة.
4. نقوم بحذف جميع التفاصيل غير المهمة والمتعلقة بالشيفرة البرمجية داخل المكونات نفسها، كما نقوم بحذف ال classes المجردة أو المساعدة التي تستخدم مع أكثر من مكون.

## أين اختفت الوحدات -Modules- والأنظمة الفرعية -Subsystems-؟

يطرح Simon إجابة لتساؤلات يمكن أن تسأل من المبرمجين ذوي الخبرة أو ممن قرأ كتابا من الكتب الكلاسيكية من كتب المعمارية (مثل كتاب Software Architecture in Practice)، والتساؤل الذي قد يطرحه البعض هنا: هناك مصطلحات شهيرة جدا تم تغييبها

من هرمنا المعماري في نموذج C4؛ وعلى رأسها Modules و Subsystems، فما هو سبب ذلك؟

أولا: استبعاد الوحدات (Modules)

بشكل أكاديمي، يُعرّف الModule بأنه "وحدة تنفيذية" دقيقة -مثل مكتبة برمجية- تتجمع مع غيرها لإنشاء المكون -Component-، ورغم صحة هذا التعريف، إلا أن إقحامه في المخططات يضيف طبقة من التفاصيل الدقيقة والمزعجة التي لا طائل منها، تحديدا عند محاولة فهم "الصورة الكبيرة" للنظام! لذلك، يتجاهل نموذج C4 هذه المعلومات التفصيلية الدقيقة، ويفضل التركيز على المكونات الشمولية الأكبر حجماً (Coarser-grained components) لتجنب التشويش البصري... أي أننا قمنا بالتضحية بالدقة الأكاديمية التفصيلية لصالح الفائدة العملية في مشروعاتنا التقنية.

ثانيا: استبعاد الأنظمة الفرعية (Subsystems)

يُستخدم هذا المصطلح غالباً للإشارة إلى "تجمع وظيفي لعدة مكونات -component-"، لكن المشكلة الكبرى مع "النظام الفرعي" هي أنه يفشل في اجتياز اختبار الواقعية! فهو غالبا ما يكون مفهوما بشكل منطقي -Logical Concept-؛ لكن يصعب إسقاطه ككيان ملموس داخل الشيفرة البرمجية الحقيقية -Codebase-، وبما أننا اتفقنا أن المعمارية الحقيقية يجب أن تكون ظاهرة ومرئية في الشيفرة البرمجية؛ فقد تم استبعاد أي مصطلح لا يمكننا إثبات وجوده الفيزيائي ^^... وقد أشرنا لذلك سابقا ^^

ثالثا: ما الذي يريده الكاتب من خلال طرح هذا الموضوع؟
إن الكاتب يريد إيصال فكرة مهمة، وهي أن نموذج C4 وُجد لتبسيط حياتك لا لتقييدها ^^، فإذا كان الفريق يعتمد بشدة على مفاهيم مثل الأنظمة الفرعية -Subsystems- أو الوحدات -Modules-، وكنتم قادرين على تمثيلها بوضوح داخل هيكل المشروع، فلا حرج إطلاقا من إضافتها إلى قاموس الفريق واعتمادها! فالمعمارية أداة لخدمتك، ونموذج C4 ليس سجنا لأفكارك...

## ما هو موقع ال Microservices في هرم ال C4؟

في ظل الهالة الإعلامية والاندفاع الكبير -Hype- المحيط في عالم تطوير البرمجيات اليوم، من الضروري أن نكون واضحين جداً حول كيفية وصف ال Microservices وكيفية إضافتها إلى داخل المعمارية، فأين سنضع ال Microservices؟ وهل هي نظام بذاته؟ أم هي مجرد حاوية -container-؟ أم هي مجرد مكون -component-؟

الإجابة ببساطة تعتمد على: "نطاق ملكيتك وسيطرتك"، وهذا يعني:

**أولا: ال Microservices كأنظمة برمجية:**

إذا كان نظامك يعتمد على Microservices تقع "خارج نطاق سيطرتك" -على سبيل المثال: تم بناؤها وتُدار من قبل فريق منفصل في شركتك، أو عبر طرف ثالث-، فيجب عليك أن تعامل هذه الخدمات "كأنظمة برمجية خارجية"! فهي بالنسبة لك مجرد "صناديق سوداء" تقع في قمة الهرم (Level 1) ولا شأن لك بتفاصيلها الداخلية ^^.

**ثانيا: ال Microservices كحاويات -Containers-:**

على الجانب الآخر، إذا كانت هذه ال Microservices جزءاً من النظام الذي تبنيه أنت وتتحكم في شيفرته البرمجية، فهنا نطبق على هذه ال Microservices قاعدتنا الذهبية التي تحدثنا عنها سابقا: "كل كيان قابل للنشر بشكل مستقل فهو حاوية"، وهذا يعني أن كل Microservices تمثل Containers -Level 2-، وأي قاعدة بيانات تستخدمها هذه ال Microservices هي أيضاً حاوية -container- منفصلة أخرى.

وهذا السلوك يشبه تماماً ال Monolith، فال Monolith يمثل حاوية -Container- تسكن بداخلها الكثير من المكونات -Components-، في حين أن ال Microservices هي أيضاً حاوية، ولكن يسكن بداخلها عدد أقل من المكونات -Components-، وعدد هذه المكونات الداخلية يعتمد على استراتيجيتك البرمجية، فقد تتكون ال Microservices البسيطة جداً من مكون داخلي واحد فقط، وقد تكون معقدة قليلاً لتشمل عدة مكونات مرتبة بأسلوب الطبقات أو المعمارية السداسية...

ملاحظة: يعد ال Hype واحد من أهم المصطلحات المستخدمة في عالم التقنية اليوم، وترجمته الحرفية تعني "الضجيج" أو "المبالغة الإعلامية"، لكن معناه أعمق من ذلك، ففي عالم البرمجيات اليوم، إذا ظهرت تقنية جديدة أو أسلوب معماري جديد واستخدمته أو اعتمدته أو تبنته واحدة من كبرى الشركات فإن ذلك يحدث "انفجار في الشعبية" لهذه التقنية أو الأسلوب، ويصبح الجميع يتحدث عنه في المؤتمرات، وتكتب عنه آلاف المقالات، ويظن

المبرمجون والمدراء أنه "العصا السحرية" التي ستحل كل مشاكلهم! هذه الهالة، وهذا الاندفاع الجماعي المبالغ فيه لتطبيق التقنية -حتى في المشاريع التي لا تحتاجها أصلا- هو ما يسمى بال Hype، وإذا كان ال Hype قبل بضع سنوات عن ال microservice وال blockchain؛ فإن ال Hype اليوم هو الذكاء الاصطناعي بكل تأكيد...

ملاحظة ٢: حقيقة وُفق الكاتب في كتابة ال Hype هنا في حديثه عن المعمارية، لأن الجميع في مرحلة ما أراد تمزيق نظامه وتحويله إلى Microservices لمجرد اللحاق بالترند^^، مما خلق حالة من "الضجيج والتشويش" في كثير من المفاهيم، لذلك، ستفهم من هذه الإضافة وكأنه يقول: "بسبب هذا الضجيج وكثرة الكلام والتنظير حول ال Microservices، دعونا نوقف هذا التشويش ونضع النقاط على الحروف بشكل علمي ومدروس، ونحدد بالضبط ما هي ال Microservices في قاموسنا المعماري البسيط وكيف سنقوم بتمثيلها بشكل واقعي وليس ترند".

الخلاصة التي نريد أن نصل لها في هذا الموضوع:
إن ال Microservices ليست شيئا جديدا في نموذج C4، فهي بالنهاية مجرد Container فيها Component...

## ماذا عن تقنيات ال Serverless؟

إذا كان نظامك البرمجي يعتمد على تقنيات "الحوسبة دون خادم مثل: AWS Lambda، فإننا نعاملها بنفس المنطق الذي عاملنا به ال Microservices تماما، والسبب في ذلك يعود أيضا لقاعدتنا الذهبية = "قاعدة النشر المستقل"، فإذا كانت هذه الدالة قابلة للنشر بشكل مستقل، فإن كل دالة سحابية نقوم ببرمجتها تُنشر وتعمل بشكل مستقل هي Container، فمثلا لو كان لديك دالة Lambda لرفع الصور ودالة Lambda لإضافة علامة مائية ودالة Lambda لتخزين الصور على s3، فإن ذلك يعني أن لديك ثلاثة Containers...
هذه القاعدة الجميلة -قاعدة النشر المستقل-، تجعل من نموذج C4 قادرا على استيعاب أي تقنية جديدة تظهر في المستقبل دون الحاجة لتغيير المفاهيم...

## أين نضع المنصات -Platforms- وأطر العمل -Frameworks- والمكتبات -Libraries- في نموذجنا المعماري؟

بعد أن حددنا شكل المكونات -components- والحاويات -containers-، قد يتبادر إلى ذهنك سؤال في غاية الأهمية: أين نضع المنصات التي نبني عليها أنظمتنا مثل .NET أو JVM؟ أو أطر العمل مثل Spring أو Laravel؟ أو المكتبات البرمجية التي نستخدمها يومياً مثل مكتبات تسجيل الأخطاء Logging أو ال ORM؟

والجواب على هذه الأسئلة بسيط جدا، إن ما ذكرناه يمثل أدوات تنفيذية، وبذلك هي ليست مكونات! وهي بذلك لن يتم رسمها داخل نموذج C4، لأنها مجرد خيارات تقنية

وتفاصيل تنفيذية داخل المكونات الخاصة بي ^^، فعلى سبيل المثال، في تطبيق PetClinic، تم الاعتماد بشكل كبير على إطار عمل Spring، لكنك لن تجد في المخطط المعماري مربعا خاصا يحمل اسم Spring! الأمر ذاته ينطبق على ال Database Drivers ومكتبات التشفير وغيرها.

وهنا يظهر لنا جليا أهمية مبدأ التغليف -Wrapping- والذي يعد واحدا من أهم قواعد التصميم الجيد... لأن المعمارية النظيفة والجيدة ستشجعك دائماً على تغليف -Wrapping- هذه المكتبات وأطر العمل الخارجية وإخفائها داخل مكوناتك الخاصة التي صممتها بنفسك، وهذا سيقودنا لمصطلح مهم مرتبط بهذا الكلام وهو ال Decouple، وترجمته الحرفية "فك الاقتران"، والمراد بهذا المفهوم هو فصل الشيفرة البرمجية الخاصة بنا عن التقنيات الخارجية بحيث لو أردنا تغيير هذه التقنيات الخارجية لاحقا سنتمكن من ذلك دون الحاجة لإعادة كتابة كل شيء! وهذا سيقدم لنا فائدتين أساسيتين وهما:

١. تأجيل القرارات -Defer Decisions-: فلا تلتزم بتقنية بشكل مبكر.

٢. تغيير التقنيات -Change Technologies-: يمكن استبدال مكتبة بأخرى بكل سهولة.

مثال: تخيل أنك تبني نظاماً للتجارة الإلكترونية، وقمت باستدعاء مكتبة شركة Stripe للدفع بشكل مباشر في كل أجزاء نظامك، لو قررت الإدارة استبدال Stripe بشركة PayPal لسبب ما، ستضطر لإعادة كتابة آلاف الأسطر البرمجية! وهذا ما لا نريده، لذلك فالتصميم الجيد يقوم على إنشاء واجهة خاصة بنا نسميها مثلا IPaymentProcessor، ونجعل كل الشيفرة البرمجية داخل نظامنا تتعامل مع هذه الواجهة فقط، ثم ننشئ class "يُغلف" مكتبة Stripe وينفذ هذه الواجهة، في هذه الحالة لو أردنا الانتقال إلى PayPal لن يتطلب منا

ذلك سوى إنشاء class جديد دون الحاجة للمساس بالشيفرة الأساسية للنظام ^^، وبالمناسبة هذه العملية تشبه فكرة بعض ال Design pattern التي أتت لحل مثل هذا النوع من المشاكل مثل ال Adapter -والذي يمثل النمط الدقيق للتعامل مع الحالة التي وصفناها-، وال Facade... يمكنك الاطلاع على هذا المثال لمزيد من المتعة ^^.

ملاحظة مهمة: هناك استثناء لهذه القاعدة:

حتى نكون واقعيين، هناك حالات قد يعتمد فيها النظام بشكل مباشر وجوهري على مكونات جاهزة توفرها المنصة -Platform- باعتبارها جزءا من المعمارية، في هذه الحالات، إن فهمك الدقيق لـ "طبيعة وكيفية استخدامك" لهذه الأدوات هو الذي سيحدد ما إذا كانت ترتقي لتُصبح مرئية في مخططك المعماري ذو البنية الثابتة، أم أنها ستبقى مجرد أدوات وتفاصيل خفية! ومن الأمثلة على ذلك، لو استخدمت داخل مشروعك Embedded Search Engine للبحث داخل النصوص، وهي إحدى الخصائص المهمة في مشروعك، فلا يمكنك اعتبار المكتبة المستخدمة لهذا الغرض مجرد أداة تنفيذية، بل هي هنا Component داخل معماريتنا، ويتم إضافتها داخل صندوق أزرق ويكتب بوصفه بشكل واضح أنه بني من خلال المكتبة س.

# فائدة

كلما رأى الإنسان نفسه معرضًا عن تدبر القرآن، أو عن بعض معانيه، ثم تذكر قوله تعالى:
{وَلَوْ عَلِمَ اللَّهُ فِيهِمْ خَيْرًا لَّأَسْمَعَهُمْ} يجف ريقه من الهلع لا محالة!

- كتاب الطريق إلى القرآن، إبراهيم السكران -فرج الله عنه-

## اصنع قاموسك الخاص: الجوهر أهم من المسميات!

هذا الموضوع يعد خاتمة الفصل الثاني من هذا الكتاب وخلاصته، وهو يدور حول هذا السؤال، هل ما تحدثنا عنه من مصطلحات -container, component- وغيرها هي تعريفات عالمية يجب استخدامها من قبل جميع الشركات؟

والجواب بكل بساطة، لا يا صديقي، فنحن لسنا ديكتاتوريين ^^، لذلك يقول Simon: "هذه المصطلحات نجحت مع أغلب الشركات التي عملت معها، لكنها ليست قانوناً عالميا، فإذا كانت شركتك تستخدم بالفعل مصطلحات متجذرة في ثقافتها، فمن الأسهل والأذكى أن تُبقي على مصطلحات الشركة بدلاً من إجبار مئات المبرمجين على حفظ مصطلحات جديدة، بشرط واحد فقط: أن تكون هذه المصطلحات مُعرّفة بدقة ومفهومة للجميع".

إذا من الفقرة نستنتج ما يلي:

١. يمكننا استخدام المصطلحات التي نرغب بها للتعبير عن الأجزاء المختلفة، فمثلا يمكنني استخدام Model مكان ال Component، وال Component مكان ال Container... وهذا لن يؤثر على المخطط المعماري! طالما أننا متفقون على ذلك.

٢. أن تكون المصطلحات معرفة صراحة -explicitly defined-.

٣. أن تكون مفهومة من الجميع -understood by everybody-.

الخلاصة:

المشكلة ليست في اختيار كلمة معينة، بل في **عدم الاتفاق على معانيها.**

# الفصل الثالث: نموذج C4!

بعد كل المقدمات التي تحدثنا عنها بالجزء الأول من هذه السلسلة، وبعد أن تحدثنا في الفصلين الأول والثاني عن المشاكل الشائعة ومشكلة المصطلحات ننتقل الآن إلى نموذج C4 بشكل عملي، وهو القلب النابض لهذا الكتاب كله، وكل ما سبق ما كان إلا مقدمات حتى نصل لهذا الفصل الذي سيظهر فيه الأثر المعماري لكل ما تعلمناه... وفي هذا الفصل سيتم تقديم نموذج C4 كنظام متكامل للمرة الأولى، وسنتعرف على المستويات الأربعة كنظام متماسك، متأثرا ومرتبطا بمبدأ Shneiderman's mantra... لن نستعجل الحديث، فالأمور الجميلة قد بدأت لتوها ^^

**سبب التسمية ^^**

يرمز اسم C4 إلى الكلمات الأربع التي تبدأ بحرف الـ C باللغة الإنجليزية: (Context, Containers, Components, Code)، وهي المستويات الأربعة التي تحدثنا عنها...

وحتى نفهم هذا النموذج ببساطة، تخيل أنك تستخدم تطبيق Google Maps للبحث عن مبنى معين، حيث ستقوم بعملية تكبير -Zoom in- تدريجية عبر أربعة مستويات:

**المستوى الأول: سياق النظام System Context - صورة الكوكب:**

هذه هي نقطة البداية. هنا نحن لا نهتم بأي تفاصيل تقنية إطلاقاً، فالمخطط في هذا المستوى يوضح لك "الصورة الكبيرة"؛ أي كيف يندمج نظامك البرمجي مع العالم من حوله؟ ومن هم المستخدمون الذين يتفاعلون معه؟ وما هي الأنظمة الخارجية التي يرتبط بها؟

**المستوى الثاني: الحاويات Containers - خريطة الدولة:**

الآن نقوم بعمل Zoom in لندخل إلى داخل نظامنا البرمجي، وهنا تبدأ المعالم التقنية الكبرى بالظهور! فالمخطط يوضح لك "الحاويات -Containers-" التي يتكون منها النظام (مثل: تطبيق الويب، تطبيق الهاتف، وقاعدة البيانات)، وكيف تتواصل هذه الحاويات مع بعضها البعض.. هذا المستوى يعطيك نظرة شاملة للهيكل التقني عالي المستوى.

**المستوى الثالث: المكونات Components - خريطة الحي:**

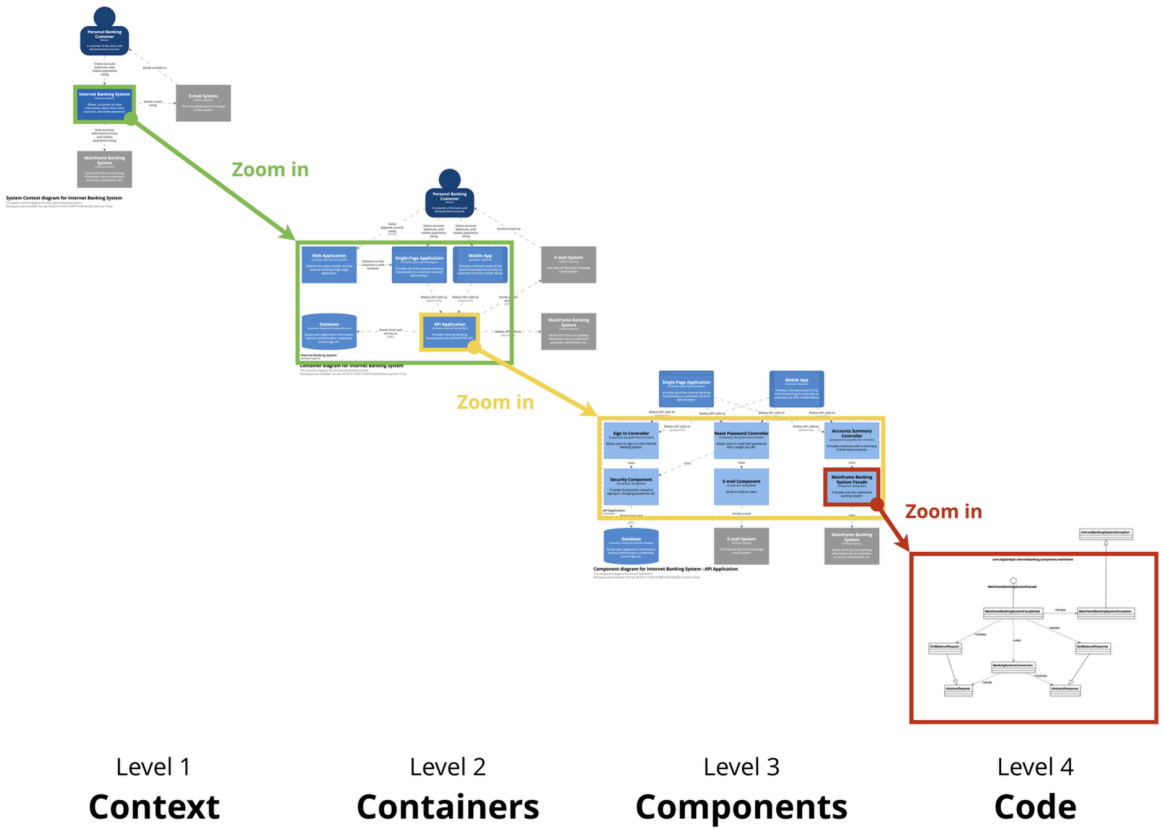
نستمر في الغوص، ونقوم بعمل Zoom in داخل "حاوية واحدة فقط"، مثلاً: تطبيق الويب.. المخطط هنا سيكشف لنا الأقسام الوظيفية والمكونات -Components- التي تعيش داخل هذه الحاوية، وكيف تتبادل البيانات داخلياً ^^

**المستوى الرابع: الشيفرة البرمجية Code - الشارع والمباني:**

هذا هو أعمق مستوى ممكن لـ Zoom in، حيث ندخل إلى داخل "مكون واحد فقط"؛ لنرى كيف تمت كتابته برمجياً... ونستخدم هنا المخططات التفصيلية مثل UML Class Diagram -وهو يعد الأشهر استخداما هنا، ويعد من الاستخدامات الجيدة لل UML- لنرى الكلاسات والواجهات -Interfaces-، وفي الواقع العملي، نادراً ما نحتاج لرسم هذا

المستوى إلا للأجزاء المعقدة جداً من النظام... فهذا المستوى اختياري، تضاف تفاصيله بناء على الحاجة، وتستخدم الأدوات لهذا الغرض، فالشيفرة البرمجية موجودة، فلا تضع وقتك في رسم ما يمكن للأدوات قراءته ورسمه! -مثال العيادة البيطرية إنموذجا-

شاهد الصورة أدناه:




**A summary of the C4 model**

بناء على ما ذكرناه، يمكننا القول أن نموذج C4 ليس مجموعة من المخططات المنفصلة، بل هو عبارة عن خريطة واحدة لنفس النظام، ولكننا نتجول فيها بمستويات تقريب -Zoom-

مختلفة؛ لنعطي لكل شخص في فريق العمل المخطط الذي يناسب احتياجاته وبمستوى التفاصيل الذي يحتاجه، سواء أكان هذا الشخص مديراً، أو معمارياً، أو مبرمجاً... إلخ.

## الخرائط الهرمية ومبدأ شنايدرمان -Shneiderman-

بن شنايدرمان -Ben Shneiderman- هو عالم حاسوب أمريكي، يعد أحد رواد مجال تفاعل الإنسان مع الحاسوب وال Data Visualization، ولقد صاغ "المانترا" الشهيرة الخاصة به عام 1996، ومانترا هي بالأصل كلمة سنسكريتية تستخدم للدلالة على جملة أو عبارة أو شعار يتكرر، وهي ترتبط ببعض الممارسات البوذية أو الهندوسية -والعياذ بالله منها ومن كل شرك-، أما في المجال التقني فقد اقتبست دلالتها فصارت تشير إلى: "مبدأ مختصر أو قاعدة أساسية يكررها الناس للتذكير بفلسفة معينة"، وهذا ما وضعه شنايدرمان في المانترا الخاصة به، وهي: "**النظرة العامة أولاً، ثم التكبير والتصفية، ثم التفاصيل عند الطلب!**"... هذه المانترا تم تبينها من قبل Simon لنموذج ال C4، والسؤال كيف تم ذلك؟ أو ماذا يقصد بذلك؟

للإجابة على هذا السؤال، دعونا نقسم هذه المانترا إلى:

**1. النظرة العامة أولاً -Overview first-:**

النظرة العامة = المستوى الأول System Context والمستوى الثاني ال Container، فعندما تفتح خريطة النظام لأول مرة، يجب ألا ترى أي تفاصيل تقنية معقدة أو شيفرات برمجية مزعجة.. فالمخطط يجب أن يعطي نظرة بانورامية عامة ومريحة للنظام وتفاعلاته مع المستخدمين والأنظمة الخارجية.

**2. التكبير والتصفية -Zoom and filter- النتيجة = L3 Components:**

هنا يأتي دور المستوى الثاني والثالث وهما ال Containers & Components، فإذا أردت معرفة المزيد، ستقوم بـ "تكبير" الصورة للدخول إلى النظام (Zoom in container حتى نشاهد ال Components الخاصة بها)، وفي نفس الوقت ستقوم بـ "تصفية -Filter-" الضجيج بإخفاء الأجزاء التي لا تهمك حاليا، فمثلا: سنقوم بالتركيز على معمارية تطبيق الويب وتقسيماته الداخلية، و ستتجاهل تفاصيل قاعدة البيانات... أي باختصار قمنا بتكبير ما يهمنا من الخريطة المدينة للوصول للحي الذي نريد.

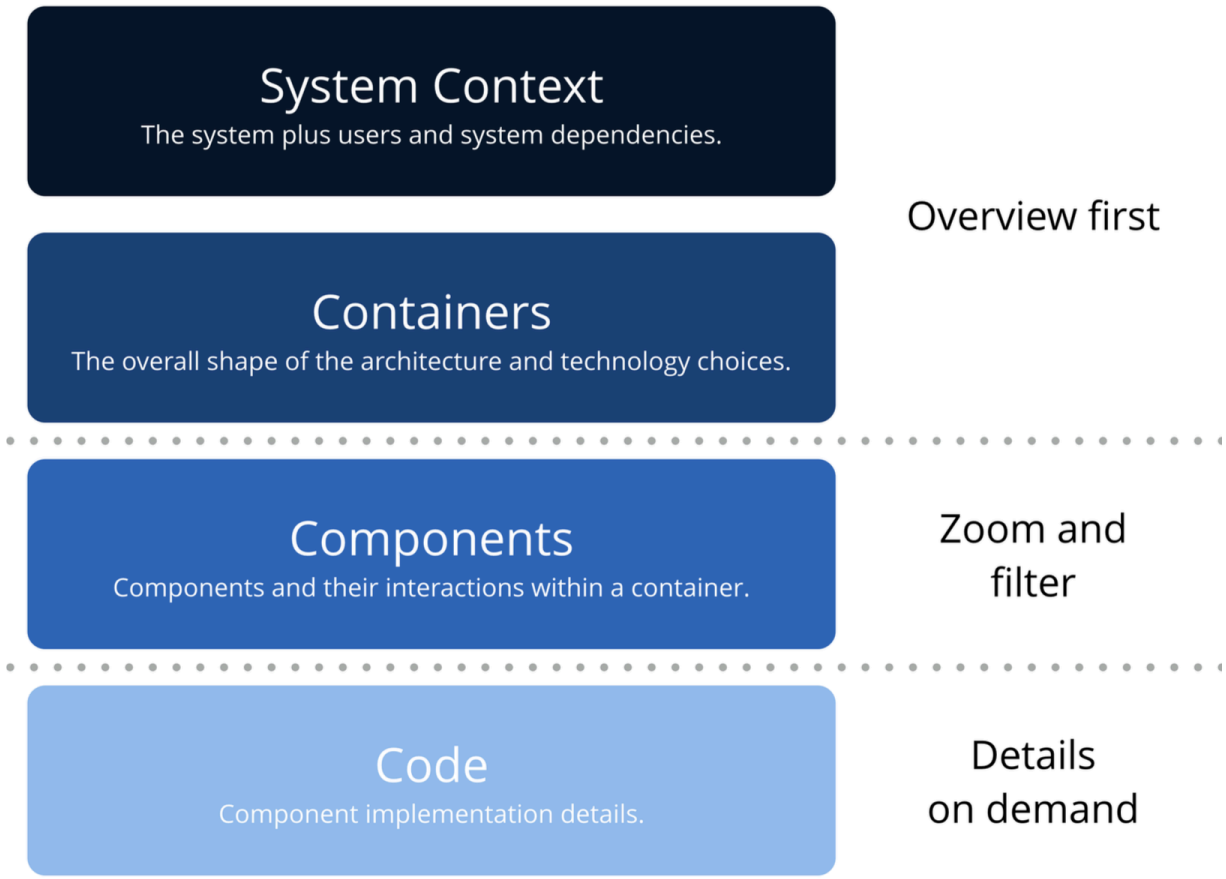
**3. التفاصيل عند الطلب -Details-on-demand-:**

هذا هو المستوى الرابع والأخير = مستوى الشيفرة البرمجية، بحيث يتم عرض التفاصيل الدقيقة مثل الكلاسات، الواجهات، وأنواع البيانات ونحو ذلك، هذه التفاصيل لا يجب أن تُعرض أبداً في البداية لأنها ستصيب المشاهد بالعمى البصري المعرفي! لذلك، تبقى هذه التفاصيل مخفية، ولا تُستدعى وتُعرض عبر مخططات UML أو غيرها إلا "عند الطلب"، أي عندما يحتاج المبرمج فعلياً للغوص في قاع المكون -component- لمعرفة كيف كُتبت الشيفرة البرمجية.

باختصار، **ابدأ بالصورة الكبيرة ثم كبر على ما يهمك ثم اعرض التفاصيل عند الحاجة فقط،** وطريقة التفكير هذه رائعة لأنها ستقودك للطريق الصحيح لتحقيق الفهم المراد بأسرع وأفضل طريقة ممكنة، فلو كان لديك مشروع معقد وكبير، لا يجب أن تبدأ بالتفاصيل! بل يجب أن تبدأ بفهم الصورة الكبيرة ثم تتدرج تدريجيا إلى العناصر التي تهمك شيئا فشيئا...

ولعل عدم تطبيق هذه القاعدة تنتج لنا واحدة من أشهر المشاكل عند انضمام عضو جديد للشركة أو الفريق، فتراه مباشرة يفتح محرر كتابة الشيفرات البرمجية ويبدأ بالعمل، مع أن الأصل أن يبدأ من الصورة الكبيرة للمشروع، ثم يقوم بالتكبير شيئا فشيئا، ولعل هذه النقطة المهمة يجب أن يعيها المسؤولين أيضا، فهذا لا يعد وقتا ضائعا، بل إهدار هذه الخطوة هي الوقت الضائع! وكمثال من الواقع العملي، لو أردت السفر لأغراض السياحة لدولة ما، فهل تبدأ بالبحث داخل جميع الشوارع والأحياء على الخريطة؟ أم أنك تبدأ بتحديد الدولة التي ترغب بزيارتها ثم تختار المدن والأحياء التي ترغب بزيارتها، ثم تنظر للشوارع التي ستستخدمها أو ستسكن فيها وهكذا؟ الجواب واضح بكل تأكيد ^^.

والآن شاهد الصورة التي تمثل ما تكلمنا عنه:




**Shneiderman's mantra**

## معضلة الرسم اليدوي (Tooling vs Hand-drawing)

هذه الفقرة تمثل إضاءة من الكاتب لتجارب مريرة عايشها، فعندما تصل إلى المستوى الثالث -المكونات-، ستجد أن عدد المربعات والأسهم أصبح كبيرا جدا، والقيام برسم كل هذا يدوياً وتحديثه كلما تغيرت الشيفرة البرمجية سيصبح مهمة مملة ومرهقة جدا -Tedious-..

لذلك، ينصح الكاتب بشدة أن تتوقف عن الرسم اليدوي في هذه المستويات العميقة، وأن تبدأ بالبحث عن "أدوات برمجية" تساعدك على أتمتة هذه العملية؛ بحيث تقوم الأداة بقراءة

الشيفرة البرمجية واستخراج مخطط المكونات منه تلقائيا، لضمان أن المخطط يطابق الواقع دائما ولا يكذب! وسنتحدث عن هذا الموضوع بالتفصيل في فصل خاص به -بإذن الله تعالى-.

ملاحظة مهمة: إن نموذج C4 **لا يمثل "عملية تصميم" -Design Process-** تخبرك كيف تبني نظامك خطوة بخطوة! بل هو مجرد "مجموعة من المخططات" التي تستخدمها لوصف وتوثيق ال static structure لنظام تم أو يتم تصميمه بالفعل! هناك فرق كبير بين عمل الشيء ووصفه!

### التنقيب البرمجي -Software Archaeology-

هل سمعت من قبل بال Software Archaeology؟ إن هذا المصطلح ترجمته الحرفية هي: "علم آثار البرمجيات".. هذا المصطلح هو وصف لعملية فهم نظام قديم كتبه آخرون أو نسيته أنت ^^، كمثل عالم الآثار الذي يحفر طبقات الأرض ليفهم التاريخ المرتبط بحضارة ما، فإن المطورين يحفرون طبقات الشيفرة البرمجية والملفات والمستندات القديمة لفهم "الحضارة الرقمية" التي بناها مبرمجون غادروا الشركة منذ سنوات -غالبا-!

في كثير من الأحيان، لن تكون أنت من رسم المعمارية من الصفر، بل غالبا ما يُطلب منك دراسة نظام قديم لا يملك أي وثائق أو رسومات، وهنا تتحول مهمتك إلى عالم آثار برمجي وظيفته التنقيب في الشيفرة البرمجية؛ وحينها ستقضي وقتك في دراسة الشيفرات البرمجية لتفكيك رموز الحضارة القديمة، وفي هذه اللحظات، يصبح نموذج C4 أحد أسلحتك القوية؛

فأنت لا تقوم برسم المخططات لتوثيق فكرتك الخاصة، بل تستخدم مستويات التجريد الأربعة -Context, Containers, Components, Code- لتفكيك النظام الحالي وفهم "كيف يعمل فعليا" وليس "كيف كتب في الأوراق القديمة إن وجدت!"... وقد تتساءل: لم أفهم ما علاقة ال C4 بهذا التنقيب؟ وهنا سأقول لك يا صديقي، واضح إنك مش مركز معنا أو أنك لم تقرأ معنا الكتاب من أوله ^^، الجواب ببساطة لأن عملية التقنيب تبدأ من سطح الأرض ثم يتم الحفر طبقة بعد أخرى، وهذا التجريد هو ما نستخدمه أيضا من خلال C4، ألم نقل أن خطوات التجريد أو خطوات البدء بفهم أي نظام أو بناء أي مخطط تبدأ من ال Context، وبعد ذلك نعمل zoom in... فإذا قمت بالقيام بهذه العملية = وصلت لما ترجو بإذن الله بأقل الخسائر وبأسرع الطرق.

وهنا تجدر بنا الإشارة للفتة جميلة، فمصطلح تنقيب لم يأتي عبثا، بل جاء بشكل مدروس، لأن الآثار لا تأتي مع دليل يشرح ما تراه أو ما تجده، وهذا يقودنا إلى التحديات الأساسية التي ستواجهها وستدعوك للقيام بهذه العملية:

- **غياب الوثائق**: غالباً ما تكون المستندات المعمارية مفقودة أو قديمة جداً ولا تطابق الواقع.
- **تغير الأجيال**: الأشخاص الذين كتبوا الشيفرة البرمجية قد لا يكونون متاحين للسؤال، أو أنهم نسوا تفاصيل المشروع أساسا ^^.
- **البناء الطبقي**: ستجد طبقات من الشيفرات البرمجية بعضها فوق بعض! شيفرات برمجية قديمة بلغة قديمة، وفوق ذلك كله "رُقع" كثيرة وتحديثات أضيفت عبر سنوات...

حسنا، يكفي هذا القدر من التفاصيل حول هذا المصطلح، والآن دعونا ننتقل لبعض الإضائات التي أشار إليها Simon حول نموذج C4 قبل الانتقال للفصل التالي:

- إن كل مستوى من مستويات ال C4 يناسب جمهورا مختلفا، أي أنك يمكنك اختيار المستوى المطلوب رسمه ومن ثم عرضه للمستخدمين بحسب طبيعتهم، وهذا العرض ليس لازما أو قانونا، بل هي مجرد إضاءات يمكن أن تساعدك:
  - المستوى الأول: يناسب الجميع، تقنين وغير تقنيين، وعادة ما يكون المستهدفون هنا هم المدراء والعملاء...
  - المستوى الثاني: يناسب التقنيون، أكانوا معماريين أو مطورين أو DevOps.. إلخ
  - المستوى الثالث: يناسب المطورين العاملين على ال Container الخاصة بهذا ال component.
  - المستوى الرابع: يناسب المطورين العاملين على المكون -component- نفسه.. وهذا نادرا ما نحتاجه.
- نموذج C4 ليس منهجية تصميم مثل RUP أو SSADM، بل مجرد مجموعة مخططات لوصف ال static structure، أي أنه لا يخبرك كيف صمم نظامك، بل يخبرك كيف تُصور نظامك بعد تصميمه أو أثناء تصميمه.
- ليست كل المستويات إلزامية ^^: إن المستويات التي تحدثنا عنها من ١ إلى ٤ ليست جميعها إلزامية، وهذه إحدى المميزات الخاصة بنموذج C4 مقارنة مثلا بال RUP، لأنها تمس الحياة الواقعية، فنحن لا نطلب التوثيق الشامل هنا - مثل RUP-، بل

نطلب التوثيق الكافي فقط ^^... بناء على ذلك، يوصي Simon بإنشاء المستوى الأول والثاني دائما، في حين يخبرك بالتفكير مليا عند الجزء الثالث، وسؤال نفسك، هل الفائدة تستحق التكلفة؟ أما المستوى الرابع فهو نادرا ما يستحق أن يكون موجودا... إن السبب وراء هذا التحفظ هو أن مخططات المكونات والشيفرة البرمجية تتقادم بسرعة بسبب تغير الشيفرة البرمجية باستمرار، وصيانتها اليدوية مكلفة ومتعبة، وإذا لم تصن بشكل جيد وقعنا بفخ التضليل ^^ ودخلنا في مشكلة ال Model-Code Gap... وسنتحدث عن هذا لاحقا بإذن الله، لذلك نستخدم هذه مع الأجزاء المعقدة أو المهمة والتي نحتاج لعرضها فعلا ^^، وهذا ينسجم مع ما ذكرناه في الفصل الثاني في الكتاب، أما المستوى الأول والثاني فنحن بحاجتهم دوما ونلزم برسمهم، وعلى الرغم من ذلك فإن هذه المستويات سهلة الرسم، وعظيمة الفائدة، وتعطي الصورة الشاملة لما نعمل عليه.

والآن، بعد أن تحدثنا عن نموذج C4 وفهمنا ماهيته ومكوناته بشكل دقيق، دعونا ننتقل للجانب العملي ونبدأ بتعلم رسم كل مستوى على حدا ^^... وسنقوم باعتماد مثال لتصميم مخطط لنظام مصرفي عبر الإنترنت -Internet Banking System-، فهيا بنا، لقد بدأت الإثارة ^^

## الفصل الرابع: Level 1: System Context diagram

من هنا نعلن الانطلاق الفعلي في رسم معمارية الأنظمة! ومن هنا بكل تأكيد سنبدأ "بالمستوى الأول"، لإنه المستوى الأهم على الإطلاق، فهو الوحيد الذي يقرأه ويفهمه

الجميع، سواء كانوا مبرمجين أو مدراء أعمال، وهو المستوى الذي سيترتب عليه وجود ما بعده...

لقد تحدثنا سابقا عن مخطط سياق النظام -System Context diagram- بشكل عام، لكننا الآن نحتاج للدخول إلى التفاصيل الدقيقة، وكثير من التفاصيل التي سنتعلمها هنا سترحل معنا وسنستخدمها في الفصول القادمة، فركز معي ^^...

## أولا: الهدف المعماري - الأسئلة الثلاث

قبل الشروع برسم أي مخطط، يجب أن نجيب على ثلاثة أسئلة جوهرية تبين النية أو القصد من وراء هذا الرسم، هذه الأسئلة هي:

1. **ما هو النظام الذي نبنيه (أو بنيناه)؟**
2. **من يستخدمه؟** (من يتفاعل معه من البشر؟).
3. **أين يعيش؟** (كيف يندمج هذا النظام مع البيئة التقنية المحيطة به والأنظمة الخارجية؟).

إن إجابة هذه الأسئلة ستعالج مباشرة مشكلة "لا سياق ولا نقطة بداية منطقية" التي كانت من أبرز المشاكل والعيوب في الأمثلة في الفصل الأول من الكتاب...

وهنا لفتة جميلة، وهي لماذا فصل Simon الأشخاص والأنظمة في هذه الأسئلة -سؤال ٢ و٣-؟ الجواب ببساطة لأن المعمارية عندما تتعامل مع شخص سيستخدم النظام فهذا يعني الحاجة لوجود UX يدعم المستخدم ويقدم له واجهات يمكنه التفاعل معها لينفذ ما يحتاجه،

في حين أن الأنظمة التي ترتبط معنا يمكن التواصل معها بشكل خفي دون أن يراه المستخدم، مثل المستخدم يحتاج لصفحة في الموقع الإلكتروني لسحب النقود، وسيصل للمستخدم بريد إلكتروني بعد عملية السحب... الأول يمثل كيف سيتدخل ال UX ليضمن وجود هذه الصفحة وقدرتها على التفاعل مع المستخدم، والثاني يمثل اتصال بين النظام وخدمة خارجية من خلال API -على سبيل المثال- حتى يتم إرسال البريد الإلكتروني... وسنشاهد هذا الأمر بشكل واقعي بعد قليل...

**ثانيا: رفع الأساسات وضبط قواعد الرسم؛ بطل قصتنا يجب أن يكون في قلب البناء!**

كيف سنرسم هذا المخطط؟ بكل بساطة:

قم برسم مربع يمثل نظامك البرمجي، وضعه في "المنتصف / القلب" -لأنه بطل القصة-، ثم أحطه بالمستخدمين (البشر والأدوار والشخصيات) و الأنظمة البرمجية الخارجية الأخرى؛ التي يتحدث معها.

ببساطة = آلية الرسم هي:

1. النظام الذي سنبينه = صندوق في المركز / المنتصف
2. نحيط النظام بالمستخدمين والأنظمة الأخرى التي تتفاعل معه

ملاحظة مهمة :إن التفاصيل التقنية هنا ممنوعة منعا باتا! فهذا المخطط هو "نظرة من الأعلى"، لذلك يجب ألا يحتوي على أي ذكر لأسماء قواعد البيانات، أو بروتوكولات الاتصال -مثل

HTTP أو -TCP، أو لغات البرمجة! إن التركيز هنا ينصب فقط على "البشر" و "الأنظمة"، وبهذا ستحصل على مخطط يمكنك طباعته وعرضه بثقة على مديرك التنفيذي غير التقني وسيفهمه فوراً *-*...

والآن، لنبدأ بالمثال العملي، مثال نظام مصرفي عبر الإنترنت -سيتم طرح المعطيات لغايات المثال-:

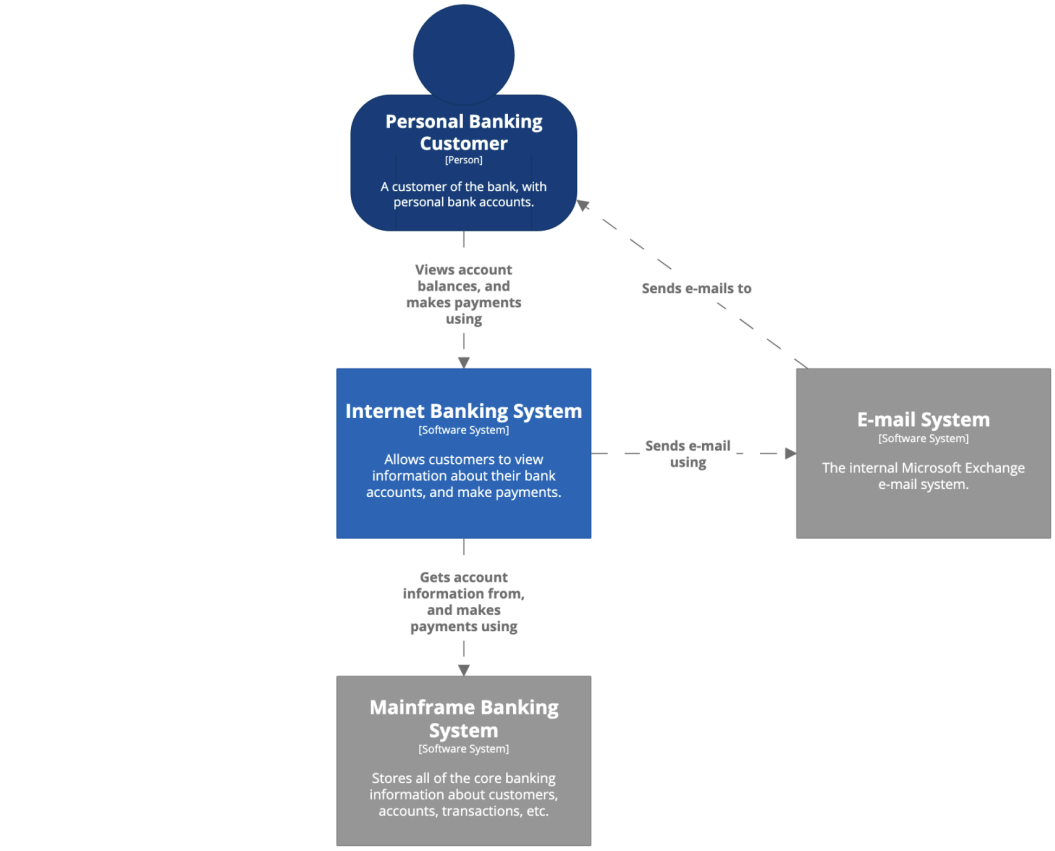
لو أردنا رسمه علينا الإجابة على الأسئلة الثلاث الأولى:

١. نظام مصرفي عبر الإنترنت

٢. المستخدمون هم عملاء البنك الذين يملكون حسابات شخصية لدى البنك

٣. الأنظمة الخارجية التي ستعمل معنا = النظام المستخدم لدى البنك والذي يقوم حاليا بهذه العمليات، ونظام البريد الإلكتروني والذي سنستخدمه لإرسال رسائل تأكيد للعملاء بعد العمليات...

بعد إجابتنا عن هذه الأسئلة سنرسم صندوق النظام في المنتصف، ونرسم صناديق الأنظمة الخارجية حولها، ونرسم المستخدمين... ونربط جميع هذه الأجزاء معا بخطوط تصف مهمتها... حاول رسم هذه الرسمة، ثم انظر للصورة أدناه وشاهد كم هي قريبة أو بعيدة من الصورة...




**A System Context diagram for the Internet Banking System**

جميل جدا، أنت الآن قد رسمت الصورة بنجاح، وأظنك قريب جدا من الرسمة الصحيحة والكاملة للمستوى الأول في نموذج C4 للنظام المصرفي عبر الإنترنت، ومع ذلك، هناك بعض التفاصيل التي أظنها قد هربت منك أو أننا لم نذكرها فأشكلت عليك عندما رأيتها في المخطط... فدعونا نشرح المخطط سريعا ^^

١. قمنا بوضع الصندوق في المنتصف، وقمنا بإعطائه اللون الأزرق لتميزه كعنصر مركزي، وداخل الصندوق كتب اسم النظام، وطبيعته والوصف الخاص به -يتيح للعملاء عرض

معلومات حساباتهم المصرفية وإجراء المدفوعات-، وهذه المكونات هي ما تحدثنا عنها سابقا بالفصل الأول والثاني، وبهذا يكتمل لدينا العرض البصري بطريقة رائعة، وهذه المكونات نفسها ستكون مشتركة مع الصناديق الأخرى.

٢. قمنا برسم صورة رمزية تعبر عن الأشخاص، وهذه ما نستخدمها عادة للدلالة على المستخدمين البشرين الخاصين بأنظمتنا، وبهذا نميز بين الأنظمة والبشر من خلال النظر مباشرة... أما وصفه: عميل البنك، لديه حسابات مصرفية شخصية.

٣. قمنا برسم الأنظمة الخارجية باللون الرمادي، وتم اختيار اللون الرمادي لتمييزه عن النظام نفسه، كما يخفف تركيز النظر عن الروابط الخارجية لتبقى العين البصرية على محور النظام... إن هذه الأنظمة مثل: "نظام البريد الداخلي، والذي وصف بكونه نظام داخلي يستخدم Microsoft Exchange"، ولاحظ الفرق هنا أن اسم النظام وضع أسفل منه كلمة Software system عكس ال Personal Banking Customer الذي كتب تحته Person.

٤. قمنا برسم الأسهم وقمنا بتسميتها، فكل علاقة في المخطط لها اتجاه يحدد المرسل والمستقبل، وتسمية لهذا السهم مثل Sends emails to... ولاحظ أن هذه الخطوة تعالج مشكلة العلاقات الغامضة التي تحدثنا عنها بالفصل الأول... -ملاحظة رسم الخطوط والعلاقات يتم أثناء عملك في الخطوات السابقة، لكنها ذكرت هنا لغايات السرد والشرح-.

٥. لاحظ وجود تفاصيل وصفية آخر المخطط على اليسار، هذه التفاصيل تحمل عنوان المخطط، والوصف الخاص به، وتاريخ آخر تعديل... وهذه الحركة الجميلة ستجعل المخطط أيضا أقوى بصريا ويقدم معلومات مهمة لكل ناظر إليه مباشرة، ويمنع تداخل المخططات... وتأكد أن هناك أدوات ستساعدك على هذه الحركة مثل: Structurizr... *_*، سنتحدث عنها لاحقا بإذن الله، الآن ركز معنا ^^

فقط وبكل بساطة ^^... فمرحبا بكم في عالم المخططات الجميلة ^^

لكن، قد تتسائل ما هي الشروط أو ما هي ماهية ما رسمناه من مربعات وصناديق؟ هل ال Person هو المحاسب أم أحمد؟ وهل النظام الخارجي هو نظام يمكن التعديل عليه أم لا؟ وهل يمكن وضع التقنيات المستخدمة على الأسهم؟...إلخ
وهذا سؤال ذكي ومهم جدا، لذلك دعونا نجيبه بالتفصيل الممل ^^:

**1. البشر -People-:**

ويقصد بهم مستخدمو النظام، ولك الحرية الكاملة في طريقة تسميتهم؛ فيمكنك تسميتهم كأفراد مثل أحمد وخالد، أو من خلال الأدوار الوظيفية مثل محاسب أو مدير، أو من خلال ال Actors، أو من خلال شخصيات اعتبارية -Personas-... لكن المهم هنا هو ماذا ستكتب داخل المربع لهذا الشخص! والمطلوب هنا معلومتين اثنتين: الاسم ويمثل اسم الدور أو الشخص، والوصف ويمثل سطر قصير جدا يشرح مسؤولية هذا الشخص أو سبب استخدامه للنظام...

**2. الأنظمة البرمجية -Software Systems:-**

وهي الأنظمة الخارجية التي يضطر نظامك للتحدث معها حتى ينجز عمله، والقاعدة الذهبية هنا لتحديد ما إذا كان هذا النظام "خارجيا" أم لا هي: **الملكية والمسؤولية.**

فإذا كان النظام مبنياً من قبل فريق آخر، أو شركة أخرى، ولا تملك أنت صلاحية التعديل عليه = فهو "نظام خارجي"، فمثلا، في مثالنا الخاص بالبنك، فريق تطوير "نظام البنك عبر الإنترنت" لا يملك صلاحية تعديل "نظام البريد الإلكتروني" الخاص بالبنك، لذلك تم رسمه كنظام خارجي مستقل، لذلك المطلوب هنا هو كتابة الاسم = والذي يمثل اسم النظام الذي نعتمد عليه، والوصف = والذي يمثل سطر قصير يوضح مسؤولية هذا النظام ولماذا نحتاجه ^^.

وهنا سيظهر سؤال مهم، ما هو الداعي لوجود هذه الأنظمة ضمن المخطط الخاص بنا؟

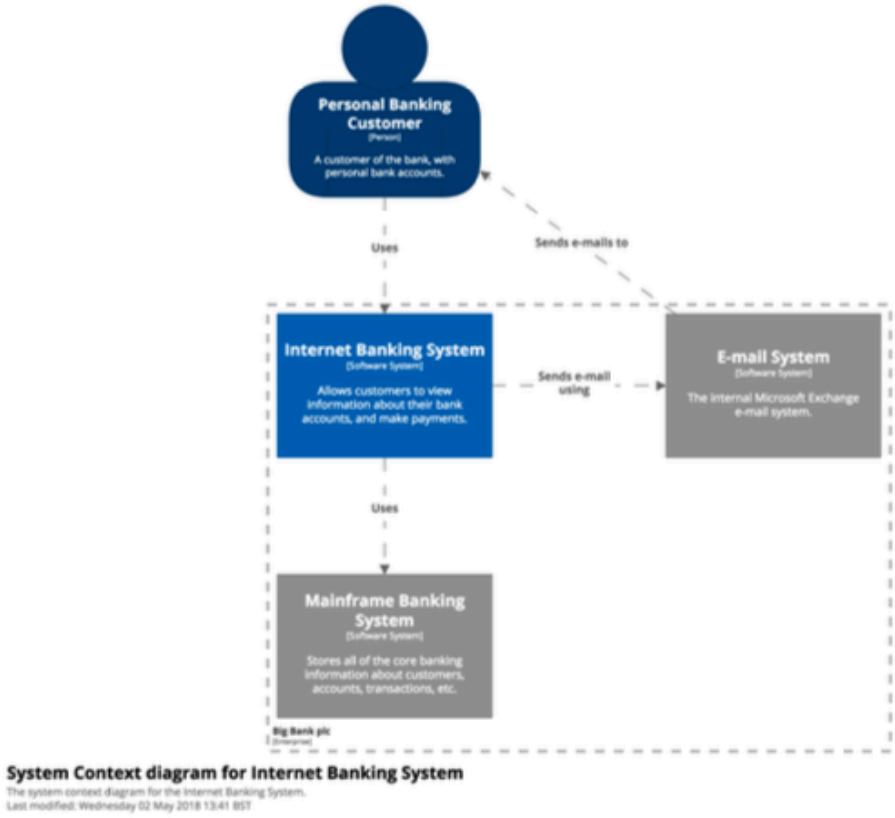
والجواب هو: بسبب الحاجة لمعرفة الاعتماديات -dependencies- الخاصة بالنظام!

والاعتمادية هي أي شيء يعتمد عليه نظامك للعمل، لو فشل = فشل نظامك أو جزء منه، وبهذا تصبح القيمة الاستراتيجية لهذه الأنظمة هي: عند التطوير سنعرف بمن سنتواصل لربط الأنظمة، وعند التشغيل سنعرف ما الذي قد يُسقط نظامنا، ، وللأعمال سنعرف الأطراف الخارجية المؤثرة على نظامنا... ولتبسيط الفكرة، تخيل البيت الذي تعيش فيه، فهو بيتك، لكنك مرتبط بشركة الكهرباء، شبكة الكهرباء لا تديرها أنت، لكن انقطاعها يؤثر عليك ..^^

**3. حدود المؤسسة -Enterprise Boundary- (عنصر اختياري):**

في الأنظمة المعقدة، قد نحتاج لتوضيح "أين" تقع هذه العناصر للغايات التنظيمية! وهنا يمكننا استخدام خط متقطع ليمثل أسوار المؤسسة، وكل هذا لتوضيح المخطط بصريا وإظهار من

يعيش "داخل" أسوار الشركة مثل الموظفين والأنظمة الداخلية، ومن يتصل بنا من "خارجها" مثل العملاء الخارجيين وأنظمة الطرف الثالث، شاهد الصورة أدناه:




**A System Context diagram for the Internet Banking System, additionally showing the enterprise boundary**

ملاحظة: plc الموجودة أسفل الحد تشبه الاختصارات الخاصة بالشركات عنا مثل: ش.م.ع -شركة مساهمة عامة-.

وهنا سيظهر لنا سؤال مهم، ما هو الفرق بين النطاق -scope- والحد -boundary-؟

وحقيقة هذا سؤال مهم جدا، وإجابته بسيطة:

إن النطاق يمثل ما نطوره نحن فعليا، والحد هو الخط الفاصل بين ما نملكه وبين ما لا نملكه! أي أننا نستخدم النطاق لنحدد المهام التي سنكتب لها الشيفرة البرمجية -ماذا سيفعل نظامنا-، ونستخدم الحدود -Boundaries- لنحدد التقسيمات التنظيمية والبيئية -أين يعيش نظامنا ومع من يشارك المكان-... وهذا يشبه أن تكون داخل قاعة مؤتمرات، فجدران القاعة وأبوابها هي حدود، أما المطعم الخاص بالفندق أو غرف النزلاء فهي خارج الحد! والنطاق هنا سيكون العقد مع الشركة المنظمة والمسؤولية عن ترتيب الطاولات وتجهيز الشاشات وتوزيع الكراسي وتسجيل الحضور، لكن حجز تذاكر الطيران للمتحدثين فهذا ليس من ضمن نطاق العقد معهم؛ مع أن المتحدثين سيحضرون وسيكونون ضمن الحد الخاص بقاعة المؤتمر...

**٤. التفاعلات بين العناصر المختلفة -Interactions-:**

عندما قمنا برسم الرسمة، وضعنا أسهم وداخل هذه الأسم هناك وصف يوضح غرض هذا السهم، وقلنا أن هذه الحركة تحل مشكلة العلاقات الغامضة التي تحدثنا عنها سابقا في قسم المشاكل ^^، لكن هنا وجب أن ننوه إلى نقطة مهمة، وهي أننا نذكر الغرض دون ذكر التقنية! مثل أن نقول: Gets account information دون أن نقول JSON over HTTPS... لأن التقنيات تنتمي إلى المستوى الثاني -container- لا الأول -context-.

إن خلاصة هذا الباب هي:

يعتبر ال System Context بمثابة ال Business Card الخاصة بمشروعك، فلو انضم موظف جديد إلى الفريق اليوم، أو أنك جلست في اجتماع مع المستثمرين؛ فأنت بهذا

المخطط لن تصيبهم بالصداع عبر الحديث عن تفاصيل تقنية دقيقة مثل React و SQL و Microservices، فكل ما يحتاجونه هو هذه الخريطة البسيطة التي تقول لهم: "يا شباب، هذا هو نظامنا في المنتصف، وهؤلاء هم العملاء الذين نخدمهم، وهذه هي الأنظمة الخارجية التي نعتمد عليها لكي نعمل".. ومتى ما فهم الجميع هذه الصورة الكبرى، يمكننا حينها فتح أي صندوق والتحدث في تفاصيله التقنية = وهذا هو بابنا للمستوى الثاني ^^

ولاحظ معنا أن المخطط الذي قمنا برسمه يحكي قصة كاملة مفهومة، حتى لغير التقني! فلا يحتاج شرحا شفهيا، وهذا تطبيق لمبدأ المخطط يقف بذاته والذي أشرنا إليه في باب: "الافتراضات الخفية - نخ: سأشرحها لكم في الاجتماع" من الفصل الأول.

# فائدة

وما من فكرة تصلح بها البشرية إلا وهي أو أصلها في القرآن، ومن أعجبته حكمة عظيمة صالحة في قول مفكر أو فيلسوف فليتحسر على نفسه أن جهل موضعها من القرآن..

- كتاب العقلية الليبرالية في رصف العقل ووصف النقل، عبدالعزيز الطريفي -فرج الله عنه-

## الفصل الخامس: Level 2: Container diagram

بمجرد أن فهمنا كيف يمكن دمج نظامنا مع العالم الخارجي عبر مخطط السياق -المستوى الأول-، فإن الخطوة المنطقية التالية هي التوغل لداخل أحد الصناديق لرؤية وتوضيح "الخيارات التقنية الكبرى"، وذلك يتم عبر رسم مخطط الحاويات -Container-... أي أننا سننتقل الآن من مجرد الفهم العام للنظام إلى خطوة أعمق تمثل الصورة الكبيرة للشكل التقني -الاختيارات التقنية عالية المستوى-، وهذه قفزة جوهرية، ففي المستوى الأول تجنبنا تماما ذكر التقنيات المستخدمة، في حين أننا هنا سنظهرها بصراحة ^^، وهذا سيعالج مباشرة مشكلة "إغفال التقنية" التي تحدثنا عنها بالفصل الأول.

بناء على هذه المقدمة نستنتج أن خصائص هذا المخطط هي:

1. أن هذا المخطط High Level = أي يعرض الصورة التقنية الكبرى لا التفاصيل الدقيقة جدا.
2. التركيز على التقنية فيه سيجعل من الجمهور المستهدف لهذا المخطط هم مطوري البرمجيات وموظفي الدعم والتشغيل -support/operation staff-، وهذه تعد مزية مزدوجة! فمخطط واحد يمكن أن يخدم فريق ال DevOps والتطوير معا ^^

حسنا، ما هي الأسئلة التي يجب أن يجيب عنها هذا المخطط حتى يكون فعالا وصحيحا؟

1. ما هو الشكل العام للنظام التقني؟ مثل بنية النظام التقنية ستكون Web + DB + API + Mobile...

2. ما هي القرارات التقنية الكبرى التي تم اتخاذها (لغات البرمجة، أطر العمل، قواعد البيانات)؟ مثل Angular، PostgreSQL، Java/Spring، إلخ
3. كيف تم توزيع المسؤوليات عبر النظام؟ أي من يفعل ماذا، مثل ال PostgreSQL ستكون مسؤولة عن حفظ البيانات الخاصة بالمقالات، وتحتفظ بتعليقات المستخدمين وتفاعلاتهم...
4. كيف تتواصل هذه الحاويات مع بعضها البعض (البروتوكولات)؟
   مثل: JDBC، JMS، HTTPS، إلخ
5. السؤال الأهم للمبرمج: كَمطور، أين يجب أن أكتب الشيفرة البرمجية الخاص بي لإضافة ميزة جديدة؟ أي المطلوب معرفة الحاوية / الحاويات -container- الصحيحة لإضافة الميزة، وهذا يحقق ما ذكرناه سابقا، بأن من المزايا التي ستعطيها المخططات المعمارية السليمة لنا هي = تقديم خريطة تساعدنا كمطورين على التنقل في الشيفرة البرمجية دون الحاجة للنظر لآلاف الأسطر البرمجية.

وحتى ننتقل من التنظير إلى التطبيق العملي، دعنا نأخذ مثالا حيا ^^:
تخيل أننا ننظر إلى مخطط الحاويات -Container Diagram- الخاص بـ "منصة لنشر المقالات التقنية"، وقمنا بعرض هذا المخطط على مطور جديد انضم للفريق، فإنه بمجرد النظر للرسم، سيستخرج الإجابات العملية التالية على تلك الأسئلة الخمسة:

1. ما هو الشكل العام لمعمارية النظام التقنية؟

الإجابة المستخرجة: "النظام يتبع معمارية Client-Server مفصولة الواجهات، ويتكون بشكل أساسي من أربع حاويات، وهم: Frontend App، و Backend API، وقاعدة بيانات علائقية لتخزين البيانات، وذاكرة تخزين مؤقت -Cache- لتسريع القراءة ووقت الاستجابة"

2. ما هي القرارات التقنية الكبرى التي تم اتخاذها؟

الإجابة المستخرجة: "لقد قرر الفريق التقني بناء واجهة المستخدم باستخدام إطار عمل ReactJS، أما القلب النابض للنظام فتمت برمجته بلغة C# باستخدام إطار عمل ال .NET Core، وتم اختيار ال PostgreSQL كقاعدة بيانات رئيسية، و Redis كذاكرة تخزين مؤقت"

3. كيف تم توزيع المسؤوليات عبر النظام؟

الإجابة المستخرجة: المسؤوليات موزعة بدقة كالتالي:

- ستكون حاوية ReactJS مسؤولة حصراً عن عرض واجهة المستخدم وجمع التفاعلات الخاصة به، ولا تحتوي على أي منطق عمل -Business Logic- معقد...
- وستكون حاوية ال .NET Core API هي العقل المدبر والمسؤولة عن التحقق من صلاحيات المستخدمين، ومعالجة النصوص، وتطبيق قواعد النشر...
- وستكون حاوية ال PostgreSQL هي المسؤولة عن التخزين الدائم والآمن للمقالات، وبيانات المستخدمين، والتعليقات...

- وستكون حاوية ال Redis هي المسؤولة عن تخزين المقالات الأكثر قراءة بشكل مؤقت لتخفيف الضغط عن قاعدة البيانات الرئيسية...

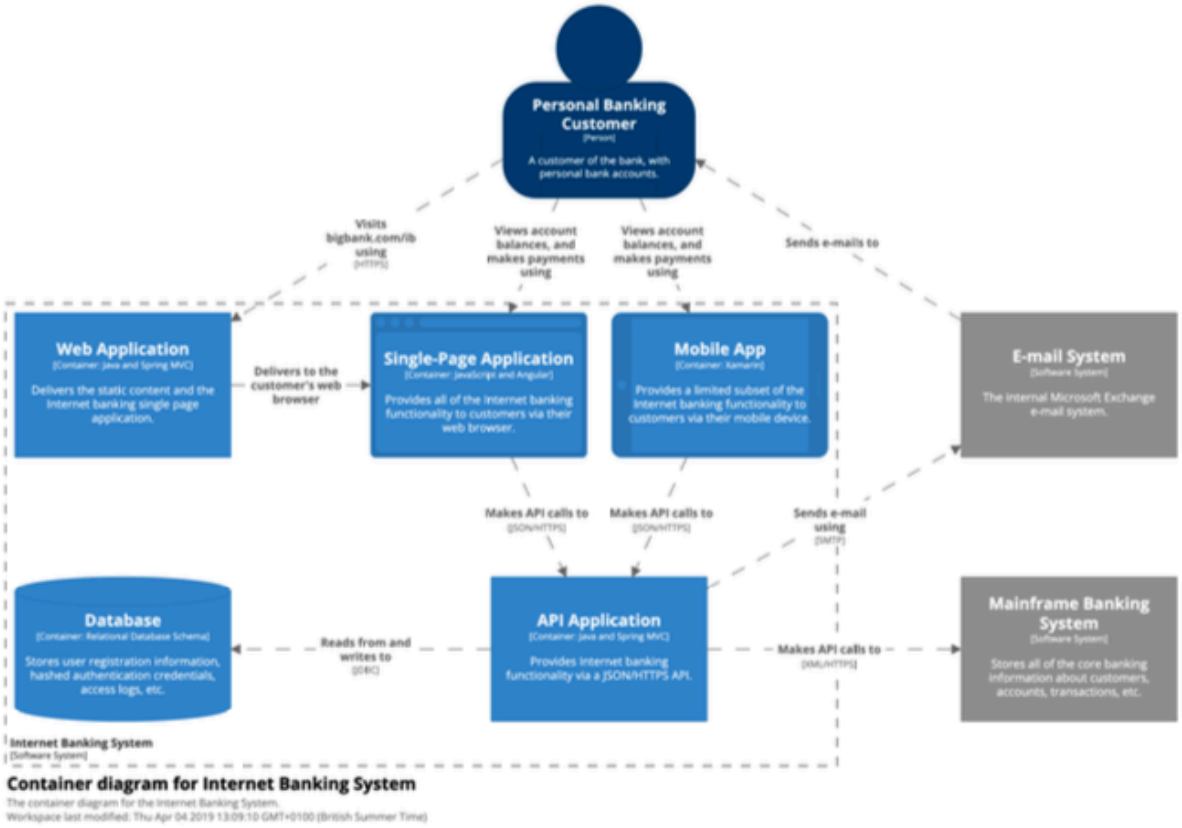
4. كيف تتواصل هذه الحاويات مع بعضها البعض (البروتوكولات)؟
الإجابة المستخرجة: "سيقوم تطبيق ال React بالتواصل مع خادم ال .NET Core عبر إرسال طلبات HTTP/REST، والبيانات سيتم تبادلها بصيغة JSON، وبدوره، سيتواصل خادم ال .NET Core مع قاعدة بيانات ال PostgreSQL باستخدام بروتوكول الاتصال (TCP/IP) القياسي لقواعد البيانات، ويتصل بـ Redis عبر ال RESP..."

5. كمطور، أين يجب أن أكتب الشيفرة البرمجية لإضافة ميزة جديدة؟ (لنفترض أن الميزة المطلوبة هي: "إضافة زر الإعجاب بمقالة ما")
الإجابة المستخرجة: "بناءً على المخطط وبصفتي مبرمجاً، أدرك أن الشيفرة البرمجية لهذه الميزة لن تكتب في مكان واحد، بل يجب أن أوزع عملي على ثلاث حاويات:
- سأحتاج لكتابة شيفرة برمجية لل UI في حاوية ال React لعرض زر الإعجاب وإرسال الطلب عند النقر...
- وسأحتاج لكتابة Endpoint جديدة في حاوية ال .NET Core لاستقبال الطلب، والتحقق من الشروط الممكنة، وتحديث عدد الإعجابات...
- وسأحتاج لإضافة حقل جديد أو جدول جديد في حاوية ال PostgreSQL لحفظ عدد الإعجابات بشكل دائم...

إن هذه الإجابات هي المزية الحقيقية لمخطط الحاويات! فهو ما يمنح المبرمج خريطة طريق واضحة ومباشرة لعمله، ودون أن يغرق في قراءة مئات الأسطر من الشيفرة البرمجية لفهم أين يجب أن يضع تعديلاته...

تحذير معماري هام: هذا المخطط لا يخبرنا بأي شيء عن عدد ال Servers أو البنية التحتية الخاصة بها! فقد يكون ال API يعمل فعليا على Server Farm 50، وقد تكون قاعدة البيانات موزعة على مجموعة خوادم مثل Oracle Cluster لضمان عدم فشل النظام، ومع ذلك، فإننا في هذا المخطط نقوم برسم كل حاوية "كمربع واحد فقط"، لأننا نهتم بال Static Structure للبرمجيات وليس بالبنية التحتية، لذلك فتفاصيل الخوادم والشبكات ونحوها لها مخطط آخر منفصل تماماً يُسمى "مخطط النشر" -Deployment Diagram-... باختصار: مخطط الحاويات يوضح "الأنواع" لا "الكميات"، فنحن هنا نوثق المنطق البرمجي، ونترك تفاصيل أعداد الخوادم الفيزيائية وتوزيعها لمخطط آخر... وهنا تظهر لنا القاعدة الذهبية مجددا: كل مخطط نقوم برسمه أو بناءه يجب أن يمثل مستوى تجريد واحد فقط، وهذا يحل مشكلة خلط مستويات التجريد التي رأيناها في الفصل الأول... (عندما نتحدث عن ال Deployment Diagram -بإذن الله- سنرى أن هذا المخطط يمثل مستوى تجريد بذاته للبنية الفيزيائية)...

والآن لنبدأ بالتطبيق العملي، ولنشاهد صورة المخطط معا، ولنتعلم كيف سنرسمه:




كيف سنبدأ بالرسم؟

إن عملية رسم هذا المخطط تتم من خلال أحد الاحتمالين التاليين:

الاحتمال الأول: التصميم المسبق -Up-front Design-: إذا كنت ترسم المخطط لنظام جديد لم تكتب شيفرته البرمجية بعد = فمن الطبيعي ألا تمتلك كل التفاصيل التقنية، وفي هذه الحالة عليك برسم المكونات الكبرى واكتب ما تعرفه فقط ^^.

الاحتمال الثاني: توثيق نظام قائم أو التوثيق العكسي -Documenting existing system-: إذا كنت توثق نظاما قائما بالفعل، فهنا يُفضل أن تضع تفاصيل دقيقة جدا تفيد المبرمجين، مثل أسماء البروتوكولات بدقة -مثلا gRPC-، وأرقام ال Ports -مثلا 443 أو 3306.. إلخ- التي تتواصل عبرها الحاويات...

والخلاصة هنا: أنت من يملك مؤشر التكبير، لذلك أضف التفاصيل التي تراها ضرورية لفريقك، وهذا تطبيق لمبدأ المرونة في C4! فلا قواعد جامدة، بل توجيهات واضحة.

الآن، لنبدأ بالرسم ^^.. إن عملية الرسم تبدأ من خلال عمل تكبير على صندوق النظام الخاص بنا، الذي يمثل نظام "نظام البنك عبر الإنترنت"، علما أن هذا المخطط سيحتوي -كما شاهدت بالصورة اعلاه- على ٣ عناصر أساسية وهي:

١. البشر - تحدثنا عنه في المستوى الأول
٢. الأنظمة الخارجية - تحدثنا عنه في المستوى الأول

وهنا توجد قاعدة بصرية هامة: فالبشر والأنظمة الخارجية الموجودة في المخطط يجب أن تحافظ على الاستمرارية -Continuity-! أي يجب أن نقوم بنسخ نفس "البشر" ونفس "الأنظمة الخارجية" التي رسمناها في مخطط السياق -Level 1- ومن ثم إسقاطها هنا في مخطط الحاويات، لأن هذا التطابق يضمن للمشاهد أن المخططين يرويان نفس القصة المتسلسلة، وأننا ببساطة قمنا بعمل Zoom in لنفس النظام... وأظنك خائف الآن، هناك الكثير من التكرارات والمشاكل التي ستحدث إذا قمنا بالنسخ هكذا، ونحن نريد مصدرا

واحدا للحقيقة *-*، وهذا كلام صحيح... لكن لا تخف، فهذه المشكلة لن نقع فيها بإذن الله، وسنتحدث عن ذلك لاحقا عند حديثنا عن الأدوات الممكنة و ال Diagrams as Code ^^.

٣. الحاويات = جديد ^^، ولكن تكلمنا عن الحاويات وماهيتها، وقلنا أنها أي وحدة قابلة للنشر بشكل مستقل...إلخ، وضربنا أمثلة عليها مثل قواعد البيانات وتطبيقات ال SPA أو ال Mobile App وال Backend Api App... إلخ، ومن هنا نتسنتج ببساطة أن الحاوية = أي تطبيق أو مخزن بيانات ^^.

لذلك، عند رسم الحاويات هنا علينا إضافة ٣ معلومات مهمة وهي:

- الاسم: مثل تطبيق الويب أو قاعدة البيانات.
- التقنية: مثل: تطبيق ASP.NET أو قاعدة بيانات PostgreSQL.
- الوصف: سطر يشرح مسؤولية الحاوية أو نوع البيانات التي تخزنها.

وأظنك الآن تتسائل -.-، لحظة لحظة، في الصورة السابقة لم يتم ذكر بعض التقنيات بشكل تفصيلي مثل قاعدة البيانات... هذا أول تناقض منك!

وهنا أحب أن أخبرك يا صديقي، أنت رائع، وهذه ملاحظة مهمة... دعنا نحللها ^^




في المخطط تم رسم الحاوية الخاصة بقاعدة البيانات دون ذكر تقنية مثل mysql or PostgreSQL وغير ذلك، والسبب يعود في ذلك لنقطة جميلة ذكرناها: التصميم المسبق -Up-front Design... ففي هذه الحالة ليس بالضرورة أن تكون التفاصيل التنفيذية التقنية الدقيقة معلومة عندنا في هذه اللحظة، لكن من المعلوم عندنا أنها قاعدة بيانات علائقية، وبهذا فقد قمنا بكتابة هذا في خانة التقنية، وهو يعطي المراد والنتيجة المرجوة، والتفاصيل التنفيذية يمكن تأجيلها لوقت لاحق... وهذا يذكرنا بما تحدثنا عنه سابقا: في ال Decouple وال Defer Decisions، ويذكرنا بال Last Responsible Moment التي تحدثنا عنها في الجزء الأول من الكتاب ^^... وطبعا هنا ملاحظة مهمة أخرى، فإننا عندما نتحدث عن تأجيل القرارات لا يعني تأجيلها للأبد، لأننا حينها سنقع ضحية لإحدى المشاكل التي تكلمنا عنها سابقا وهي: "إغفال التقنية"، فإذا كان ما يميز مخطط الحاويات ذكر التقنيات ولم نقم بذكرها فما الفائدة التي حصلنا عليها!! ثم إذا كان القرار الذي سيتم تأجيله قرار جوهري ويؤثر على المخطط المعماري فهذا يجب ألا يؤجل = نتوقف عن الرسم، نصل إلى قرار = ثم نتابع، كما أن التأجيل لن يكون صحيحا إذا كنت ترسم مخططا لنظام قائم فعلا! فالتقنية موجودة فلماذا لم تكتبها؟!

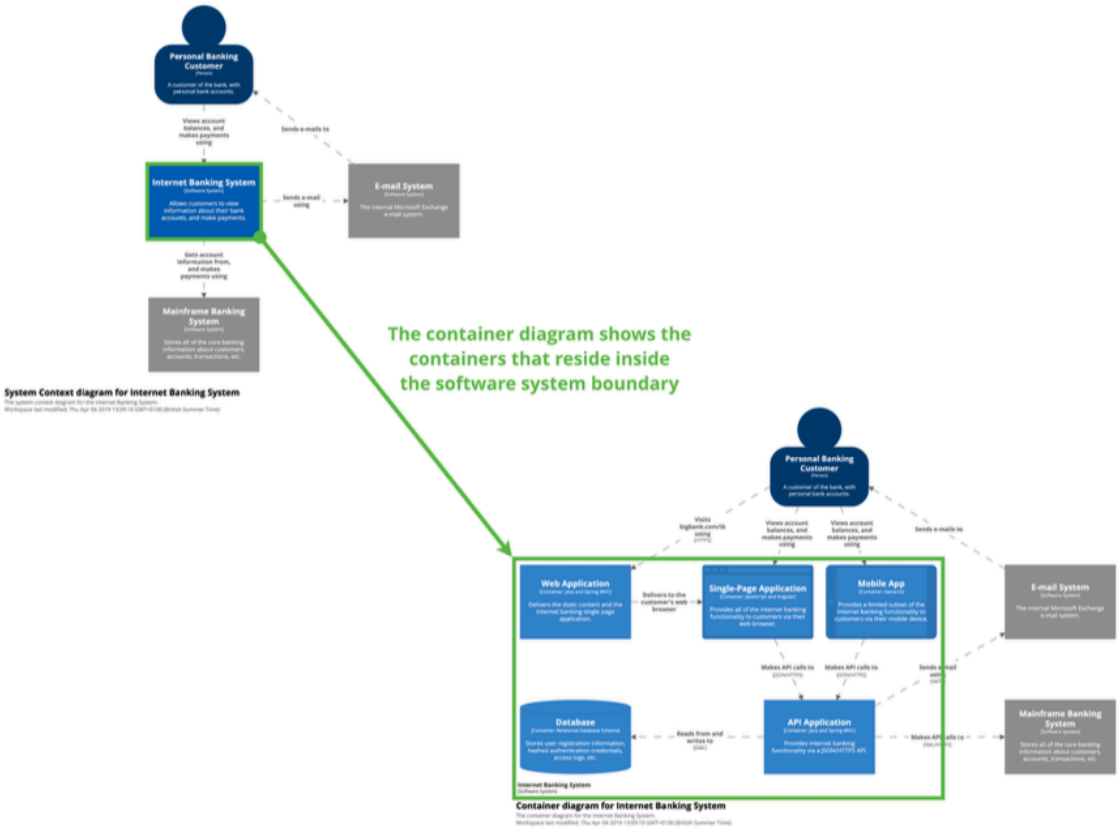
إذا، يمكننا القول إن القاعدة الأساسية في رسم الحاويات هي ذكر التقنية المستخدمة بوضوح، لكن، قد تلاحظ في بعض المخططات الاحترافية أن المعماري يكتفي بكتابة عبارة عامة مثل [Relational Database] دون تحديد اسم التقنية نفسها مثل PostgreSQL، وهذا يعد من الخطوات التقنية الصحيحة، وهو تطبيق عملي لمبدأ تأجيل القرارات...

إن المراحل الأولى من ال Up-front design قد لا نكون فيها قد حسمنا خيارنا التقني بعد، لكن المعمارية المرنة تسمح لك برسم الصندوق، وتحديد وظيفته، وترك التفاصيل التنفيذية ليتم ملؤها لاحقاً عندما تتضح الرؤية.

بعد أن تحدثنا عن رسم الحاويات، علينا بتعلم كيفية رسم الأسهم، وهي ببساطة من خلال كتابة الوصف وكتابة التقنية داخل [] مثل [JSON/HTTP]... كما أن هذه الأسهم ستربط الحاويات معا، وستربط الحاويات بالأنظمة الخارجية والأشخاص...

كما يمكننا رسم إطار على الحاويات ممثلا بخط متقطع، وبهذا يصبح ما داخل هذا الإطار هو الخاص بهذا المخطط والخاص بنا، وما هو خارجه لا نملكه (كما فعلنا مع المخطط في المستوى الأول)

شاهد الصورة التي ستظهر التسلسل البصري لك عندما تنتقل من المخطط ذو المستوى الأول إلى المستوى الثاني:




**The Container diagram shows a zoom in of the software system boundary**

ملاحظة: لاحظ أثناء الرسم أننا نهتم بالتفاصيل البصرية، فمثلا قاعدة البيانات تم رسمها داخل صندوق اسطواني، لأن هذا الرمز البصري عادة ما يستخدم مع قواعد البيانات... كما إن استخدام الألوان له هدف بصري... كما أن هناك اتساق بصري للألوان حتى باختلاف المستويات.

إن هذا المخطط إلزامي، والفئة المستهدفة منه هم التقنيون بشكل أساسي.

لو أردنا أن نقارن المستوى الأول مع الثاني:

| وجه المقارنة | **السياق - المستوى الأول** | **الحاويات - المستوى الثاني** |
|---|---|---|
| **التركيز** | النظام وعلاقته بالعالم الخارجي (أشخاص وأنظمة) | النظام من الداخل (القرارات التقنية وتوزيع المسؤوليات) |
| **العناصر المسموحة** | أشخاص + أنظمة (2) | أشخاص + أنظمة + حاويات (3) |
| **التقنية** | مُغفلة تماماً (ممنوع ذكرها) | تفاصيل تقنية بشكل عالي المستوى |
| **مستوى التفاصيل** | عالية المستوى | عالية المستوى مع تفاصيل تقنية |
| **الجمهور** | الجميع (تقنيون، إداريون، وعملاء) | التقنيون أساساً (مطورون، مهندسو تشغيل) |
| **معلومات العنصر** | الاسم + الوصف | الاسم + التقنية + الوصف |
| **العلاقات (الأسهم)** | الوصف فقط (الغاية من التفاعل) | الوصف + [البروتوكول/التقنية] |
| **الفائدة** | فهم النطاق (Scope) وحدود النظام | فهم البنية التقنية وخريطة عمل المبرمج |
| **الإلزامية** | دائماً | دائماً |

قبل الانتقال من هذا المستوى للمستوى الذي بعده، هناك حالات معمارية خاصة يجب أن نتعامل معها، وهي:

- ملفات النظام وسجلات النظام (Log files vs Data storage):

السؤال القوي: هل نرسم ملفات النظام File System كحاوية مستقلة؟

الجواب القوي: إذا كانت تحتوي على بيانات العمل -Business Data- مثل نظام يقرأ صور الأشعة للمرضى المحفوظة على disk، فهنا يجب رسم "نظام الملفات" كحاوية مستقلة، أما إذا كانت لسجلات المراقبة -Log Files- فإننا نتجاهل رسمها تماما للحفاظ على نظافة المخطط وإيجازه -رغم أهميتها-، فكل حاوية تكت logs على ملفات النظام، لذلك، إذا كان نظامك يُخزن بيانات حرجة على ال file system، فأضف "File System" كحاوية صريحة!

- ال Data Storage as a Service:

يعد هذا السؤال من الأسئلة الأكثر شيوعاً، وهو: أين نضع خدمات مثل Amazon S3 لتخزين الملفات، أو Azure SQL لقواعد البيانات؟ أي بمعنى آخر، هل نعاملها معاملة "الأنظمة خارجية" لأننا لا نملك خوادمها، أم نعاملها "كحاويات" لدينا؟

الجواب ببساطة: تُعامل "كحاويات -Containers-" تقع داخل حدود نظامنا! فصحيح أننا لا نملك الخوادم الفيزيائية لشركة أمازون، لكننا نملك ونتحكم كليا في شكل قاعدة البيانات -Schema- أو مساحة التخزين -Bucket-، لذلك، وبما أنها جزء لا يتجزأ من معمارية النظام الخاص بنا = فهي حاويات أصيلة؛ حتى لو كانت مستضافة في مكان آخر!

فكرة: تبرز عبقرية هذا المخطط -مخطط الحاويات- باعتباره "منطقة وسطى -Intermediate Diagram-"؛ فهو يفتح الصندوق الأسود الكبير، ولكنه لا يغرقك في تفاصيل الشيفرة البرمجية! بل يرتب لك المعمارية في "حاويات" واضحة التقنيات والتفاعلات.. ومن خلال هذا المخطط الواضح، سُتقاد بسلاسة إلى المستوى الثالث؛ حيث ستختار "حاوية واحدة فقط" لتفتحها وترى المكونات التي تسكن بداخلها...

## الفصل السادس: Level 3: Component diagram

لقد انتهينا من رسم الحاويات -containers- في المستوى الثاني، والآن قد حان الوقت لفتح "صندوق واحد فقط" من تلك الحاويات -containers- التي قمنا برسمها لنرى مما تتكون هذه الحاوية من الداخل، وهذا هو المستوى الثالث... المستوى الذي يشكل "اللحظة التي يعود فيها المبرمج إلى بيئته المألوفة" ^^

لقد رأينا في المستوى الثاني النظام كصناديق كبيرة تتواصل عبر الشبكة، مثل تطبيق ويب، وال API، وقاعدة بيانات! والخطوة المنطقية التالية هي تطبيق مبدأ التكبير والتصفية -Zoom and filter-؛ بأن نختار حاوية واحدة فقط، ثم نفتحها، ونقوم بتفكيكها لنرى "المكونات -components- الداخلية" التي تشغلها.

لقد صمم هذا المستوى -المستوى الثالث- ليجيب على هذه الأسئلة:

١. مم تتكون هذه الحاوية؟ أي ما هي الأجزاء البرمجية أو المكونات التي تتضافر معا لتشغيل هذه الحاوية؟

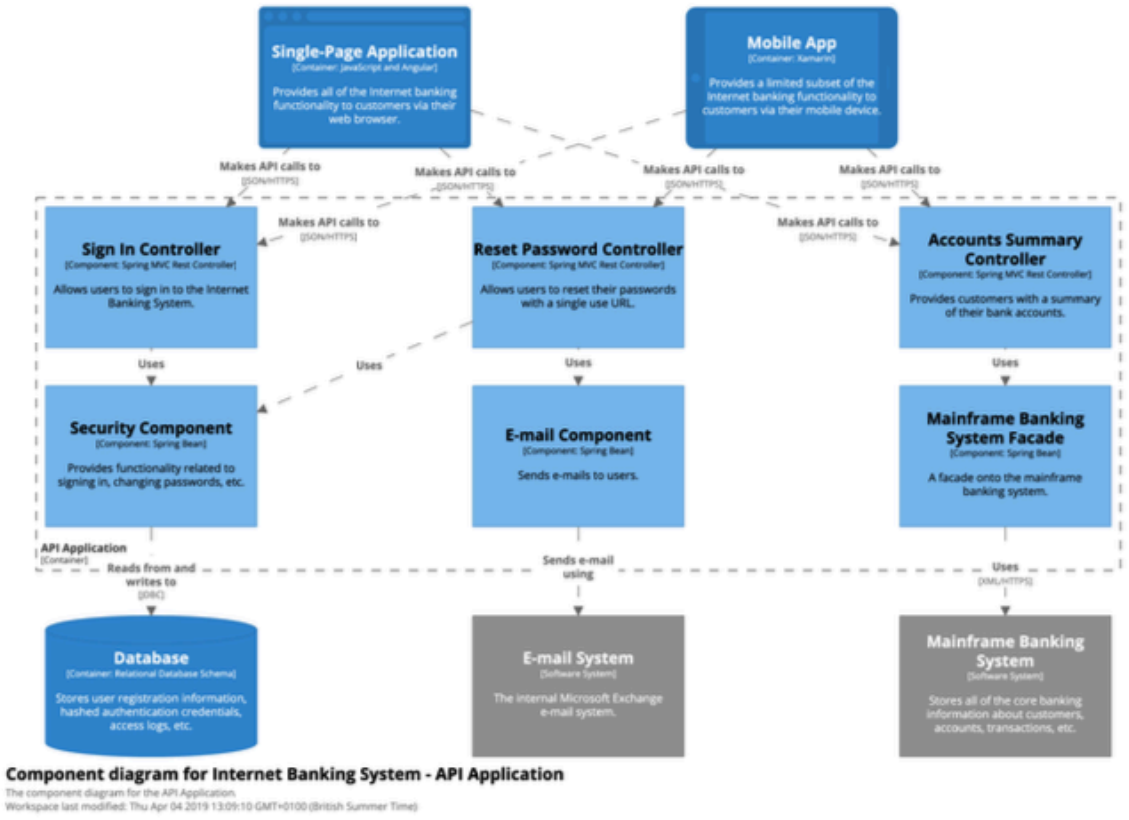
٢. هل كل مكون -component- يمتلك منزلا يسكن فيه؟ أي التأكيد على أن كل مكون -component- برمجي قد تم وضعه بشكل صحيح داخل حاوية معينة، ولا توجد أجزاء "عائمة" أو تائهة في النظام ^^، وهذا السؤال هو الأهم من بين الأسئلة! تذكر مشكلة ال HOCO التي تحدثنا عنها سابقا.

٣. هل طريقة عمل النظام واضحة؟ أي هل من السهل فهم كيف تتدفق البيانات داخل هذه الحاوية بمجرد النظر إلى مكوناتها؟

إجابة هذه الأسئلة ستقودنا بشكل تلقائي إلى البنية الخاصة بهذا المخطط، لكن قبل البدء بالتفاصيل الدقيقة، أحب أن أنقل ملاحظة مهمة طرحها Simon هنا، وهي -بتصرف-:
"عادة، عندما يُطلب من المبرمجين رسم مخطط معماري، فإنهم يندفعون مباشرة لرسم هذا المستوى -أي أن السلوك الافتراضي لهم هو البدء برسم مخطط المكونات-! ولكن الخطأ الكارثي الذي يقعون فيه هو محاولة رسم كل المكونات الموجودة في كل الحاويات في لوحة واحدة، فتنتج لوحة فوضوية لا تُقرأ"...
هذه الملاحظة المهمة تقودنا إلى القاعدة الذهبية في نموذج C4 لهذا المستوى، وهي:
"في مخطط المكونات -components- = يجب ألا ترى سوى المكونات -components- التي تعيش داخل حاوية -container- واحدة فقط في كل مرة!"... وهنا نضيف قاعدة

أخرى مهمة وهي: "لو كان لديك 5 حاويات = **فقد** يكون لديك حتى 5 مخططات مكوّنات، لكن ربما لن تحتاجها كلها، وكل ذلك حسب الأهمية! وداخل نفس الحاوية قد لا تظهر جميع المكونات الخاصة بها؛ يكفي منها ما نحتاجه لنفهم القصة! باختصار = "ليس عليك إظهار كل شيء. أظهر ما يكفي لإيصال الفكرة"

الآن شاهد الصورة:




بعد مشاهدتك للصورة ستكتشف الموضوع ببساطة... لقد قمنا بإنشاء المكونات التي نحتاجها بعد أن اخترنا حاوية محددة، وفي مثالنا قمنا باختيار حاوية ال API Application.

والآن، لقد ذكرنا في المستوى الثاني عن احتمالين اثنين سنواجههما عند رسم المخطط، وهنا لدينا نفس الحالة، فمقدار التفاصيل التي سنقوم بكتابتها يعتمد على حالتنا الحالية:

١. في حالة التصميم المسبق -Up-front design-: سنكتب ما نعرفه فقط، ولن نقلق إن لم نكن نعرف أسماء المكتبة التي سنستخدمها بشكل دقيق.

٢. في حالة توثيق نظام قائم / توثيق العكسي -Documenting existing system-: فهنا يجب أن نضع التفاصيل الدقيقة، مثل أسماء أطر العمل -Frameworks- المستخدمة لبناء كل مكون، مثل: Entity Framework للتعامل مع قاعدة البيانات، وهنا يمكننا إضافة التفاصيل التي نرى أنها تنقذ المبرمج الذي سيقرأ المخطط من الضياع في الشيفرة البرمجية.

حسنا، الآن يجب علينا أن ننتقل إلى العناصر المسموح استخدامها في هذا المخطط، وهي:

١. البشر
٢. الأنظمة الخارجية
٣. الحاويات
٤. المكونات -جديد-

لاحظ أن المكونات الثلاثة الأولى لقد تحدثنا عنها بالفعل، وانتقلت معنا تصاعديا من المستوى الأول ثم الثاني ثم الثالث، وهذا ينطبق مع مبدأ التكبير والتصفية ^^

ملاحظة مهمة: الحاويات هنا تظهر كإطار خارجي -Boundary- يحيط بالمكونات ليوضح أننا ننظر داخل حاوية محددة، والحاويات الأخرى تظهر كصناديق ترتبط بها المكونات إذا كان هناك اتصال بين أحد المكونات معها... وهذا يشبه ما قمنا به بالمستوى الثاني أيضا ^^

إن العنصر الجديد لدينا في هذا المستوى من التخطيط هو المكون -component-، وقلنا سابقا بأن المكونات هي الكتل البرمجية الكبيرة -Coarse-grained- التي تعيش داخل حاوية! لذلك، عند رسم أي مربع يمثل مكون -component-، يجب أن نرفق معه بطاقة تعريفية صارمة تتضمن ثلاثة عناصر:

١. الاسم -Name-: اسم المكون بوضوح, مثل: Sign in Controller أو Security Component.

٢. التقنية -Technology-: كيف تمت برمجته؟ مثل: كائن جافا بسيط POJO، أو Express.js Route Handler أو React Context Provider أو Laravel Eloquent Model... إن الفكرة الأساسية هنا ليس ذكر لغة البرمجة بشكل عام، بل هي لتوضيح طريقة التنفيذ، أو إطار العمل -Framework-، أو النمط -Pattern- المستخدم لبناء هذا المكون التقني ^^

٣. الوصف: جملة قصيرة تلخص مسؤولية هذا المكون.

وهنا ستظهر لنا قاعدة معمارية:

يجب أن يعكس شكل المخطط وتوزيع المكونات بداخله "النمط المعماري" -Architectural Style- الذي تستخدمه، فإذا كنت تستخدم ال Layered Architecture، فيجب أن تُرسم

المكونات فوق بعضها كطبقات! وإذا كنت تستخدم ال Hexagonal Architecture، فيجب أن يظهر ذلك في ترتيب المكونات حول قلب النظام -Core Domain-، الشكل المعماري يجب أن يقرأ من الرسمة!

## لغة الأسهم

إذا دققت النظر في مخطط المكونات في الصورة، ستلاحظ أمرا غريبا: بعض الأسهم يكتب عليها كلمة بسيطة مثل Uses، وبعضها الآخر يحمل تفاصيل تقنية مثل JSON/HTTPS! وأظنك تتسائل، لماذا؟

السر يكمن في عبقرية وصف الاتصال، فيمكننا تفصيل الاتصال إلى داخلي وخارجي:

- الاتصال الداخلي: عندما يتحدث مكون -component- مع مكون آخر داخل نفس الحاوية، فهما يتخاطبان داخل نفس الذاكرة عبر استدعاء الدوال، وهنا نكتفي بكتابة الغاية المنطقية مثل كلمة Uses.

- الاتصال الخارجي: أما عندما يضطر المكون لمد يده خارج أسوار الحاوية (ليتحدث مع قاعدة بيانات، أو نظام خارجي، أو حاوية أخرى) = فهذا يعني أن البيانات ستسافر عبر الشبكة، وهنا يصبح إلزاميا أن تبرز جواز السفر الذي تعبر به هذه البيانات، فتكتب الغاية بالإضافة إلى البروتوكول التقني (مثل: Reads from and writes to [JDBC]) أو Uses [XML/HTTPS]... إلخ

إن هذا الذي ذكرناه حاليا ليس بمفهوم جديد، فقد ذكرناه سابقا هنا، هل تتذكر العنوان؟ نعم، أحسنت ^^ لقد ذكرنا ذلك في خضم حديثنا عن ال Out-of-process Call && Remote Procedure Call (RPC

باختصار، إن الوصف الذي سيكتب على الأسهم هنا يجب أن يكون ذا غاية منطقية، مثل: Uses (يستخدم)، أو Delegates to (يفوض المهمة إلى)، أو Reads from (يقرأ من)..إلخ، وهذا ما رأيته في الصورة أعلاه ^^.

**معضلة البنية التحتية والاهتمامات المتقاطعة -Cross-cutting Concerns-**

كما تحدثنا في المستوى السابق عن بعض التحديات والأسئلة الشائعة مثل هل سنعتبر ال file system حاوية مستقلة بذاتها أو لا، فهنا لدينا تحدي آخر، وهو: ماذا سنفعل مع المكونات "الخدمية" التي يستخدمها الجميع في النظام؟

تخيل أن لديك ال Logging Component، من الطبيعي أن كل المكونات الأخرى داخل هذه الحاوية ستحتاج للاتصال بهذا المكون لتسجيل الأخطاء والعمليات، لكنك إذا قمت برسم سهم من كل مكون في نظامك ليتجه نحو الـ Logging، فحينها سيتحول مخططك إلى طبق معكرونة فوضوي تتشابك فيه المعكرونة -الخطوط- بشكل يجعله غير مقروء! وحتى

نتخلص من هذه المشكلة بذكاء، علينا أن نضع استراتيجية للتعامل مع المكونات المشتركة، ولديك ٣ خيارات هنا:

١. التجاهل التام -Don't include-: إذا كان وجود مُكوّن ال Logging لا يضيف أي قيمة حقيقية للقصة التي يرويها المخطط -لأن الجميع يعرف أنه موجود ضمنا-، فببساطة.. لا تقم برسمه!

٢. الملاحظة النصية -Write a note-: بدلاً من رسم الأسهم، يمكنك وضع صندوق نصي صغير في زاوية المخطط تكتب فيه: "ملاحظة: جميع المكونات المعروضة هنا تقوم بتسجيل بياناتها عبر مكون الـ Logging".

٣. الرسم بدون أسهم -Use a legend/key-: ارسم مربع مُكوّن ال Logging في زاوية المخطط، ولكن لا ترسم أي خطوط تتجه إليه! بدلا من ذلك، استخدم وسيلة بصرية مثل وضع نجمة * بجانب أسماء المكونات التي تستخدمه، أو تلوينها بلون مميز، ثم اكتب في ال Legend الخاص بالمخطط: "أي مكون يحمل علامة النجمة * يعني أنه يتصل بمكون الـ Logging"

الخلاصة هي: عندما نبدأ برسم مخطط المكونات، سنواجه حتما ما يعرف برمجيا بالاهتمامات المتقاطعة - Cross-cutting Concerns-، مثل خدمات الـ Logging أو الـ Security! هذه المكونات تتصل بها جميع أجزاء النظام تقريباً.. والخطأ الساذج هنا هو رسم أسهم من كل مكون نحو المكونات الخدمية مثل ال Logging! لأن النتيجة ستكون شبكة عنكبوتية

معقدة تدمر وضوح المخطط، والمعماري المحنك يتلاعب بطريقة العرض؛ إما بتجاهل رسم هذه الخدمات البديهية تماماً، أو بوضع ملاحظة نصية في زاوية المخطط تغني عن رسم عشرات الأسهم المتقاطعة أو من خلال حيلة بصرية جميلة تبعد هذه الشبكة عن عيوننا!
وتذكر دائما: إن الهدف من المعمارية هو الوضوح، وليس حشر كل سطر من الشيفرة البرمجية في رسمة واحدة!

**معضلة المكتبات والمكونات المشتركة -Shared components and libraries-:**

لابد أنه قد خطر على بالك سؤال بعد حديثنا عن معضلة الاهتمامات المتقاطعة، وهو: ما الذي سأفعله إذا كان "لدي Static Library أو ملف (DLL/JAR) مشترك تستخدمه عدة مكونات، هل سأرسم ذلك في المخطط؟"
والجواب المعماري ببساطة: نعم، إذا كان يخدم القصة! فإذا كان إظهار هذه المكتبة المشتركة يوضح كيف يعمل النظام مثل: مكتبة أمان مخصصة للشركة، فيجب رسمها، ولكن لكي تميزها عن المكونات العادية يجب علينا استخدام "دلالة بصرية -Notation-"؛ مثل تلوينها بلون مختلف، أو وضع أيقونة مميزة بجانبها ليعرف القارئ أن هذا المكون عبارة عن "مكتبة مشتركة" يتم استدعاؤها، وليست شيفرة برمجية مكتوبة حصرا لهذه الحاوية!

## تحدي تعدد أشكال التنفيذ للمكون الواحد -Multiple component-implementations

هذه النقطة إحدى التحديات التي ستمر بها بكل تأكيد، فهي تلامس صميم عمل المبرمجين الذين يطبقون مبادئ البرمجة الكينونية -OOP-، وتحديداً عند استخدام ال Interfaces...

التحدي ينطلق من سؤال:

ماذا لو كان لديك component واحدة، ولكن لهذه ال component تنفيذات -Implementations- متعددة تتغير حسب إعدادات العميل؟ مثلاً: عميل يستخدم SQL Server، وآخر يستخدم MySQL؟

شاهد المثال أدناه:

```
interface UserAuthenticator {
    boolean authenticate(String user, String password);
}

class LDAPAuthenticator implements UserAuthenticator { ... }
class ActiveDirectoryAuthenticator implements UserAuthenticator { ... }
class DatabaseAuthenticator implements UserAuthenticator { ... }
```

إن السؤال الذي سيتولد لديك الآن، هو كيف سنرسم شيئاً يتغير شكله أثناء وقت التشغيل -Runtime- أو النشر -Deployment-؟ وهنا لدينا ثلاثة استراتيجيات:

١. الاستراتيجية الأولى (التجريد والتجاهل): في هذه الاستراتيجية نكتفي برسم مكون واحد فقط بوصف عام يركز على ال Contract أو المسؤولية، وبغض النظر عن طريقة التنفيذ، مثل: "مكون تسجيل الدخول"!

٢. الاستراتيجية الثانية (التوضيح النصي): في هذه الاستراتيجية نقوم برسم المكون مع وضع ملاحظة نصية صريحة داخله تقول: "هذا المكون يستخدم لتسجيل الدخول من خلال الاعتماد على قاعدة البيانات أو وسيط خارجي أو ملف نصي، يتم اختيار الأسلوب وقت التنفيذ من قبل مدير النظام"، كما يمكننا تلوينه بلون مميز وشرح ذلك في مفتاح الخريطة -legend-

٣. الاستراتيجية الثالثة (تعدد المخططات): إذا كانت الاختلافات جذرية وتجلب معها مكونات أخرى جديدة، فالأفضل رسم "مخطط مكونات منفصل" لكل حالة، وهذه نقطة مهمة، لأننا نبحث عن الأثر المعماري هنا، ليس مجرد أسماء classes or interface... مثلا لو النظام الذي سنبنيه سيتعامل مع عميلين مختلفين، أحدهما يريد LDAP والآخر يريد نظام المصادقة من خلال قاعدة البيانات.. ففي هذه الحالة سنضطر لاستخدام هذه الاستراتيجية... وعادة ما نستخدم هذه الاستراتيجية إذا كان تغيير المكونات يمكن أن يحدث تغييرا جوهريا... وتذكر أن أي استراتيجية من الثلاث يمكن استخدامها، لكن حدد ما هو احتياجك!

القاعدة الذهبية هنا: لا توجد إجابة صحيحة بشكل مطلق؛ لذلك، اختر الاستراتيجية التي تروي "القصة" بوضوح لجمهورك ^^.

## الحفاظ على السياق (البشر، الأنظمة، والحاويات)

كما حافظنا على السياق في المستوى الأول عندما انتقلنا للمستوى الثاني، فالآن سنحافظ على السياق عند انتقالنا من المستوى الثاني إلى المستوى الثالث، فلا تنسى أننا نقوم بعمل Zoom In ^^

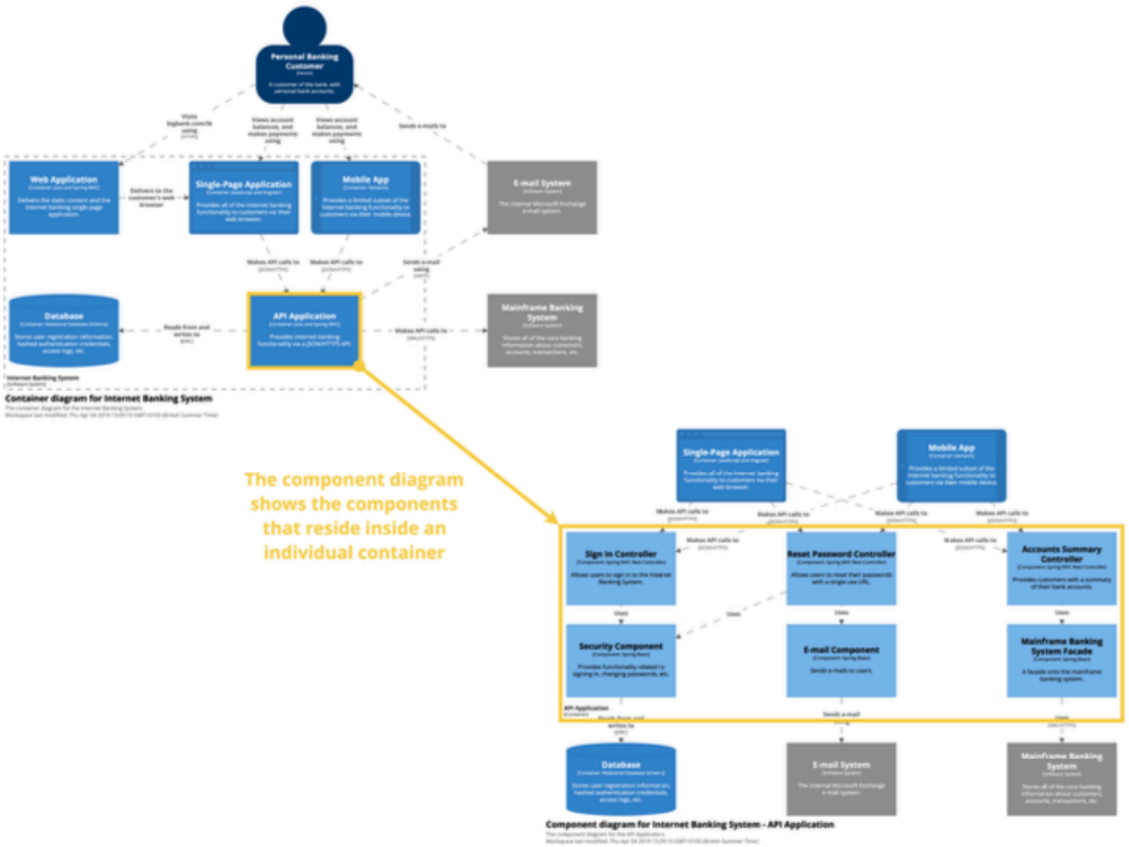
إن الحفاظ على عناصر المخطط السابق -بعضها أو كلها أو المستخدم منها- له دور كبير في الحفاظ على رؤية بصرية واضحة ومفهومة، لذلك: يُنصح بشدة أن تُبقي في رسمتك بعض العناصر من المستويات السابقة، وهذا يشمل المستخدمين، والأنظمة الخارجية، أو حتى حاويات أخرى من النظام؛ إذا كانت المكونات الداخلية تتواصل معها مباشرة.. مثال: إذا كان لديك مكون داخل الحاوية وظيفته إرسال تنبيهات، ارسم سهماً منه يخرج إلى صندوق يمثل "نظام البريد الإلكتروني الخارجي"، فهذا يربط المستويات ببعضها ذهنيا.

### حدود الحاوية -Container boundary-

بما أننا قررنا إظهار عناصر خارجية -كما في النقطة السابقة-، فكيف نفرق بين ما هو "داخل" الحاوية وما هو "خارجها"؟

الحل بسيط: نقوم برسم إطار -Bounding Box- يغلف كل المكونات الداخلية، ونكتب عليه اسم الحاوية، مثلاً: حاوية الـ API! كل ما هو داخل هذا الإطار يمثل المكونات البرمجية التي تعيش في الحاوية، وكل ما هو خارجه يمثل العالم الخارجي أو حاويات أخرى.

شاهد الآن الصورة بعد ال zoom in:




**The Component diagram shows a zoom in of a single container**

إن الدافع الأساسي لعمل zoom in على أي حاوية ورسم تفاصيلها الداخلية يكمن في تحقيق أربع غايات جوهرية، هذه الغايات هي: **تفكيك المسؤوليات** بوضوح عبر تقسيم الحاوية الكبيرة إلى كتل برمجية أصغر ذات مهام محددة، و**كشف الاعتماديات** -dependencies- بشفافية لتبيان شبكة التبعيات والعلاقات بين مختلف أجزاء الشيفرة البرمجية، وإلى جانب ذلك، يعمل هذا المستوى بمثابة **توثيق تقني** سريع يبرز أطر العمل -frameworks- والمكتبات المستخدمة في التنفيذ، ليتحول في نهاية المطاف إلى **خريطة حقيقية تمثل الشيفرة البرمجية** -Code Map-، لا سيما إذا التزمنا بمطابقة أسماء المكونات في المخطط مع الشيفرة البرمجية المكتوبة، مما يمنح أي مبرمج جديد الدليل الأمثل لاستيعاب هيكلة المشروع بشكل

فوري من خلال النظر... وبناء على هذا الكلام، يظهر لنا جليا أن هذا المستوى من التخطيط سيكون موجها للفريق التقني، وتحديدا الفريق الذي يعمل برمجيا على هذه المكونات.

**إليك هذه الصدمة المعمارية: هل تعتقد أن هذا المخطط إلزامي أم اختياري؟**

الجواب باختصار = اختياري!

إن هذه الصدمة التي تعرضت لها، أعتقد أننا جميعا تعرضنا لها ^^، وهذه النقطة واحدة من النقاط الإيجابية والمميزة في هذا النموذج المعماري، لأن الحاجة لإضافة هذا المخطط من عدمه أو تحديثه من عدمه يعتمد على قاعدة مهمة وهي: "التكلفة والفائدة" ^^، لذلك ستجد أن ال Microservices الصغيرة أو حاويات تخزين البيانات ونحو ذلك، لا تحتاج لإضافة مخطط لمكوناتها، لأنها لن تضيف أي قيمة إلى معماريتنا! في حين أنها قد تكون مفيدة جدا في تطبيقات ال Monolithic، خصوصا عند التصميم المسبق -Up-front Design-؛ وذلك لترتيب أفكار الفريق قبل البدء بكتابة الشيفرة البرمجية.

إن صيانة هذا المخطط وتحديثه مع كل تغيير في الشيفرة البرمجية يتطلب **جهدا عاليا**، فبمجرد أن يتجاوز عدد المكونات حدا معينا، سيتحول المخطط بسرعة إلى لوحة فوضوية مزدحمة -Cluttered Diagram- تفقد قيمتها، لذلك، تفضل الكثير من الفرق الاكتفاء بالمستويين الأول -السياق- والثاني -الحاويات- كوثائق دائمة، والاكتفاء بتوليد المكونات آليا عند الحاجة فقط.

الخلاصة:

إن رسم كل هذه المربعات، وتحديثها كلما أضفنا Service جديدة أو Controller جديد في الشيفرة البرمجية، يبدو كابوسا لنا عند عمليات الصيانة! لكن الخبر السار هو أن المعمارية الحقيقية ليست بيروقراطية عمياء، لأن هذا المستوى من النموذج -المستوى الثالث- اختياري للتوثيق طويل الأمد.

فإذا كنت تبني Microservice لا تتجاوز بضع مئات من الأسطر، فلا تضيع وقتك في رسم مكوناتها! وإذا كنت تصمم نظاماً Monolith، فقد يكون رسمها في البداية مفيداً لتوزيع المهام، لكن تحديثها اليدوي المستمر سيكون مرهقاً! لذلك، النصيحة المعمارية لك هي: استثمر وقتك في رسم المستوى الأول والثاني بدقة، واترك المستوى الثالث للأدوات الآلية التي تقرأ الشيفرة البرمجية وتستخرج المخطط منه مباشرة"...

والآن، لنشاهد مقارنة حول المستويات الثلاث:

| وجه المقارنة | **السياق - المستوى الأول** | **الحاويات - المستوى الثاني** | **المكونات - المستوى الثالث** |
|---|---|---|---|
| **التركيز** | النظام وعلاقته بالعالم الخارجي (أشخاص وأنظمة) | النظام من الداخل (القرارات التقنية وتوزيع المسؤوليات) | حاوية واحدة من الداخل (ال class / modules... الخ، أطر العمل، وآلية التنفيذ) |

| العناصر المسموحة | أشخاص + أنظمة (2) | أشخاص + أنظمة + حاويات (3) | أشخاص + أنظمة + حاويات + مكونات (4) |
|---|---|---|---|
| **التقنية** | مُغفلة تماماً (ممنوع ذكرها) | تفاصيل تقنية بشكل عالي المستوى | تفاصيل تقنية دقيقة |
| **مستوى التفاصيل** | عالية المستوى | عالية المستوى مع تفاصيل تقنية | عميقة (تحت المجهر البرمجي) |
| **الجمهور** | الجميع (تقنيون، إداريون، وعملاء) | التقنيون أساساً (مطورون، مهندسو تشغيل) | التقنيون داخل فريق التطوير (المبرمجون) |
| **معلومات العنصر** | الاسم + الوصف | الاسم + التقنية + الوصف | الاسم + التقنية + الوصف |
| **العلاقات (الأسهم)** | الوصف فقط (الغاية من التفاعل) | الوصف + [البروتوكول/التقنية] | الوصف + [الغاية المنطقية / التقنية أو البروتوكول إذا لزم الأمر] |
| **الفائدة** | فهم النطاق (Scope) وحدود النظام | فهم البنية التقنية وخريطة عمل المبرمج | خريطة حية للشيفرة البرمجية، وتوضيح الاعتماديات داخل الحاوية |

| | | | |
|---|---|---|---|
| **الإلزامية** | دائماً | دائماً | اختياري حسب الفائدة والتكلفة |

# فائدة

الإنسان يمكن أن يتطبع ويألف ما يخالف بعض الفطرة، إذ يحتاج إلى كثرة مخالطة ومشاهدة ومجاورة، وتدرج بذلك مع صبر حتى يتشربها كتشرب الإسفنج للماء، فيستطيع الإنسان أن يجاور أنتن الروائح وأكرهها، كجيفة الميتة، فإن جاورها شق عليه، حتى إذا طالت مجاورته لها لأيام، اعتاد عليها، ولم يشعر بما يستنكره المارة عليه، كذلك الأفعال والأفكار، ومنها السفور والتعري، فإذا تكاثر الناس على مجاورتها وتشربها، آنس بعضهم بعضا، ونظر بعضهم إلى بعض فلم يستنكروا شيئًا، وظنوا أن صاحب الفطرة الصحيحة هو الذي شذّ عن الصواب..

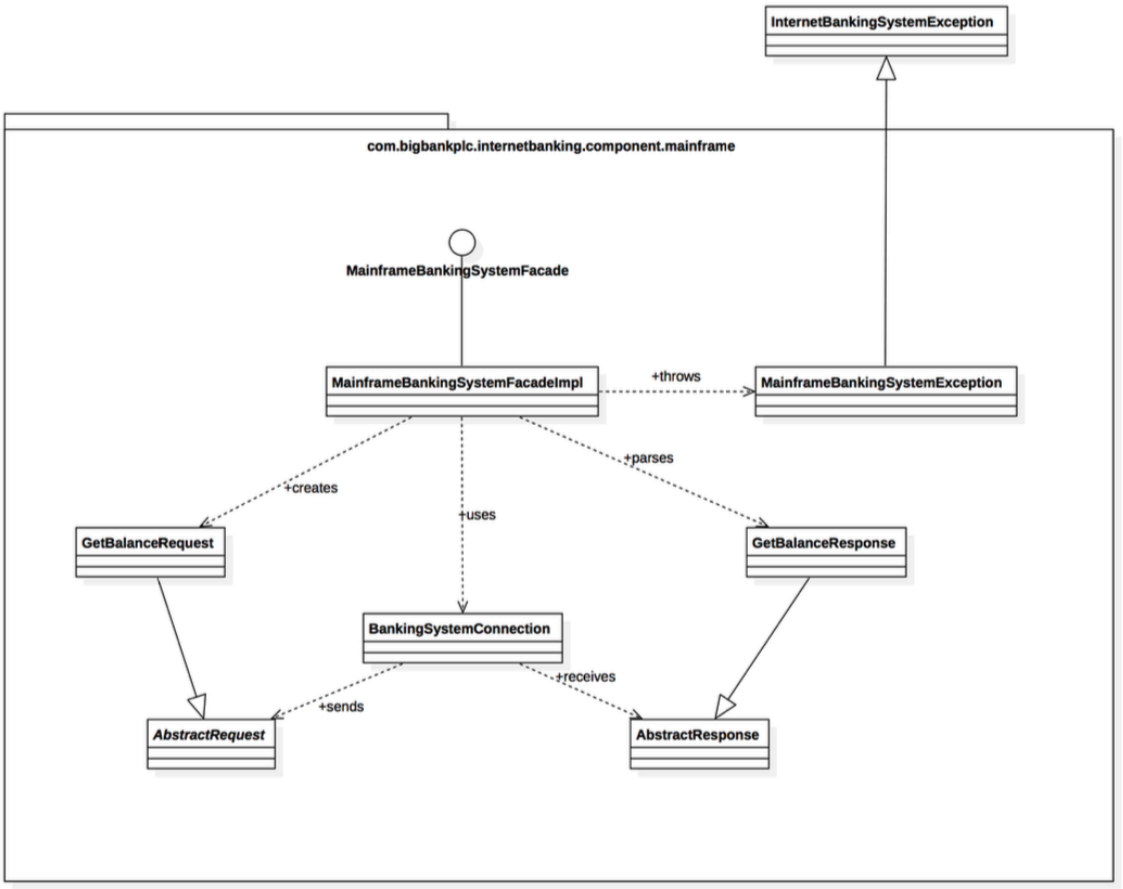
- كتاب الحجاب في الشرع والفطرة بين الدليل والقول الدخيل، عبدالعزيز الطريفي -فرج الله عنه-

## الفصل السابع: Level 4: Code-level diagram

بعد أن فتحنا الحاوية في المستوى الثالث لنرى المكونات التي بداخلها، نأخذ الآن مُكوّناً واحداً فقط ونضعه تحت ميكروسكوب عالي الدقة لنرى شيفرته البرمجية الداخلية وهيكليته وكيف تمت كتابته فعلياً.

إن الهدف من هذا المخطط هو عرض التفاصيل التنفيذية -Implementation Details- لمُكوّن محدد، فإذا كنت تستخدم لغة برمجة كائنية التوجه -OOP- مثل Java أو C#، فإن الأداة المثالية هنا هي ال UML Class Diagram، لذلك، يمكنك رسم هذا المخطط يدويا أو باستخدام الأدوات الآلية -Auto generation tools- التي تقرأ الشيفرة البرمجية لمكون ما، ونستخرج المخطط منه.

شاهد الصورة أدناه:




**A UML class diagram**

## نفخ التفاصيل: كم مقدار التفاصيل المطلوب في هذا المستوى؟

يطلق هنا Simon تحذيرا شديد اللهجة من خطأ شائع جداً وهو: نفخ التوليد الآلي!

عندما نستخدم أدوات توليد مخططات ال UML = فإنها ستقوم برسم كل متغير -Attribute- وكل دالة -Method- في الكلاس، والنتيجة هي لوحة مليئة بالتفاصيل المزعجة والمشتتة -تذكر مشكلة خريطة الطيران-، لذلك، عليك بهذه القاعدة: قاوم رغبتك في عرض

كل شيء! واعرض الخصائص والدوال "التي تخدم القصة" أو التي توضح المنطق الأساسي للمُكوّن فقط.

المخططات الكلاسيكية باستخدام ال UML فشلت في الماضي لأن المطورين أو المعماريين حاولوا استخدامها لرسم "النظام بأكمله"، فنتجت شبكة معقدة من الخطوط المتقاطعة، لكن في نموذج C4 = فنحن نحصر استخدام ال UML داخل حدود مكون واحد فقط، مما يبقي المخطط صغيراً ومقروءاً ^^

وقد تتسائل عن الفائدة المرجوة من هذا المستوى، وهذا سؤال واقعي، فهذا المستوى يمثل جسر بين عالمين يبدوان منفصلين أحياناً: عالم المعمارية -المربعات عالية المستوى- وعالم البرمجة -الشيفرة البرمجية-، لذلك، فإن الفئة المستهدفة من هذا النموذج هم المطورون داخل فريق العمل، فلا تعرض هذا المخطط على أي شخص خارج فريق البرمجة!

**إليك هذه الصدمة المعمارية: هل تعتقد أن هذا المخطط إلزامي أم اختياري؟**

على غرار المستوى الثالث، فهذا المستوى اختياري أيضا! وليس هذا فقط ^^، بل بشكل عام = لا يُنصح برسمه!

والسبب في ذلك يعود لماذا؟ لأن الحقيقة تعيش معنا داخل بيئة التطوير -IDE-، وبضغطة زر واحدة يمكننا الحصول على تفاصيل الكلاسات والواجهات عند الطلب مباشرة -On-demand-. فلماذا نضيع وقتنا في رسم وتحديث مخططات لشيء تمتلك الآلة القدرة

على قراءته وتوليده في أجزاء من الثانية؟ كما أن هناك مشكلة أخرى، فالأدوات التي تنشئ مثل هذه الرسومات لا تعرف ما يهم؛ لذلك فهي تقوم بتوليد وتمثيل كل شيء أمامها، وهذا يذكرنا أيضا بمثال ال PetClinic في الفصل الثاني.

هذا كلام جميل جدا، لكن متى نستخدم هذا المخطط؟
يمكن القول أننا نرسم المستوى الرابع في حالتين نادرتين، هما:

1. إذا كان المكون معقدا جدا ويحتوي على خوارزمية حرجة تحتاج لشرح.
2. إذا كنت تريد توثيق Design Pattern معين يتم استخدامه بكثرة في مشروعك وتريد إيضاحه باعتباره قالب تعليمي للمبرمجين الجدد.

الخلاصة:

إذا كنت تظن أن المعمارية الجيدة تعني رسم كل Class وكل Method في نظامك، فإليك هذا الخبر السعيد: توقف عن الرسم!
إن المستوى الرابع هو مستوى اختياري، ولا يُنصح بتضييع الجهد فيه إلا في أضيق الحدود، لأن المعمارية الذكية لا تنافس بيئة التطوير -IDE-؛ طالما أن المبرمج قادر على استكشاف تفاصيل الشيفرة البرمجية بنقرة زر داخل ال Visual Studio أو IntelliJ، فلا داعي لتحويل هذه الشيفرة البرمجية إلى مربعات صماء!
إن المعمارية هي فن معرفة متى نتوقف عن التجريد.. وكما رأيت: لقد قمنا برسم السياق، ثم الحاويات، ثم المكونات... وهنا، نضع أقلامنا جانبا، ونضع أيدينا على لوحة المفاتيح لنكتب الشيفرة البرمجية!

فكرة: هل سمعت بقانون ميلر من قبل؟ هذا القانون يمثل مبدأ أساسي وشهير في علم النفس المعرفي صاغه عالم النفس الأمريكي "جورج ميلر" عام 1956 في ورقة بحثية بعنوان: "الرقم السحري سبعة، زائد أو ناقص اثنان"، وفكرته ببساطة أن الذاكرة العاملة القصيرة -Working Memory- للإنسان البالغ لا يمكنها معالجة أو الاحتفاظ بأكثر من 5 إلى 9 عناصر -أي 7 ± 2- في نفس الوقت وبشكل نشط، فإذا زاد عدد العناصر المعروضة عن هذا الحد، يبدأ العقل البشري في فقدان المعلومات والشعور بالتشتت والإرهاق المعرفي، ثم حدث تطور على هذا القانون الكلاسيكي بحيث تم تقليل القيمة إلى 4 ± 1... وهذا كله مهم في مواقع متعددة مثل المخططات عند رسم المكونات أو تصميم المواقع أو برمجة ال method...إلخ

## الفصل الثامن: Notation

بعد أن تحدثنا عن المفردات المشتركة في الفصل الثاني -الأساس-، وتحدثنا عن المستويات الأربعة في نموذج C4 من الفصل الثالث وحتى السابع -البناء-، ينبغي علينا أن نتابع لنتعلم الرسم بشكل صحيح، فإذا كنت تذكر، قلنا أن ال Vocabulary أهم من ال Notation، وهذا التسلسل المنطقي بعد أن وضعنا الأساس ورفعنا البناء، فعلينا أن نعرف كيف سنزينه .^^

تترجم كلمة Notation بشكل حرفي إلى كلمة "ترميز"، وقد تشير إلى معنى التدوين، ويقصد بها هنا مجموعة الرموز والاتفاقيات البصرية للتعبير عن الأفكار، ويشمل هذا الأشكال والألوان والخطوط...إلخ، ولعلي سأستخدم مصطلح "الترميز البصري" للإشارة لهذا المفهوم.

### 1. العناوين -Titles-

إن أول قاعدة في الرؤية البصرية هي الوضوح ^^، فإذا كنت تستخدم لغة معيارية مثل ال UML، فقد يستنتج القارئ نوع المخطط من شكل العناصر، لكن في نموذج C4، فنحن نرسم مربعات وخطوط، لذا دون عنوان = سيبدو المخطط مجرد لغز هندسي!
والنصيحة البسيطة والمباشرة هي: "أدرج عنواناً واضحاً لكل مخطط"، ومع أن هذه النصيحة أساسية وبسيطة إلا أنها مهملة، ولأثبت لك ذلك، سأسألك سؤالا: كم مخططاً رأينا في الفصل الأول دون عنوان واضح؟
أظنك تذكرت الآن أن مخططات الفصل الأول معظمها كان دون عناوين! لذلك، تبدأ التوصية البصرية الأولى عندنا بالأبسط: وهي اكتب عنوان المخطط ^^

كيف نكتب العنوان؟

- يجب أن يكون قصيراً وذا معنى، أي لا فقرات طويلة، ويُخبرنا بمحتوى المخطط.
- يجب أن يوضح "نوع المخطط": لا تكتب "نظام المخاطر المالية" فقط، بل اكتب [مخطط سياق النظام] - نظام المخاطر المالية، وهذا الأسلوب هو الذي رأيناه بالأمثلة الخاصة بنموذج C4.

- الترقيم -التسلسل-: إذا كانت المخططات تُقرأ كقصة متسلسلة (مستوى 1 ثم 2 ثم 3)، فاستخدم أرقاماً في العناوين لترشد القارئ للترتيب الصحيح، وهذه النقطة ترتبط فيما تحدثنا عنه بالفصل الأول: "غياب نقطة الانطلاق -No context-".

**2. مفاتيح الخريطة -Keys and legends-: لا تفترض أن القارئ هو المحقق كونان!**

في الصفحات الأولى من هذا الكتاب، تحدثنا عن مشكلة مضحكة وقع بها فريق برمجي أثناء رسم مخططه المعماري، وكان ذلك في خضم حديثنا عن ال The Functional View، وذكرنا يومها تعليقا ساخرا ل Simon على المخطط حول سمك الخط في أحد الصناديق، فقال: "بصراحة لا أعرف! لكنني أظن أن الشخص الذي كان يرسم على السبورة قام بإمالة القلم قليلاً فتغير سمك الخط!" ^^

هذه المشكلة حان وقت حلها والحديث عنها، والحل ببساطة هو ألا تظهر أي تغيير بصري دون إيضاحه! فالجميع يجب أن يعرف المكتوب من خلالك، لا أن يقوم بدور المحقق كونان ليكتشف جريمتك! وهذا سيقودنا بكل تأكيد للتحذير من "فخ الافتراضات"، فقد ترسم نظامك الجديد باللون الأحمر، والأنظمة القديمة باللون الرمادي، وسيبدو هذا منطقياً جداً في رأسك! لكن ما تراه أنت بديهياً، قد يفسره شخص آخر بخلفية مختلفة بطريقة خاطئة تماما، مثلا: قد يظن مهندس التشغيل أن المربع الأحمر يعني نظاما معطلا أو به خطأ!

إن ال UML التي تحدثنا عنها سابقا -تذكر المقارنة مع الهندسة الكهربائية في الفصل الثاني-، كانت مزيتها الخرافية هي بكونها لغة موحدة، يفهمها الجميع بنفس الطريقة وبغض النظر عن مكانك، لكن عندما نتحدث هنا عن رسم معماريتنا فنحن نتحدث عن رسم صناديق وخطوط بشكل حر، لذلك، فلغتنا البصرية التي سنقوم باختراعها يجب أن تكون ذا معنى واضح لكل من يقرأها وسيقرأها! وهذا لا يمكن دون وضع مفتاح ودليل للمخطط الخاص بك! وهذا الأمر لا عيب فيه، فلا عيب في ابتكار أسلوبك الخاص في تلوين المربعات وتغير شكل الخطوط ونحو ذلك، لكن العيب أن لا يفهم أحد هذه الدلالات البصرية التي قمت برسمها، وحتى أنت قد تنسى اختراعك في المستقبل ^^... لذلك، قم بوضع Legend في إحدى زوايا المخطط يشرح كل دلالة لأي خيار بصري قمت به، وهذا يشمل هذه النقاط:

- الأشكال مثل ماذا يعني المربع ذو الزوايا الدائرية مقابل المربع الحاد؟
- الخطوط مثل ماذا يعني الخط المتصل مقابل الخط المتقطع مقابل الخط المنقط؟
- **الألوان مثل** ماذا يعني اللون الأزرق مقابل الرمادي مقابل الأحمر...إلخ؟
- **الحدود مثل** ماذا يعني الإطار الخارجي المتقطع؟
- **الاختصارات مثل AHAAH**، وهنا علينا شرح أي اختصار تقني كُتب داخل المربعات! - حتى لا تتعب، فهذه ال AHAAH تشير لإسمي فقط ^^، أرأيت أهمية مفتاح المخطط :D

هل تخيلت الآن الفرق؟ وهل أدركت جمال ومرونة وبساطة هذا النموذج وهذه الملاحظات الجميلة وأثرها على المخطط الذي ستقوم برسمه؟!

حسنا، بعد أن تحدثنا عن هذه الدلالات، ماذا عن الصناديق نفسها، أشكالها وما يكتب بداخلها؟

وهذا السؤال المهم سيقودنا لموضوع جديد، وهو الأشكال الهندسية وهويتها

## الأشكال الهندسية وهويتها

بعد أن تحدثنا عن العناوين والدلالات البصرية وأهمية المفاتيح، ننتقل إلى جزئية مهمة مرتبطة بما تحدثنا عنه ارتابطا وثيقا، وهي جزئية أشكال الصناديق التي نرسمها وما تحتويه في داخلها = أي أننا سنتحدث عن شكل المربعات من الداخل وما ستكتبه فيه = الهوية الخاصة بكل مربع!

ضمن حياتك البرمجية أظنك شاهدت الكثير من المخططات المعمارية، ولعل لاحظت أن هذه المخططات تعج بالأشكال الهندسية المعقدة، مثل استخدام الاسطوانات للدلالة على قواعد البيانات، والغيوم للدلالة على السيرفرات، والتروس للدلالة على الخدمات! لكن، وعلى الرغم من أن الناس اعتادت على تفسير الأسطوانة كقاعدة بيانات، إلا أن هذا ليس قاعدة ثابتة دائما، لكنها قد تستخدم أيضا للدلالة على أي شكل من أشكال تخزين البيانات، أو تشير إلى مخزن ملفات -Blob storage-... وهنا دون مفتاح سيخمن القارئ! لذلك يقترح هنا Simon منهجية ممتازة وتوصية مهمة لأي مبرمج أو معماري، وهي باختصار: "ابدأ بالأبسط"... يقول Simon: "ابدأ دائماً بأبسط شكل ممكن وهو المربعات والخطوط -Boxes and lines-، فهذه البساطة المتعمدة -Utilitarian notation- تجعل رسم المخطط سهلاً

جداً سواء على ال Whiteboard، أو باستخدام ال Sticky notes، أو عبر أدوات البرمجة! وبعد أن تكتمل خريطتك -مخططك-؛ يمكنك لاحقاً إضافة الألوان أو تغيير شكل الإطار الخارجي لإضافة لمسة جمالية أو دلالية بصرية"... ويمكن تمثيل كلام Simon بخطوات منهجية وهي:

الخطوة 1: ابدأ بـ "صناديق وخطوط" بسيطة

الخطوة 2: أَضف أشكالا للمعنى

الخطوة 3: أَضف ألوانا للتمييز

الخطوة 4: أَضف حدودا للتأكيد

ولاحظ أن الخطوة الأولى هي المهمة والمطلوبة فقط، أما باقي الخطوات في خطوات جمالية مفيدة للدلالة البصرية وتحسينها... وهذا له إيجابيات كثيرة، فالبساطة تُجبرك على الوضوح في المحتوى، ومن ثم فإن الإضافات تعزز الرسالة، لا تحل محلها، ثم يأتي دور الجمال والذي تكون أهميته ثانوية أمام الفهم!

ملاحظة: بالمناسبة هذه المنهجية أو هذا الأسلوب نستخدمه في حياتنا الواقعية بكثرة، فمثلا عند كتابتي لأي موضوع في هذا الكتاب أقوم أولا بقراءة الموضوع من الكتاب الذي أرغب بشرحه، ثم أقوم بكتابة أهم النقاط والأفكار فيها، ثم أبدأ بتحويل هذه الأفكار إلى فقرات ذات معنى، ثم أقوم بترتيبها وتزينها وإضافة المكنهات عليها :P... لذلك، لا شيء جديد هنا، وإنما هي استخدام لمهاراتنا في الحياة العملية لتكون عونا لنا في حياتنا المعمارية.

حسنا، الآن تحدثنا عن الشق الأول من العنوان "الأشكال الهندسية"، لكن ماذا عن الشق الثاني منها، ماذا عن "هويتها"؟

إن الهوية التي سنتحدث عنها الآن هي اختراع Simon ^^، نعم يا صديقي، فهو يطبق القاعدة التي كنا نتكلم عنها طوال هذا الكتاب، فلا بأس باختراع تدوين خاص بك، لكن بشرط شرحه!
إن المخططات التي رأينها أثناء شرح نموذج C4 كانت تقسم هويتها المعمارية لأربعة أنواع وهي:

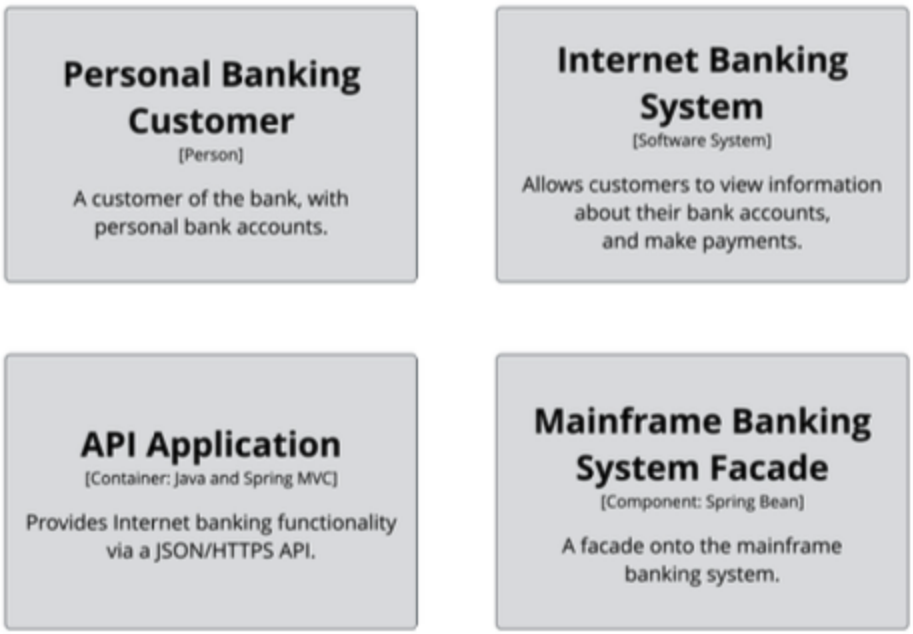
| النوع | المعلومات التي كتبناها |
|---|---|
| **Person** | الاسم + الوصف |
| **Software system** | الاسم + الوصف |
| **Container** | الاسم + **التقنية** + الوصف |
| **Component** | الاسم + **التقنية** + الوصف |

هذا النمط من الرسم والمعلومات أنشئ لدينا هوية بصرية رأيناها في كل المخططات، ولاحظنا بعض المميزات البصرية أيضا فيها، لكن قبل الحديث عنها دعونا نتحدث أكثر عن المشكلة، وهي مشكلة المربعات الخرساء! والتي رأيناها في أول صفحات الكتاب في مشاكل الصناديق الغامضة والتي عنونت بمشكلة قائمة التسوق ومشكلة العرض الوظيفي ونحوها...

إن معضلة التسمية تعد واحدة من أشهر وأكثر النقاط التي تضرب العصب السابع عند المبرمجين ^^؛ في محاولة للوصول لأفضل تسمية لأي شيء نكتبه أو نرسمه أو نعرفه، لذلك هناك مقولة مشهورة عندنا: "أصعب شيء في البرمجية هو التسمية الصحيحة".

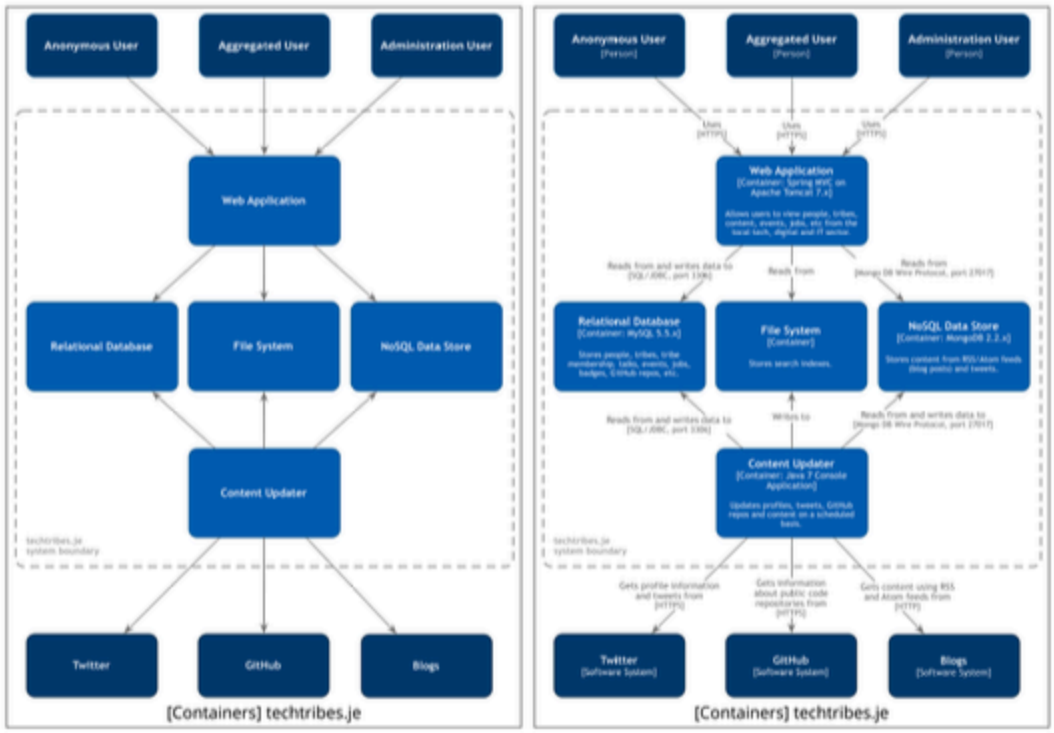
والآن خذ هذه المفاجأة = إن الخطأ الكارثي والشائع الذي يقع فيه المبرمجون هو رسم المخططات المعمارية بأسماء فقط! نعم بأسماء فقط! إن المربعات التي تحتوي اسم مثل Manager أو Data Processor وإن كانت صحيحة فهي بذاتها غامضة وتفتح بابا واسعا للتفسيرات والغموض، فمثلا هل المربع الذي يحمل الاسم Data Processor يعالج الصور أم يحسب الضرائب؟ وهنا يأتي دور نموذج C4 كما رأينا بقاعدته الذهبية = لا تترك الاسم وحيدا في المربع، بل يجب عليك إضافة الوصف من خلال كتابة سطر قصير يشرح وظيفة هذا المربع -يحدد مسؤوليته- أو قائمة نقطية صغيرة ويفضل ألا تتجاوز ال ٩ نقاط كحد أقصى -تذكر قانون ميلر، والنسخة المحدثة الحد الأقصى هو ٥-؛ هذا النص الصغير سيحول المربع الغامض إلى بطاقة تعريفية تتيح لأي شخص فهمه بلمح البصر! ويتم استخدام خط أصغر -Smaller font- لكتابة الوصف تحت العنوان، وهذا يحافظ على جمالية المربع ويبقي الإسم بارزا.

حسنا الآن انظر لهذه الصورة:




**Examples of the elements I use on my diagrams**

هذه الصورة تمثل التطبيق العملي لما ذكرناه هنا... فهي في قمة البساطة وتحتوي اسم ووصف، وهي نفسها التي يعتمدها Simon عند رسم أي صندوق... يبدأ ببساطة، دون أشكال وتعقيدات، ويضع اسم ووصف، ومبروك، لديك معمارية جميلة وجاهزة وسهلة الرسم والتعديل بشكل يدوي أو من خلال الأدوات... ويمكن تحسين جماليتها بأي وقت لاحقا... وشاهد هذه الصورة الآن والتي تظهر أثر إضافة الوصف على المخطط:




**Adding additional descriptive text to diagram elements can remove ambiguity**

## الألوان: ليس للزينة فقط، بل تنقل تصورات ورؤى بصرية!

المخططات المعمارية لا يفترض أن تكون كئيبة ومجردة ومرسومة باللونين الأبيض والأسود فقط! لأن إضافة الألوان واحدة من الطرق العبقرية لإضافة أبعاد جديدة للمخطط دون زيادة عدد المربعات أو النصوص، ومع ذلك عليك الحرص على أن المخطط يجب أن يكون مفهوماً قبل إضافة اللون، فاللون يُحسن ولا يجب أن يكون الحل! إن اللون يستخدم للتمييز وتسليط الضوء والتخفيف من بروز العناصر أخرى!

يمكننا استغلال الألوان لعدة أهداف، منها:

- حالة النظام: للتمييز بين نظام قائم بالفعل مقابل نظام جديد سيتم بناؤه.
- طبيعة البناء: للتمييز بين منتج جاهز تم شراؤه مقابل نظام برمجه الفريق داخليا.
- البيئة والملكية: للتمييز بين أنظمة داخلية نملكها مقابل أنظمة خارجية لا نملكها.
- نوع التقنية أو المنصة: للتيميز بين التقنيات مثل أحمر لل JAVA application وأزرق لل .NET.
- مستوى الخطر أو التعقيد: إبراز أن اللون الأحمر يمثل خطر/معقد جداً، والأصفر يمثل مستوى متوسط، والأخضر يمثل شيء آمن/بسيط.
- خطة العمل -Sprint-: مثل العناصر الحمراء سيتم تعديلها في التحديث القادم، في حين أن الأخضر سيشمل العناصر التي لن يتم المساس بها، أما الأصفر فهي التي سيتم حذفها.

لكن جميع هذه الأهداف ستصبح هباءا إن لم تراعي بعض الملاحظات المهمة وهي:

- تذكر ال Legend، فعدم إضافتك لمفتاح الخريطة عند استخدامك للألوان يهدم كل ما تحدثنا عنه من فوائد، ويضيف أثرا سلبيا على الفهم!
- تذكر أنك عضو من فريق، والفريق قد يكون بداخله من هو مريض بعمى الألوان، بدرجاته المختلفة، لذلك احرص على استخدام تباين لوني واضح، ولا تجعل اللون هو الطريقة الوحيدة للتمييز، مثلا يمكن استخدام لون مختلف وخط مختلف.
- تأكد من أن مخططك إذا طبع من خلال طابعة غير ملونة -أبيض وأسود- فسيبقى مفهوما!

**الأشكال -Shapes-:**

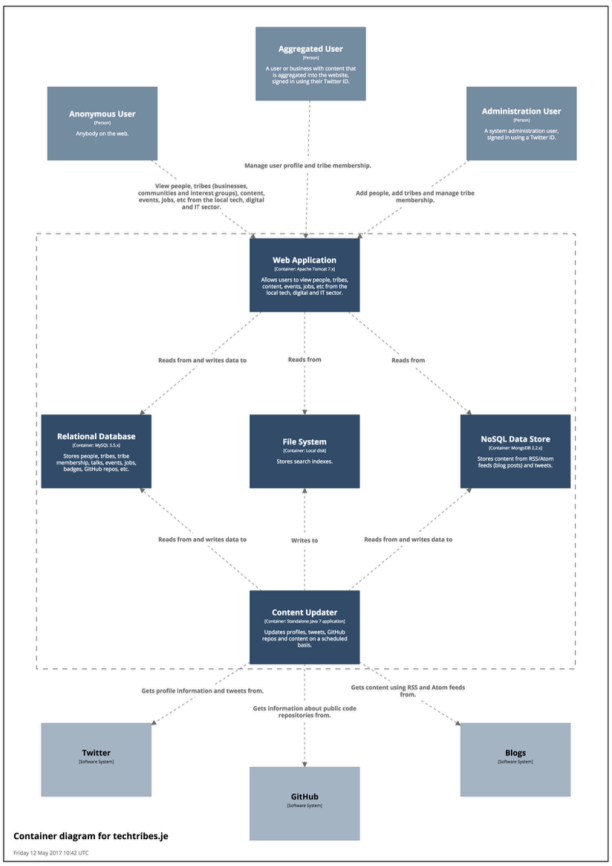
الناس يميلون لقراءة المخططات المريحة للعين!

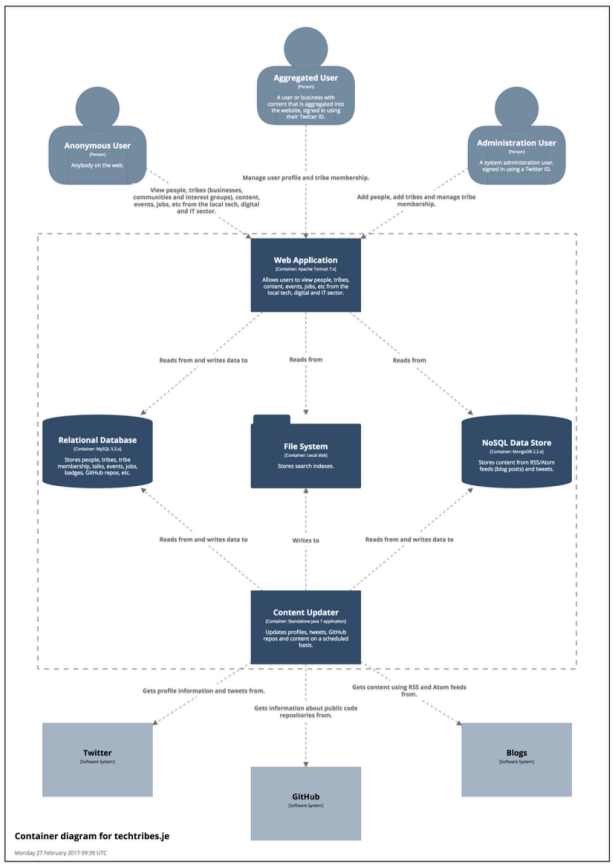
إن السطر السابق يمثل نقطة نفسية محورية لنا كبشر، وهي أن العين ترتاح للجمال، وهذا الارتياح يمكننا استغلاله لتحصيل فوائد كثيرة وجمة -وهذا ما حدث فعلا على أرض الواقع عندما رفض المبرمجون نماذج ال UML مع أنها تمثل قاموسا ضخما!-... ومع ذلك؛ يظن كثير من المطورين أن الاهتمام بجمالية المخطط ليس سوى أمر سطحي لا يليق بالتقنيين المحترفيين! لكن الحقيقة النفسية تثبت العكس: فالمبرمجون يتجاهلون المخططات القبيحة أو المزدحمة! بل إن استخدام الأشكال مثل شكل الأسطوانة لقاعدة البيانات، أو أيقونة شخص للمستخدم؛ يمنح الفريق قدرة على فهم المخطط بلمح البصر -At a glance-، ودون الحاجة لقراءة كل حرف، ولكن احذر من السقوط في فخ ال UML المتمثل في استخدام عشرات الرموز المعقدة التي تحتاج إلى قاموس لفكها أو تحتاج منك لحفظها!

والقاعدة هنا: ابق بسيطا = استخدم المربعات بشكل أساسي، وأضف الاسطوانات والأيقونات البشرية لتحسين القراءة، وتذكر أهم قاعدة معمارية: الشكل مجرد عامل مساعد، أما المعلومة الأساسية فيجب أن تُكتب كنص واضح داخل العنصر، مع توضيح دلالة الأشكال في مفتاح الخريطة -Legend-.

إن هذه النقطة ترتبط بما تحدثنا عنه سابقا تحت عنوان: "المقاتلون النمطيون، جنود بلا ملامح -Stormtroopers-"، الأشكال وحدها لا تكفي، النص أساسي والأشكال للجمال!

شاهد الصورة أدناه لمعرفة الفرق بين مخططين صحيحين، أحدها أجمل من الآخر فقط!




## الحدود -Borders-: فن التجميع والتأكيد

الحدود الخارجية للمربعات ليست مجرد ديكور، فيمكنك فيها استخدام أنماطا مختلفة من الخطوط -مزدوجة، أو ملونة، أو متقطعة- لتحقيق هدفين أساسيين:

*. التجميع -Grouping-: كما رأينا في مخطط الحاويات، قمنا باستخدام الخط المتقطع كإطار يحيط بمجموعة من الحاويات لنقول للقارئ: "كل هذه العناصر تعيش معا داخل نفس النظام".

*. التأكيد -Emphasis-: يمكنك جعل إطار أحد المكونات "سميكاً" أو "مزدوجاً" لتوضيح أنه المكون المركزي (Core) في النظام.

وبكل تأكيد، تستمر معنا القاعدة اللولبية = كما هو الحال مع الألوان والأشكال، أي إطار مميز تستخدمه يجب أن تشرح معناه فوراً داخل المخطط -Legend-.

**الأحجام ليست عشوائية وذات معنى: الفخ النفسي!**

إذا قمت برسم مخطط وجعلت أحد المربعات "أكبر حجماً" من البقية لأي سبب مثل: أن اسمه طويل أو فقط لملء الفراغ في الرسمة؛ فإن عين القارئ الذي يقرأ المخطط ستترجم هذا الكبر -الحجم- تلقائياً إلى كونه أكثر تعقيداً، أو أكثر أهمية، أو يمثل عبئاً برمجياً أكبر!
ويكمن الخطر هنا إذا كان المربع الكبير يمثل في الحقيقة خدمة بسيطة جداً، فنكون حينها قد أرسلنا رسالة معمارية مضللة للفريق!

وهنا تأتي نصيحة Simon: ارسم كل العناصر بنفس الحجم تقريباً، إلا إذا كنت تقصد "عمداً" إيصال رسالة بأن هذا المكون أضخم أو أكثر تعقيداً من غيره ^^

وهنا يجب علينا أن نلخص ما ذكرناه تحت عنوان مهم:

## احذر الرسائل البصرية الخفية في مخططاتك!

فالمخطط المعماري يتحدث حتى عندما لا تكتب فيه أي نص! فرسمك لمربع حجمه كبير بجوار مربع صغير سيرسل رسالة فورية ومضللة لعقل المبرمج بأن المربع الكبير هو الأكثر تعقيداً والأكثر أهمية في الشيفرة البرمجية!

إن المعماري المحترف يوحد أحجام المربعات، ولا يغيرها إلا إذا أراد إيصال رسالة متعمدة حول حجم أو تعقيد هذا المكون، وهذا أيضا يدعونا لنصحح خطأ تقليدي ورثناه من ال UML، ففي ال UML نستخدم أحياناً أيقونة Stick man لتمثيل أي نظام خارجي يتصل بنظامنا، وهنا يدعونا Simon للتمرد على هذه القاعدة غير المنطقية! فالنظام البرمجي الخارجي ليس إنساناً، فلماذا نرسمه كإنسان ونربك القارئ؟

القاعدة في C4 بديهية ومريحة: البشر نرسم لهم أيقونة إنسان، والأنظمة البرمجية نرسمها على شكل مربعات! المعمارية الناجحة لا تحتاج إلى فك طلاسم، بل تعتمد على المنطق السليم المباشر! وهذا ما رأيناه فعلا عندما تحدثنا عن نموذج C4.

## الخطوط!

تمثل الخطوط الغراء الذي يربط الصناديق معا، وهي بذلك تمثل العلاقات بين الصناديق الموزعة، وهذا يعني أنها جوهر أي معمارية...

إن المطورين عادة ما يركزون على الصناديق نفسها وينسون العلاقات التي بينها -الخطوط-، وبذلك كثير ما نحصل على خطوط عشوائية وغير مسماة وغير موجهة... وهذا تماما ما شاهدناه في خريطة الطيران ^^... لذلك لا تتجاهل العلاقات -الخطوط- في معماريتك!

حسنا، هذا سيطرح سؤالا مهما، كيف سنرسم هذه الخطوط؟ وماذا عن اتجاه الخطوط؟ وكيف سنصف الخطوط؟...إلخ

وحقيقة هذه أسئلة خطيرة ومهمة، خصوصا الجزء المتعلق باتجاه الخطوط... لذلك تعال معي لنتحدث عن آليات وتحديات رسم الخطوط...

أولا: اتجاه الخطوط -Directionality-:

إن البيانات أو مخطط سير العمل إما أن يكون أحادي الاتجاه -Uni-directional- بحيث يكون مخطط السير في اتجاه واحد فقط (الاتجاه الأساسي) وإما أن يكون ثنائي الاتجاه -Bi-directional- بحيث يكون سير العمل ذهابا وإيابا، وهذا سيطرح سؤالا مهما، كيف سنتعامل مع أمثال هذه الخطوط؟ وأين سأقوم برسم رأس السهم؟ كيف سينعكس هذا على نموذج C4 الذي تعلمنا رسمه سابقا؟

وهذه الأسئلة الجميلة إجابتها ببساطة من خلال اختيار الاتجاه الأهم أو اتجاه الاعتمادية ليكون هو اتجاه السهم، ورسم السهم ليكون أحادي الاتجاه! وحقيقة هذا الاختيار هو الخيار المفضل Simon، فإذا كان لديك موقع إلكتروني س يعتمد على خدمة ص، فإذا بدأ س بإرسال Request ل ص وقام ص باستقباله، فإن اتجاه السهم سيكون من س إلى ص!

ومع ذلك، هناك مجال من الحرية هنا، لكنها يجب أن تكون منضبطة ومتسقة، بحيث بتم استخدام نفس الأسلوب في جميع المخططات، كما يجب أن يكون هناك تطابق بحيث يطابق النص الخاص بالسهم اتجاه السهم!

مثال: إذا قلنا أن س -> ص = فالنص هنا يجب أن يكون س يستخدم ص أو س يجلب البيانات من ص، وهذا ما يسمى التطابق، لكن لو قلت هنا أن ص يرسل البيانات إلى س فهذا النص يعارض الاتجاه وإن كان صحيحا!

حسنا، ماذا عن الوصف؟

حتى الوصف الخاص بالسهم لديه مجموعة رائعة من القواعد، فمثلا ينبغي علينا استخدام حروف الجر خصوصا الشائعة منها مثل: "to, from, with, on, in" أو "إلى، ومن، ومع، وعلى، وفي"... والفكرة الأساسية من استخدام حروف الجر هي توضيح اتجاه العلاقة... شاهد المثال التالي:

الوصف الجيد: جلب بيانات الحساب من -> MySql DB

الوصف الأقل جودة: جلب بيانات الحساب -> MySql DB

لاحظ أن قراءتك لحرف الجر حسن من الفهم الخاص بك وجعل العنصر البصري فعالا حتى قبل قراءة النص كاملا... وهذا له أثر مهم في المخططات الحقيقية...

حسنا، ما رأيك في سؤال آخر جميل؟

لو أردت تمثيل أمر ما مثل الخدمات المتزامنة وغير المتزامنة، فكيف ستقوم بتمثيلها؟

هنا نأتي لملعب أنماط الخطوط، فمثلا يمكن استخدام الخط المتصل للعمليات المتزامنة والخط المتقطع لغير المتزامنة ^^، وهذا بكل تأكيد = يجب أن يتم وضعه في مفتاح الخريطة -لا نريد أن ننسى ما تعلمناه- ^^

مثال:

ال Synchronous Interaction -تفاعل متزامن-: المُرسِل يَنتظر استجابة فورية. مثل: استدعاء HTTP REST... قمنا باستخدام الخط المتصل مع ال HTTP REST

ال Asynchronous Interaction -تفاعل غير متزامن-: المُرسِل لا يَنتظر. مثل: إرسال رسالة لـ JMS Queue... قمنا باستخدام الخط المتقطع مع ال JMS Queue.

في هذه الحالة ومجرد نظرة خاطفة على المخطط سنعرف نوع التفاعل ^^...

وهذا ينطبق على الألوان أيضا... فمثلا يمكن استخدام اللون الأخضر لأي اتصال HTTPS والبرتقالي لأي اتصال HTTP... إلخ.

هل نتابع بالخطوط؟ حتى لو سئمت، يجب أن نتابع :P، فهذا الموضوع المهم لا يمكن تجاوزه ^^.. والسؤال الجديد الآن، ماذا لو كانت هناك أكثر من علاقة مختلفة بين صندوقين؟ ماذا

لو كنا نقوم باسترجاع معلومات الملف الشخصي من Profile Information API ونقوم باسترجاع تغريدات هذا الشخص مثلا Tweets API، فكيف سنتعامل مع هذه الحالة؟

للتعامل مع هذه الحالة هناك طريقتين مختلفتين، الأولى من خلال رسم خط واحد مع وصف يجمع العلاقتين، والثاني هو رسم خطين منفصلين كل خط يمثل علاقة محددة... ويتم الاختيار بين هذين الأسلوبين بناء على رغبتنا، فإذا كان التركيز على الصورة الكبيرة نقوم برسم خط واحد -مخطط المستوى الأول-، في حين إذا كان التركيز عندنا على التفاصيل التقنية فحينها قد يكون عدة خطوط هو الأفضل -مخطط المستوى الثاني والثالث-.. تذكر ما تحدثنا عنه سابقا "**شنايدرمان**".

## الشاشات مقابل السبورات -Electronic tools vs Whiteboards-

والآن، لنعد للوراء قليلا، لقد تحدثنا سابقا عن مزايا اللوح الأبيض وعيوب استخدام التقنيات الخاصة بالرسم... لكن دعونا الآن ننتقل من جانب التنظير إلى الجانب العملي، وكيف ومتى سيكون من المفيد استخدام الأدوات ومتى سيكون من المفيد استخدام اللوح الأبيض! وإن كنت تظن أن هناك تعارض أو فصل؛ فأنت مخطئ! بل هي تكامل بين أسلوبين مختلفين في مرحلتين مختلفتين.

- إن أدوات الرسم الإلكترونية: تمنحك مرونة مطلقة؛ بحيث يمكنك سحب المربعات وتغيير حجمها بضغطة زر، لكن عيبها أنها "فردية" أو تعطي شعور مالك الفأرة ولوحة التحكم أو المخطط "بالفردية"، وتجعل التعاون مع الفريق صعباً.

- اللوح الأبيض -Whiteboard-: هي البيئة المثالية والذهبية لتعاون الفريق -Collaboration-، لكن المشكلة هنا هي في "التخطيط المسبق للمساحة" -Layout-، فالتعديل على اللوح مزعج جداً، فإذا نفدت المساحة أو أردت تحريك مربع ما، ستضطر لمسح أجزاء كبيرة من الرسمة وإعادة رسمها مراراً وتكراراً!

حسنا، لو أردنا الجمع بين مزايا الأسلوبين، فماذا سنفعل؟

هنا يظهر خيار هجين وهو استخدام ال Sticky Note مع اللوح الأبيض، أي في استخدام "الورق اللاصق" أو "البطاقات" لتكون بديلاً عن رسم المربعات بالأقلام، وهذا الأسلوب له مزايا ممتازة، كما أنه يتماشى تماماً مع تقنيات التصميم الجماعي مثل ال CRC أثناء الاجتماع! فإذا كنت تريد نقل مكون ما من حاوية لأخرى، فببساطة = انقل البطاقة بيدك!

وإذا اكتشفت أن أحد المكونات غير ضروري = انزع البطاقة وارمها في سلة المهملات بكل بساطة!

هذه المرونة ستجعل من جلسة التصميم حيوية وفعالة، وتسمح للفريق بتجربة عدة هياكل معمارية في دقائق دون عناء المسح والرسم في كل مرة ^^

ومع ذلك، فإنه كثرة البطاقات اللاصقة قد تجعل المخطط مزدحما خصوصا في أول الأمر، وسيتأثر الترميز البصري لأن الأشكال المتنوعة غير متوفرة، كما أن اللاصق على اللوح قد يجعلك تعاني من كثرة وقوعه على الأرض ^^ فستحتاج إلى لاصق احتياطي معك :P...

شاهد الصورة أدناه كمثال على هذا الموضوع:



**Examples of where sticky notes and index cards have been used instead of drawing boxes**

## المختصرات -Acronyms-

كل فريق برمجي يعاني مما يُسمى للعنة المعرفة؛ بحيث يعتاد هذا الفريق على مصطلحات داخلية خاصة بالبزنس -Domain Concepts- ويبدأون باختصارها، مثلاً: بدلاً من كتابة نظام إدارة الموارد البشرية فإنهم يكتبون HRMS، وبدلاً من كتابة نظام المخاطر يكتبون RS.

إن المشكلة هنا تكمن في أن الفريق الحالي يفهم هذه الاختصارات جيداً، ولكن عندما ينضم مبرمج جديد للفريق، أو يقرأ موظف من قسم آخر هذا المخطط = فإنه سيشعر وكأنه يقرأ طلاسم فضائية ^^، لذلك ينبغي علينا تجنب استخدام الاختصارات قدر الإمكان، وهذا يعني

أن علينا كتابة الاسم كاملا، فإذا اضطررت لاستخدام الاختصارات لتوفير المساحة داخل المربع، فيجب عليك وبشكل إلزامي كتابة معنى هذا الاختصار بالكامل في مفتاح الخريطة أو في قاموس مصطلحات ملحق بالمخطط...

لكن هناك ملاحظة مهمة جدا، هذا الكلام لا ينطبق على كل المصطلحات! فمثلا كتابة JavaScript Object Notation بدلا من JSON هذا = عبث و"فزلكة" ما لها من داع! فأي مصطلح تقني مشهور عالميا لا داعي لكتابته كاملا، بل يمكن الاكتفاء بكتابة اختصاره كما في JSON أو HTTP أو API أو PHP!

## سمات الجودة -Quality attributes-

هل تتذكر سمات الجودة؟ وهل تتذكر كم تحدثنا عنها في الجزء الأول من الكتاب؟!

الآن آتانا تحدي مهم يتعلق بها، والتحدي المعماري هنا هو:

نحن نرسم "مكونات برمجية"، فكيف نرسم أشياء غير ملموسة مثل "السرعة" أو "الأمان" على المخطط المعماري؟

يمكن الإجابة على هذا السؤال من خلال تجزئة الإجابة لثلاثة أجزاء، وركز معي هنا:

1. **الإضافة النصية المباشرة للأرقام والمقاييس**: إذا كانت سمة الجودة عبارة عن "رقم" أو "مقياس أداء"، فإن أسهل طريقة لإظهارها هي إضافتها كنص داخل المربع أو على السهم أو داخل المربعات، وذلك من خلال إضافة ملاحظة صغيرة في وصف العنصر توضح حجم

المستخدمين مثل: يتوقع 10,000 مستخدم متزامن أو على الأسهم مثل أن تكتب على السهم: ينقل 50GB يومياً أو أن زمن الاستجابة المستهدف أقل من 200ms... إلخ، هذه الإضافات البسيطة تجعل المخطط ينبض بالحياة ويوضح "حجم المشكلة" التي يحلها النظام.

2. **معضلة الأمان والاهتمامات المتقاطعة:** ماذا عن الأشياء المعقدة مثل "الأمان" (Security) أو التشفير؟ هنا يرفع Simon راية التحذير، فيقول: لا تحاول رسم كل شيء! لأن محاولة رسم آليات الأمان المعقدة ستدمر بساطة المخطط، والأفضل دائماً هو إخراج هذه التفاصيل من الرسمة، وكتابتها في وثيقة مساعدة بسيطة -Lightweight supplementary documentation- يتم إرفاقها مع المخطط... إن المخطط يرسم الهيكل، والوثيقة تشرح القواعد ^^.

3. **قابلية التوسع والبنية التحتية:** هنا نصل إلى خطأ شائع جداً يقع فيه المعماريون؛ وهي محاولة رسم ال Load Balancers أو ال Clusters/Replicas Servers في مخطط الحاويات لبيان أن النظام قابل للتوسع -Scalable- أو متاح دائماً -Highly Available-، وهذا مرفوض تماما! لأن مخططات السياق -المستوى الأول-، والحاويات -المستوى الثاني-، والمكونات -المستوى الثالث- هي مخططات "برمجية/منطقية"، وليست مخططات بنية تحتية! والحل لهذا النوع من السمات يكون من خلال لنوع آخر من المخططات يُسمى "مخطط النشر" -Deployment Diagram-، والذي يوضح كيف يتم تسكين الشيفرة البرمجية على خوادم

مادية أو سحابية حقيقية = أي البنية الفيزيائية... -وسنتحدث عن هذا المخطط لاحقا بإذن الله-.

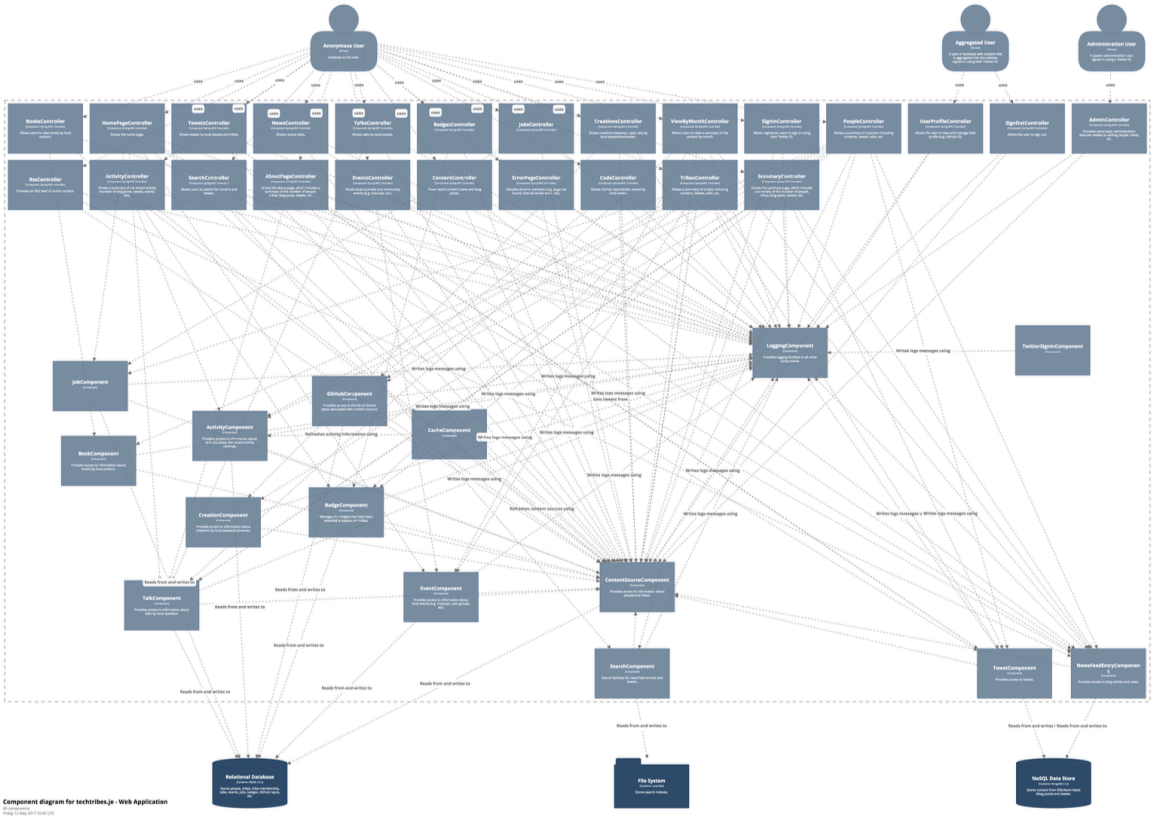
إن ما نحاول قوله في هذا الباب المهم = إن المعماري الحقيقي يعرف متى يتوقف عن الرسم، ومتى يترك الكلمة للوثائق المكتوبة!

**نطاق المخطط -Diagram scope-**

بعد هذا الحديث المطول... ستظهر لدينا ملاحظة مهمة، ألا ترى أن معظم أو جميع الأمثلة أو الرسومات الجميلة التي قمنا بطرحها كانت بسيطة وواضحة! ألا ترى أن المشاريع الحقيقة والواقعية أعقد من ذلك بكثير وأكبر من ذلك بكثير؟!

والجواب، نعم بكل تأكيد! إن الأمثلة التعليمية بكل تأكيد لن تمثل إلا الجزء البسيط والواضح والمفاهيمي لأي مخطط، لكن مع ذلك إن الأنظمة المعقدة والكبيرة هي في كينونتها أنظمة مصغرة مجمعة معا ^^

شاهد هذه الصورة:




**A component diagram for the techtribes.je web application**

حسنا، كيف يمكننا معالجة هذه المشكلة في مخططاتنا؟ ماذا سنفعل إن لم يعد يتسع المخطط لأن يكون في صفحة واحدة؟

لحل هذه المشكلة سيخطر على بالنا عدد من الحلول، وهي:

١. الحل المباشر والسريع هو تكبير حجم الورق، فإذا كنا نرسم على ورق A4 سنتجه لورق A3، وإذا كان اللوح صغيرا نذهب للوح أكبر... وأظن لا داعي لنكمل، فمع أن هذا الحل بسيط إلا أنه حل ساذج ^^... وسيجعلنا نقع بالعديد من المشكلات، فزيادة الحجم ليس الحل! وتذكر أن زيادة الحجم= زيادة في المعلومات = مخطط مزدحم... وهذا سيعود بنا

للمشاكل التي تحدثنا عنها في أول هذا الكتاب مثل خريطة الطيران! ومشاكل الحمولة المعرفية -Cognitive Load- وقانون ميلر...إلخ

٢. تشريح الفوضى: إذا قمت بتوليد مخطط مكونات بشكل آلي يشمل كل النظام، فستحصل على "طبق معكرونة"، المشكلة هنا أنك لن تستطيع التمييز: هل الشيفرة البرمجية نفسها سيئة، أم أن الرسمة تحتوي على تفاصيل مفرطة؟
بالتدقيق في المخطط الفوضوي الذي بالأعلى، سيتبين أن سبب الازدحام فيه يعود لثلاثة عوامل:
أ. الطمع البصري: محاولة رسم كل ال Controllers وكل مسارات التبعية في تطبيق الويب في لوحة واحدة.

ب. المكونات المركزية: وجود مكون أساسي مثل ContentSourceComponent تستدعيه أغلب المكونات الأخرى، مما يخلق شبكة من الأسهم تتقاطع في المركز.

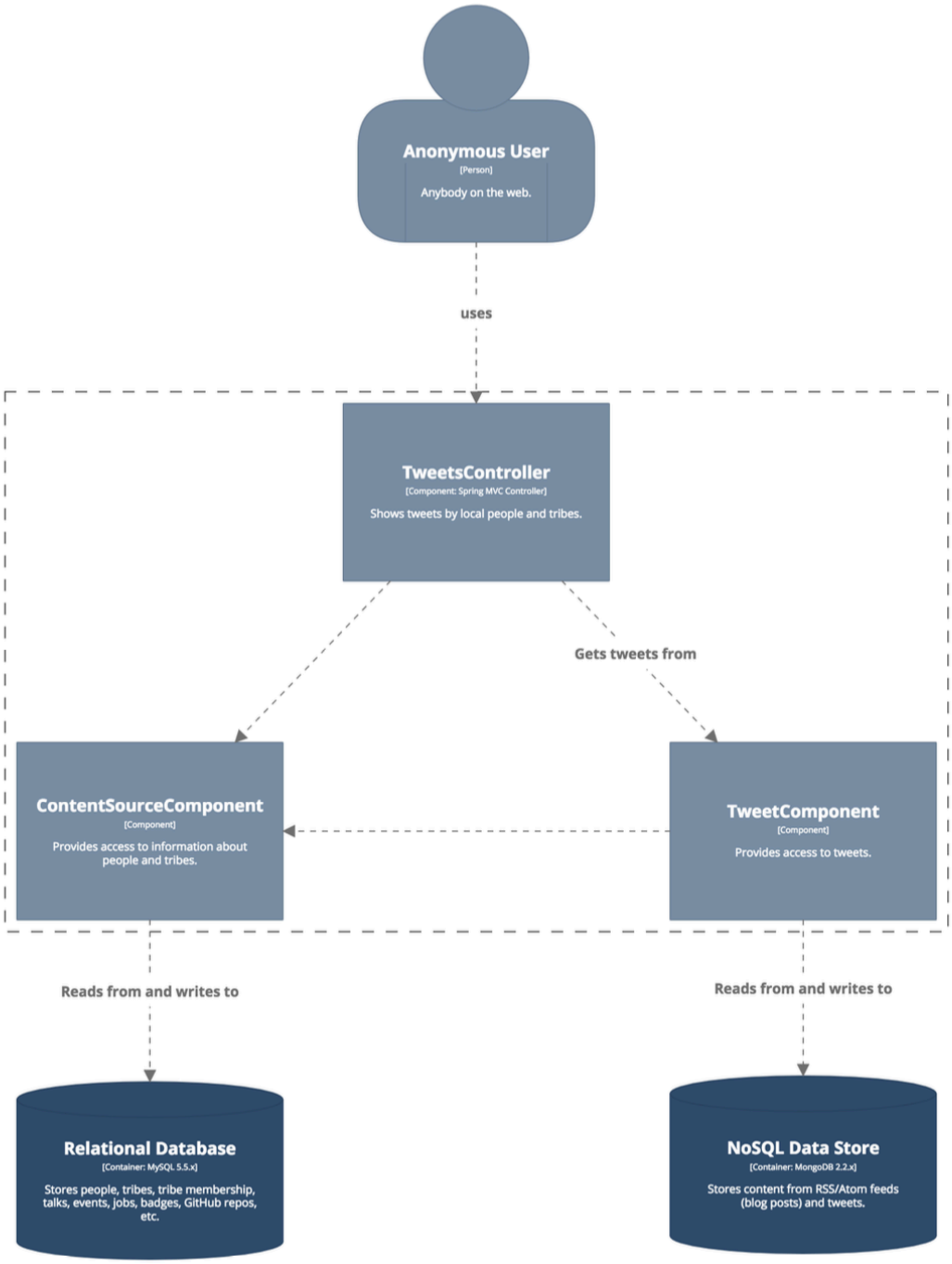
ج. الاهتمامات المتقاطعة: مكونات البنية التحتية مثل خدمة تسجيل الأحداث -LoggingComponent- يتصل بها الجميع تقريباً.

بعد تشريح هذه الفوضى = نستنتج أن إزالة مُكوّن الـ Logging لن يحل المشكلة بشكل جذري، كما أن تكبير الصفحة لن يجدي نفعا! نحن بحاجة إلى تغيير "عقلية الرسم" بالكامل!

وهذا سيجعلنا نضع فائدة جميلة: إن المخطط المُنظَّم يكشف الحقائق سواء أكانت سيئة أم جيدة بالنسبة لك ^^

٣. سياسة فرق تسد:

إن الحل الجذري والاحترافي هو تمزيق المخطط العملاق إلى مجموعة من المخططات الصغيرة والمستقلة! فبدلاً من رسم تطبيق الويب بالكامل في مخطط مكونات واحد، قم بتقسيمه بناءً على سياق محدد، مثل: حالة استخدام -Use Case- أو ميزة برمجية -Feature set- أو تفاعل مستخدم معين -User interaction-، وكمثال عملي على نذكره = بدلاً من رسم كل الـ Controllers، ارسم مخطط مكونات خاصاً بـ TweetsController فقط، هذا المخطط سيعرض حصرياً المكونات والخدمات التي تعمل معاً لإنجاز مهمة "عرض التغريدات"، وبهذا ستظهر لنا مثل هذه الصورة:




## Component diagram for techtribes.je - Web Application

This diagram shows a slice through the web application, starting at TweetsController.
Friday 12 May 2017 10:42 UTC

لاحظ بساطة هذا المخطط مقارنة في المخطط الشامل، فلو وجد ١٠ مخططات مثل هذا لتغطية المخطط الكبير، فإنها ستكون بكل تأكيد أفضل بمليون مرة من المخطط المعقد الكبير ...^^

إن الشرط الأساسي لهذا التقسيم هو أن يروي كل مخطط صغير "جزءاً" من القصة الكلية للنظام، وبنفس مستوى التجريد، وهنا يجب أن ننتبه أن القيام بهذه العملية بشكل يدوي سيكون مرهقا بشكل كبير وغير عملي، ومن هنا تظهر أهملية استخدام الأدوات التقنية لتوليد مثل هذه المخططات بشكل آلي، وهذا ما سنتطرق إليه لاحقا بإذن الله...

الخلاصة:

إن أكبر خطأ يقع فيه المبرمج عند محاولة تطبيق مخطط المكونات على نظام حقيقي، هو إصراره على حشر النظام بأكمله في رسمة واحدة. وعندما تضيق به الصفحة، يلجأ للحل الساذج: تكبير مساحة الرسم أو حجم الورق!

تكبير اللوحة لا يحل المشكلة، بل يضاعف (العبء المعرفي) على من يقرأ المخطط! والمخطط المليء بمئات المربعات المتداخلة هو مخطط ميت سريريا؛ لأن هذا المخطط لن يقرأه أحد ^^

المعماري المحترف يطبق مبدأ فرق تسد، فلا ترسم تطبيقك في لوحة واحدة، بل جزّئه! وارسم مخطط مكونات منفصل لكل حالة استخدام Use Case أو لكل Controller على حدة. أو لكل منطقة وظيفية أو حسب منطق الأعمال...إلخ، إن النظر إلى 10 مخططات صغيرة وواضحة، أفضل بمليون مرة من التحديق في مخطط عملاق يشبه شبكة العنكبوت!

ملاحظة أخيرة قبل الانتقال للفصل التالي:

إن الكلام اللفظي الذي يقال في غرف الاجتماعات يتبخر بمجرد انتهاء الاجتماع، لكن المخطط سيبقى في وثائق الشركة لسنوات، لذلك، إن القاعدة الصارمة هنا هي:

أي جملة توضيحية تضطر لقولها بلسانك = يجب أن تأخذها وتكتبها فوراً في مفتاح الخريطة!

فالمخطط المعماري الحقيقي يجب أن يكون مستقلاً وقائماً بذاته -Stand-alone-، بحيث يقرؤه مبرمج جديد بعد 3 سنوات ويفهمه بالكامل دون الحاجة للاتصال بك هاتفياً لتشرحه له! هذه الخلاصة تمثل الحل أو النتيجة التي يجب أن تصل إليها إذا وقفت في اجتماع لتشرح مخططك للشباب الطيبة... فعندما تقف في غرفة الاجتماعات لعرض مخططك المعماري الجديد على فريق التطوير، فإن مهمتك لا تقتصر على الشرح، بل يجب أن تستمع بدقة لما يدور في الغرفة، فالأسئلة التي يطرحها الحضور مثل: "ماذا يقصد هذا المربع؟" أو "هل هذا السهم يعني اتصالاً عبر الشبكة؟" ليست مجرد أسئلة عابرة، بل هي في الحقيقة (ثغرات بصرية وإخفاقات تواصل) في مخططك! كما أن المعماري الذي رسم المخطط قد يكون مدركا بوجود غموض ما في رسمته، وبدلا من تصحيح ذلك الخطأ يحاول ترقيع هذا الغموض بالكلام، فتجده يقول أثناء العرض: "للتوضيح فقط يا شباب، هذه الأسهم تمثل تدفق البيانات (Data Flows)، وليس استدعاء الدوال"، وهنا نقول: إذا وجدت نفسك مضطراً لنطق مثل هذه الجمل التوضيحية لتنقذ الموقف، فهذا يعني أن مخططك يحتاج إلى تعديل فوري! ^^

# فائدة

وما زال المسرح يمزق هالة الكمال الزائف التي وضعها العلم على وجه الحضارة. يتدخل العلم عنوة ببياناته عن وفرة السلع، ومعدلات الإنتاج بالجملة، وعن الطاقة، وعن القوة البشرية، بينما تشير الفنون إلى الضياع الإنساني، والبؤس الفكري والأخلاقي، والعنف، والوحشية والخواء النفسي في قلب الثراء والقوة للعالم المتقدم، فيكتشف المسرح إنسانًا يائسًا عنيفًا تطوقه النفس اللوامة. إن الشعراء هم جهاز الحس في الجنس البشري، ومن مخاوفهم وشكوكهم نستطيع أن نحكم بأن العالم لا يسير في طريق الإنسانية، وإنما في طريق اغتراب الإنسان واستلاب انسانيته...

- كتاب الإسلام بين الشرق والغرب، علي عزت بيجوفيتش -رحمه الله-

# الفصل التاسع: المخططات يجب أن تعكس الواقع -Diagrams must reflect reality-

إن أكبر أزمة تواجهنا كمعماريين هي الانفصال التام بين ما نرسمه على اللوح وبين ما نكتبه من شيفرة برمجية!، وقد أطلق جورج فيربانكس على هذه الأزمة اسم The Model-Code Gap أي فجوة النموذج والكود.

على اللوح الأبيض نحن نتحدث عن مكون إدارة المستخدمين وطبقة قواعد البيانات، ولكن عندما نفتح محرر الشيفرة البرمجية، فإننا لن نجد هذه المصطلحات! سنجد فقط Classes و Interfaces متناثرة في مجلدات Packages! هذا الاختلاف في المفردات يجعل المبرمج ينظر للمخطط المعماري كلوحة فنية لا تمت للواقع بصلة، ويردد مقولته الشهيرة: الشيفرة البرمجية هي مصدر الحقيقة الوحيد ^^

إن المعمارية الحقيقية الناجحة ليست مجرد رسم مفاهيم عالية المستوى تحلق في الفضاء الطلق! بل هي التي تُجبر المخطط على أن يعكس واقع الشيفرة البرمجية تماماً، فإذا كان المخطط يقول شيئاً، والشيفرة تقول شيئاً آخر = فالمخطط ليس له أي قيمة!

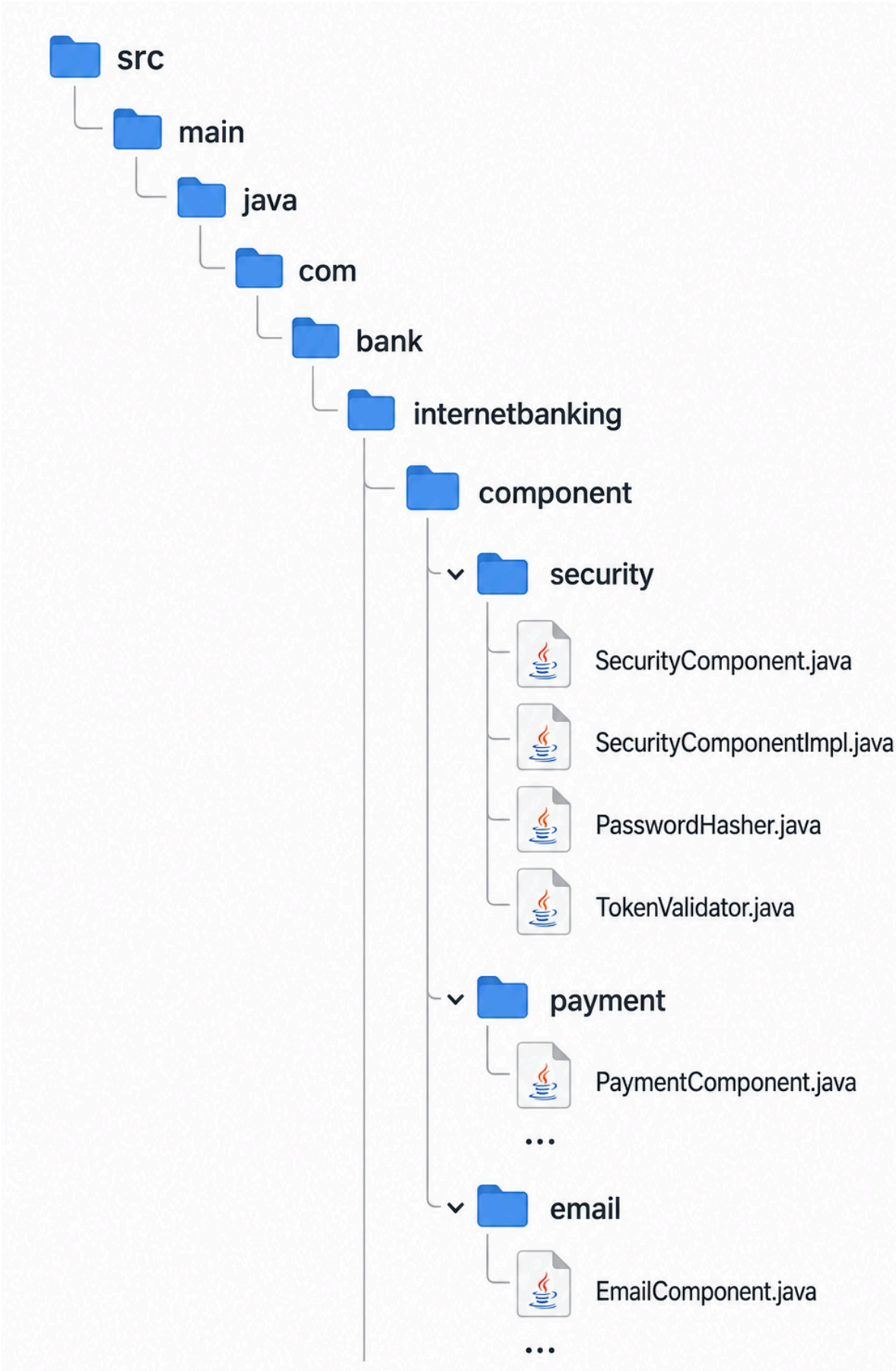
إن الفكرة الجوهرية هنا هي: لا يَكفي أن ترسم مخططات جميلة؛ بل يَجب أن تعكس هذه المخططات الجميلة الشيفرة البرمجية بشكل فعلي، فالمخطط الذي لا يَعكس الواقع هو كذبة معمارية تُضلّل الفريق، لذلك اجعل الشيفرة البرمجية تعكس المعمارية، لا العكس.

إن أكبر إهانة لأي معماري هي أن يرسم مخططاً جميلاً، ثم يفتح المبرمجون الشيفرة البرمجية فلا يجدون أي أثر لتلك الرسمة! ولتجنب هذا، يجب أن تتبنى أسلوب = الشيفرة البرمجية الناطقة بالمعمارية أو الموضحة للمعمارية -Architecturally-Evident Coding Style-،

إذا رسمت في مخططك مربعاً أسميته Payment Component، فيجب أن يفتح المبرمج الشيفرة البرمجية ليجد مجلد أو Package اسمه Payment، وداخله واجهة برمجية تحمل نفس الاسم، وربما موسومة بـ @Component! إن التطابق بالاسم والمكان هو ما يحول المخطط من رسمة تجريدية إلى دليل استخدام للشيفرة البرمجية، واحذر من التحول إلى معماري البرج العاجي -راجع ذكرياتك من الجزء الأول-! المعماري المنفصل عن الواقع هو من يرسم مربعات مفاهيمية ويرفض كتابة اسم التقنية بحجة أنها تفاصيل تنفيذية، والمخططات التي لا تُذكر فيها التقنيات بوضوح مثل Node.js أو PostgreSQL هي مخططات خيالية تفترض أن النظام سيعمل بالسحر، وينسى صاحبنا هنا أن المعمارية الحقيقية قرار.. فكن شجاعا واكتب قراراتك التقنية بشجاعة وهمة عالية داخل المربعات ^^.

وحتى يكون الكلام ملامسا للفهم أكثر، دعونا ننظر لأمثلة سريعة... إن حل مشكلة الفجوة بين الكود والمخطط -حسب الفقرة أعلاه- يمكن حلها من خلال اتفاقيات التسمية -Naming conventions-، فمثلا يعتبر اسم class Payment مثال غامض في حين أن class PaymentComponent اسم واضح... وال package المسماة ب com.bank.internetbankin.payment ستكون غامضة في حين أن

com.bank.internetbanking.component.payment ستكون واضحة.. وعند البحث في الشيفرة البرمجية سيرتبط المخطط بالشيفرة بشكل تلقائي... وسيجد المبرمج الذي دخل المحرر ما يريده بسرعة وبشكل مرتبط بالفهم العميق لما رآه بالمخطط... وهذا سيجعلنا أيضا نهتم برطيقة تنظيم المجلدات أو ال packages داخل المشروع، لأن هذا سيجعل من ال component قابلة للتعرف عليها بشكل بصري مباشرة داخل المشروع، وذات حدود واضحة وتبعيات ظاهرة... هل تتذكر كيف تم تنظيم المجلدات في مثال عيادة الحيوانات الأليفة؟
شاهد المثال أدناه:


src
main
java
com
bank
internetbanking
component
security
SecurityComponent.java
SecurityComponentImpl.java
PasswordHasher.java
TokenValidator.java
payment
PaymentComponent.java
...
email
EmailComponent.java
...

يضاف إلى ما ذكرناه سابقا استغلال المزايا الحديثة للمحرات وللغات البرمجة مثل استخدام ال Machine-readable metadata، ويقصد بها البيانات الوصفية التي يمكنك استخدامها لتكون قابلة للقراءة بشكل آلي، أي معلومات مخزنة بشكل تفهمه البرامج ^^... وهذا مثل ال Annotations داخل الجافا، شاهد المثال:

```
@Component(name = "SecurityComponent")
@ArchitecturalLayer("Service")
public class SecurityComponentImpl : implements SecurityComponent {
// ... }
```

من خلال كل هذا، استخدامنا للأدوات سيكون فعالا وسيتنج لنا مخططا يوافق الشيفرة البرمجية وبشكل تلقائي ^^...

## المعمارية دون تقنية ليست سوى أحلام يقظة

أكثر ما يُفقد المبرمج احترامه للمخطط المعماري، هو أن يرى رسمة مليئة بالأسهم والمربعات الطائرة التي تتصل بعنصر ما، دون أي ذكر للغة البرمجة أو البروتوكول!

الكثير من مدعي العمارة يتهربون من كتابة التقنيات بحجة أنها تفاصيل تنفيذية يجب أن يتدبرها المبرمج! والنتيجة هي مخططات عاجية تعمل بالسحر وتستحيل برمجتها!
لذلك: لا عذر لإخفاء التقنية، فإذا كنت ترسم مخططاً لنظام تم بناؤه وانتهى بالفعل، فلا يوجد أي مبرر منطقي لإخفاء أسماء التقنيات! الشيفرة البرمجية موجودة والقرارات اتُخذت

بالفعل! أما الأعذار مثل "التقنية ستجعل المخطط مزدحماً!" فهذا عذر ساذج لتجنب بذل الجهد والقيام بوظيفتي! وعذر مثل: "الجميع في الشركة يعرف أننا نستخدم فقط ASP.NET و Oracle!" فهذا عذر آخر يتجاهل الموظفين الجدد ويوثق للمجموعة الحالية فقط!

إن بعض المعماريين هم سبب السمعة السيئة لوظيفة المعماري -تذكر المعماري صاحب البرج العاجي، الجزء الأول-، لإن هؤلاء يفترضون أن النظام سيعمل تلقائيا من خلال إطلاق بعض الشعوذات السحرية! فهؤلاء المعماريون يعتبرون لغة البرمجة وقاعدة البيانات مجرد تفاصيل تنفيذية لا تليق بمقام المخطط المعماري! التقنية ليست تفصيلاً هامشياً! وضع أسماء التقنيات صراحة يزيل الغموض ويربط المخطط بأرض الواقع، وحتى في مرحلة "التصميم المسبق"، لا يوجد شيء اسمه تصميم نقي خالي من التقنية، فأنت كمعماري تفكر بالتقنية لا شعورياً وأنت ترسم! فإذا كنت محتاراً بين تقنيتين، فاكتبهما معاً! ارسم مربع قاعدة البيانات واكتب داخله: [MySQL أو Microsoft SQL Server]، والفائدة المرجوة هنا: أن هذا التصرف يوضح للفريق أن هناك قراراً يجب اتخاذه، ويفتح باب النقاش الفني لاختيار الأنسب، بدلا من تركه كمربع صامت يحمل اسم قاعدة بيانات.

حسنا، برأيك، ما أكثر فائدة بعيدا عن وضوح المخطط يمكن أن نحصل عليها من هذا الكلام الذي ذكرناه بالأعلى؟
قد تتعجب من الإجابة، إلا أن أفضل فائدة هي إنقاذ المخططات من المثالية الجوفاء ^^، لأنك عندما تُجبر الفريق على كتابة التقنيات في المخطط، فإنك ستحول النقاش من نقاش تنظيري إلى نقاش واقعي! إن المخطط المفاهيمي الخالي من التقنية يخبئ المشاكل، لكن إضافة

التقنية تفجر الأسئلة الحقيقية مثل: كيف سيتواصل هذا المكون مع ذاك إذا كان أحدهما مكتوباً بال Java والآخر بال Node.js وفي Processes منفصلة؟
إن اشتراط وجود التقنية في المخططات يضمن أن من يقوم بالرسم هم المعماريون والمطورون الحقيقيون الذين يفهمون الشيفرة البرمجية!

الخلاصة: إن إخفاء التقنية في المخططات المعمارية ليس علامة على النقاء المعماري! بل علامة على الانفصال عن الواقع، فالمخطط الذي لا يذكر التقنية يعمل في برج عاجي.

## الخيال المعماري مقابل الواقع البرمجي

قبل أن توجه يدك لرسم أي مربع أو سهم، اسأل نفسك هذا السؤال: هل ستكتب الشيفرة البرمجية فعليا بهذه الطريقة التي رسمت المخطط بها؟ أو هل يعكس هذا الرسم بدقة الطريقة التي سنبني/بنينا بها الشيفرة البرمجية؟

إن إجابة هذا السؤال تمثل المبدأ والترياق المضاد للمخططات النظرية المنفصلة عن الواقع ^^؛ سواء أكنت ستصمم نظاما جديدا أو ستوثق نظاما قائما بالفعل... ولفهم عمق هذا السؤال، يطرح Simon حالتين شائعتين جداً:

**الأولى: المكونات المشتركة -Shared Components:**

تخيل أن لديك تطبيقي ويب، وكلاهما يحتاج إلى استخدام ال Logging Component، إن الخطأ الشائع هنا هو ميل المعماريون لرسم مربع الـ Logging خارج كلا التطبيقين، ورسم أسهم تتجه إليه من التطبيقين، وذلك لمجرد إيضاح أنه "مشترك" بينهما، والسؤال القوي الآن: "هل ستبرمج هذا المكون بهذا الشكل؟ هل سيكون هذا المكون عبارة عن خادم مستقل أو حاوية منفصلة تعمل خارج التطبيقين؟

بالطبع لا! في الغالب سيكون هذا المكون عبارة عن مكتبة أو package سيتم نسخها ولصقها ليعمل هذا المكون داخل كلا التطبيقين... مثل استخدام إحدى ال Packages من ال NPM أو استخدام ومشاركة ملف DLL أو JAR... لذلك، يكمن الحل المعماري = بارسمه كما ستبرمجه! ارسم مربع الـ Logging داخل التطبيق الأول، وانسخه لترسمه داخل التطبيق الثاني، وضع عليه ملاحظة صغيرة [Shared Component].. رسمه بالخارج سيجعل الفريق يظن خطأً أنه خدمة منفصلة ك Microservice!

**الثانية: استراتيجية الطبقات -Layering Strategies & CQRS-:**

تخيل أنك تبني تطبيق ويب تقليدي مقسم إلى ثلاث طبقات، وهن: طبقة العرض -Presentation- وطبقة الخدمات -Services- وطبقة البيانات -Data Access-، ثم قام المعماري برسم الأسهم لتمر دائما بشكل صارم بالترتيب عبر الطبقات حتى لو كان الواقع مختلف مثل وجود حاجة لاستخدام نمط CQRS، وال CQRS هي اختصار ل Command Query Responsibility Segregation؛ أي فصل مسؤولية الأوامر والاستعلامات، بحيث تمر أوامر الكتابة -writes- عبر ال services لتطبيق القواعد والمنطق

الخاص بعملنا، في حين يتم الاستعلام مباشرة من طبقة البيانات متجاوزين طبقة ال Services...

في هذه الحالة يجب علينا كمعماريين ألا نخفي الحقيقة من أجل رسمة أنيقة ^^، بل علينا رسم سهمين، ومع كل سهم تعليق يختص به، مثل السهم الأول For Write والثاني For Reads CQRS pattern...

## الفصل العاشر: مخططات أخرى

**"المعماري المحترف يعرف أن النظام أعقد من أن يُختزل في زاوية واحدة!"**

إن النظام البرمجي يشبه المبنى الضخم؛ لا يمكنك فهمه من خريطة واحدة! مهندس الكهرباء يحتاج خريطة تمديدات، والمدني يحتاج خريطة أعمدة... لذلك، ظهرت في معمارية البرمجيات معايير ونماذج عالمية مثل IEEE 1471 وISO/IEC/IEEE 42010 و Philippe Kruchten's 4+1 model والتي تعتمد على فكرة "وجهات النظر -Views-"، فكل منظور يسلط الضوء على زاوية مختلفة من النظام لجمهور مختلف (منظور منطقي، منظور مادي، منظور تطويري.. إلخ).

في هذا الجزء من الكتاب يحاول Simon أن يربط نموذجه C4 بالمدارس المعمارية الكلاسيكية، وتحديداً النموذج الأشهر = **نموذج Philippe Kruchten's 4+1**، وكمعماري برمجيات لا يمكنك قيادة فريق دون فهم هذا النموذج التاريخي الذي تُبنى عليه أغلب

النقاشات المعمارية المعقدة، وهذا ما سيقودنا للإجابة عن سؤال مهم: لماذا نحتاج مخططات أخرى؟ ولماذا نحتاج للنظر من خلال زوايا أو مناظير مختلفة للمشروع والمخططات؟
إن العناصر الثلاثة في المخططات = السياق، والحاويات، والمكونات تكفي في أغلب المشاريع، ولكن النظام البرمجي يشبه المبنى الضخم؛ لا يمكنك فهمه من خريطة واحدة! -مثال المهندس الكهربائي والمدني-... لذلك، ظهرت في معمارية البرمجيات معايير ونماذج عالمية تعتمد على فكرة "وجهات أو زوايا النظر -Views-"، وكل منظور منها يسلط الضوء على زاوية مختلفة من النظام لجمهور مختلف، مثل: منظور منطقي، ومنظور مادي، ومنظور تطويري.. إلخ.

بعد هذه المقدمة البسيطة يجب علينا أن نبدأ بنموذج **Philippe Kruchten's 4+1**، لأنه يمثل الأب الروحي لهذه الفكرة، والتي أطلقها في ورقة بحثية عام ١٩٩٥، وقد اقتبس منها الكثير من المعماريين وعدلو عليها عبر الزمن بما يلائم احتياجاتهم في محاولة للوصول لنماذج أفضل، وفكرة هذا النموذج هي أن أي نظام برمجي يمكن وصفه بالكامل من خلال 4 مناظير أساسية، ويربط بينها منظور خامس (+1)، وهذه المناظير هي:

١. المنظور المنطقي -Logical View-:
إن هذا المنظور يصف الوظائف التي يقدمها النظام -Functional Requirements-، وهو يمثل مربعات عالية المستوى تمثل اللبنات الأساسية للمشروع وكيف ترتبط ببعضها، وهذا يشبه إلى حد كبير مستويات C4 الأولى...

٢. المنظور العملياتي -Process View-:

إن هذا المنظور يصف آلية تعمل هذه اللبنات معاً في وقت التشغيل -Runtime-، والتركيز هنا على الأداء، والتزامن -Concurrency-، وتواصل العمليات مع بعضها -Inter-process communication-.

٣. المنظور المادي -Physical View-:
إن هذا المنظور يصف أين ستعيش هذه الشيفرة البرمجية = أي البنية التحتية، لذلك فالتركيز هنا على الخوادم، والشبكات، وكيفية نشر النظام -Deployment Topology-...

٤. المنظور التطويري -Development View-:
إن هذا المنظور لرؤية كيف سيقوم المبرمجون ببناء الشيفرة البرمجية فعلياً، لذلك فالتركيز هنا على ال Packages، والكلاسات، والطبقات، والمكونات، وهذا يشبه المستوى الثالث والرابع من C4...

٥. الرقم الصعب (+1) = منظور السيناريوهات -Scenarios View-:
هذا المنظور يمثل الصمغ الذي يربط المناظير الأربعة السابقة، لذلك نختار فيه حالات استخدام حقيقية -Use Cases- ونتتبعها عبر النظام، لنثبت أن تصميمنا في المناظير الأربعة صحيح ومتماسك...

الخلاصة المهمة:

لا يكفي الاعتماد على مخطط رسمت فيه ال classes أو رسمت في العلاقات في قاعدة البيانات لوصف النظام البرمجي! ووفقا لنموذج **Philippe Kruchten's 4+1** يجب علينا تصميم النظام من أربع زوايا تمثل: الوظائف الأساسية، وتنظيم وتطوير وبناء الشيفرة البرمجية، وآلية التشغيل والتواصل بين المكونات، والبنية التحتية للنشر، ثم تُختبر هذه الجوانب عبر سيناريوهات استخدام حقيقية للتأكد من أن التصميم عملي ويلبي احتياجات جميع الأطراف، مثل المطورين وفرق التشغيل ومديري المنتجات.

بعد هذه الرسالة البحثية من فيليب، ظهر نموذج آخر أيضا أطلق عليه = وجهات النظر السبع ل Woods & Rozanski.. و سأختصر هذا الجزء بقولنا أن أن هذه النماذج وغيرها ستقودنا لمشكلة أساسية ومهمة = وهي أهمية المصطلحات وفجوة المعرفة وفجوة الشيفرة البرمجية مع المعمارية، لذلك عندما بدأ Simon هذا الجزء من الكتاب وقبل حديثنا عن نموذج C4 قام بطرح موضوع المصطلحات أولا؛ وذلك لأثرها البالغ هنا... وهذا لا يعني بالضرورة وجود خطأ في هذه النماذج، وإنما المشكلة التي ستواجهها في تحويل هذه النماذج التي كتبت بطريقة أكاديمية لتستخدم وتترجم في المشاريع الحقيقية، وحل هذه المشكلة يكمن في بناء قاموس مشترك للمفاهيم وآليات التمثيل ونحو ذلك... وهذا ما قدمه Simon في نموذجه C4 بشكل أو بآخر!

كما أنه من المهم لنا أن نفهم أن البنية الثابتة التي تعلمناها ورأيناها، وأن المخططات التي قمنا باستعراضها ورسمناها ستمثل جوانب أو مناظير معينة لا كل المناظير، وهذا ما واجهناه سابقا أثناء حديثنا عن ال Container Diagram وغيره من المواضع، وإطلاقنا تحذير شديد

اللهجة لمنع الخلط بين ما نقوم ببناءه في البنية الثابتة -الهيكل العظمي- وبين ال Deployment Diagram -عملية الجري-، والتي اقترب الحديث عنها بفضل الله...

الخلاصة لربط هذه الأفكار العديدة هي:

بعد أن تعرفنا على النماذج المعمارية الكلاسيكية مثل (4+1)، فإنك ستجدها تقسم النظام إلى منظور منطقي يرسمه المعماريون، ومنظور تطويري ينفذه المبرمجون، وهذا التقسيم يخلق فجوة عميقة؛ فالمعماري يعيش في برج عاجي يرسم صناديق لا تشبه الشيفرة البرمجية، والمبرمج يعاني من تحويل هذه الصناديق إلى ملفات ومجلدات حقيقية!

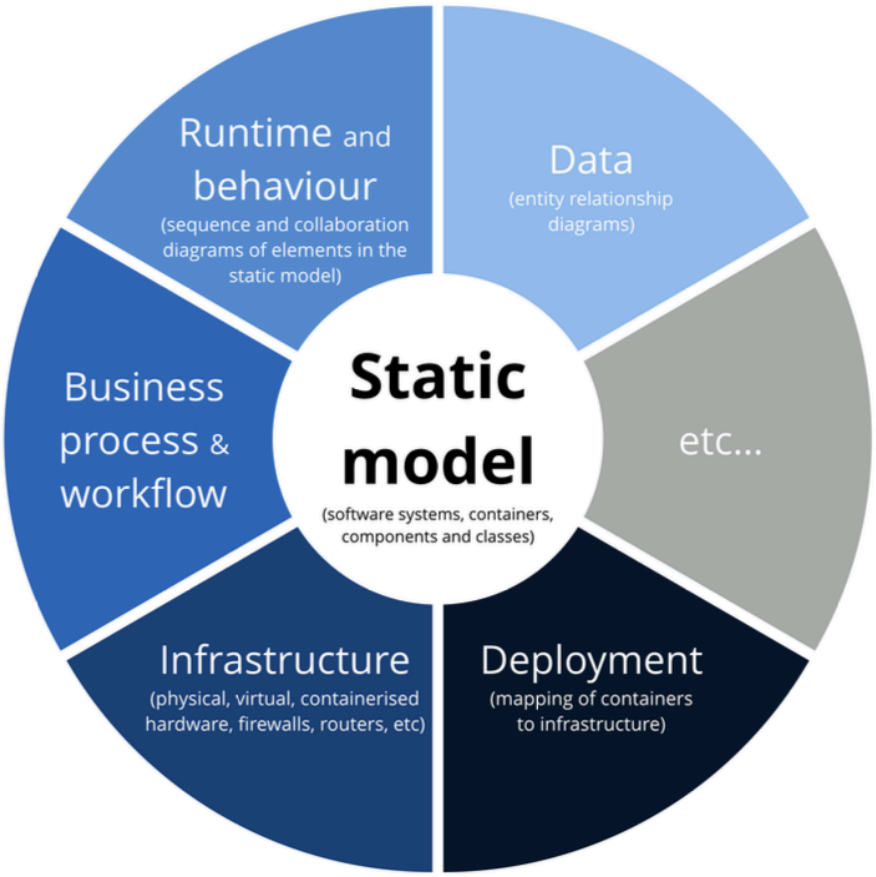
وهنا تكمن عبقرية نموذج C4 في كونه حطّم الجدار بين المعماري والمبرمج، وحول الصيغة الأكاديمية لصيغة عملية يمكن استخدامها في المشاريع الحقيقية، فنموذج C4 لا يطلب منك رسم مخطط منطقي منفصل عن الشيفرة البرمجية، بل يدمج المنطق والتطوير في هيكل واحد، فعندما ترسم مخطط المكونات - Level 3- في نموذج C4، فأنت فعلياً توثق المنطق الوظيفي وهيكل الشيفرة البرمجية في نفس اللحظة، وعندما ترسم المستوى الأول فإتك تنظر من منظور منطقي وسياقي للنظام، وعندما ترسم المستوى الثاني فأنت تجمع بين المنظور التطويري ومنظور العمليات...إلخ، وبهذا يرتقي نموذج C4 لأن يكون أكثر من مجرد آلية رسم، بل ليكون أداة لإجبار المعمارية على التحدث بلغة المبرمجين!

ومع ذلك، إياك أن تظن أن نموذج C4 نموذج كامل! فهناك حدود لأي أسلوب، والتقنية والأساليب التي يمكنك استخدامها غالبا ما تكون عبارة عن مقايضات شيء مقابل شيء، فمثلا نموذج C4 لوحده لا يغطي كل شيء! وهذا ما سيجعلنا نبني مخططات أخرى متخصصة

ترتبط مع النموذج مثل وجود ال Synchronisation أو Concurrency... إلخ، فهنا سنحتاج إلى مخطط حاويات معدل يظهر عدد ال threads أو آليات التزامن أو كيف تتدفق الرسائل... وستشاهد ذلك عمليا بعد قليل... فهيا بنا ^^

ملاحظة مهمة:

قبل أن ننتقل بالحديث عن أنواع جديدة من المخططات، يجب أن نضع قاعدة أساسية: إن الهدف الأول والأهم لنموذج C4 هو بناء الهيكل الثابت -Static Structure- للنظام، وهذا يشمل السياق، والحاويات، والمكونات، والشيفرة البرمجية حسب ما شرحناه سابقا...
والسبب في ذلك يعود لأن تأسيس اللغة مشتركة والمفردات الواضحة مع الفريق حول المكونات الثابتة للنظام = سيجعل من السهل علينا بناء أي مخططات أخرى لاحقا بالاعتماد على هذا الهيكل العظمي، فالهيكل العظمي -البنية الثابتة- هي الخريطة الجغرافية التي سنرسم عليها كل تحركاتنا اللاحقة... إذا الهيكل العظمي أولا ثم أي شيء آخر.




**The static structure defines the core of the software architecture model**

ولنبدأ بالأشياء الأخرى على بركة الله ^^

**مخطط المشهد العام -System Landscape-**

نموذج C4 بحد ذاته من المستوى الأول للرابع يُركز دائمًا على نظام برمجي واحد فقط، لكن في الواقع العملي، الأنظمة لا تعيش في عزلة؛ فقد يكون نظامك هذا هو جزء من شبكة / أنظمة ضخمة داخل الشركة! لذلك، وضع Simon مخططا جديدا يجلس "فوق" مخططات

C4، وهو مخطط المشهد العام -System Landscape-، هذا المخطط في الترتيب سيكون C0 ^^، وهذه الإشارة الفكاهية تعني أن مخطط C4 لم يتغير، وأن هذا ال C0 مخطط اختياري ليس بالضرورة وجوده، وسيكون كطبقة فوق ال System Context، أي أنه أكبر أو أعلى أو أول طبقة في نموذج C4 في حال حاجتنا لإضافته...

إن هذا المخطط -مخطط المشهد العام-: لا يوجد فيه "بطل واحد للقصة"، فهو يمثل لقطة بانورامية تظهر فيها كل أنظمة الشركة، وكيف تتواصل مع بعضها البعض، وكيف تتواصل مع المستخدمين الخارجيين! حتى مع تلك الأنظمة التي لا علاقة لها بالنظام المباشر الذي نعمل عليه نحن ضمن المؤسسة...

وهذا سيطرح سؤالا مهما، ما الفائدة من هذا المخطط ومن سيحتاجه؟!

وهذا سؤال مهم ورائع، وقد تتعجب أن هذا المخطط له استخدامات كثيرة ومهمة... ^^

إن الجمهور المستهدف من هذا المخطط عادة ما يكون أحد هؤلاء:

١. معماريو المؤسسة -Enterprise Architects-: مثل معماري مسؤول عن ٥ أو ١٠ أو ٢٠ نظاما في هذه الشركة.

٢. ال CTOs

٣. ال Portfolio Managers: ويقصد بهم المدراء أو المسؤولين التقنيين عن أكثر من مشروع داخل المؤسسة...

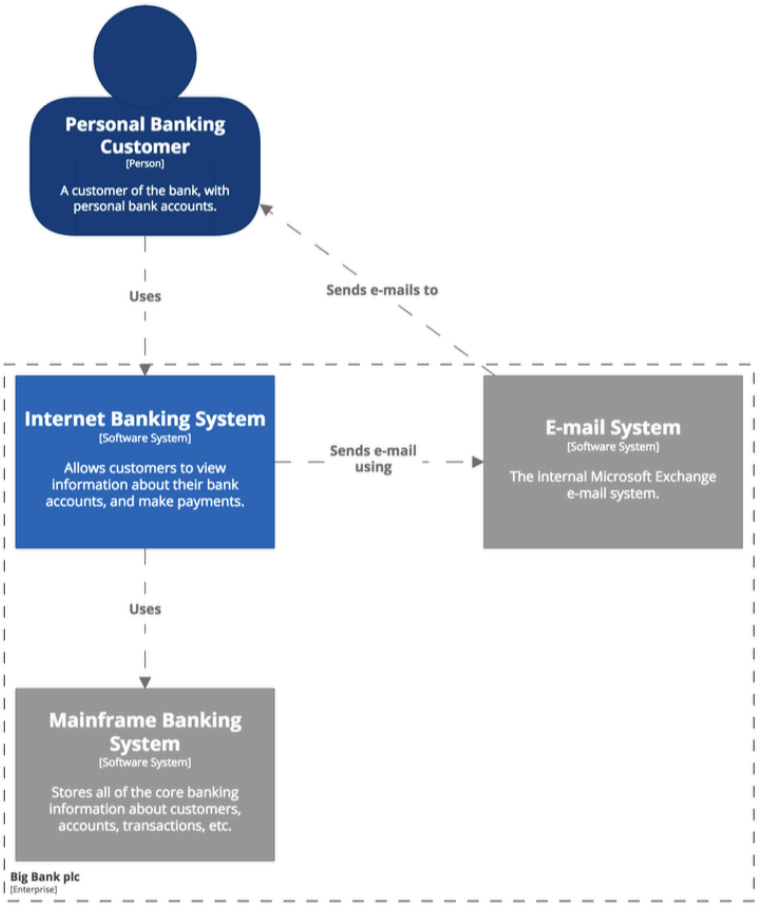
ومن هذه الفئة المستهدفة يظهر لنا أهمية هذا المخطط، فلو أردنا استبدال نظام قديم مثلا بنظام جديد، فما أثر استبدال هذا النظام على الأنظمة الأخرى المرتبطة معه؟ وما هي تداعيات هذا التغيير؟ ومن هي الفرق التي يجب عليها العمل على هذا التغيير أو ستتأثر به... إلخ، ولنأخذ أمثلة عملية على ذلك:

إذا كان معماريي المؤسسة مسؤولين عن ٢٠ نظام داخل مثال البنك الخاص بنا ^^، فنحن هنا بحاجة لمعرفة كيف تتصل هذه الأنظمة ببعضها، وسنحتاج لتحديد الاعتماديات، والتخطيط لعملية التطوير!

وهذا يفيد أيضا المدراء بأقسامهم المختلفة في تخطيطهم الاستراتيجي، فلو قررنا استبدال نظام البنك القديم بنظام جديد، فما هو التأثير المتوقع؟ وأي أنظمة تعتمد عليه، وما هي البدائل الممكنة؟ وما هي كلفة الاستبدال؟

حسنا، ماذا مدراء الفرق، كيف سيتمكنون من الانسجام مع الأنظمة الأخرى ضمن هذه المؤسسة، أي كيف سيتكامل مشروعنا مع مشاريع المؤسسة؟ فنحن بحاجة لفهم كيفية تدفق البيانات، وتحديد ال Contracts لل interfaces بين الأنظمة أو معرفتها ومشاركتها... إنها آلية تخطيط كاملة...

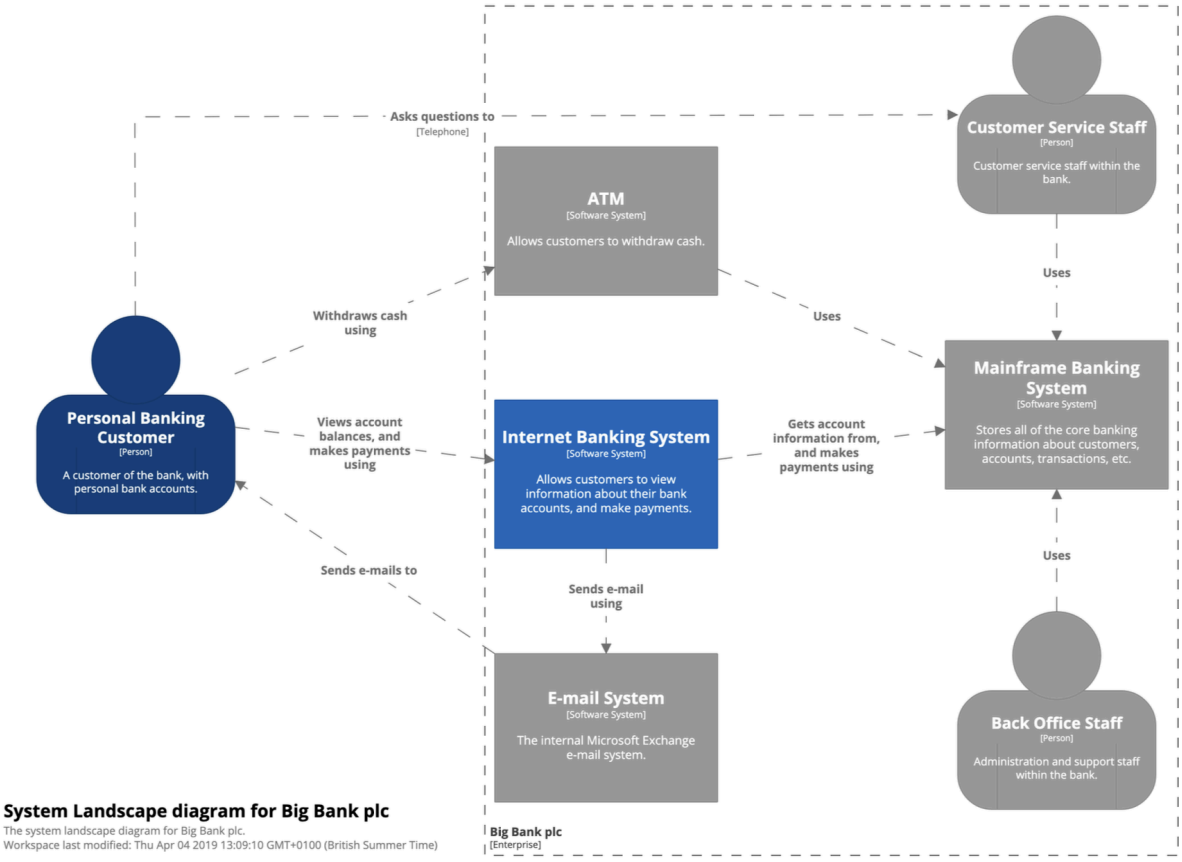
كل هذه الأمثلة وما فيها من فوائد سيجيب عنها مخطط المشهد العام System-Landscape- مباشرة... والآن، تعال معي وانظر للصور أدناه لمعرفة الفرق بين هذا المستوى وبين المستوى الأول.




**System Context diagram for Internet Banking System**

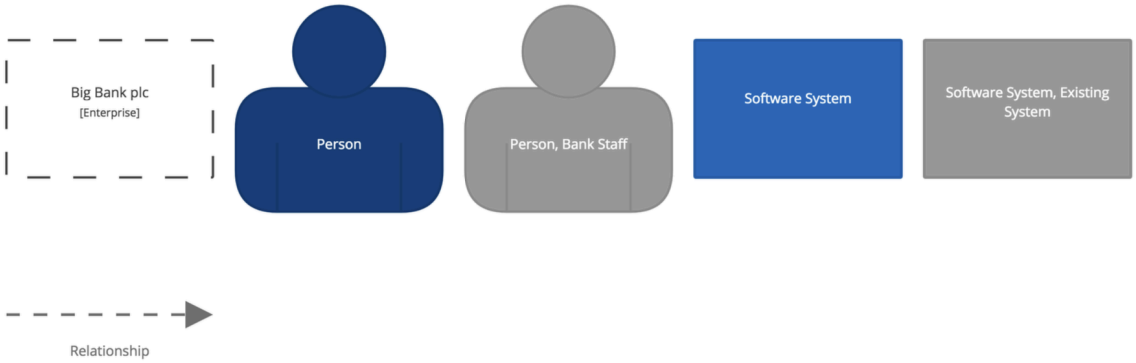
The system context diagram for the Internet Banking System.
Last modified: Wednesday 02 May 2018 13:41 BST

هذه الصورة لقد شاهدناها من قبل، فهي الصورة الجميلة الأولى التي تمثل أول مستوى تعلمناه من مستويات ال C4، وهو مستوى ال System Context، حسنا... الآن دعنا نضع هذه النظام ليكون جزءا من المؤسسة ولنحصل على نظرة شمولية ^^




**A System Landscape diagram for the fictional "Big Bank plc"**

هذه الصورة الجميلة تعطينا تصورا شاملا لنظام البنك الذي نعمل عليه، وكما تلاحظ فنظام البنك الداخلي الذي قمنا بتصميمه يمثل أحد الأنظمة في المؤسسة لا كل المؤسسة، كما تلاحظ أن القواعد التي تحدثنا عنها مثل وضع الحدود Boundary context مستخدمة كما تعلمنا... وهذه الصورة ينقصها شيء طالما تحدثنا عنه ^^، وهو مفتاح الخريطة، وها هو مفتاح هذا النموذج^^




**The diagram key for the System Landscape diagram**

إن مشكلة عدم وجود هذا المخطط في المؤسسات الكبيرة أو المؤسسات ذات المشاريع والأنظمة الكثيرة هي بناء صوامع تجعل كل فريق يعمل في عزلة، فتصبح المؤسسة نفسها معرضة للخطر من الداخل قبل الخارج، فهناك ستجد مشاكل التكرار والتعارض والفوضى! فكل فريق في صومعته يعيش! لذلك أتى هذا المخطط ليغطي هذه النظرة الشمولية والتي تفتقدها الكثير من الشركات، وبكل تأكيد هذا المخطط لا يعني أو يساوي ال Enterprise Architecture.

ملاحظة: كثيراً ما يختلط الأمر على المبرمجين بين مصطلحي مخطط المشهد العام System Landscape وبين تخصص معمارية المؤسسات Enterprise Architecture، مع أن الفارق بينهما كالفارق بين خريطة جوجل ووزارة التخطيط! مخطط المشهد العام هو خريطة بصرية ترسمها لتعرف ما هي الأنظمة البرمجية الموجودة في شركتك وكيف تتخاطب معا، وهي بذلك أداة تقنية بحتة لمساعدة المطورين على تجنب العزلة والعيش داخل صوامعهم

الخاصة! أما (معمارية المؤسسة - EA) فهي تشبه وزارة التخطيط، وهي تخصص إداري واستراتيجي شامل، يهدف إلى مواءمة التقنية مع أهداف العمل لذلك فال EA ليس مجرد رسمة، بل هي إطار عمل ضخم يحتوي على سياسات، وميزانيات، وخطط مستقبلية، وحوكمة، وبذلك فال EA لا يسأل فقط كيف تتصل الأنظمة؟، بل يسأل: هل هذا النظام يخدم أهداف الشركة المالية؟ وهل نحتاج لشرائه أم برمجته؟ وما هي خطتنا التقنية للسنوات الخمس القادمة؟..إلخ ببساطة: المخطط العام يخبرك أين أنت الآن تقنيا، بينما معمارية المؤسسات تخبرك لماذا نحن هنا، وإلى أين نحن ذاهبون كمؤسسة!

**النماذج المبدئية والنهائية للواجهات -UI Mockups & Wireframes-**

لقد ذكرنا مرارا وتكرارا أن لكل أداة دورها التي تؤديه، والزاوية التي تغطيها، والتي قد يعجز عنها غيرها من الأدوات.. هذا المفهوم يتجلى بوضوح هنا؛ فنموذج ال C4 ليس مصمما لإظهار تجربة المستخدم -UX- أو التركيز على ذلك، والحل سيكون من خلال استخدام النماذج المبدئية -Wireframes- أو النهائية -Mockups-، في حين أن هذه الواجهات تعجز أيضاً عن عرض البنية الثابتة للنظام كما يفعل C4 ^^، لذلك، يظهر هنا الأثر الحقيقي لتكامل الأدوات، وعليه، احفظ هذه القاعدة: إن **واجهة مستخدم واحدة قد تشرح للمستخدم غاية النظام في ثوان، بينما يشرح مخطط C4 الصحيح كيف بُني هذا النظام، فلا الواجهات تغني عن المخطط المعماري، ولا المخطط المعماري يغني عن الواجهات؛ فلكل أداة منظورها الذي تكشفه، ولا تكتمل الصورة إلا بتكاملهما...** وبناءً على ذلك، فالممارسة المعمارية الأفضل هي تصميم هذه الواجهات وإرفاقها مع المخطط المعماري كملفات مساندة أو روابط خارجية

مثل روابط Figma، لتكتمل الصورة أمام فريق العمل وتترابط الأفكار بشكل جلي...
ولاحظ أن هذه الإضافة تمثل عامل مساعد لا مكون أصيل في المخطط المعماري ^^.

**العمليات التجارية وسير العمل -Business process and workflow-:**

إن نموذج C4 يشرح مِمَّ يتكون النظام، لكنه لا يشرح كيف تسير العملية التجارية داخله، وهذا يأخذنا لبعدين مهمين، فالعمليات التجارية تمثل سلسلة خطوات لتحقيق هدف تجاري ما مثل عملية طلب القرض، وسير عمل المستخدم يمثل خطوات المستخدم لإنجاز مهمة ما مثل تسجيل حساب جديد... والقيمة المطلوبة هنا أن نفهم ماذا يحدث هنا دون الانغماس في كيف يحدث ذلك تقنيا! ولعل من أشهر الأدوات المستخدمة لهذا الغرض ال Flowchart وال UML Activity Diagram...

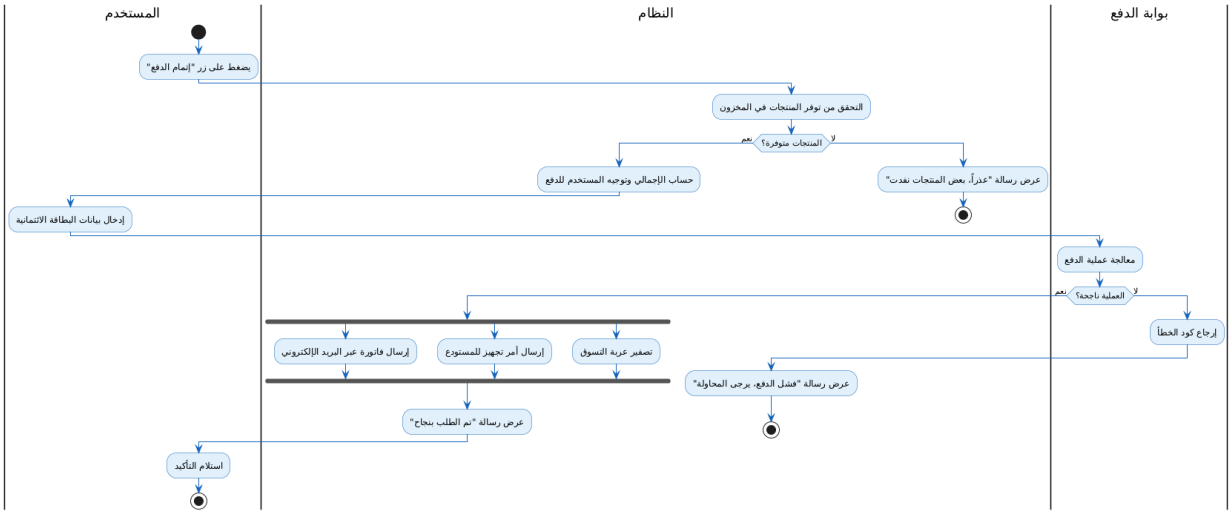
إن المطورين سيعرفون أين سيكتبون الشيفرة البرمجية عند اطلاعهم على نموذج ال C4، وسيستطيعون استخدام الحاوية المناسبة لهذا الغرض، ومع ذلك سنحتاج لمثل هذه المخططات -نخطط العمليات التجارية وسير العمل- عند حاجتنا لمعرفة المنطق الذي سيكتبه المطور داخل الشيفرة البرمجية... فمثلا، يستحيل إظهار أو وضع الشروط داخل المخطط المعماري...
وهناك فرق كبير بين:

مِمَّ يتكون النظام؟ وكيف ترتبط أجزاؤه؟ => C4 => هيكلي وثابت

وبين

كيف يعمل النظام؟ وما هي خطوات إنجاز المهام بناء على مسار المستخدم؟ => العمليات التجارية وسير العمل => سلوكي وديناميكي

شاهد المثال:




## نموذج النطاق والمفاهيم -Domain Model-

في العالم الحقيقي سيكون عملك وأنظمتك قائمة على خدمة مجال / مجالات معينة أو محددة، وهذا ما يطلق عليه ال Business Domain، فمثلا قد يكون نظامك يعمل في المجال الصحي أو التعليمي أو مجال البنوك...إلخ، وكل نظام يعمل في مجال معين؛ يحوي في داخله مفاهيم أساسية يتفق عليها الجميع...

ومن الأمثلة على ذلك:

مجال البنوك: Account, Customer, Transaction, Loan.

مجال الصحة: Patient, Doctor, Appointment, Prescription.

مجال التجارة الإلكترونية: Product, Cart, Order, Payment.

حسنا، كيف يمكننا تمثيل هذا الشيء في مخططاتنا المعمارية؟

يوصي Simon هنا باستخدام ال UML Class Diagram باعتباره أفضل طريقة لذلك، وسيتم استخدامه لوصف مفاهيم البيزنس لا لوصف الشيفرة البرمجية، وهذه هي النقطة الفارقة هنا!

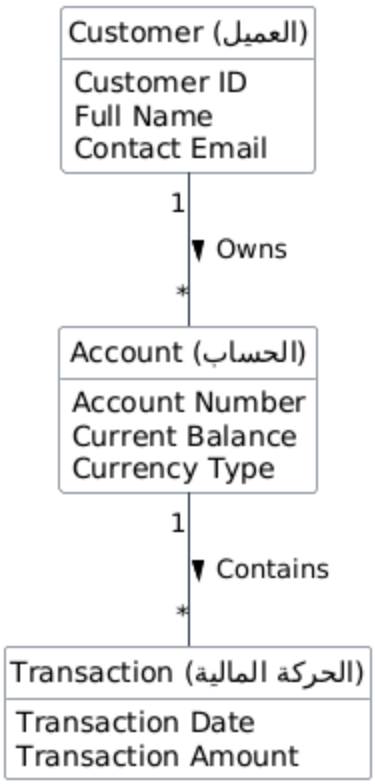
وآلية العمل هنا ستكون كما يلي:

ال Classes بال UML = سيتم استخدامها لتمثيل كيانات المجال -Domain Entities-...

وال Attributes = سيتم استخدامها لعرض خصائص الكيانات...

والعلاقات = الروابط بين الكيانات...

شاهد هذا المثال:




لاحظ أن المخطط الموجود في الصورة لا يعرض أي خصائص برمجية مثل "-" للدلالة على ال Private أو ال type الخاص بأي attribute... إلخ، لأن هذا النموذج وظيفته هي استخدام مخططات ال UML ليقرأها خبير البزنس لا المطور وال compiler ^^...

والملاحظة الأخيرة هنا، أن هذا النموذج مهم جدا، ويجب أن يتواجد ضمن ملفاتك المعمارية لأي نظام؛ وذلك لأن كل نظام يحتاج منا لفهم مفاهيمه المركزية والقدرة على مشاركتها واستخدامها في التواصل، حتى وإن لم نكن DDD...

## السلوك الديناميكي ووقت التشغيل -Runtime & Behaviour-

قد تتعجب من هذه البداية، لكن Simon يعترف أن 80% إلى 90% من المخططات التي يراها بما فيها C4 تركز فقط على الهيكل الثابت، مع البرمجيات ليست جماداً، بل تتحرك وتتفاعل في وقت التشغيل -Runtime-!

إن البرمجيات والأنظمة التي نعمل عليها مصممة لتفعل شيئا ما، لذلك نحن بحاجة لشيء يرينا الحياة في هذه الأنظمة، وهذا هو دور أمثال هذه المخططات!

إن حل هذه المشكلة -عند حاجتنا لها- هي استخدام المخططات الديناميكية، لذلك هناك مصطلح: Dynamic View والذي يشير إلى المنظور الحي لا الثابت!

لكن هذا النوع من المخططات يجب استخدامه في السيناريوهات المهمة فقط، منها ال Use Cases الأساسية والمهمة، وال User Stories المركزية ونحو ذلك...

وهنا يأتي السؤال اللولبي، كيف سنستطيع رسم هذه الحركة؟

وهنا يظهر Simon ليبجيب عن ذلك بطريقة جميلة أيضا = وهي من خلال استعارة الأدوات من ال UML مجددا ^^...

مثل:

- ال Sequence Diagrams
- ال Collaboration Diagrams أو ال Communication Diagrams في الإصدار الثاني من UML.

هذه الاستعارة ستكون كما يلي:

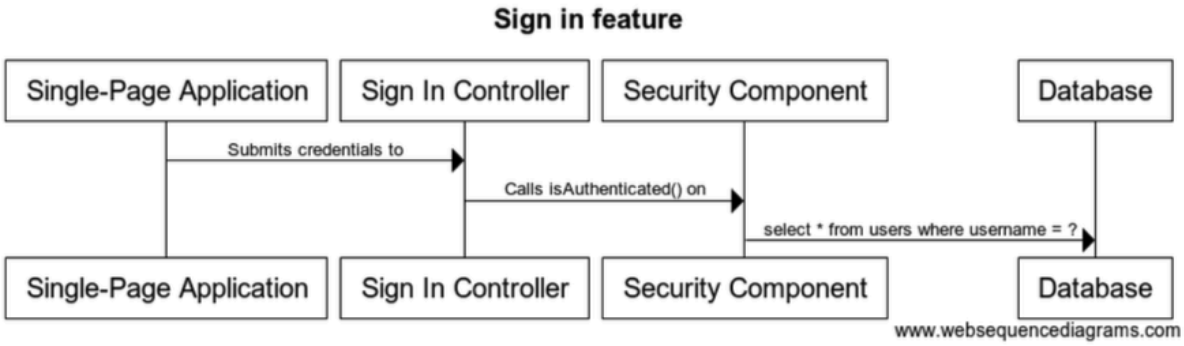
- بدلاً من رسم كيف يتخاطب كلاس (A) مع كلاس (B)، سنرسم كيف تتخاطب الحاوية A مع الحاوية B في ال Microservices لتنفيذ حالة استخدام -Use Case- معينة... أي أننا سنقوم باستخدام هذه المخططات مع مكونات نموذج C4 لا ال Classes فقط ^^

وهذا يقودنا لملاحظة مهمة، فهي وإن كنا نرى أن استخدام ال UML لوصف المخططات المعمارية وبناء الأنظمة بشكل دقيق هو أمر سيء وغير عملي؛ إلا أن استخدامها كأدوات للرسم وكتدوين رمزي أمر مفيد جد ورائع... وهو ما رأيناه في الأمثلة الأخيرة والكلمات السابقة...

لذلك، ميز بين ال Standard UML -الرسمية- بكونها معقدة وفيها قواعد دقيقة كثيرة ويرفضها المبرمجين، وبين ال sketching notation UML والتي تأخذ الأفكار الجيدة وتتجاوز التعقيدات وتنجح عمليا...

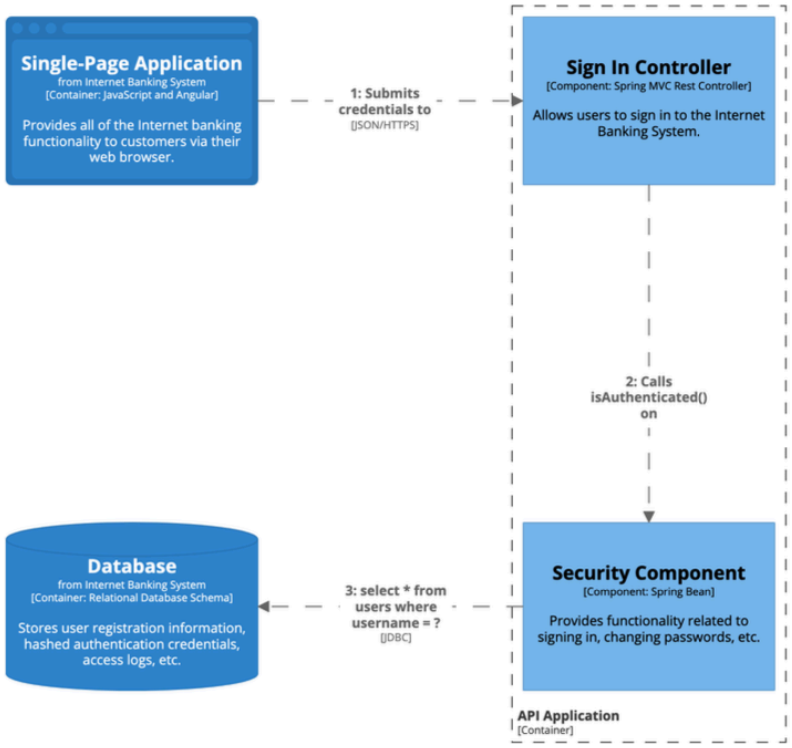
حسنا، الآن دعونا نلقي نظرة على أمثلة متعددة على هذا الموضوع:

- مثال على Sequence diagrams:




إن هذا المخطط كما ترى، يركز على الزمن؛ لذلك فهو يقرأ من الأعلى للأسفل عبر خطوط عمودية.

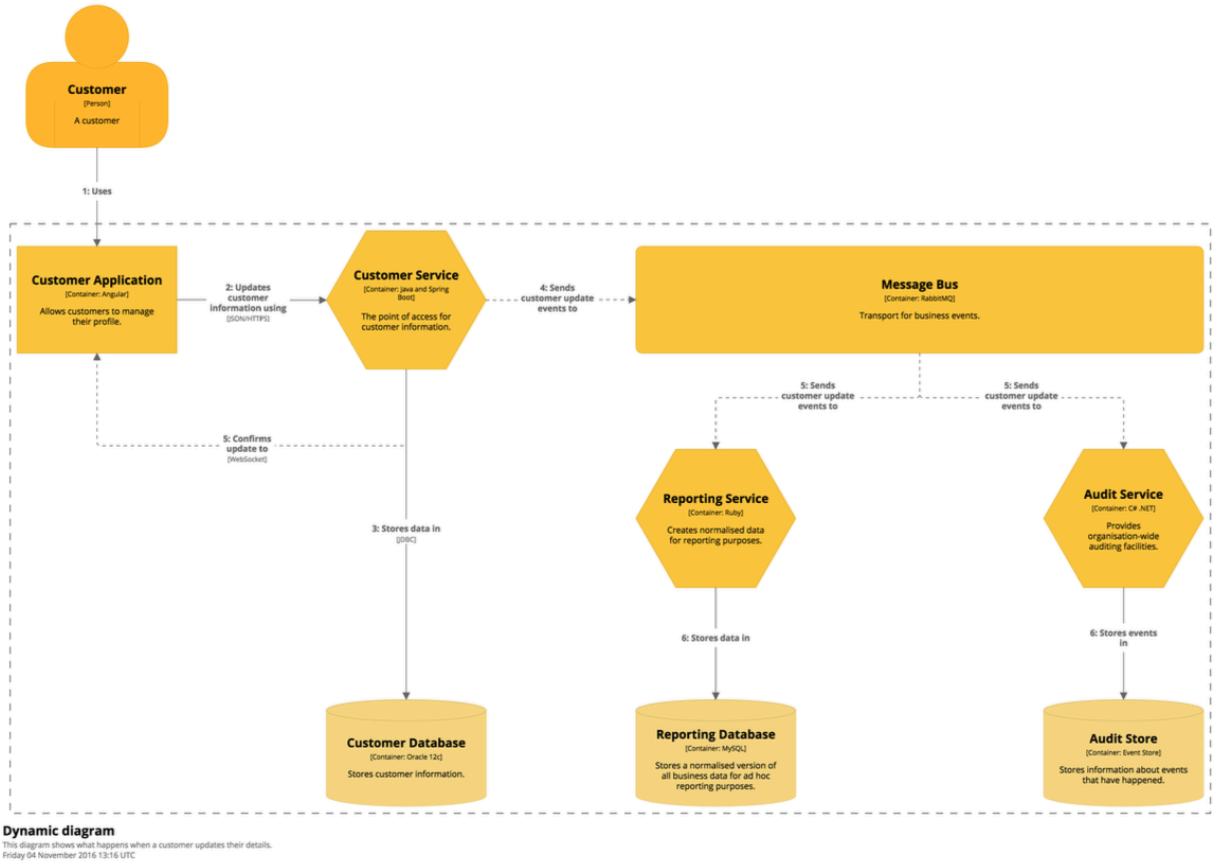
- مثال على ال Collaboration diagrams:




**Dynamic diagram for API Application**

Summarises how the sign in feature works in the single-page application.
Workspace last modified: Thu Apr 04 2019 13:09:10 GMT+0100 (British Summer Time)

إن هذا المخطط يركز على المكان والروابط، لذلك ستجد مربعات تتصل ببعضها بخطوط، لذلك ستجده يشبه مخططات C4 الثابتة، لكننا نضيف للأسهم أرقاماً تسلسلية 1، 2، 3.. لنخبر القارئ بترتيب الأحداث، لكن ماذا لو كان عندنا أكثر من أحدث في نفس الزمن؟ والجواب ببساطة من خلال استخدام تكرار الرقم الخاص بترتيب هذا الحدث... شاهد الصورة التالي:




وبالمناسبة، هذا الأسلوب وهذا النوع من المخططات هو المفضل لدى Simon، لأنه أسهل بالرسم على اللوح... لكن استخدم أنت ما تجده أفضل لك!

ملاحظة مهمة ذكرناها وسنعيد ذكرها:

المخططات الديناميكية / الحية قوية جداً لكنها ليست للتوثيق الشامل! لذلك، استخدمها بحكمة فقط للسيناريوهات المهمة أو المُعقّدة أو غير الواضحة من الشيفرة البرمجية، فالإفراط في

التوثيق الديناميكي مضيعة للجهد، والتفريط سيجعل جزء من القصة مفقودا! والاعتدال هو الحكمة...

ملاحظة مهمة ٢: إياك أن ترسم المخطط الديناميكي / الحي إلا إذا كانت الشيفرة البرمجية غامضة! أما إذا كانت العملية واضحة بمجرد قراءة الشيفرة البرمجية، فلا داعي لرسم هذا النوع من المخططات لأنه لن يضيف أي قيمة برسمه! بل على العكس، ستقع في فخ التوثيق المفرط -Over-documentation! وتذكر أن هناك حدا فاصلا بين المعماري العملي الذي يقدر وقته ووقت فريقه، وبين المعماري النظري الذي يضيع أشهرا في رسم كل شاردة وواردة! لأننا نعلم أن أي نظام برمجي -حتى الصغير منها- يحتوي على مئات المسارات -مثل مسارات النجاح، والفشل، والشروط المنطقية، والمعالجات المتزامنة..إلخ-؛ فمحاولة رسم مخطط لكل User Story بتكون بالتأكيد ضرب من الجنون ومضيعة للوقت... لذلك، الوقت الأفضل لاستخدام هذه المخططات هي:

- عند وجود مسارات متعددة أو خدمات خارجية للأمان والمصادقة (SSO, OAuth): وهنا سنهتم بمعرفة مسار ال Tokens بين هذه الأنظمة...
- الأحداث غير المتزامنة -Asynchronous Events-: عندما تستخدم أي Message Brokers وتفقد القدرة على تتبع الشيفرة البرمجية، فسيكون من الجيد استخدام هذا المخطط هنا...
- أنماط التصميم المعقدة: فبعض الأجزاء قد تكون معقدة ويصعب تتبعها من خلال الشيفرة البرمجية... فالأفضل هنا هو معرفة هذه الأجزاء ورسم المخططات الحية لها... حتى تبقى مفهومة مع تغير الزمان والأجيال ^^...

وبهذا تكون النتيجة النهائي لنا في هذه الملاحظة = لا ترسم مسارا حيا يمكن فهمه بسهولة من خلال قراءة الشيفرة البرمجية! والتوثيق لا يكون على مستوى ال Classes & Method بل على مستوى ال Container أو ال Components... وبالمناسبة، حتى Simon يقول: "نادراً ما أوثّق الجوانب الديناميكية"، وهذا يعني أن الأمر ليس ضرورة يومية!

ملاحظة ٣: يمكن استخدام ال Automated tests أو ال Debugger أيضا كأسلوب لفهم الشيفرة البرمجية ومسار العمل من خلال تتبع السلوك أو التتبع التنفيذي، وهذه من الوسائل المستخدمة أيضا لفهم الشيفرة البرمجية...

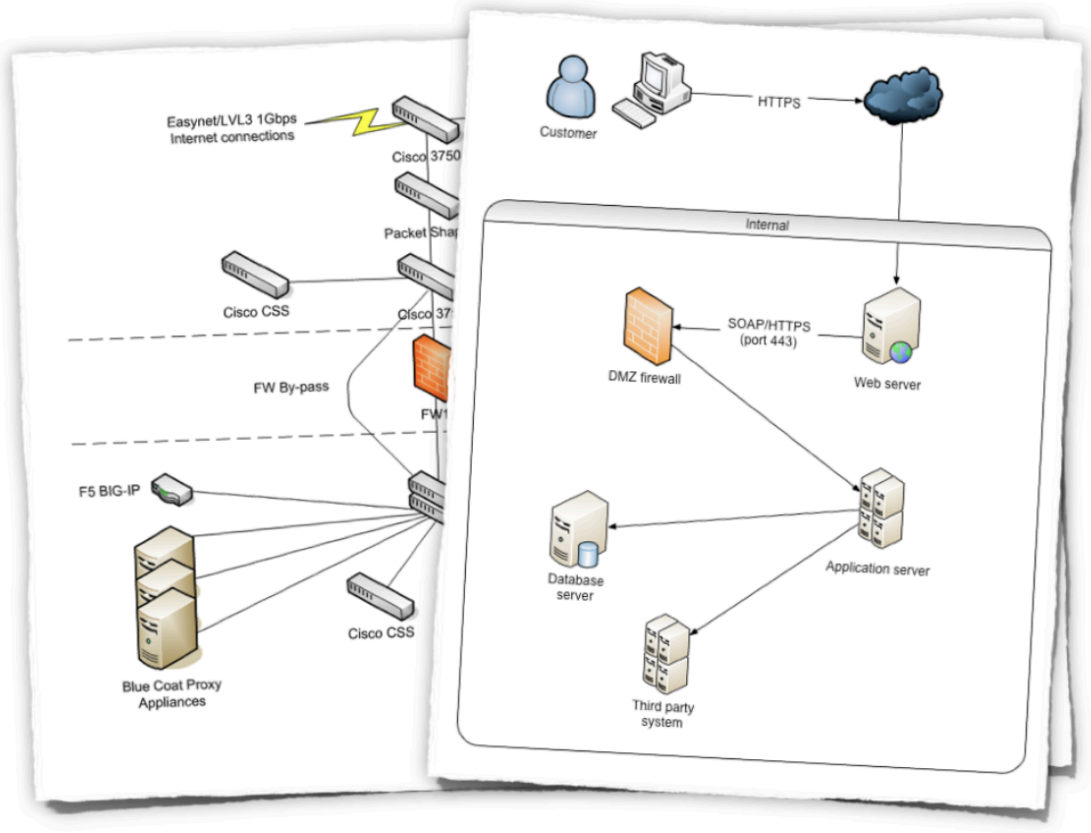
ملاحظة ٤: التوثيق يكن للسيناريوهات المعقدة أو المهمة أو غير الواضحة من الشيفرة البرمجية = وهذا ما شرحناه قبل قليل، لذلك يمكنك اختبار ما اذا كنت بحاجة إلى مخطط ديناميكي أم لا من خلال هذا السؤال: هل يمكنني فهم السلوك من الشيفرة البرمجية؟ إذا نعم = لا حاجة للمخطط، إذا لا = قد تكون بحاجة لمخطط، فالأمر يستحق... واعلم أن الجوانب الديناميكية وإن كانت مهمة فهذا لا يعني أننا بحاجة لتوثيقها إن لم تضف قيمة كبيرة، وهذا عادة ما يكون باستثناء ما ذكرناه...

## البنية التحتية -Infrastructure-

هذا الموضوع أحد المواضيع الجوهرية التي طالما أشرنا لها في ثنايا سطور كلماتنا في هذا الكتاب... فهذا الباب هو مدخلنا لمخططات النشر -Deployment Diagrams-.

إن البنية التحتية تمثل الأجهزة والبيئات التي تعمل عليها البرمجيات، وهذا يشمل ال Servers وال Networks وال Data Centers وال Virtual Machines وال Docker... إلخ... وهذا يقودنا لقاعدة مهمة جدا، وهي: إن الشيفرة البرمجية الجيدة بلا بنية تحتية مناسبة = لن تكون سوى ملفات نصية لا قيمة لها! وهذه المقدمة المهمة هي زاوية النظر لنا للجزء المادي المادي -Physical View-، والذي تحدثنا عنه أو أشرنا إليه سابقا في العديد من المواضيع والأماكن، وبالمناسبة، شهد هذا الباب تطورا كبيرا في السنوات الأخيرة، وقفزات رهيبة! فمثلا اعتاد المعماريون على رسم المخططات المعهمارية بشكل يدوي من خلال أحد البرامج مثل Microsoft Visio، لكن الآن أصبح الموضوع يمكن أن يكون تلقائيا من خلال ال IaC، والذي يشير إلى Infrastructure as Code، ومع أن استخدام الطريقة اليدوية ما زالت مستخدمة، إلا أن المعمارية الحديثة تجاوزت ذلك بكثير! فاليوم، يمكن أن يكون توثيق البنية التحتية عبارة عن مجموعة سكربتات تمت برمجتها لهذا الغرض -Automated Scripts-، وهذا مثل استخدام ال AWS CloudFormation، بل ويتجاوز الأمر في هذه السكربتات موضوع رسم البنية التحتية فقط، فهي تقوم ببنائها وإنشائها فعليا على ال Cloud بضغطة زر!

شاهد المثال:




**Example infrastructure diagrams, created in Microsoft Visio**

حسنا، بعد هذه المقدمة الجميلة، لنبدأ ب:

## مخطط النشر -Deployment Diagram-

عندما نتحدث عن هذا النوع من المخططات، فنحن نسعى لأن نجيب عن السؤال التالي: أين ستعمل هذه الشيفرة البرمجية وكيف سيتم توزيعها؟ وإجابة هذا السؤال أكثر من يهتم بها هم فريق الـ DevOps ^^... فنحن ننقل هنا تركيزنا من البنية المنطقية للنظام البرمجي، إلى كيفية ارتباطه بالواقع المادي -العقد فيزيائية-، فالنظام متمثلاً بحاوياته سيعمل في مكان ما على أرض الواقع، مثل وجود Server لخدمة نظامنا... وهنا قام Simon بتطويع هذا

المخطط الذي نرغب برسمه والحصول عليه في نموذج ال C4 ليكون أبسط وأكثر واقعية من تعقيدات ال UML التقليدية ^^... وذلك من خلال ربط الحاويات من نموذج C4 مع البنية التحتية ^^، وهذا سيقودنا لمصطلح ال Deployment Mapping، ويقصد بها كيف توزع الحاويات على الأجهزة عند النشر...

حسناً، قد تتساءل الآن:

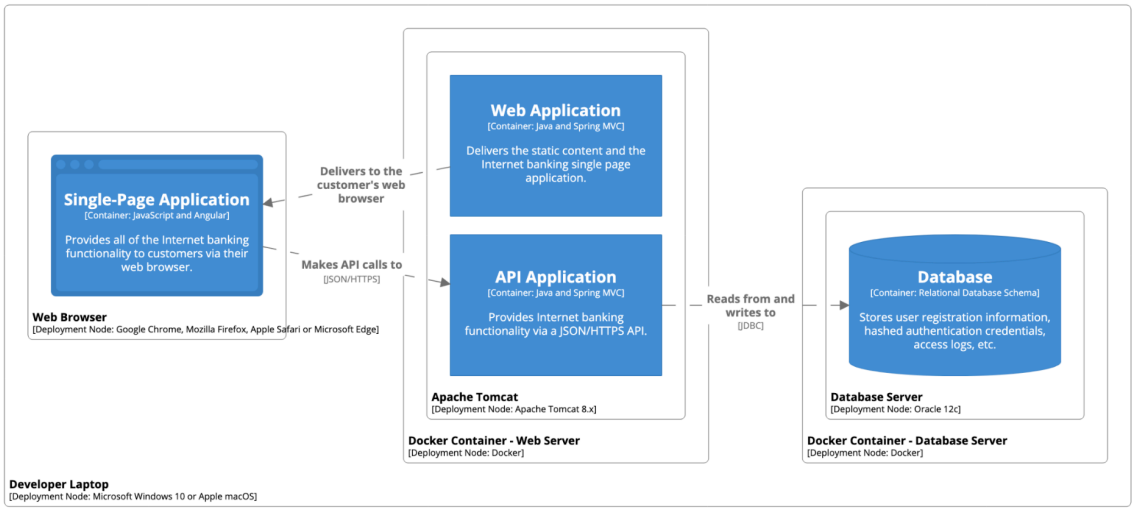
لماذا نرسم مخطط النشر حتى مع وجود الأتمتة التي تحدثنا عنها (IaC)؟

أو طالما لدينا سكربتات Terraform وأتمتة كاملة، فلماذا نرسم مخططا أصلا؟!

يجيب Simon على هذه النقطة بقوله: إن المخطط البصري يروي قصة لا يمكن للسكربتات إبرازها بلمحة عين، فكيف سنعرف أن نظام الويب الخاص بك هل يعمل على خادم واحد -Single Server- أم على Server Farm تضم 100 خادم؟ هذا المخطط هو الذي يوضح قرارات معمارية حاسمة مثل: التوسع -Scalability-، والمرونة -Resilience-، والأمان... فحتى مع الأتمتة الكاملة ستحتاج إلى النظرة العامة الأولى -تذكر مانترا شنايدرمان-.

وهنا تظهر لنا مفارقة مهمة، وهي أن البنية المنطقية نفسها قد يكون لها بنى فيزيائية مختلفة تماما! ومخطط النشر هو من سيكشف هذه الاختلافات بكل تأكيد.

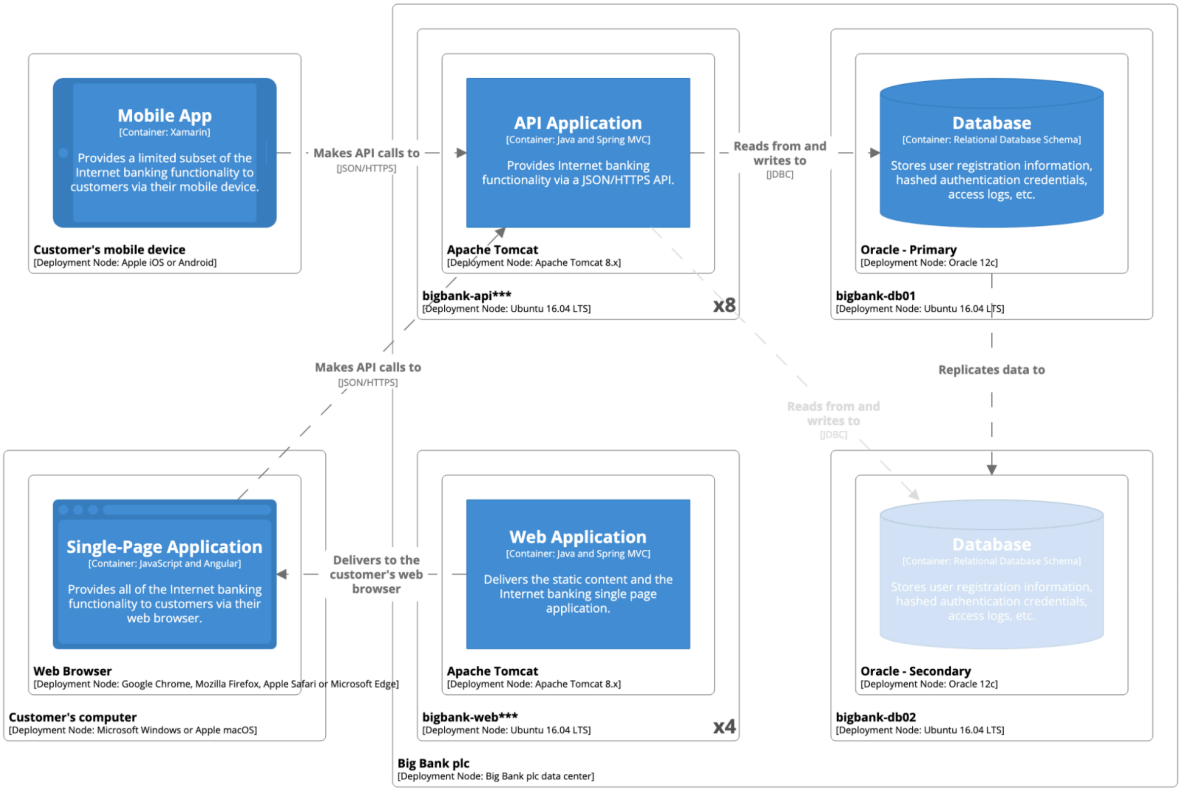
حسنا، الآن لنأخذ مثالا:




**Deployment diagram for Internet Banking System - Development**

An example development deployment scenario for the Internet Banking System.
Workspace last modified: Thu Apr 04 2019 13:09:10 GMT+0100 (British Summer Time)

في هذه الصورة نشاهد مخطط النشر، مخطط النشر هذا يمثل نظام البنك الذي عملنا عليه، لكن على جهاز المطور، والآن لاحظ نفس المخطط على ال Production:




**Deployment diagram for Internet Banking System - Live**

An example live deployment scenario for the Internet Banking System.
Workspace last modified: Thu Apr 04 2019 13:09:10 GMT+0100 (British Summer Time)

حسنا، دعونا نشرح هذه المخططات سريعا ^^

مصطلحات مهمة:

- ال Deployment Artifact ويشير إلى عنصر النشر: أي الشيء الذي يُنشَر مثل ملف JAR أو ملفات ال build الناتجة من موقع react.
- ال Deployment Node وتشير إلى عقدة النشر: أي الجهاز أو البيئة التي يُنشَر فيها ال Artifact

- ال Nesting وتشير لوجود أو إمكانية التداخل للعقد القابلة للنشر داخل بعضها، مثلا Web App داخل Docker Container داخل Virtual Machine داخل Physical Server ... إلخ

فتصبح العلاقة

Deployment Artifact → deployed on → Deployment Node

لكن Simon قام بتعديل هذه العلاقة لتصبح:

Container → deployed on → Deployment Node

وهكذا تخلص Simon من تفاصيل الملفات، وأصبحنا ننشر الحاويات مباشرة!

إذا، الشرح:

في المخطط الأول نجد مخطط النشر على جهاز المطور والذي يمثل بيئة التطوير، وكا ترى فهي معمارية بسيطة جداً! فالمبرمج يستخدم حاسوبه الشخصي كعقدة نشر، ويفتح المتصفح كعقدة متداخلة لتشغيل واجهة الويب من نوع SPA، وفي الخلفية، تعمل حاويتا Docker على الجهاز، واحدة لـ API والأخرى لقاعدة البيانات ^^... ببساطة لقد رأيت أن ال SPA هي نفسها الحاوية من C4 التي قمنا بكتابتها، وتم وضعها داخل المتصفح لتعمل فيه! وهنا نقول مبارك، لقد أصبح عندك مخطط نشر... (وهذا ينطبق على بقية الحاويات)

أما المخطط الثاني فهو يمثل بيئة الإنتاج -Live-: فواجهة المستخدم تعمل على أجهزة العملاء المتنوعة، أما في مركز بيانات البنك -Data Center-؛ فالأمر مختلف: هناك خوادم Ubuntu متعددة، وقواعد بيانات أساسية وأخرى ثانوية متزامنة -Replication-، كما يمكنك إضافة ما يحلو لك مثل ال Load Balancer، أو أرقام ال IP، أو راوترات؛ طالما أنها تساعد في سرد قصتك المعمارية ^^

## نصائح خطيرة؛ لكنها ذهبية

بعد أن استعرضنا المخطات المتعددة مثل C4، ومخططات التتابع، والأنشطة، والنشر، وغيرها، يضع Simon هنا "مكابح" تمنع المعماري من الغرق في بحر الرسم والتوثيق ^^، ويمكن تلخيصهم بما يلي:

- الاكتفاء بالحد الأدنى -Minimalism-:

المخططات ليست غاية في حد ذاتها، بل هي وسيلة للفهم، فمجموعة من مخططات C4 مع عدد قليل من المخططات الداعمة مثل ال UML أو النشر تكفي بنسبة 99% لشرح أي نظام مهما كان معقداً... لذلك، أبقِ عدد المخططات في أدنى حد ممكن، لأن كثرة المخططات تعني جهداً أكبر عند تحديثها مستقبلاً = مما سيؤدي إلى تقادمها -Outdated- وفقدان الثقة بها.

- نفخ التوليد التلقائي واختبار الورقة A0

بعض المبرمجين يتباهى بتوليد مخطط قاعدة بيانات -ERD- عملاق من الشيفرة البرمجية، مع أن المخطط الناتج معقد لدرجة أنك تحتاج لورقة بحجم A0 لطباعته! وهنا يسأل Simon سؤالاجوهريا: هل هذا المخطط مفيد حقا؟ هل هناك أحد سيقرأه؟ = لذلك، المخطط الذي لا يمكن للعقل البشري استيعابه بلمحة، هو مخطط فاشل حتى لو كان دقيقاً 100%

- مقياس القيمة -Value-Driven Architecture-

المعيار الوحيد لبقاء أي مخطط أو حذفه هو القيمة، فإذا واجهت أي فكرة غريبة لا يناسبها أي مخطط قياسي فقم باختراع مخططك الخاص وارسمه! وإذا رسمت مخططاً قياسياً حسب الأصول لكن الفريق لا يعود إليه ولا يستفيد منه؟ تخلص منه فوراً!

سؤال للتفكير: لماذا سنتخلص من المخطط القياسي والصحيح إذا لم يعد إليه المبرمجين؟

# فائدة

وأصل فساد العالم إنما هو من اختلاف الملوك والخلفاء، ولهذا لم يطمع أعداء الإسلام فيه في زمن من الأزمنة إلا في زمن تعدد الملوك من المسلمين واختلافهم، وانفراد كل منهم ببلاد، وطلب بعضهم العلو على بعض...

- كتاب الداء والدواء، ابن القيم -رحمه الله تعالى-

# الوحدة الثانية: التوثيق

في هذه الوحدة من الكتاب سنتناول ذلك الموضوع الجوهري الذي نكرهه ونحبه في آنٍ واحد ؛ وهو كتابة التوثيق! إن هذه المفارقة -نكرهه ونحبه- هي الخيط الذي سيبنى عليه الفصل كله = نحب الفكرة، لكن نكره الفعل...!

## الفصل الحادي عشر: توثيق البرمجيات كدليل إرشادي

في عالمنا الجميل = عالم البرمجة، كثيرا ما يردد المبرمجين عبارة ال Agile الشهيرة: "برمجيات تعمل أهم من توثيق شامل"، ولكننا كمبرمجين استخدمنا هذه العبارة كعذر شرعي للتهرب من كتابة أي توثيق! وترى الكثير منا يعتقد أن الشيفرة البرمجية النظيفة والمقروءة -Clean Code- تغني عن التوثيق، وهذا وهم كبير؛ فالشيفرة النظيفة أمر ممتاز ولا بد منه، لكنه لا يروي القصة الكاملة!

تخيل أنك انضممت لمشروع جديد وفتحت الشيفرة البرمجية للمشروع، ستجد أنك فهمت من أسماء الملفات أن النظام مكتوب بلغة مثل ال C# وأن المشروع مقسم لعدة طبقات، لكن هل ستخبرك الشيفرة البرمجية لماذا تم اختيار قاعدة البيانات هذه؟ وهل ستشرح لك كيف يتم تأمين النظام؟ أو أين ستُرفع هذه الملفات في ال Production؟ وكيف تم تحقيق قابلية التوسع في هذه الشيفرة؟ ولماذا اختيرت هذه التقنيات لهذا المشروع؟ وأين سيتم عمل Deploy لهذه المكونات؟ وكيف تم توحيد سياسات الأمان...إلخ

إن الشيفرة البرمجية تخبرك بكيف يعمل النظام في هذه اللحظة، لكن التوثيق المعماري هو الوحيد الذي يخبرك بلماذا بُني بهذه الطريقة، لذلك أكبر مشكلة عند انضمامك كمبرمج جديد لهذه المنظومة هي: "أنك لا تعرف ما لا تعرفه"، لذلك لعلك لن تسأل السؤال الصحيح أصلًا للوصول لما تريده أو ما هو موجود حقا!

## التوثيق كواجب أخلاقي ومهني -Our duty to deliver documentation-

هل من أخلاقك المهنية أن تسلم زبونك "الشيفرة البرمجية" فقط -دون أي توثيق-؟

يضرب Simon على وتر حساس، ففي مشاريع ال Outsourcing، الكثير من الشركات تبني أنظمة معقدة وتسلمها للعميل كملفات فيها مجموعة من الشيفرات البرمجية فقط!

لكن، ماذا لو غادر الفريق الأصلي العمل؟!

سيقف الفريق الجديد عاجزا أمام هذه الشيفرة البرمجية، وأي محاولة لتطويره ستؤدي غالباً إلى تدمير المعمارية الأصلية لأنهم ببساطة لا يعرفون "الرؤية" التي بُني عليها النظام، لذلك يعد تسليم الشيفرة البرمجية دون توثيق مساعد هو خيانة مهنية للعميل وللفريق الذي سيكمل المهمة من بعدك...

**عقدة التسعينيات: التوثيق المرعب -The 1990's template-**

لقد تعرض المطورين في تسعينات القرن الماضي إلى صدمة نفسية عميقة بسبب قوالب التوثيق القديمة! فبمجرد ذكر كلمة "توثيق"، سيتخيل المبرمج ملف Microsoft Word عملاق ومملّ، ويحتوي في ثناياه على مئات الصفحات، وسيجبره على رسم UML Class Diagram لكل Use Case حتى ولو كانت تافهة في النظام! لذلك كان هذا النوع من التوثيق الثقيل لا يطيقه أحد ولا يحب أحد كتابته! والأهم من ذلك: لا أحد يقرؤه ^^...

لذلك، ولمعالجة هذه العقد النفسية التي صارت لدينا، اقترح Simon فكرة أخرى، وهي بناء توثيق تكميلي ليكون كدليل سياحي -The Travel Guidebook- متجدد باستمرار لا مجرد سجل تاريخي ثابت وشامل!

إن أي دليل سياحي لمدينة ما لا يخبرك عن التركيب الكيميائي لحجارة المباني وعدد طبقات الإسفلت ونوعه وكم مقدار الحرارة التي يتحملها... إلخ (والتي تمثل تفاصيل الشيفرة البرمجية)، بل يخبرك هذه الدليل عن المعالم الرئيسية والقوانين العامة للمدينة وكيف تنتقل فيها... إلخ (وهذا يمثل الرؤية المعمارية).

**وهنا يظهر السؤال المحوري: ما هو الشيء الذي يجب أن يحتويه هذا الدليل؟**

إن ما يجب أن يحتويه الدليل يمكن تقسيمه إلى عدة عناصر وهي:

1. الخرائط -Maps-
2. المعالم -Sights-

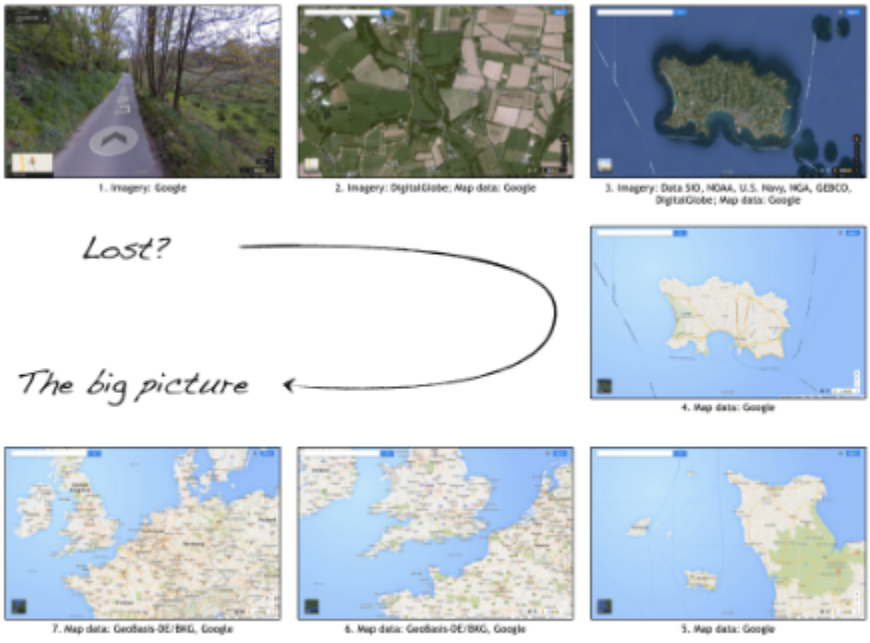
3. التاريخ والثقافة -History and culture-

4. المعلومات العملية -Practical information-

## العنصر الأول: الخرائط -Maps-

تخيّل أنك نُقلت فجأة إلى طريق ريفي هادئ ومجهول في مكان ما من العالم، فكيف ستعرف أين أنت؟ ستقول: يمكنني أن أنادي وأن أطلب المساعدة، أو يمكنني أن أمشي حتى أتعرف على شيء مألوف، لكن هذه الحلول ليست عملية في حالتنا هذه، لذلك، فالحل العملي في عصرنا هو: فتح تطبيق الخرائط على هاتفك المحمول ^^.

شاهد الصورة:




**From the detail to the big picture**

في هذه الصورة تجد أنك في هذا الطريق الريفي في مكان ما من العالم، كل شيء تقريبا سيكون مجهولا بالنسبة لك، فستقوم حينها بإبعاد العدسة قليلا لترى الحي ثم المدينة ثم الدولة ثم القارة وهكذا... إن خلاصة الصورة ستخبرك أنك دون الصورة الكبرى ستبقى تائها مهما بدت التفاصيل الأرضية وهذا الشارع الريفي مألوفا لديك! وهذا الشيء يشبه ما تحدثنا عنه سابقا أو أشرنا إليه: فهذا التشبيه يُمهّد مباشرة لفكرة أن أي نظام معماري بلا "خريطة عامة" مثل System Context أو Container diagram سيجعل أي عضو جديد في الفريق "تائهًا" داخل المشروع، وهذا ما لا نريده... لذلك كانت الصورة العالية والمستويات الأولى مهمة جدا حتى يستطيع أي مطور من الانتقال التدريجي بين أجزاء النظام وفهمها بشكل صحيح...

**العنصر الثاني: المعالم -Sights-**

بعد أن تحدثنا عن الخرائط يجب أن نعي أن الخرائط نفسها قد تكون مربكة، فتخيل أنك الآن على جزيرة سياحية، الخرائط العادية ستكون مربكة وفيها كثير من الشوارع والأبنية والتفاصيل التي لا تهمني كسائح، وهنا تظهر حاجتنا لأهم المعالم التي تستحق الزيارة، وهذا هو ما يميز السائح الذكي والمتمرس عن غيره، فالسائح الذكي لا يقرأ الخريطة التفصيلية لكل شيء في المدينة، بل يبحث في الدليل السياحي عن أهم المعالم التي ينبغي عليه زيارتها، وكذلك الأمر في حالة التوثيق البرمجي! فمن العبث توثيق 200 شاشة من صفحات الويب المتشابهة والتي تؤدي وظائف عادية مثل ال CRUD، بينما الصحيح هو توفير طاقتك وطاقة الفريق

لتوثيق النقاط المعمارية المثيرة للاهتمام: مثل كيف صُممت آلية الأمان؟ ما هي أنماط التصميم المستخدمة؟ وكيف يتعامل النظام مع الضغط العالي؟ باختصار وثّق ما هو مهم حقاً واترك الشيفرة البرمجية تشرح التفاصيل المملة!

**العنصر الثالث: التاريخ والثقافة -History and culture-**

في جزيرة Jersey توجد قلعة حجرية اسمها Elizabeth Castle من القرن السادس عشر، إذا تجولت فيها وأنت سعيد بهذه الرحلة؛ ستجد فجأة أسطوانة خرسانية ضخمة وقبيحة لا تتناسب أبداً مع العمار الكلاسيكي للقلعة! وستقول ما هذا الشيء الغبي وكيف أتى إلى هنا؟! لكنك إذا قرأت اللوحة التاريخية، ستكتشف أن الألمان في الحرب العالمية الثانية أضافوا هذه الخرسانة كتحصين عسكري، وهذه العبارة التاريخية ستفسر لك الشيء الذي بدا غريبا! وهنا يمكننا القول أن الشيفرة البرمجية الغريبة أو القبيحة التي وجدتها في مشروع ما؛ قد تخفي بداخلها قصة تاريخية تجعل من وجودها منطقيا أو صحيحا...

إن أي مبرمج جديد عندما ينضم لفريق ما، سيجد أجزاء من الشيفرة البرمجية تبدو قبيحة أو غير منطقية، وقد يظن حينها أن المبرمجين السابقين كانوا سيئين! لكن، إذا كان هناك توثيق يشرح تاريخ وثقافة المشروع، سيكتشف المبرمج سريعا أن هذه الشيفرة الغريبة كُتبت لغرض منطقي أو فرض عليهم كتابتها في هذا الشكل، مثل رفع مزية ما في ليلة إطلاق المشروع تحت ضغط الإدارة، أو أن تكنولوجيا معينة فُرضت عليهم لسبب أمني...إلخ، لذلك إن معرفة

التاريخ تحمي المبرمجين الجدد من إطلاق الأحكام الخاطئة، وتمنعهم من تدمير الشيفرة البرمجية بحجة ال Refactoring غير المدروسة...

وهنا أود العودة إلى فكرة مهمة كنت قد شاركتها سابقاً من خلال موقع قبيلة -بتصرف-: من الطبيعي جدا أن نراجع أعمالنا وما كتبناه من شيفرات برمجية قديمة؛ فنجد أنها كانت تحتمل تحسينات وتعديلات كثيرة، وكثيرا ما نضحك عليها ونقول: "وين مخي كاين"، وهذه شيفرة نحن كتبناها بأنفسنا! وهذه النظرة في الحقيقة هبة وفضل من الله -سبحانه وتعالى- علينا، فهي دليل على تطورنا المستمر، فالتعلم رحلة لا تنتهي ولا تتوقف، والكمال شيء مستحيل!

لذلك، ستضعنا هذه الحقيقة أمام مقاربة واقعية عند تقييم عمل الآخرين؛ فعندما تقع عيناك على شيفرة برمجية غير متقنة، فأول ما ينبغي أن يأتي على خاطرك هو أن تلتمس العذر لكاتبها قائلا: "أنا أجهل تماماً السياق والظروف التي كُتبت فيها"، وهذا النهج المبني على حسن الظن هو درس ثمين تعلمناه من أحد مسؤولينّا في الماضي، وما زال حتى اليوم يمثل ركيزة أساسية ترتقي بمهنيتنا وأخلاقنا... وبكل تأكيد هناك فرق كبير بين من كتب مشروعا برمجيا جيدا اختلط ببعض السوء في ثناياه، وبين من بنى نظاما سيئا بكل ما فيه! وحتى مع من بنى نظاما سيئا بشكل كامل قد تجد عذرا له! وهذا حدث معي شخصيا عندما رأيت شيفرة برمجية وضع اتصال قاعدة البيانات فيها داخل الجافا سكربت الخاص بال Client، وهذا يعني أن معلومات الاتصال مع قاعدة البيانات كانت مكشوفة للجميع، وهذا عام ٢٠١٤، لكن الشاب الذي عمل عليها لم يكن هذا مجاله، وكان بتوصية أو توجيه من الشركة أن يقوم هذا الشخص بتعلم البرمجة وتصميم الموقع، وكل ذلك خلال شهر أو شهرين... فكانت النتيجة شيفرة سيئة وموقع مدمر داخليا، مع أنه كان يعمل! وبالنهاية تم اختراق الموقع بسهولة وتدمير

البيانات أو سحبها عدة مرات... الخلاصة من هذا كله: إن معرفة التاريخ قد تفسر لماذا يبدو ما نشاهده على ما هو عليه، وحسن الظن مقدم على سوء الظن.

## العنصر الرابع: المعلومات العملية -Practical information-

إن آخر ما يتضمنه أي دليل سياحي هي المعلومات العملية ممثلة بالعملة، والكهرباء، وقوانين الهجرة، والأعراف المحلية، وطرق التنقل وأرقام الطوارئ ونحو ذلك، وهذه جميعها تمثل الجزء اليومي الذي يبحث عنه أي سائح أو زائر!

وإذا أردنا أن نسقط هذا على البرمجة، فإن المبرمج يحتاج إلى نفس الشيء في التوثيق، فأين يوجد رابط Git Repo؟ وما هو الأمر البرمجي لتشغيل أو بناء المشروع -How to build/run-؟ وما هي المعايير -Coding Standards- التي يتبعها الفريق؟

إن مثل هذه المعلومات يجب أن تكون واضحة وفي الصحة الأولى من الدليل السياحي ^^، فلا تجعل المبرمج الجديد يتسول هذه المعلومات من زملائه!

## وثّق ما تعجز الشيفرة البرمجية عن وصفه وقوله!

جميعنا يحب الاستكشاف، فهو أمر ممتع، لكن في عالم الأعمال "الوقت = المال"، والمبرمج الجديد لا يملك رفاهية الاستكشاف لمدة شهر وهو يتجول في الشيفرة البرمجية حتى يفهمها! وهنا تظهر أهمية التوثيق التكميلي كدليل يمنع الناس معلومات كافية للانطلاق، ويسرع من

عملية الاستكشاف، لكن عليك مقاومة إغراء الدخول في التفاصيل التقنية الدقيقة، فهذا نفق قاتل!

إن المبرمج المحترف قادر على قراءة الشيفرة البرمجية لمعرفة كيف تعمل دالة معينة، لذلك لا تكتب له شيفرة برمجية داخل ملف ال Word! فالتوثيق وُجد ليجيب على الأسئلة التي لا يمكن للشيفرة البرمجية الإجابة عنها مثل: القرارات، والقيود، والصورة الكبرى.

وهنا بناء على هذا كله، وبعد هذه المقدمات الجميلة التي بدأنا فيها الفصل الحادي عشر، يبدأ فعليا ذكر المحاور والقائمة التي سيبنى عليها ما تبقى من الكتاب، ويمكن سرد هذه القائمة على شكل ثلاثة عشر قسما، والتي تمثل عناصر ال Software Guidebook.

1. السياق -Context-
2. نظرة عامة على الوظائف -Functional Overview-
3. سمات الجودة -Quality Attributes-
4. القيود -Constraints-
5. المبادئ -Principles-
6. معمارية البرمجيات -Software Architecture-
7. الشيفرة البرمجية -Code-
8. البيانات -Data-
9. معمارية البنية التحتية -Infrastructure Architecture-
10. النشر -Deployment-
11. التشغيل والدعم -Operation and Support-

12. بيئة التطوير -Development Environment-
13. سجل القرارات -Decision Log-

إن هذه الأقسام الثلاثة عشرة، تمثل القالب -template- الذي اقترحه Simon ليمثل الدليل السياحي الذي سنبنيه، وهو ممتاز لفرق ال Agile التي تريد توثيقا سريعا ودون تعقيد، وهو ما سيشرحه Simon من الآن فصاعدا، ومع ذلك، فإن Simon أشار لنقطة مهمة، وهي إمكانية استخدامك لأي توثيق قياسي تراه موجودا أو منتشرا مثل قالب arc42... المهم هنا أن تبتعد عن قوالب التسعينات المعقدة ^^ ،أن تختار من القوالب ما يناسب ويغطي ما ذكرناه من أبواب...

**المشروع -Project- لا يساوي المنتج -Product-!**

أظنك تعجبت من العنوان، لكن هذا العنوان وضع لإزالة اللبس عن واحدة من أكثر الأخطاء التي نقع فيها كمطورين أو مديري فرق! فكثيرا ما ترانا نستخدم كلمة مشروع أو منتج كمترادفات أو للدلالة على نفس الشيء، في حين أنها في الواقع العملي شيء مختلف مثل الرحلة والسيارة :P

إن المنتج يمكننا تعريفه بالكيان الدائم الذي يعيش طويلا ليقدم خدمة للمستخدمين، وله دورة حياة من الولادة وحتى الممات... أما المشروع فهو يمثل فترة زمنية مؤقتة لها بداية ونهاية محددة، وميزانية معينة الهدف منها إحداث تغيير ما على منتج سواء من خلال بنائه لأول مرة أو من خلال تعديلة وإضافة ميزة عليه...

في مثالنا الخاص بالبنك يمكننا القول مجازا:

- في عام 2020 قام البنك بإنشاء "نظام البنك الإلكتروني" لأول مرة.. وهذا كان **المشروع الأول.**
- في عام 2023 أراد البنك إضافة نظام الدفع عبر Apple Pay لنظام البنك.. وهذا هو **المشروع الثاني.**
- في عام 2026، قرر البنك تغيير قاعدة البيانات من Oracle إلى PostgreSQL.. وهذا هو **المشروع الثالث**

إن نظام البنك هو المنتج، وكما تلاحظ قد تكون تتقاطع نقطة الانطلاق فيكون المشروع والمنتج وجهين لعملة واحدة، لكن بعد انتهاء المرحلة الزمنية المخصصة للمشروع وانطلاقه -بدأ المنتج يعيش- تكون هذه النقطة الفارقة، وهنا يكمن اللبس...

الآن عودة لنظام البنك، لو أن البنك حسب تسلسلنا الزمني وثق المشروع الأول وسماه ثويق تأسيس النظام، ثم وثق المشروع الثاني وسماه توثيق الدفع عبر أبل ثم وثق المشروع الثالث وسماه توثيق تغيير قاعدة البيانات... في هذه الحالة حتى يفهم المبرمج المنتج الحالي عليه أن يقرأ هذه الوثائق كلها ويربطها معا في عقله ليستنتج شكل النظام الجديد، وهذا ما لا نريده أبدا! وهنا يقول Simon: "توقفوا عن ذلك! اكتبوا وثيقة واحدة تصف المنتج، وكلما انتهى مشروع، قوموا بتعديل وثيقة المنتج لتعكس التحديث الجديد، ثم تخلصوا من أوراق المشروع لأن فترته انتهت!".. وحتى تتضح الفكرة من الواقع العملي بشكل أفضل انظر لهذا المثال اللطيف:

تخيل أنك تمتلك منزلا جميلا... هذا المنزل هو المنتج الذي ستعيش فيه لسنوات طويلة، هذا المنتج مر دورة حياة وهي:

- أولا اشتريت الأرض واستأجرت مقاولاً لبناء المنزل يبدأ بشهر ١ من عام ٢٠٢٠ وينتهي بشهر ٤ من عام ٢٠٢٠، وهذا كان مشروعك الأول.
- بعد ٣ سنوات، قررت إضافة مسبح في الفناء الخلفي للمنزل.. وهذا كان مشروعك الثاني.
- بعد ٥ سنوات، قررت هدم المطبخ القديم وتجديده، وفتح واجهة المطبخ مع واجهة معلقة وبروز ٣ متر للخارج، وهذا هو مشروعك الثالث.

الآن، تخيل أنك أردت بيع المنزل بعد أن سكنت فيه ١٥ سنة، وطلب منك المشتري المخطط الهندسي للمنزل الحالي، فماذا ستعطيه؟
تخيل أنك قمت بإعطاءه مخطط بناء المنزل القديم ومخطط المسبح ومخطط المطبخ الجديد، وقلت له اجمع هذه الأوراق لتعرف شكل المنزل الآن، فماذا تظن ستكون ردة فعله؟!
هذا بالضبط ما سيحدث لو قمت بالاعتماد على توثيق المشاريع فقط التي قد تكون انتهت وماتت منذ سنوات، والصحيح هو عكس توثيق المشروع على توثيق المنتج وتحديثه بعد أن ينتهي كل مشروع، وبهذا سيكون هناك مخطط واحد محدث للمنزل يمكنك مشاركته مباشرة مع أي مشتر جديد...

الخلاصة التي نريدها من هذا الكلام كله هي:
أنشئ دليل إرشادي واحد -Single Software Guidebook- لكل نظام برمجي تبنيه، وهذا لا يعني إلغاء التوثيق على مستوى المشروع (project-level)، بل يعني أن يكون هناك مكان واحد مرجعي يروي كيف يعمل المنتج وكيف تطوّر عبر الزمن، ومتى وُجد هذا الدليل

سيصبح كل مشروع أو تغيير لاحق مجرد تعديل صغير -delta- يضاف إلى توثيق المنتج؛ بدلا من أن يُضطر القادم الجديد لتجميع الفتات من عشرات الوثائق لمشاريع متراكمة عبر 20 عاما...

والآن:

**كيف يمكننا أن نمنع موت التوثيق بسبب الإهمال والتجاهل؟ وكم يجب أن يكون طول التوثيق حتى لا يموت بهذا السبب؟**

هذا السؤال الجميل يمكن الإجابة عليه بعبارتين بسيطتين: "قليل دائم خير من كثير منقطع"! و "معيار فنجان القهوة".

إن الفرق التقنية عادة ما تترك مهمة التوثيق إلى نهاية المشروع، فيصبح هذا التوثيق كابوسا ثقيلاً يحتاج لأسابيع من الكآبة المملة *-*، لذلك يقترح Simon هنا حلا بسيطا وجميلا من خلال تعديل مفهوم ال DoD، والذي هو اختصار ل Definition of Done، أي أنه عدل تعريف الإنجاز لأي مهمة برمجية، فمثلا لا يمكننا إغلاق أي مهمة تصل لنا كمبرمجين إلا بعد التحقق من أن الشيفرة البرمجية تعمل وأن الاختبارات تمت بنجاح، والشيفرة البرمجية تمت مراجعتها والموافقة عليها... وهنا نقوم بإضافة عنصر جديد اسمه: "تم تحديث التوثيق والمخططات"، وبذلك يصبح التوثيق من المهام اليومية الصغيرة، وتحديث صغير في أي وثيقة لن يأخذ أكثر من ٥ دقائق = أسهل مليون من مرة من تخصيص شهر كامل للتوثيق بعد نهاية المشروع، وهذا يمثل العبارة الأولى...

أما العبارة الثانية فهي تمثل مقياس الطول لأي توثيق، فإذا استطعت قراءة التوثيق خلال ساعتين -معدل شرب فنجانين من القهوة- فهذا يعني أن التوثيق ممتاز، وهذا يغير نظرتنا عن عدد الصفحات المسموح؛ لأننا لا نقدم مرجع أكاديمي هنا بل نقدم دليل سياحي، وهذا يمثل جرعة مركزة من الفهم حتى يكون جاهزا بعدها المبرمج للغوص في الشيفرة البرمجية، وكلما زاد الوقت زاد الكابوس، وزادت احتمالية عودتنا لكابوس التسعينات...

الخلاصة:

المعماري الذكي يقيس طول التوثيق بمعيار فنجان القهوة، أي أنه يجب أن يكون التوثيق قابلاً للقراءة والفهم في غضون ساعتين أو وقت شرب فنجانين من القهوة! فإذا احتاج المبرمج الجديد لأكثر من ذلك ليفهم الصورة الكبرى، فاعلم أنك تورطت في التفاصيل التقنية الدقيقة التي كان يجب أن تتركها للشيفرة البرمجية لتحكيها! وحتى تحافظ على التوثيق محدثا ويمثل المشروع حقيقة، اربط التوثيق بتعريف الإنجاز، ولا تقبل أي طلب Pull Request ما لم يقم المطور بتحديث التوثيق المعماري بما يتناسب مع الشيفرة البرمجية الجديدة، فتحديث سطرين في التوثيق اليوم سيوفر عليك شهراً من الكوابيس غداً!

والآن، لنذهب سويا ولنبدأ بأقسام التوثيق التي ذكرناها...

## الفصل الثاني عشر: القسم الأول من الدليل = السياق -Context-

يُمثل قسم السياق البوابة الأولى لفهم أي نظام برمجي، حيث يُجيب على الأسئلة الكبرى للصورة العامة للنظام بعيداً عن التعقيدات البرمجية، ويهدف هذا القسم إلى سرد قصة النظام وتوضيح ماهيته، وكيفية اندماجه ضمن بيئة العمل الأوسع، وتحديد كافة المستفيدين منه -مستخدمين وأنظمة خارجية-، كما تأتي أهمية هذا التوثيق من كونه المنقذ الحقيقي للفرق التقنية من "كابوس الصفحات الثلاثين"؛ فبدلاً من إضاعة وقت القارئ في تفاصيل قواعد البيانات قبل أن يفهم الغاية الأساسية للنظام أصلا، يمنحك هذا القسم -السياق- رؤية واضحة ومباشرة تمنع التشتت وتضع الجميع على نفس الخط، وتتجلى قوة هذا القسم في هيكلته شديدة البساطة التي تجعل منه "وجبة خفيفة" لا تتجاوز الصفحتين! معتمدين على مبدأ "صورة تغني عن ألف كلمة"، وهنا نستدعي ما تعلمناه سابقا في المستوى الأول من نموذج C4 ^^، وهو مخطط السياق -System Context Diagram-، حيث تكفي صورة واحدة مدعومة ببضعة أسطر لاختزال النظام بأكمله، ولأنه يُكتب بلغة واضحة خالية من التعقيدات التقنية، فهو يُعد قسما إلزاميا لا غنى عنه؛ إذ أن الجمهور المستهدف هنا لا يقتصر على المطورين الجدد فحسب، بل يشمل الجميع بلا استثناء، بدءاً من مديري المشاريع ومختبري الجودة، وصولا إلى الإدارة التنفيذية.

إن الأسئلة الكبرى لهذا القسم يمكن تلخيصها ب:

ما هي قصة هذا النظام؟ وماذا نبني نحن بالضبط؟ وكيف يندمج هذا النظام مع بيئة العمل الحالية (مثل هل هو نظام مستقل أم جزء من شبكة أنظمة)؟ ومن هم الذين يستخدمون هذا النظام (مثل المستخدمين، أم الأنظمة الخارجية...)
إذا استطعت الإجابة عن هذه الأسئلة مع توضيح الدافع والهيكلية للنظام، ووصلت للجمهور المستهدف بالشكل الصحيح... فمبروك عليك هذا النظام الجميل ^^

الآن بعد أن حددنا السياق في هذا الدليل، ننتقل لإجابة سؤال ماذا يفعل هذا النظام بالضبط من الداخل؟

## الفصل الثالث عشر: القسم الثاني من الدليل = Functional Overview

في هذا القسم نحن لا نريد شرح كل زر في النظام، فهذا ليس دليل استخدام! بل ما نريده هو ربط الوظائف الموجودة بالمعمارية = أي التركيز على المهام الرئيسية التي تؤثر على القرارات التقنية، مثل هل النظام يخدم مستخدمين عاديين أم مدراء؟ وما هي ال User Stories وال Use Cases ذات الأهمية المعمارية (مثل وجود مزية تتطلب تحديثا بالوقت الفعلي)، وإذا كان نظامنا عبارة عن عملية أتمتة لمجموعة من العمليات، فيجب أن نوضح مسار تدفق البيانات عبر النظام..

بناء على هذا الكلام، أسهل طريقة لكتابة هذا الجزء من التوثيق ببساطة هو وضع روابط المهام -Tickets- الموجودة على Jira في هذه الصفحات ونكون انتهينا! ولا تتعجب من ذلك يا صديقي، فأي وثيقة متطلبات -Functional Specs- جاهزة وموجودة يمكن استخدامها مباشرة، فلماذا نعيد بناء العجلة؟!

ومع ذلك، قد تكون هناك بعض الحركات اللطيفة التي يمكنك استخدامها لتسهيل القراءة وتحسين تجربة القارئ، مثل استخدام ال UML USE CASE، ونماذج ال WireFrames، ونعرض فيها رسومات سريعة لأهم الشاشات، وهذه مفيدة جدا في توضيح الفكرة دون قراءة الكثير من الأسطر، كما يمكن استخدام ال Activity Diagram للأنظمة التي تعتمد على مسارات عمل معقدة...

وبهذا نستنتج أن الدافع لهذا الجزء من الدليل هو أن ننظر للنظام من الداخل، وهو يمثل الجزء الذي يمهد الطريق قبل الغوص في تفاصيل الشيفرة البرمجية وقواعد البيانات... وبهذا نكون نظرنا للنظام من الخارج -أين يعيش- من خلال دليل السياق، ونظرنا له من الداخل -ماذا نفعل- من خلال النظرة الوظيفية، كما أن هذا المستوى من الدليل إلزامي ولا غنى عنه، وهو يستهدف الجمهور التقني وغير التقني، فحتى مدير التسويق مطالب بفهمه.

## الفصل الرابع عشر: القسم الثالث من الدليل = Quality Attributes

لقد تحدثنا عن سمات الجودة في الجزء الأول من الكتاب، لذلك لن نطيل الحديث هنا عنها، لكننا سنذكر ما يهمنا في هذا القسم ليتم توثيقها مع الدليل السياحي بطريقة صحيحة...

إن سمات الجودة تخبرنا "كيف وبأي كفاءة" سيفعل النظام مهمة ما، وغالبا ما تكون هذه السمات هي سبب قلب المعمارية وتعقيد الأنظمة! فبناء صفحة تسجيل دخول أمر سهل جدا، لكن بناء صفحة تسجيل دخول تتحمل مئة ألف مستخدم في الثانية هو ما سيغير المعمارية! لذلك، يجب أن تكون هناك قواعد صارمة لتعريف وكتابة سمات الجودة مثل:
١. هل هناك فهم واضح لسمات الجودة التي يجب أن تحققها المعمارية؟

٢. يجب أن تكون SMART: أي محددة وقابلة للقياس وقابلة للتحقيق وذات صلة ومحددة بوقت!
٣. هل السمات التي تُؤخذ كأمر مسلّم به عادة ذُكرت صراحة كتفاصيل "خارج النطاق" إن لم تكن مطلوبة؟ مثل: أن واجهة المستخدم ستكون باللغة الإنجليزية فقط، للإشارة إلى أن دعم تعدد اللغات غير مُخطط له... ولا يوجد تطبيق للهواتف المحمولة!

٤. هل بعض السمات غير واقعية؟ (تذكر الأمثلة من الجزء الأول أريد نظام سريع، وأريد توافرية بنسبة ١٠٠٪...إلخ)، فدورنا أن نوضح أن هذه الأهداف غير واقعية أو مكلفة أو تحتاج لتوضيح لضبطها وتعديلها لتصبح واقعية بما يتناسب مع المشروع وطبيعته...

بناء على ما ذكرناه، فهذا الجزء من التوثيق السياحي سيحتوي قائمة ال "ilities" التي تحدثنا عنها بالجزء الأول من الكتاب، مثل الأداء والتوسع والأمان...إلخ، لكن احذر هنا من أن تقع في نفس فخ الكلمات المطاطية! فإياك أن تستخدم جمل مثل: "يجب أن يكون النظام سريعا" و"يجب أن يتحمل أكبر عدد من العملاء" ... بل يجب أن نستخدم شيء مثل: "يجب

أن يستجيب الخادم في أقل من ٢٠٠ جزء من الثانية لأي طلب"، ويجب أن تكون التوافرية "٩٩،٩٪ - أي يسمح بالتوقف لحوالي ٨ ساعات سنويا"... إلخ

وكما تعلمنا في الجزء الأول من الكتاب، فهذا القسم لن يقدمه لك العملاء على طبق من ذهب! بل وظيفتك كمعماري هي تعقب كلماتهم المطاطية ومحاولة استخراج هذه السمات من تفاصيل الكلام والمعلومات التي يمكن جمعها.. وبناء على كل هذا يظهر جليا أن هذا القسم من الدليل يستهدف القسم التقني، وهو إلزامي وبشدة! لإنها من تشكل المعمارية بالواقع!

## الفصل الخامس عشر: القسم الرابع من الدليل = Constraints

لقد تحدثنا أيضا عن القيود في الجزء الأول من الكتاب، ومع ذلك، نعود لاقتباس Simon الجميل مجددا: "البرمجيات تعيش ضمن سياق العالم الحقيقي، والعالم الحقيقي له قيود"... إن من أهم الأفكار التي خلصنا إليها عند حديثنا عن هذا الباب هي أن القيود تمثل الأسوار التي رسمت لنا، والتي ستجبرنا على اتخاذ قرارات معمارية قد تبدو غريبة أو غير مثالية في الظروف الطبيعية، لذلك كان أخذها والاهتمام بها من العوامل والركائز الأساسية لأي معمارية، ولهذا وجب وجود جزء لها في الدليل السياحي...

ومع ذلك، فإن القيود التي تذمرنا منها لها فائدة كبيرة جدا، فهي تقلل من شلل التحليل، فعالم البرمجيات مليء بالخيارات التي لا حصر لها، وعندما يفرض عليك قيد معين مثل ممنوع استخدام الخدمات السحابية لأسباب أمنية، فأنت بذلك قلصت خياراتك وسهلت عملية اتخاذ

القرار عليك... وهذا القسم سيوضح صراحة ما هي القرارات التي اتخذت بالنيابة عنك، وأنت لا تملك سلطة لتغييرها...

نستنتج من المقدمة السابقة أن هذا القسم سيحتوي على قائمة بسيطة تشرح القيود الموجودة ومن فرضها ولماذا، وهذا يشمل جميع أنواع القيود مثل القيود المالية والزمانية، والقيود التقنية، والقيود البشرية، والقيود السياسية والتنظيمية... وقد تحدثنا عن كل ذلك بالتفصيل في الجزء الأول من الكتاب ^^ وبهذا إذا انضم مبرمج جديد وقال: "لماذا استُخدمت هذه التقنيات القديمة والغبية مع أنه يتوافر غيرها في السوق! فستقول: كانت سياسة العميل تجبرنا على استخدام هذه التقنية -قيد-، وهو ما منعنا من استخدام غيرها في ذلك الوقت"... ومن سيجيب هو الدليل السياحي، ولن يضطر الزائر الجديد من السؤال أصلا ^^

إن المستهدف من هذا القسم من الدليل هم التقنيون وغير التقنين، لأن القيود قد تكون تقنية وقد تكون مالية وقد تكون قانونية... إلخ، وهي إلزامية بكل تأكيد، أي يجب كتابة هذا الجزء من الدليل.

## الفصل السادس عشر: القسم الخامس من الدليل = Principles

لقد تحدثنا أيضا عن المبادئ في الجزء الأول من الكتاب، وهي تشمل المبادئ التي تتبعناها واعتمدناها للعمل على تصميم وبناء التطبيق، سواء طلبها ال stakeholders صراحة أو تبناها

الفريق نفسه... وقد ذكرنا سابقا بأن المبادئ تمثل القوانين التي يتبناها الفريق أو يفرضها الفريق على نفسه، في حين أن القيود تفرض من الخارج كقرارات تم اتخاذها مسبقا ^^
وبهذا، يكون الهدف من هذا القسم في الدليل السياحي هو جعل المبادئ التي يتبعها الفريق "صريحة ومكتوبة"، وهذه المبادئ إما أن يفرضها ال stakeholders مثل أن يتم استخدام استراتيجية الشراء بدلا من البناء -Buy rather than build-، أو يتبناها الفريق نفسه لتوحيد الرؤية التقنية.

ملاحظة: للتذكير، فإن الخط الفاصل بين المبادئ والقيود كالشعرة، ويمكنك العودة للجزء الأول من الكتاب للاطلاع على مزيد من التفاصيل، لكن للتذكير فإن المبادئ تمثل قواعد إرشادية تساعدك على اتخاذ القرارات في حين أن القيود هي قرارات اتخذت بالنيابة عنك، والمبادئ يمكن تجاهلها في بعض الأحيان بعد تحقيق بعض الشروط اللازمة لها، في حين أن القيود لا يمكن تجاهلها!... إلخ، لذلك Buy rather than build تمثل مبدأ وإن كانت ال Stakeholder، لإنها وضعت مبدأ سيسير عليه الفريق أثناء تطويره للناظم، بينما لو كانت استخدم Oracle Database لأن البنك يريدها فهنا صارت قيدا.

حسنا، كيف سنقوم بكتابة هذا الجزء من التوثيق؟
ببساطة إذا كان لديك صفحة README أو Wiki أو أي مصدر تم اعتماده ليكون بداخله المعايير التي اتفقتم عليها فاكتفي بوضع الرابط الخاص بها ^^.. أما إن لم يكن هذا الملف موجودا فبكل بساطة قم بكتابة الكلمات المفتاحية فيها مثل:

- تطبيق مبادئ ال SOLID وال DRY

- ممنوع كتابة ال Business Logic داخل ال Views
- ال Component يجب أن تكون Stateless لتسهيل عملية ال Scaling.
- قمنا بتوحيد طريقة التعامل مع الأخطاء وتسجيلها بالشكل الفلاني أو من خلال الطريقة التالية...

حسنا، هل تذكر فائدة المبادئ؟ إذا لم تذكر فارجع للجزء الأول من الكتاب :P...
إن أهم فائدة للمبادئ هي تحقيق الاتساق في آلية كتابة الشيفرة البرمجية، وبهذا بدلا من تقرأ شيفرة برمجية مكتوبة بخمسة عقليات مختلفة من خمسة مبرمجين مختلفين، ستقرأ شيفرة برمجية واحدة مكتوبة من قبل خمسة مبرمجين... مثلا: تخيل أن يستخدم مطور ال ORM وآخر يستخدم ال SQL بشكل يدوي؟ وتخيل أن أحدهم يقوم بكتابة ال Business logic داخل ال view والآخر داخل ال Model والثالث في مكان آخر... في هذه الحالة سيتحول المشروع لزومبي ^^، أما لو قلنا أننا سنستخدم ال ORM كمبدئ، ووجب على الجميع اتباع ذلك = انتهت المشكلة قبل أن تبدأ.

وبكل تأكيد، هذا الجزء موجه للتقنين داخل فريق التطوير، وهو إلزامي أيضا لأن الدليل السياحي يجب أن يوضح المبادئ التي بني عليها النظام... وعند انضمام عضو جديد كل ما عليه فعله هو اتباع هذه المبادئ...

# الفصل السابع عشر: القسم السادس من الدليل = Software Architecture

إن كل ما تحدثنا عنه في الأقسام السابقة كان تمهيدا لهذا الجزء من الدليل، فبعد قراءتك للأجزاء السابقة فأنت تحتاج للخريطة التي تبرز هيكل النظام، والتي من خلالها يجب على المبرمج أن يعرف ما هي الحاويات الأساسية والتقنيات المستخدمة؟ وما هي ال component الداخلية وكيف تتفاعل مع بعضها؟ وما هي نقاط الاتصال الرئيسية بين أجزاء النظام؟

وهذا كله موجود مسبقا من خلال نموذج C4 من خلال تضمين مخططات المستوى الثاني والثالث ^^، أرأيت فائدة نموذج C4 وجماليته... ومع ذلك، فنحن نقوم بوضع مخطط الحاويات ومخطط ال component للأجزاء المهمة، ويمكننا ارفاق كل مخطط مع نص قصير يشرح ما يراه القارئ، وإذا كان هناك تفاعل معقد بين المكونات المختلفة، فيمكن استخدام ال UML Sequence Diagram الذي تحدثنا عنه سابقا... لكن، تذكر ما حذرنا منه سابقا = قاوم بشدة الإغراء الخاص برسم مخطط تتابع لكل حالة استخدام، واستخدمه فقط عندما يضيف قيمة حقيقة...

والفائدة من هذا الجزء من التوثيق هو أن يعرف المبرمج الأقسام المهمة في المعمارية وكيف ترتبط فيما بينها، و سيتمكن من خلالها من الذهاب للموقع الذي يريد التعديل عليه بسهولة أو معرفة أماكن التعديل واستكشافها ببساطة بدلا من أن يقوم بفتح مئات الملفات بشكل يدوي ليستنتج كيف يعمل النظام... وهذا أصلا من أهداف المستويين الثاني والثالث من

النموذج ^^... وبناء على ذلك فالمستهدفين من هذا الجزء من الدليل هو الفريق التقني، وهو إلزامي دون نقاش! فلا يوجد دليل يمكنه أن يكون موجودا دون أن يشرح المعمارية التي بني عليها المشروع!

## الفصل الثامن عشر: القسم السابع من الدليل = Code

قد تظن من الاسم أن هذا القسم هو المكان الذي ننسخ فيه الشيفرات البرمجية! لكن هذا الكلام عير صحيح، فهذا المكان الهدف منه شرح الآليات المعقدة والأطر المخصصة التي بناها الفريق وصارت تمثل تعويذة غامضة للمبرمج الجديد!

والأمثلة على ذلك كثيرة مثل كيف تعاملنا مع الأمان والصلاحيات، وهل بنينا أداة متخصصة لمعالجة الأخطاء وتسجيلها؟ وكيف قمنا بتطبيق الطبقات المعمارية في الشيفرة البرمجية؟... إلخ، لذلك احذر من إنشاء المخططات بشكل آلي من الشيفرة البرمجية لأنك ستحصل على شبكة عنكبوتية لا أحد يريدها ^^... بل ما نحتاجه هنا UML Class Diagram أو Sequence Diagram، ولا نحتاج لرسم كل دالة وكل متغير، بل ما نريده هو رسم مخطط عالي التجريد يدويا نضع فيه الخصائص والدوال الجوهرية، وهذا المخطط غالبا ما يصمد لسنوات ولن يتأثر بتعديلات الشيفرة البرمجية الطفيفة... وهذا الجزء من التوثيق مفيد لحماية المبرمجين الجدد من الإحباط، وضمان أن يقوم الجميع بصيانة النظام بنفس الطريقة، ودون كسر الآليات المعقدة التي أسسها المبرمجين المحترفين في الفريق، وهو ليس إلزامي، لكن ينصح بكتابته لأي مشروع غير تافه ^^.

# الفصل التاسع عشر: القسم الثامن من الدليل = Data

هناك حقيقة قاسية في هندسة البرمجيات: "المبرمجون يعشقون الشيفرة البرمجية، لكن الشركات لا تكترث إلا للبيانات!"... إن البرمجيات تُكتب، وتُمسح، ويُعاد كتابتها من الصفر، لكن البيانات هي الثروة التي تتوارثها الأنظمة وتعيش أطول من الشيفرة البرمجية نفسها!، وهنا يقول Simon: "لا تُعد البيانات المرتبطة بنظام برمجي عادة محور التركيز الرئيسي، ومع ذلك فهي ربما أهم من البرنامج نفسه، لذا من المفيد غالبا توثيق شيء ما عنها"

من هذا المنطلق، يأتي هذا الجزء من الدليل السياحي ليجيب عن أسئلة مثل:
كيف يبدو نموذج البيانات؟ وأين تُخزَّن البيانات؟ ومن يملك البيانات؟ وكم مساحة تخزين مطلوبة -سؤال مصيري إذا كنت تتعامل مع Big Data-؟ وما هي استراتيجيات الأرشفة والنسخ الاحتياطي؟ وهل هناك قوانين تنظيمية تجبرنا على الاحتفاظ بملفات السجلات -Logs- لعدة سنوات؟

ومع ذلك، كما حذرنا في القسم السابق من خطورة التوليد التلقائي لمخطط الشيفرة البرمجية، فلنحذر هنا من نفس الشيء؛ من خطورة توليد مخططات ال ERD الذي يحتوي على كل الجداول وكل الأعمدة التي فيها وكل key... لأننا سنقع بنفس المشاكل الموجودة هناك ^^، بدلاً من ذلك، ارسم مخططا عالياً التجريد لل Domain Model يوضح الكيانات الرئيسية فقط، مثل: العميل والطلب والفاتورة، ومنها دورة الحياة مثل العميل لديه علاقة بعربة التسوق، والعربة تتحول إلى طلب، والطلب يصدر فاتورة = هذا كل شيء، أما من يريد

معرفة نوع بيانات حقل معين، فليذهب لقراءة قاعدة البيانات نفسها مباشرة! وهنا تبرز أهمية توثيق القرارات المعمارية لتصميم البيانات، مثل توضيح سبب اختيارك لهيكلة البيانات بمرونة عبر أعمدة JSONB في PostgreSQL مدعومة بفهارس GIN، بدلاً من التورط في تعقيدات ال EAV، هذا هو النوع من المعرفة المعمارية الذي يحتاجه الفريق في هذا القسم!

مثال عملي: تخيل أن لديك نظام يحتاج إلى إضافة منتجات بخصائص متغيرة مثل لابتوب له حجم ذاكرة ومعالج، بينما القميص له لون ومقاس، المطورون قد يلجؤون لنمط EAV لمرونته من خلال بناء ثلاثة جداول هي Entity, attribute, value، لكن المشكلة كانت هنا أننا سنحتاج إلى ٥ عمليات join في بعض جمل الاستعلام لاستخراج منتج بخمس خصائص... أما من خلال ال JSONP فسيتم تخزين هذه الخصائص داخل عمود مع فهرسة GIN، مما سيوفر سرعة هائلة في استرجاع البيانات... هذه جميعها قرارات معمارية -مقايضات- تتعلق بالبيانات يجب توثيقها بطريقة صحيحة حتى نعرف تأثيرها على البيانات وسرعة استرجاعها أو كتابتها أو تأثيرها على فهرسة قواعد البيانات... إلخ.

بناء على هذه المعطيات نستنتج أن الجمهور المستهدف لهذا الدليل هم المطورون، ومهندسو التشغيل، ومديري قواعد البيانات -DBAs-، وهو ليس مخطط إلزامي، لكنه عمليا مفيد جدا وينصح به بشدة...

# الفصل العشرون: القسم التاسع من الدليل = Infrastructure Architecture

عادة ما نميل كمبرمجين للبقاء في منطقة الراحة الخاصة بنا: مثل منطقة الشيفرة البرمجية، والخوارزميات، وقواعد البيانات! لكن ماذا عن الخوادم؟ والشبكات؟ وال Load Balancers...إلخ؟ وهذا يذكرنا فيما تحدثنا به في الجزء الأول من هذا الكتاب، مختصرا: "إن المعمارية البرمجية تتعلق بالبرمجة والبنية التحتية، بال Software وال Hardware".

إن إهمال البنية التحتية يشبه تماما أن تقوم بتصميم برنامج خطير وقوي يشبه سيارة الفيراري؛ لكنك تجاهلت أن هذه السيارة ستسير على شوارع ترابية *_*، أي سيعمل البرنامج القوي والخطير على خوادم قديمة وبطيئة، وهنا النظام سيفشل بكل تأكيد والجميع سيلوم هذه الشيفرة البرمجية ^^... لذلك، يقوم Simon بوضع هذا القسم في الدليل ليجبرك كمعماري برمجيات على الخروج من فقاعتك البرمجية! ويلزمك بتوثيق ما يلي:

- أين سيُرفع النظام؟ على Cloud server أم local server؟
- هل لدينا خطة للتعافي من الكوارث -Disaster Recovery-؟
- هل لدينا بيئات عمل منفصلة مثل ال Dev وال Test وال Prod؟
- من المسؤول عن صيانة هذه الخوادم؟ هل لدينا فريق DevOps أو فريق IT متخصص لهذا الغرض؟
- هل من الواضح كيف حددت حجم مكونات الأجهزة؟
- كيف سنتجاوز الفشل إن حصل؟

إلخ من الأسئلة

يمكن الإجابة عن هذه الأسئلة من خلال رسم Network Diagram، وبكل بساطة إذا كنت تعمل في شركة ضخمة، فاطلب مساعدة صديق واذهب لمهندس البنية التحتية الموجود عندكم بالشركة، خذ منه المخطط وضعه في هذا القسم مع شرح بسيط ^^، فهذا يثبت للإدارة وللفريق القادم أنك صممت نظاما قابلا للتشغيل على أرض الواقع، وليس فقط على جهازك الشخصي ^^... أما إذا كانت الشركة صغيرة، فاستعن بالله تعالى، وشمر عن ساعديك، وابدأ بالرسم ^^، وبالطبع هذا المخطط إلزامي وهو يستهدف المطورين وال devops وفريق الدعم الفني.

## الفصل الواحد والعشرون: القسم العاشر من الدليل = Deployment

إن هذا القسم من الدليل سيخبرنا ببساطة كيف سترتبط -mapping- البنية التحتية بالبرمجية ^^، أي كيف ستعيش الحاويات داخل الخوادم؟

إن بيئة التطوير المحلية -localhost- غالبا ما تعمل دون مشاكل تذكر، والحياة سعيدة وبسيطة ^^، لكن بمجرد أن تذهب إلى العالم الحقيقي ستجد أن الأمور أكثر تعقيدا، وأنك بحاجة للاهتمام بالعديد من النقاط وإدارتها... وهذا ما سنتطرق له في هذا القسم من الدليل ^^.

ملاحظة: تذكر مخطط النشر الذي تحدثنا عنه في هذا الكتاب في الفصول السابقة ^^.

هذا القسم هدفه الأساسية أن يجيب على الأسئلة الغامضة حول كيفية تشغيل النظام في بيئة ال Production، ولا يقتصر هذا الأمر على "أين سنضع الشيفرة البرمجية؟"، بل يغوص في استراتيجيات التشغيل مثل:

- خريطة التوزيع: هل سنضع كل الحاويات في خادم واحد One-to-one، أم سنوزع تطبيق الويب على عدة خوادم -Server Farm-؟
- تقسيم الموارد: كيف سيتم تقسيم ال RAM وال CPU بين العمليات المختلفة على نفس الخادم؟
- استراتيجية التوافر: هل الخوادم تعمل بنظام Active-Active أي كلها تستقبل الطلبات معاً؟ أم Active-Passive خادم يعمل والآخر احتياطي صامت يتدخل عند الطوارئ (Hot/Cold Standby)؟
- الخطة "ب": ما هي استراتيجية التراجع -Rollback- إذا فشل التحديث الجديد ودمر النظام؟

هذه الأسئلة وغيرها يمكن توثيق إجابتها من بطريقتين، الأولى من خلال جدول بسيط يربط اسم الحاوية بالخادم الذي سترفع عليه، أو من خلال رسم مخطط نشر يوضح الحاويات ومكانها داخل الخوادم -كما تعلمنا سابقا-، ويمكن بكل تأكيد من باب التوجيه البصري الفعال يمكننا أن نقوم بتلوين الخادم النشط -Active- باللون الأخضر، في حين يتم تلوين الخادم الاحتياطي ال passive باللون الرمادي... وهكذا

وقد يتسائل أحدنا الآن مجددا، لكننا نستخدم أدوات النشر الآلي، لماذا نحتاج لهذا القسم من التوثيق! ومع أننا تطرقنا لإجابة هذا السؤال من قبل هنا، لكن لأهميته سنقوم بتكرار

الإجابة بطريقة أخرى، يقول Simon: «البرامج النصية -Scripts- الخاصة بهذه الأدوات هي بالنهاية "شيفرة برمجية"، لذلك، أنت لا تزال بحاجة إلى "نظرة عامة مقروءة للبشر" تشرح استراتيجية النشر المعقدة دون أن نضطر لفك طلاسم كود الأتمتة!»، بهذا نستنتج أن الجمهور المستهدف لهذا القسم هم التقنيون وفريق ال DevOps، وهو إلزامي بكل تأكيد.

## الفصل الثاني والعشرون: القسم الحادي العشر من الدليل = Operation and Support

هل رميت في حياتك نظام قمت ببنائه في ثقب أسود وتمنيت له التوفيق في حياته؟
أتمنى ألا تكون قد فعلت ذلك ^^
كمطورين للبرامج يجب أن نعلم أن هذا النظام الذي سنبنيه ينبغي له العيش لسنوات عديدة، لذلك، فأنت تحتاج لمراقبته ودعمه ومعالجة أمراضه! وتحتاج إلى آلية واضحة تحدد كيف سيتم ذلك أو من أين سنحصل على السجلات الخاصة بهذا النظام لنتمكن من علاجه زو استكشاف مشاكله...إلخ، فإذا لم تفعل ذلك فقد رميت نظامك في ثقب أسود، ثم تمنيت له التوفيق ^^، وحتى أقرب لك الصورة، هل عملت من قبل على موقع ما احتجت فيه لتتبع مشكلة معينة ولم تعلم أين توجد ملفات ال log أصلا، وقضيت وقتا طويلا في البحث عن معلومة بسيطة كهذه؟ إن الزمن يتغير، وأعضاء الفريق يتغيرون، وتوثيق مثل هذه المعلومات يمنع الوقوع في موقف مثل هذا.

إن هذا القسم من الدليل يجب أن يجب عن أسئلة مثل:

- كيف نراقب حالة النظام -Monitoring-؟
- أين تُحفظ ملفات السجلات -Logs- وكيف نقرؤها لتشخيص المشاكل؟
- هل تغيير الإعدادات -Configuration- يتطلب إعادة تشغيل النظام أم سيتم تطبيقها فورا؟
- هل هناك مهام تنظيف يدوية يجب القيام بها دوريا، مثل تفريغ مساحات التخزين المؤقتة أو أرشفة البيانات القديمة؟

أمثال هذه الأسئلة الجميلة يمكن الإجابة عنها من خلال أسلوب سردي، فلا داعي لرسم مخططات كالأقسام السابقة :P، بل يكفي أن تضع عناوين رئيسية واضحة مثل: التشخيص، والإعدادات...إلخ، واكتب تحت كل عنوان فقرة تشرح الآلية المتبعة...
إن مثل هذا القسم من التوثيق = ستشكر صاحبه عليه إذا رن ناقوس الخطر الساعة الثالثة صباحا على خطأ مفزع في النظام، وليس لديك تصور كيف ستقرأ الأخطاء أو أين مكانها ^^... والجمهور المستهدف لهذا الجزء من التوثيق هم الفريق التقني والدعم الفني وال devops.

## الفصل الثالث والعشرون: القسم الثاني العشر من الدليل = Development Environment

هذا القسم من الأقسام الجميلة في أي دليل سياحي، فهي تنقذ المطورين الجدد من الأسئلة الكثيرة والمشاكل العبيطة! والآن سأسألك: كم مرة انضممت إلى فريق جديد، وقضيت أسبوعك الأول بأكمله تائها تسأل زملائك: "من أين أحمل قاعدة البيانات؟ وما هي كلمة المرور لبيئة الاختبار؟ ومن هم المستخدمين الافتراضيين لأستطيع تجربة النظام؟ ولماذا لا يعمل النظام على جهازي؟" كل هذه الأسئلة جاء ليقضي عليها هذا القسم وليخلصنا من هذه المأساة...

بناء على ما ذكرناه، يكون هدف هذا القسم من الدليل أخذ المبرمج الجديد من نقطة الصفر -جهاز حاسوب فارغ بنظام تشغيل جديد- إلى نقطة العمل -بيئة تطوير متكاملة جاهزة لكتابة الشيفرة البرمجية وتشغيل النظام-، أما آلية تمثيل هذا القسم في الدليل فببساطة من خلال استخدام قائمة المهام -CheckList- مثل:

- إصدارات البرامج المطلوبة مثل: NodeJs v24+
- روابط تحميل الأدوات والبرامج التي يحتاجها هذا المطور لتشغيل المشروع.
- ال Environment Variables التي يجب أن يضيفها هذا المطور لبيئته حتى يعمل النظام.
- ال IDE Configuration

- نصوص تجهيز قاعدة البيانات لملئها ببيانات وهمية للتجارب أو الأوامر الخاصة بذلك -Database population scripts-...
- أسماء المستخدمين وكلمات المرور لبيئة التطوير
- ملاحظة: حتى لو كنت تستخدم أدوات حديثة لأتمتة بيئة التطوير مثل Docker، لا تفترض أن المبرمج الجديد سيعرف ما يفعله فورا، يجب أن تكتب هنا الأوامر البرمجية صراحة، مثلاً: اكتب docker-compose up في هذا المجلد لتعمل قواعد البيانات.. أعطه تسلسل الأوامر لا تجعله يتعذب!

أما الجمهور المستهدف فهم الفريق التقني، وتحديدا الأعضاء الجدد، وهو إلزامي بكل تأكيد.

## الفصل الرابع والعشرون: القسم الثالث عشر من الدليل والأخير = Decision Log

نصل الآن إلى مسك الختام، والقسم الثالث عشر والأخير من الدليل السياحي، وهو من أهم الأقسام على الإطلاق في المعماريات الحديثة: سجل القرارات -Decision Log-، والذي يُعرف اليوم في كبرى الشركات باسم -ADR - Architecture Decision Records-، هذا القسم الذي يمثل الصندوق الأسود الذي يحفظ أسرار المشروع وتاريخه...

لقد تحدثنا عن ال ADR من قبل في كتاب ال Micro Frontend -الصفحة ٣٦٧-، وسننقل التعريف كما هو: "ال ADR هي اختصار ل: Architectural Decision Records، وهي وثيقة تستخدم لتوثيق القرارات الهندسية المهمة، وهي ليست مجرد سجل

للقرارات، بل تمثل snapshot يمكن استخدامها لتقديم السياق الكامل وراء اتخاذنا لقرار معماري معين على حساب آخر، باختصار هي آلية الهدف منها مساعدة المطورين الحاليين والمستقبليين على الحصول على إجابة السؤال التالي: **لماذا تم اختيار هذا النهج المعماري على حساب غيره ؟!"**

إن المشكلة الكبيرة التي لدينا هي أن المطورين آخر ما يفكرون فيه هو إضافة سجل بالقرارات التي اتخذت أثناء تطوير النظام! وهذا يجعلنا نضيع ونرتكب أخطاء وتجارب سابقة أثبتت عدم جدواها للمشروع، فلماذا اخترنا التقنية A ولم نختار التقنية B أو رفضناها؟
من دون سجل اتخاذ القرارات -ADR- لن نعرف الإجابة!
ومن الأمثلة على ذلك:

- لماذا قمنا باختيار التقنية X بدل من Y و Z
- ولماذا اخترنا المعمارية البرمجية الحالية، وما الخيارات الأخرى التي كانت موجودة أو تجاهلنا ورفضناها...
- كيف تأكدنا أن هذا القرار يحقق سمات الجودة
- لماذا اخترنا ال React بدلا من ال Angular أو ال Vue؟
- استخدمنا ال Strangler pattern للانتقال من المشروع القديم للجديد لأننا لا نستطيع إيقاف التطوير على القديم ونريد بناء النظام الجديد...

حسنا، كيف يمكننا توثيق هذه القرارات؟ بكل بساطة من خلال استخدام قالب ال ADR الشهير الذي يوضح السياق والقرار والنتيجة، ويمكن إضافة أي روابط مفيدة مثل

اختبارات الأداء للمقارنة بين نقطتين...إلخ، كما يمكنك كتابة فقرة قصيرة لكل قرار، اختر ما يناسبك!

إن كل قرار اتخذناه هو عبارة عن تنازل أو مقايضة بناء على ظرف زماني ومكاني، هذا الظرف قد يتغير بمرور الوقت، وقد يتغير السياق تماما، وهذا التوثيق سيحميك من اتهامات الغباء! وسيحمي الشركة من الوقوع أو الرجوع لمشاكل سابقة تم حلها وتجاوزها...

إن المستفيد من هذا القسم من التوثيق هم الفريق التقني وأي شخص سيقوم بصيانة النظام، وهو ليس إلزاميا، لكن يفضل أن تقوم بتوثيق التقنيات أو القرارات التي قضى فيها الفريق وقتا في التفكير والمفاضلة، كما أن هذا القسم يعد مهم لدرجة الخطر إذا كان هذا النظام مبني لعميل خارجي، لأنه يوثق أيضا موافقة العميل أو أوامره على الخيارات التقنية...

# فائدة

اعلم أنّ أنفع المحبة على الإطلاق وأوجَبها وأعلاها وأجلّها محبةُ مَن جُبلت القلوب على محبته، وفطرت الخليقة على تألّهه. وبها قامت الأرضَ والسماوات، وعليها فُطِرت المخلوقات. وهي سرّ شهادة أن لا إله إلا الله، فإنّ "الإله" هو الذي تألَهه القلوبُ بالمحبة والإجلال والتعظيم والذلّ والخضوع، وتعبدُه. والعبادة لا تصحّ إلا له وحده، و"العبادة" هي كمال الحبّ مع كمال الخضوع والذلّ...

- كتاب الداء والدواء، ابن القيم -رحمه الله تعالى-

# الوحدة الثالثة: الأدوات

في هذه الوحدة سننتقل الآن من مرحلة "ماذا نكتب؟" إلى مرحلة "**كيف نرسم؟**"، وذلك بعد أن أنهينا الدليل السياحي، وهنا يفتح Simon محورا جديدا يناقش فيه دورة حياة المخططات المعمارية: من مجرد شخبطة على اللوح، إلى مخططات احترافية ^^... وهنا سنتناول كيفية استخدام الأدوات لتصوير المعمارية البرمجية وتوثيقها واستكشافها.

# الفصل الخامس والعشرون: المسودات الأولية، والمخططات، والنماذج، والأدوات -Sketches, diagrams, models and tooling-

## المسودات الأولية / الرسومات اليدوية -Sketches-

إذا كنت تصمم نظاماً جديداً أو توثق نظاماً قديماً، فإن البداية المنطقية ليست بفتح أدوات الرسم المعقدة، بل بالوقوف أمام اللوح -Whiteboard- أو من خلال استخدام الورقة والقلم، وهذه لها ميزة كبيرة أشرنا لها سابقا، فاللوح يعد من أفضل الأماكن للتعاون والتفكير مع الفريق، والأداة كذلك بسيطة، مجرد مساحة بيضاء وأقلام ملونة ^^، لكن مع ذلك تكمن الخطورة هنا من أن تتحول هذه العفوية أو البساطة إلى مربعات وخطوط مبهمة، وهنا سنعود مجددا للفخاخ التي حذرنا منها سابقا في بداية الكتاب! لذلك: قبل أن تمسك القلم = اتفق مع الفريق على لغة مشتركة، مثلاً: سنرسم الآن مخطط حاويات C4، والمربع الأزرق

يمثل نظاماً، والأسطوانة تمثل قاعدة بيانات...إلخ، وكُن دقيقاً في المعنى، ولا تقلق بشأن جمالية الرسمة -Fidelity- لأن عمرها سيكون قصيرا.

## المخططات -Diagrams- ونفخ الصور الفوتوغرافية

اللوح مكان رائع للنقاش، لكنه لا يصلح ليكون مرجع تاريخي، ولا يمكن إرفاق اللوح مع مقترح مالي للعميل، أو استخدامها لأغراض أرشيفية أو لإدراجها في مواصفات تقنية ما...
وهنا تبرز الحاجة لتحويل هذه الشخابيط إلى مخطط رسمي، وسيخطر على بالك الآن الحل البديهي والسريع، وهو: "ببساطة سأقوم بالتقاط صورة لهذا اللوح من الهاتف واستخدامها"، وهذا الحل فعلا سهل وسريع ومقبول في حالة واحدة = وهي الاحتفاظ السريع بالفكرة ^^، أما ما دون ذلك فلا يمكنك الاعتماد عليها! والسبب في ذلك يعود للكثير من الأسباب أهمها اكتشاف وجود خلل بالمخطط أو احتجت لإضافة عنصر جديد عليه، فماذا ستفعل؟ ثم من عيوبها المهمة أنها تفتقر للمظهر الاحترافي التي تحتاجه الوثائق الرسمية.
حسنا، ماذا عن استخدام أدوات الرسم؟ ماذا عن استخدام أدوات مثل ال draw.io أو LucidChart ونحوها؟
إن هذا أحد الأنفاخ الشهيرة التي يقع بها الكثير من المطورين، فبمجرد انتقالهم من الورقة والقلم أو اللوح يذهبون مباشرة لجهاز الحاسوب ويستخدمون أدوات الرسم التقليدية، لكنها بالحقيقة تخفي خلفها كوابيس مخيفة ^^ وهي:

- نفخ التجميل وإهدار الوقت -The Pretty Trap-: في أدوات الرسم التي يستخدم فيها السحب والإفلات ستجد نفسك لا شعوريا تقضي ٥ دقائق أو ١٠ دقائق في التفكير

في المعمارية و٥٠ دقيقة في محاذاة المربعات بالبكسل وتغير درجات الألوان وتعديل شكل انحناءات الأسهم... والسبب في هذا صعوبة مقاومة الإغراء النابع من رغبة في تحسين المظهر ليعجب الآخرين ولو على حساب المعنى... مما يعني إهدارا للوقت.

- غياب الاتساق -InConsistency-: مع تزايد عدد المخططات، يصبح ضمان تسمية العناصر وتنسيقها بشكل متسق عبر جميع المخططات تحديًا متزايد الصعوبة. إذا كان لديك مخطط واحد، فالأمر سهل وبسيط، لكن الحياة الواقعية غالبا ما تتطلب العديد من المخططات المختلفة، وهنا كيف ستضمن أن "حاوية قاعدة البيانات" ستظهر بنفس اللون والحجم في جميع المخططات؟ وماذا لو تغير اسم أحد المكونات؟ الجواب ببساطة = ستضطر لفتح جميع الملفات التي تستخدم هذا المكون، والبحث عن هذا المربع في كل ملف، وتعديله، وهذا عمل يدوي مرهق وعرضة للخطأ البشري.
- ال Version Control: هذه واحدة من أهم المشاكل إن لم تكن الكارثة الكبرى لاستخدام أدوات الرسم التقليدية مثل التي بالأعلى! لأن ملفات أدوات الرسم غالبا ما تكون عبارة عن ملفات Binary مثل .vsdx أو تنسيقات XML معقدة جدا، لذلك، تخيل معي أن ترفع هذا الملف على Git، وقام زميلك بتعديل المخطط، هنا لا يمكنك استخدام أمر بسيط مثل git diff لتعرف ما الذي تغير بالتحديد ^^، فهل أضاف سهما جديدا؟ أم هل حذف مربعا؟ والأسوأ من هذا كله: كيف ستحل ال Merge Conflicts بين ال Develop Branch وال HotFix Branch إذا حصل بين هذه ال Branches والمخطط عبارة عن ملف ثنائي مغلق؟! بالتأكيد = لا توجد طريقة منهجية لدمج التعديلات بأمان في هذه الحالة...

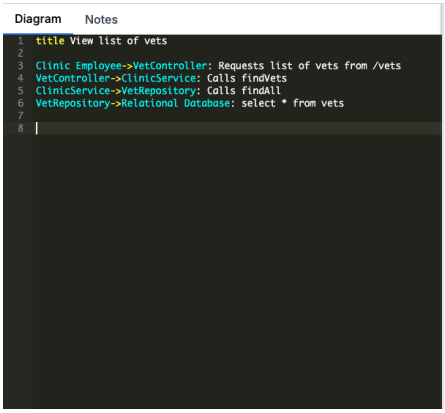
حسنا، الآن ستقول: "أكلت راسنا، وكل ما نحكيلك حل بتحكيلنا هذا نفّ... شو الحل؟؟؟؟"، على رسلك يا صديقي، لنعيد صياغة السؤال بطريقة جذابة ^^:
كيف سنرسم مخططاتنا بطريقة تضمن الاتساق، وتدعم Git بشكل أصيل، وتغنينا عن تضييع الوقت في تلوين المربعات؟
والإجابة ببساطة هي في الانتقال من عقلية الرسم إلى عقلية البرمجة، وهو ما يطلق عليه ال: Diagrams as Code، وهنا يأتي دور الأدوات السحرية التي ستحل كل هذه المشاكل مثل: ال PlantUML أو ال WebSequenceDiagrams -وغيرها مما يشبه هذه الأدوات-!

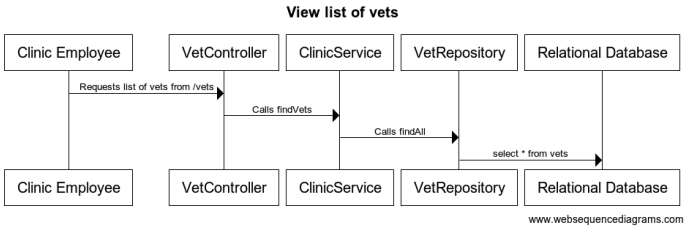
## أدوات المخططات النصية -Text-based diagramming tools-

إننا كمبرمجين عادة ما نفضل كتابة مئة سطر برمجي على أن نستخدم الفأرة لرسم ومحاذاة مربعين ^^، لذلك أدوات مثل WebSequenceDiagrams ستكون مفيدة لنا جدا عند كتابة نص بداخلها لتقوم هذه الأداة بتحويل هذا النص إلى رسمة مولدة بشكل تلقائي! وأكبر انتصار عند استخدام هذا الأسلوب هو حل مشكلة ال Version Control ^^، فال git diff ستكون كافية لمعرفة التغيير.. كما أن هذه الأدوات ستريح عقولنا وأعيننا عن البحث عن المكان المناسب لرسم كل مربع على الشاشة، وكم المسافات التي بين هذه المربعات، فالأداة ستفعل ذلك بشكل تلقائي، وهي غالبا ما تكون ذات جمال أكبر بكثير من الجهد النابع من المبرمج :P...
لكن، مع كل هذا الجمال، هناك صدمة كبيرة لك، فهذه الأدوات لن تحل مشكلة غياب الاتساق! تخيل أن لديك خمسة ملفات PlantUML مختلفة، وجميعها فيها مربع يمثل ال

User، إذا قررت الإدارة تغيير ال User إلى Customer فماذا ستفعل؟ ستضر لفتح الملفات الخمسة و تعديلها يدويا... مع أن الموضوع هنا أسهل، لكن مشكلة الاتساق لم تحل بالكامل...

شاهد هذا المثال -الذي عرضناه سابقا- عند حديثنا عن العيادة البيطرية، وكيف قمنا برسمه بهذا الشكل بسهولة من خلال أداة WebSequenceDiagrams:




## النماذج -Models-

عليك أن تعلم يا صديقي أن هناك فرقا كبيرا يميز بين من يفكر بعقلية "الرسام" وبين من يفكر بعقلية "النمذجة"!

إن المخططات ليست سوى مجرد مخططات غبية! وأقصد بهذا أنها بالنهاية رسومات أو صور ثابتة لا حياة فيها، وظيفتها إيصال الفكرة البصرية دون أن تدرك ما فيها من مؤثرات، فالأسطوانة لا تعني قاعدة البيانات، بل تعني مجرد شكل هندسي تم رسمه من خلال مجموعة

من ال Pixels، لهذا لا يمكنك توجيه أسئلة وعمل استعلام عليها، والمسؤولية وضمان الاتساق تقع بالكامل على المعماري، لذلك جاءت النماذج لحل هذه المشكلة ^^

تمثل النماذج مصدر الحقيقة الموحد لأي مخطط سيتم رسمه، وبذلك فالنموذج هنا ليس رسمة؛ بل قاعدة بيانات أو تعريف برمجي لنظامك... وهنا بدلا أن تبدأ برسم المربعات تبدأ بتعريف الكائنات برمجيا في مكان مركزي واحد، وبمجرد الانتهاء من هذه العملية، يمكنك أن تطلب منه توليد أي عدد من المخططات -الرسومات- التي تحتاجها ^^... وبهذا إذا تغير اسم قاعدة البيانات = فأنت لن تفتح المخططات لتعديلها! بل ستذهب إلى النموذج المركزي لتعديل الاسم هناك ولمرة واحدة، وستتحدث جميع المخططات التي تعتمد على هذه القاعدة تلقائيا! ولأن النموذج يمكن قراءته -Machine-readable-، فيمكنك توجيه أسئلة برمجية له مثل: "أعطني قائمة بكل المكونات التي تتصل ببوابة الدفع؟"، وسيعطيك الإجابة فورا، وهو أمر مستحيل فعله مع الصور المجردة...

ومن هنا ننتقل إلى:

### أدوات النمذجة -Modeling tools-

الآن بعد أن عرفنا أن النمذجة هي الحل السحري لمشكلة الاتساق، سندرك بكل تأكيد أن السوق مليء بأدوات تقوم بهذه المهمة مثل ال Enterprise Architect و StarUML...

كيف تعمل هذه الأدوات؟

كما قلنا سابقا، أنت لا ترسم مربعات فيها، بل تُعرف مجموعة من الكيانات، وذلك من خلال الذهاب لقاعدة بيانات الأداة ونقول مثلا: لدي نظام اسمه X، ثم تسحب هذا النظام لأي مخطط تريده، وأي تغيير في القاعدة المركزية، سيحدث تغييرا فوراً في كل المخططات!
لكن، مع هذه المزايا الجميلة، يكره المبرمجون هذه الأدوات ^^، والسبب في ذلك يعود للتعقيد والتقييد، فهذه الأدوات متعصبة لمعايير ال UML المعقدة، وإذا أردت استخدام بساطة نموذج C4، ستشعر وكأنك تقود دبابة للذهاب إلى البقالة! كما أن بناء النموذج يتطلب ساعات من النقر وتعبئة النوافذ المنبثقة المملة، فهربنا من التصميم فوقعنا في إدخال البيانات -Data Entry-، وهذا الفخ المزعج يمكن تجاوزه إذا كنت تملك شيفرة برمجية جاهزة من خلال الهندسة عكسية.

حسنا، ما الذي تريدونه أيها المبرمجين المدللين؟
نحن نريد قوة النمذجة، ولكننا نريدها كشيفرة برمجية سهلة وسريعة، ومتوافقة مع نموذج C4، وبعيداً عن تعقيد أدوات ال UML التقليدية! هل تقصد Structurizr؟ ^^

**ملاحظة حول الهندسة العكسية**: هناك رغبة جامحة لدى المبرمجين دوما في أتمتة التوثيق المعماري لأي برمجية نعمل عليها باستخدام أدوات التحليل مثل NDepend وStructure101 وغيرها، وغالبا ما يتعرض هؤلاء المبرمجين إلى صدمة عاطفية بعد استخدامهم لهذه الأدوات لأنها تنتج مخططات معقدة وغير مفيدة للمعمارية -شبكة عنكبوتية-، فهي تقوم بتحليل الشيفرة البرمجية نفسها بدلا من أن تفهم المعمارية والمكونات ذات المعنى، وهذه هي مشكلة فجوة النموذج والشيفرة البرمجية -Model-Code Gap-:

فالأداة ترى جميع الفئات والعلاقات بينها تقريبا على المستوى نفسه، ولا تميز بين مكوّن معماري مهم مثل Controller رئيسي وفئة صغيرة تؤدي عملية بسيطة، وبالتالي يصبح المخطط أشبه بشبكة عنكبوتية بدل من معمارية واضحة وصحيحة، وهنا يطرح Simon نقطة جميلة جدا: «نحن كمبرمجين نؤمن بأن الشيفرة البرمجية هي مصدر الحقيقة، فإذا كان الأمر كذلك، فلماذا تفشل هذه الأدوات في استخراج المعمارية من الشيفرة البرمجية؟؟ الإجابة ببساطة: لأن المعمارية في مستوياتها العليا (السياق، والحاويات) لا تُكتب داخل الشيفرة البرمجية أصلا!»، وهذا لا يعني أننا نحارب استخدام مثل هذه الأدوات، بل على العكس قد تكون مفيدة في كثير من الحالات، حتى عند حاجتنا لرسم المعمارية! لكن علينا أن نعلم أن المستوى الأول من نموذج C4 مثلا سيكون من الصعب استخلاصه، والمستوى الثاني قد ننجح جزئيا فيه عبر قراءة ملفات مثل ال Docker، وهذه العملية مفيدة جدا اذا كان لدي 100 microserveice على سبيل المثال! أما المستوى الثالث فهو ما نريد أتمتمه حقا، لكنه هو نفسه من ينتج لنا الشبكة العنكبوتية، لذلك وجب الانضباط ووضع قواعد واضحة حتى تتمكن هذه الأدوات من رؤية المكونات المعمارية بطريقة أفضل.

الخلاصة: لكي تنجح الأدوات في رسم المخططات تلقائيا، يجب أن تتفق مع فريقك على ترك علامات ودلائل واضحة في الشيفرة البرمجية، مثل: استخدم ال Annotations مثل @Service، أو من خلال توحيد طريقة التسمية مثل أن تنتهي بـ *Controller... بهذا الانضباط = ستتمكن الأدوات من رؤية المكونات المعمارية بشكل أفضل بدلا من الغرق في بحر من سطور الشيفرة البرمجية العادية.

إن الوضع المثالي الذي نحلم فيه هو استخدام أداة نقرأ بها الشيفرة البرمجية فتولد لنا المعمارية بالكامل! لكن Simon يحسمها هنا، فيقول: هذا غير واقعي حاليا، والسبب في ذلك يعود لأن الشيفرة البرمجية لا تحتوي على القصة الكاملة، فالشيفرة البرمجية لا تعرف لماذا اخترت هذه التقنية، ولا تعرف شكل الأنظمة الخارجية التي تتحدث معها... والحل العملي لهذه المشكلة هي الإثراء -Enrichment-، فبما أننا لا نستطيع استخراج 100% من المعمارية من الشيفرة البرمجية، فالخطوة الصحيحة هي = استخراج ما يمكننا استخراجه آليا من الشيفرة البرمجية، ثم إثراء هذا النموذج يدويا بالمعلومات التي نعجز الشيفرة البرمجية عن البوح بها.

## لغات وصف المعمارية -Architecture Description Languages -ADLs

بناء على ما تحدثنا به في الفقرة السابقة، حاول الأكاديميين وعلماء البرمجيات حل هذه المشكلة بابتكار ال (Architecture Description Languages - ADLs)، والفكرة هنا جميلة، فبدلاً من رسم المخططات نقوم "بكتابة" وصف دقيق للنظام باستخدام لغة نصية مخصصة لذلك. وهي أشبه ما يكون بكتابة شيفرة برمجية لكن لوصف المعمارية، ولأنه نص فيمكن تتبعه عبر ال Git، ويمكن مقارنة نسختين منه عبر ال Diff، والأهم أنه يبني نموذجا مركزيا يمكننا توليد المخططات منه لاحقا، وهذا يبدو حلا مثاليا، لكنه ليس كذلك! فمع أن الفكرة كانت جذابة جدا للمبرمجين، إلا أنك بالكاد ترى أحدا يستخدمها في الواقع العملي، والسبب يعود في ذلك لعدة نقاط وهي:

- ضغط الميزانية والوقت: فالشركات تريد برمجيات تعمل، ولا تريد دفع أموال للمطورين لكتابة وصف نظري طويل للنظام، فالوقت والمال هنا حاكم أساسي لدفع المطورين لعدم استخدام هذه الأدوات أو تجاهلها في مراحل المشروع... مما سيقتل المعمارية.
- الانفصال عن الواقع: هذه اللغات منفصلة تماما عن الشيفرة البرمجية المصدرية الحقيقية، فأنت تكتب المعمارية في ملف، وتكتب الشيفرة البرمجية في ملفات أخرى، وسرعان ما سيحدث انفصام بينهما ويصبح الوصف المعماري كاذبا ولا يمثل الشيفرة البرمجية!
- ال Agile Manifesto ينتصر: سنعود لعالم ال agile وما تحدثنا عنه في الجزء الأول من الكتاب، والخلاصة ستكون من قبل المطورين: "برمجيات صالحة للاستعمال أهم من توثيق شامل"، وهذا مبدأ صحيح في حالتنا، تسليم برمجيات تعمل بكفاءة بشكل فعلي أهم بكثير من كتابة مستندات وتوثيقات مطولة ومفصلة قبل البدء في العمل، فإن ما نحتاجه هو ال Just enough من التوثيق... بينما انطلق ال ADLs من محاولة صارمة لوصف هيكلية الأنظمة برمجيا وبتفاصيل دقيقة، باختصار: سقطت ال ADLs لأن عالم البرمجيات أدرك أن المعمارية الحقيقية تعيش في الشيفرة البرمجية التي تعمل، لا في النماذج النظرية المنفصلة عن الواقع.

### ال Structurizr

تمثل أداة ال Structurizr نقلة نوعية من عقلية الرسم اليدوي أو النصي إلى تبني مفهوم "المعمارية كشيفرة برمجية -Architecture as Code-" بشكل حقيقي، وتعمل الأداة من خلال اعتمادها على مكتبة برمجية تسمح للمعماري ببناء نموذج مركزي متكامل باستخدام

لغات البرمجة مثل Java و .NET، ثم تصدير هذا النموذج برمجيا إلى منصة سحابية تحوله إلى مخططات بصرية تفاعلية، وهذا النهج يضمن أن التوثيق المعماري ليس مجرد صور ثابتة، بل قاعدة بيانات حية ومقروءة آليا تعكس واقع النظام.

،لردم الفجوة بين الشيفرة البرمجية والمعمارية، تتبنى هذه الأداة نهجا هجينا، فبما أن الشيفرة البرمجية المصدرية تجهل بطبيعتها الأنظمة الخارجية وهويات المستخدمين، سيقوم المعماري بتعريف المستويات العليا لمعمارية C4 (المستوى الأول والثاني - السياق والحاويات) برمجيا وبشكل يدوي، أما في المستويات العميقة، فستتولى الأداة هذه المهمة عبر مكتشف المكونات -Component Finder- الذي يغوص في الشيفرة البرمجية ويبحث عن العلامات المعمارية مثل ال Annotations التي تمت كتباتها، ليستخرج من خلالها المكونات وعلاقتها واعتماديتها آلياً بالكامل دون تدخل بشري.

من خلال هذا النموذج المركزي ستتغير طريقة إنشاء المخططات لتصبح أشبه بتحريك عدسة الكاميرا نحو النظام، فالمعماري يطلب برمجيا عرض حاويات أو مكونات معينة، فتقوم الأداة بجلبها ورسم روابطها تلقائيا بناء على ما يمتلكه النموذج من بيانات، وهذه المرونة تتيح استبعاد المكونات المزعجة بصرياً وتوليد مخططات مصغرة ومخصصة، والأهم من ذلك، برفع الملف عبر الـ API، يُدمج التوثيق المعماري ضمن مسارات النشر المستمر (CI/CD)، لتتحدث المخططات المعمارية تلقائياً مع كل تعديل يطرأ على الشيفرة البرمجية، ويمكنك زيارة موقع https://structurizr.com/ لمزيد من التفاصيل، كما يمكنك الاطلاع على المثال للنمذجة

التي حصلت للنظام البنك من خلال هذا الرابط:

https://structurizr.com/share/1/diagrams#SystemLandscape

ملاحظة: على الرغم من هذه الأتمتة الجميلة، تترك هذه الأداة لمسة الترتيب البصري للمعماري؛ حتى يضبط طريقة عرض قصة النظام بوضوح، بحيث تعجز الخوارزميات عن ترتيب المربعات هندسيا بشكل منطقي مريح للعين! وهذا هو اختيار Simon في أداته! كما أن الأداة تعمل كجهاز كشف الكذب الذي يفضح فوضى الشيفرة البرمجية، فإذا اضطر الفريق لكتابة قواعد معقدة ومليئة بالاستثناءات لاستخراج المكونات، فهذا دليل قاطع على غياب الاتساق المعماري داخل الشيفرة البرمجية، وخلاصة هذا الكلام أن هذه الأداة تهدف إلى فرض أسلوب برمجة يبرز المعمارية بوضوح، مما يجعل استخدام أدوات الرسم اليدوي التقليدية مجرد ذكرى من الماضي [٨].

ملاحظة ٢: بالنسبة لكيفية استخدام الموقع وكتابة الشيفرات الخاصة بإنشاء النماذج، يمكنك تعلمها لوحدك من خلال الموقع أو أدوات الذكاء الاصطناعي، فالموضوع سهل لكونك مبرمج أصلا.

# الخاتمة

وصلنا الآن إلى نهاية الجزء الثاني من هذه الرحلة، والذي بدأت بمحاولة للإجابة عن أسئلة قد تبدو مملة للوهلة الأولى! ثم ما لبثنا حتى تعمقنا شيئاً فشيئاً، لنجد أنفسنا نبني دليلا سياحيا متكاملا نغوص فيه، ولو أردنا أن نُلخص ما تعلمناه في كلمات قليلة لقلنا: إن التوثيق المعماري ليس مجرد ملفات نصية يكسوها الغبار، ولا مخططات صامتة نرسمها لنرضي أصحاب العمل! بل هو نموذج حي يتنفس مع النظام، وهو القدرة على الانتقال من عقلية "الرسم" إلى عقلية "النمذجة"، وردم تلك الفجوة المرعبة بين ما ندّعي أننا صممناه، وبين ما تحتويه الشيفرة البرمجية بالفعل...

إن الذي يجب أن يبقى معنا بعد إغلاق صفحات هذا الكتاب، ليس الهوس بأداة معينة مثل ال Structurizr أو ال PlantUML! بل هو هذا المبدأ: "هل تعكس مخططاتي ما كتب في الشيفرة البرمجية حقا؟ وهل تروي القصة التقنية الصحيحة لمن يقرؤها؟"، فإن تمكنت من الإجابة على هذا السؤال بثقة، 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**