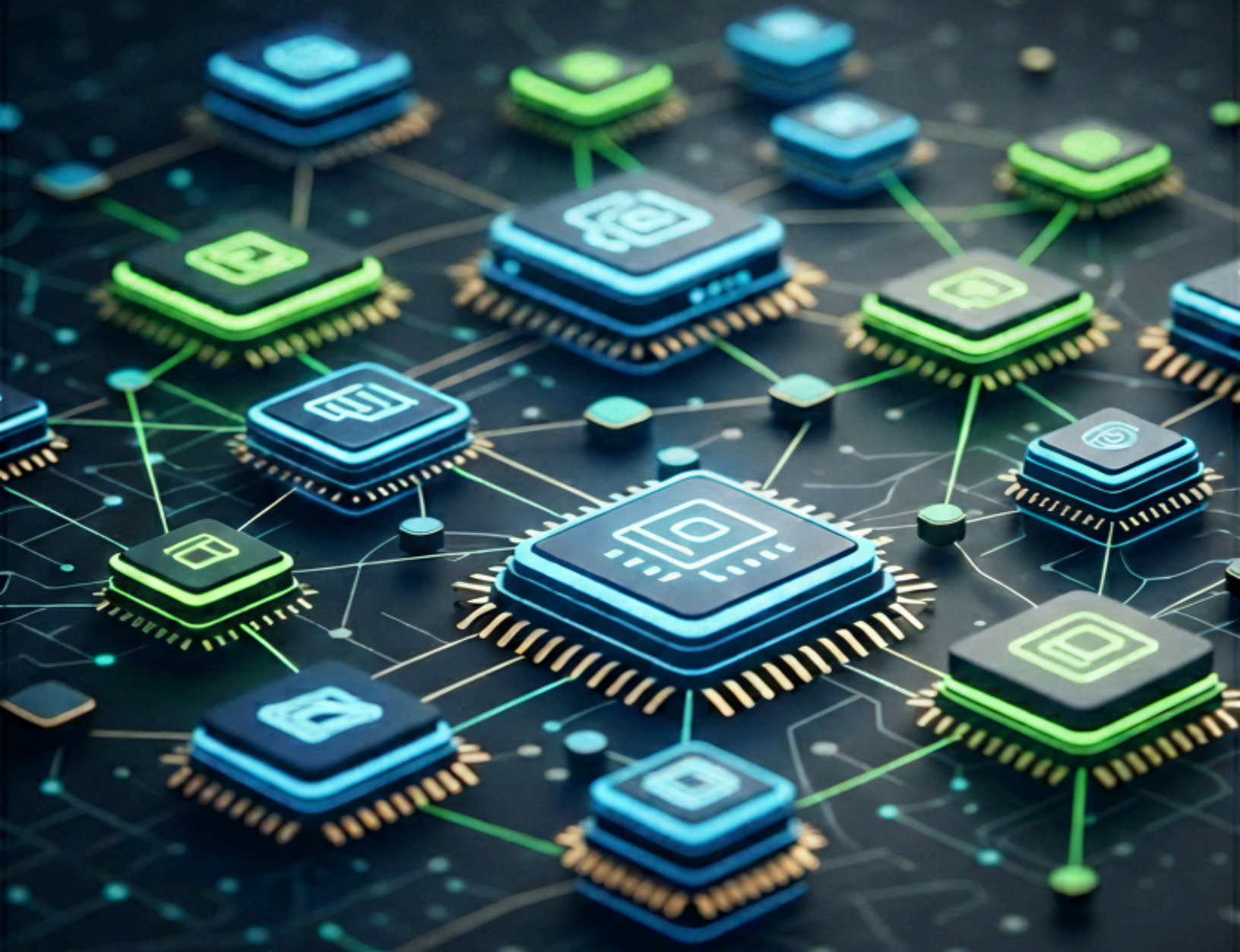
MICRO FRONTEND

Building Scalable and Maintainable Web Applications



By Anees Hikmat

بسم الله الرحمن الرحيم

Micro-Frontend

شرح مفصل لتحويل أو بناء مشاريع الواجهات الأمامية من خلال معمارية تقوم على فصل مكونات المشروع لأجزاء متعددة اعتمادا على مجالات محددة.

تأليف: أنيس حكمت أبوحميد

الموقع الإلكتروني: 2nees.com

إهداء

إلى أبطال غزة الصامدين، الذين أزالوا زيف هذا العالم عن أعيننا...
وإلى شهدائنا الأبطال؛ الذين سُفكت دمائهم لأجل لا إله إلا الله...
وإلى المجاهدين والمرابطين؛ الذين نكلوا في عدونا وعدوهم...
وإلى أمهاتنا وبناتنا وأخواتنا في غزة؛ اللواتي قدمن من معاني العزة والكرامة والعفة والصبر والتضحية ما نعجز عنه...

هذا الكتاب إهداء لكم؛ تقديراً لشجاعتكم وعزتكم وصبركم وإخائكم...
هذا الكتاب لكم؛ لأنكم كنتم الأحرار حقا، حين كنا نحسبكم أسرى!
هذا الكتاب لأننا نحبكم، ولأننا منكم...
هذا الكتاب لكم؛ لأننا نرى أن النور قادم لا محالة، وأن النصر والعز لنا مهما طال الزمان...
فهذا وعد صاحب الجلال والإكرام!

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.

في عالمنا الحديث تتغير احتياجات المشاريع الرقمية بسرعة، فأصبح بناء وتطوير واجهات المستخدم الكبيرة والمعقدة تحديًا مستمرًا، ومع تزايد حجم التطبيقات وتعدد فرق التطوير، ظهر مفهوم الـ Micro Frontend كنهج مبتكر يهدف إلى تفكيك التطبيقات الأمامية (FE) الكبيرة إلى وحدات صغيرة مستقلة يمكن تطويرها واختبارها ونشرها بشكل منفصل، وكما ساعدت ال Microservices في تنظيم خدمات الـ Backend، يوفر ال Micro Frontend القدرة على توزيع مسؤوليات واجهة المستخدم بين فرق مختلفة دون التضحية بالتماسك العام للتطبيق، وهذا النهج لا يقتصر على تحسين الإنتاجية فحسب، بل يعزز القابلية للصيانة ويقلص المخاطر المحتملة، ويتيح استخدام تقنيات متعددة ضمن نفس المشروع بسلاسة.

في هذا الكتاب، سنستكشف الأسس النظرية لمفهوم الـ Micro Frontend، وسنناقش الفوائد والتحديات لهذا الأسلوب، وسنحاول تقديم أفضل الممارسات في هذه المعمارية، وسنستعرض بعض الأمثلة العملية لبناء تطبيقات واجهة مستخدم مرنة وقابلة للتوسع.

إن الهدف من هذا الكتاب هو تزويد المطورين بفهم شامل يمكنهم من اعتماد هذا النمط من العمل واستعماله بشكل فعّال، وتحقيق أقصى استفادة ممكنة منه في مشاريعهم، لذلك، فهذا الكتاب ليس مجرد دليل تقني؛ بل هو خارطة طريق لتطبيق فلسفة جديدة يمكن أن تُحدث ثورة حقيقة في طريقة العمل وكفاءته...

والآن، لنبدأ على بركة الله -جل في علاه- في هذا العمل المتواضع، سائلين المولى -عز وجل- أن يوفقنا لما يحبه ويرضاه، وأن يبارك لنا فيه، وأن يعيننا على إتمامه على خير وبركه... ونسأله -سبحانه وتعالى- أن يغفر لنا تقصيرنا وخطأنا، فما نحن إلا بشر، واللهُ المُستعان وعليه التُّكلان.

ملاحظة: المرجع الأساسي لهذه السلسلة كتاب Building Micro-Frontends ل Luca Mezzalira، لكن ليس بالضرورة التقيد بالتفاصيل كما هي، لكن سنحاول الحفاظ على التسلسل الخاص بالكتاب والمؤلف قدر الإمكان، والله ولي التوفيق.

فائدة

ومن أهم ما يصنعه استحضار لقاء الله في النفوس الزهد في الفضول، فضول النظر، وفضول السماع، وفضول الكلام، وفضول الخلطة وفضول النوم، وفضول تصفح الإنترنت، ونحوها، فيصبح المرء لا ينفق نظره وسمعه ووقته إلا بحسب الحاجة فقط....

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

الفهرس

الفصل الأول: ما قبل ال Micro-Fronted حتى ال Micro

قبل الخوض في غمار هذا الشرح وهذه الفصول وقبل الحديث عن أي جزئية، أريدك أن تجعل هذه العبارة في مخيلتك: "لا تستخدم المدفع الرشاش لتقتل نملة"!

هذه العبارة نفسها مقدمة مفيدة ومهمة لأي مطور برمجيات حتى يضع نصب عينه ما يحتاجه فعلا من تقنيات وبنية تركيبية للمشروع ونحو ذلك، وألا يستخدم الرشاش بكل ما فيه ليقتل نملة صغيرة! ومن هنا ننتقل للفقرة الأولى في هذا الكتاب، وهي ولادة ال micro-frontend.

لقد ولد ال micro-front من رحم معاناة مطورين ال FE، تلك الولادة التي تم استنساخها من ال Micro-service بعد تشابه المشاكل التي دعت ال BE لبنائه، ومنها إمكانية تطبيق نفس الحلول للتخلص من نفس المشاكل، ومن أهم المشاكل التي يعانيها مطوري ال FE في هذا الباب هي:

- ازدياد حجم المشروع وبالتبعية ازدياد حجم الفرق التي تعمل عليه، وبالتالي ازدياد الصراع بين الفرق وفرد العضلات وازدياد التداخلات والاعتمادية بين المكونات بين الفرق... وهذا كله ممكن أن يكون قنبلة موقوتة داخل المشروع ويزيد من نسبة

حدوث الأخطاء أو إضاعة الوقت من خلال انتظار أحد الفرق من الانتهاء من component معينة ليعمل في نفس المكان ونحو ذلك.

- في الصفحات التي تكون مثلا SPA تعديل بسيط أو جزء بسيط من المشروع يلزم إعادة بناء كاملة لكامل مكونات المشروع.

- حجم ملفات ال css وال js عادة ما يكون كبيرا ومتداخلا فيما بينه خصوصا في المشاريع الكبيرة، وهذا يزيد من صعوبة فهم الشيفرة البرمجية أو تطويرها أو التحقق من الأخطاء فيها، كما أن ذلك يزيد من معدل الخطورة عند إجراء أي تعديل بسيط!

- محدودية استخدام التقنيات المختلفة والموجودة في عالم ال FE، فقد يحتاج فريق ما أو خبرة فريق ما في تقنية معينة أو حاجة المشروع لاستخدام خاصية معينة توفر حلولا جاهزة لمشكة معينة وتتعارض مع التقنيات المستخدمة حاليا... مما يمنع من استخدامها.

هذه المشاكل وغيرها دعتنا لاستنساخ مفهوم ال micro-service وتحويله إلى micro-frontend!

والآن قبل الخوض في تفاصيل ال micro-frontend لنأخذ بعض النماذج المستخدمة حاليا من قبل مطوري ال FE:

- ال SPA: وهي اختصار ل Single-Page Applications، ويعد هذا الأسلوب الآن واحدا من أشهر وأكثر الأساليب المستخدمة لسهولة بناءه وسرعته العالية في التجاوب مع الأحداث إذا بني بالشكل الصحيح، لكنه يصبح أكثر تعقيدا مع مرور الوقت خصوصا إذا كانت التعديلات تتم بشكل كبير، كما أن هذا النوع يشتهر بمشاكله المتعلقة بال SEO، لذلك يلجأ البعض لاستخدام بعض الحلول الأخرى.

- ال Isomorphic: وهي تلك التطبيقات التي تعمل وتبنى على ال server ثم يتم تقديمها لل client (المتصفح) حتى يتم التفاعل معه، وهذا الأسلوب مفيد جدا للمواقع التي تحتاج SEO احترافي وتحتاج إلى أداء عالي، وذلك لأن الملف المراد تشغيله بنى على ال server وتم إرساله للمتصفح، وجوجل مثلا يستطيع قراءته ولأنه صغير فسيعطي سرعة تجاوب عالية في أول render، ثم يعمل المتصفح مع الملف وكأنه SPA، لكن مشكلة هذه الأنظمة مستوى التعقيد فيها وصعوبة البناء الهندسي أو المعماري الصحيح للحصول على نتائج ممتازة.

- ال Static-Page Websites: هي تلك الصفحات الجميلة التي تعلمنا بناءها في أول حياتنا البرمجية في عالم الويب، وهي تلك الصفحات السريعة التي يمكننا استخدامها عند وجود محتوى ثابت لا يتغير كثيرا، وهي تلك الصفحات سهلة التطوير، ومشاكل هذا الأسلوب أو محدداته واضحة للجميع [٨].

هذه النماذج التي ذكرناها لكل منها حسناته وسيئاته، ولكل واحد من هذه الأساليب مكانه المميز، وحتى "لا تستخدم المدفع الرشاش لتقتل نملة"، فإن معرفة هذه التفاصيل البسيطة سيساعدك في اختيار الأسلوب المناسب لبناء مشروعك دون اتباع للموضة أو فقط اسم micro-*!

فلو كانت لدي صفحات تسويقية أو إعلانية لا تتغير عادة... فال static pages كافي لهذه المهمة وستكون ذا كفاءة وأداء ممتاز وتكلفة رخيصة! ثم تخيل أنك استخدمت ال micro-fronted لهذا الغرض؟!

والآن لننتقل للحديث عن مقدمات تخص ال Micro-Frontend ^^

لقد ذكرنا في الصفحات الأولى: "لقد ولد ال micro-front من رحم معاناة مطورين ال FE، تلك الولادة التي تم استنساخها من ال Micro-service بعد تشابه المشاكل التي دعت ال BE لبنائه"... ومن هنا نعلم أن ال micro-frontend هي معمارية ناشئة مقارنة مع غيرها، ومستلهمة من ال microservices، وهذا يعني أن نفس المميزات التي جعلت من معمارية ال microservice معمارية شائعة هي التي تدعونا لاستنساخها إلى ال FE، ومنها:

- المرونة
- القابلية للتوسعة
- القابلية للنشر المستقل

وهذ يعني إمكانية وجود فرق مستقلة أو تطوير مكونات بشكل مستقل = مما يؤدي إلى نشر مستقل، وهذا بالضرورة يقوم بتقليل الاعتمادية بين المكونات وبالتالي تحسين تجربة المستخدم والأداء.

في النظر إلى واقعنا، فعادة ما يتم بناء المشاريع ككتلة واحدة في أول الأمر حتى تتضح معالم البيزنس ومنها معالم المشروع وصولا لإصدار أول نسخة تجريبية من المشروع وإطلاقها (MVP)، هذا المشروع الكبير ذو الكتلة الواحدة يسمى معماريا ب: monolith، وهو خيار ممتاز يخدم الكثير من الحالات أو أكثر الحالات... لكن في لحظة ما أو في مشروع ما قد تصل إلى نقطة حرجة من أعداد المستخدمين وطلباتهم، وهنا تظهر الحاجة لحل المشاكل التي ذكرناها سابقا أو استنساخ المميزات السابقة، هذه النقطة الحرجة هي نقطة البداية للتحول من ال monolith إلى ال micro-service أو/و micro-frontend... ومن هنا تظهر الحاجة لأول مهارة وأهمها وهي تحديد نقاط الألم أو الضغط في المشروع، ويطلق عليها ال Pain Points، هذه النقاط قد تكون هي المتسببة في المشاكل المتعلقة بصعوبة التوسع أو صعوبة الصيانة...

مواضع الألم هذه يتم تحديدها لتحديد الأجزاء التي سيتم تقسيمها وفصلها إلى وحدات صغيرة تعمل عليها فرق محددة أو بطريقة منفصلة، ولنأخذ مثالا عمليا على هذا:

لو افترضنا أن لديك متجر إلكتروني، فإن ال Pain Point الرئيسية لديك قد تكون في هذه المكونات:

1. بطء في نظام الدخول والتسجيل
2. بطء + أداء سيء لصفحة عرض المنتجات
3. بطء + صعوبة إدارة سلة المشتريات وما فيها
4. بطء + الحاجة لوضع قيود أمان أكثر على صفحات الدفع وتأكيد الطلب

ثم نذهب للخطوة التالية وهي:

1. اختيار فرق مسؤولة عن هذه الخدمات الجديدة.
2. منح كل فريق القدرة على اتخاذ القرارات المناسبة لهذه الجزئية والتقنيات التي يحتاجها ضمن حدود معينة.

والآن، وبعد حديثنا عن نقاط الألم -الضغط- (Pain Point) ننتقل إلى مبادئ ال Microservice ثم ننظر كيف يمكننا تطبيقها في عالم ال FE ^^... شاهد الصورة التالية:

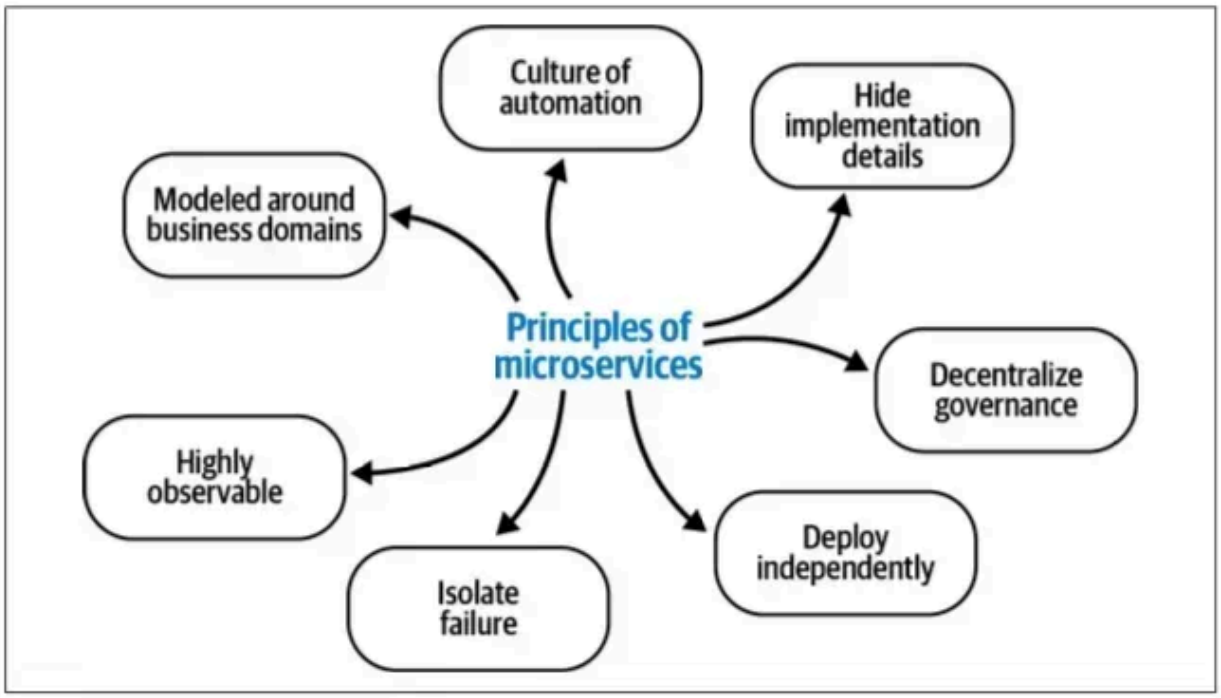


Figure 2-4. Microservices principles

١. ال Modeled Around Business Domains:

يقصد بهذا المبدأ أن تصميم النظام مبني حول المجالات (ال Domains)، والتي تمثل بدورها مفاهيم وأقسام العمل الحقيقي داخل الشركة، وال Modeled Around Business Domains أحد أهم المبادئ التي تم طرحها من قبل ال (DDD). وينطلق هذا المبدأ من افتراض أن كل برنامج يجب أن يعكس ما تقوم به المؤسسة، وأننا يجب أن نصمم معماريتنا بناء على domains وال sub domains الخاصة بنا كمؤسسة، ومستفيدين من اللغة المشتركة

التي يتم بناؤها لهذا الغرض (Ubiquitous Language) بين الموظفين والنابعة من وجود بيزنس يربطهم ^^.

لكن قبل الانتقال هل خطر على بالك سؤال، ما هو الفرق بين ال Pain Point وال Business Domain؟ هذا السؤال رائع حقيقة ويدل على تركيز عالي... لأن ما تم طرحه من مثال في الجزء الثاني يمثل أو يقترب من كونه Modeled Around Business Domains ... والجواب صحيح.

إن ال Pain Point تمثل نقاط الألم داخل المشروع الحالي، أي أننا نركز على أماكن المشاكل أو الأماكن التي تسبب إزعاجا للمستخدمين على النظام الخاص بنا، وبهذا فإن هذه العملية تمثل نقطة التحليل والبحث عن المشاكل ليتم فيما بعد البدء في تصميم ال domains الخاص بالبيزنس لحل هذه المشاكل التي تم تسليط الضوء عليها... وبذلك يكون ال Paint Point أولا ثم ال Modeled Around Business Domains... لذلك تكون النتيجة النهائية لل Pain Point هي قائمة المشاكل التي أردنا حلها في حين أن ناتج ال Modeled Around Business Domains تكون ال domains التي تم تقسيمها وبنائها لحل تلك المشاكل.

ويمكن تلخيص هذه الفكرة من خلال هذين السؤالين:

- تبدأ بتحديد ال Pain Points: ماذا يزعج المستخدمين؟

- ثم هل نستخدم هذا الفهم لبناء Domains صحيحة: ما الأقسام التي يجب أن نبنيها لحل هذه المشاكل؟

ملاحظة ١: يقصد بال Ubiquitous Language اللغة المشتركة التي يتم وضعها داخل حدود معينة (Bounded Context) لضبط المصطلحات المستخدمة بين الفرق حتى تكون لغتهم واحدة وذات دلالة واحدة على عنصر ما في المشروع... مثلا لا يصح أن يستخدم المبرمجين كلمة Shopping Cart والتسويق Shopping Basket أثناء حوارهم في نفس الحدود (وهذا ينعكس حتى على تصميم قواعد البيانات وأسماء الحقول وأسماء المتغيرات :P والشباب بضيع)... لهذا يتم وضع جدول فيه المصطلح وما سيقوم به، مثلا:

المصطلح Cart - المفهوم مكان تخزين المنتجات قبل إتمام الشراء.

ملاحظة ٢: داخل ال DDD هناك مفهوم مهم وهو ال Bounded Context، ويقصد بها الحدود الخاصة بالنطاق الذي تم تعريفه ليكون مستقلا عن باقي النظام، وهو مفهوم رئيسي في ال (DDD)، ويستخدم لتقسيم الأنظمة المعقدة إلى أجزاء أصغر وأكثر وضوحا، بحيث يحتوي كل جزء (سياق) على نموذج خاص به ولغة مشتركة (Ubiquitous Language) تستخدم فقط داخل هذا الجزء، والهدف الرئيسي هو تجنب الغموض والتناقضات بين النماذج المختلفة في أجزاء مختلفة من النظام، ومن الإسم ندرك أن من وظائفه الأساسية وضع الحدود لمنع تداخل المسؤوليات داخل ال domains من خلال وضع حدود واضحة...

٢. ال Culture of Automation:

يقصد بهذا المبدأ أن العمليات في بيئة ال micro service، يجب أن تكون عمليات مؤتمتة، وهذا يشمل ال build وال test وال deploy، بمعنى آخر، هذا يعني اعتماد عقلية تجعل الأتمتة جزءا أساسيا من تطوير وتشغيل البرمجيات الخاصة بنا، والسبب في ذلك يعود لاستحالة أو صعوبة متابعة جميع الأجزاء الصغيرة، مع تضاعف كمية الأخطاء التي يمكن أن تنتج اعتمادا على العنصر البشري -الطريقة اليدوية-.

٣. ال Hide Implementation Details:

يقصد بهذا المبدأ أن ال Service التي ستقوم ببنائها يجب أن تخفي تفاصيل عملها داخليا عن ال client أو ال service الخارجية التي ستقوم باستخدامها، وإنما ما يهم هو الواجهة (API) التي تقدمها الخدمة لهم ومن خلالها يتم التواصل، أما كيف تنفذ المهام داخلياً فهذا ليس من شأنها :P...

هذا المبدأ مهم جدا لأنه يحقق عدة أهداف وهي:

- الفصل أو الاستقلالية (Decoupling): الخدمات لا تعتمد على تفاصيل بعضها البعض، ويمكن تطوير الخدمة وتحديثها دون التأثير على الخدمات الأخرى ما دامت واجهتها (API) لم تتغير.

- المرونة: يمكن تغيير التكنولوجيا الداخلية لهذه ال Service من لغة إلى لغة أو تحديث إصدارتها أو إجراء تغييرات معينة، دون إعلام أو توقف للخدمات الخارجية (مجددا ما دمنا نحافظ على ال API).

- الأمان: إخفاء التفاصيل عن ال service الخارجية يزيد من الأمان من خلال تقليل فرص الهجمات وإساءة الاستخدام ونحو ذلك.

- الاختبار والتطوير السهل: يمكنك اختبار الخدمة كوحدة مستقلة باستخدام واجهتها (API) فقط، ودون الحاجة لفهم تفاصيل الكود.

٤. ال Decentralize Governance:

يقصد بهذا المبدأ أن القرارات المتعلقة بالتطوير والأدوات التقنية لا تدار من مركز واحد أو فريق واحد فقط، بل يمنح لكل فريق حرية اتخاذ قراراته، لكن ضمن حدود أو أطر معينة! ومن الأمثلة على ذلك قد يتاح لكل فريق اختيار لغة البرمجة المناسبة لل Service التي يعمل عليها... فقد يختار فريق A ال PHP لتنفيذ مهام عرض المحتوى ويختار الفريق B ال python للتواصل مع نماذج الذكاء الاصطناعي...

وهذا يقدم عدة فوائد منها:

- السرعة: فالفرق لا تنتظر موافقة من جهة مركزية لتبدأ أو تغير شيئا.

- الابتكار: فكل فريق يمكنه تجربة أدوات وتقنيات جديدة مناسبة لمشكلته.
- المرونة: فلا حاجة لتقييد الجميع بتكنولوجيا واحدة قد لا تناسب كل الحالات.
- قابلية التوسع الإداري: من السهل إدارة عشرات الفرق حين لا يحتاج كل شيء لموافقة مركزية.

لكن هذا أيضا ينشئ لنا مجموعة تحديات حقيقة -سبحان الله، الكمال لله وحده جل في علاه-، أهمها:

- التنوع المفرط: إن لم تكن هناك خطوط واضحة أو guardrails واضحة للفرق يتبعونها، ستصبح الصيانة صعبة وسيكون المشروع مكتملا عبارة عن حاوية تجمع بداخلها كل شيء وإن كان لا يتماشى مع المشروع وماهيته أو إن كان حقيقة الخيار الأفضل أم لا... وتخيل أن يقوم فريقين في نفس المشروع بشراء خدمة كل واحد منهما من مكان مختلف لتحقيق نفس الغرض!

- فقدان الرؤية الشاملة: سيكون من الصعب وجود رؤية شاملة لكل خدمة من الخدمات دون وجود أدوات تساعد على ذلك.

- الأمان والامتثال: قد تنتهك بعض الفرق معايير الأمان التي تضعها المؤسسة أو ضبط امتثال المطورين داخل الفرق دون مراقبة خارجية...

٥. ال Deploy Independently (النشر المستقل):

يعد هذا المبدأ من الركائز الأساسية والجوهرية في تصميم microservices، وهو ما يميز الأسلوب الذي نتحدث عنه عن أنظمة ال monolith...، ويقصد بهذا المبدأ أن ال service التي نعمل عليها يجب أن تكون قابلة للتطوير والفحص والنشر بشكل مستقل عن باقي ال service الأخرى، ومجرد تطوير خدمة واحدة لا يعني الحاجة لتطوير أو نشر جميع الخدمات الأخرى!

وكما تلاحظ فإن هذا المبدأ فعلا مهم، ولولاه لقلنا ما الفائدة أصلا من ذهابنا إلى ال microservice! وهذه الأهمية نابعة من:

- السرعة في التطوير والنشر: فكل فريق يمكنه العمل على خدمته ونشرها في اللحظة المناسبة، ودون انتظار بقية الفرق.

- تقليل المخاطر: إذا حصل خطأ يمكن التراجع عنه (rollback) بسهولة.

- تحقيق مبادئ ال DevOps بشكل فعلي مع ضبطها بشكل فعال ضمن دورة ال CI/CD.

(وسيتم الحديث عن هذه الجزيئة بشكل مفصل بإذن الله تعالى في الأجزاء القادمة).
لكن يجب أن تيقي بذهنك نقطة مهمة جدا، وهي أن أي release خاصة ب service معينة يجب ألا توقف service أخرى، وهذا ما يطلق عليه: backward compatibility،

أي إذا تغيرت خدمة معينة فيجب الحفاظ على التوافق بينها وبين الخدمات الأخرى، كما يجب الانتباه والحفاظ وتنظيم آلية الإصدارات وأرقامها، مع وجود آلية فحص (test) شاملة للمكونات وهي متصلة فيما بينها (Integration Testing).

٦. ال Isolate Failure (عزل الأعطال):

يقصد بهذا المبدأ أن فشل خدمة واحدة يجب ألا يؤدي إلى انهيار باقي الخدمات أو النظام، أو التسبب بما يطلق عليه الإنهيار التسلسلي أو الفشل المتتالي (cascading failure)، ويتم تصميم ذلك من خلال بناء ال service على احتواء الخطأ الممكن حصوله ومنع انتشاره، وكمثال بسيط، تخيل أن لديك في الصفحة الرئيسية جزء يتعلق بالتوصيات الخاصة بالزبائن وجزء يتعلق بالمنتجات... فتعطل الجزء الخاص بالتوصيات، ففي هذه الحالة هناك أكثر من خيار، الأول أن يتعطل النظام كاملا أو أن نقوم بعملية تحايلية مثلا فنخفي هذا الجزء من الصفحة أو نظهر مكانه رسالة تفيد بوجود خطأ معين دون إيقاف بقية الخدمات!

فكرة معالجة الأخطاء ومنع حصول فشل متسلسل فكرة رهيبة حقيقة، فهي تدير أكثر من حالة ممكن أن تتسبب بالخطأ منها:

- وجود خطأ فعلي أدى لانهيار الخدمة تماما.
- وجود ضغط على خدمة ما أدى لتأخير زمن الاستجابة.

وبناء على ما ذكرنا، فإن هذا:

- يبرز الاهتمام بال Availability و Reliability ^^.

- كما أنه يحسن من تجربة المستخدم من خلال تقليل نسبة تأثير الخطأ عليه .
- وهذا يمكننا / يساعدنا من تمكين أو بناء ال Self-healing (التعافي الذاتي) والذي من خلاله يمكن مراقبة الخدمة فإذا وقعت قام شيء مثلا بإعادة تشغيلها.
- وهذا كله في المجمل يدعم مبدأ ال resilience أي المرونة التي تزيد من قدرة النظام على تحمل الأعطال والتعامل معها ومن ثم التعافي منها بأسرع وقت ممكن.

على الهامش: من الجميل أن تعرف أو أن تقرأ عن ال CAP Theorem.

والسؤال الآن: كيف يمكننا تطبيق هذا المفهوم -Isolate Failure- باختصار؟
والجواب من خلال ما يلي:-

- ال Timeouts: وذلك يعني وضع حد لأقصى وقت للاتصال بين الخدمات وقبل أن يعتبر فاشلا...

- التحكم بال Retries with backoff، ويقصد بذلك: إعادة المحاولة عند الفشل، ولكن بشكل محدود وتدريجي (exponential backoff)، لتفادي زيادة الحمل.. لهذا تجد في هذه النقطة Retries و backoff، فإعادة "المحاولة - Retries" يجب أن ترتبط بجزئية مهمة وهي "التراجع - backoff"، وهذا يعني بوجود المعامل الأسي الذي يستخدم عادة من أن الوقت المستغرق لأعادة أي محاولة ستتضاعف بشكل أسي، ويضاف إلى ذلك أنه يمكن إضافة بعض الضجيج لمنع تفاقم المشكلة من خلال

وجود الكثير من الطلبات على نفس هذه الخدمة وقت محاولة الإرجاع...

- ال Circuit Breaker: وذلك من خلال قطع الخدمة مؤقتا إذا زادت نسبة الفشل.

- ال Bulkheads: هذه الفكرة من اسمها مستوحاة من عالم السفن، فالسفن مكونة من حواجز ومقصورات، فإذا حدثت مشكلة وتسرب الماء يمكن إغلاق الحواجز لمنع تسرب الماء لباقي المقصورات، ومن خلال نفس الفكرة إذا حصلت مشكلة لدينا في نظام الدفع مثلا بسبب زيادة الضغط على الخدمة فيجب أن تبقى موارد اختيار المنتجات فعالة... وهذا يتقاطع ويترابط مع ال Circuit Breaker (أي أنهما يستخدمان معا عادة).

- ال Fallbacks: هذه الفكرة من النقاط الجميلة أيضا في عالم تحسين تجربة المستخدم ^^، ففيه يمكن اختيار مسار بديل أو نتيجة معينة في حالة حدوق الفشل، فمثلا إذا تعطلت الخدمة المتعلقة بتقييم المنتجات، سيكون الخيار البديل هو عرض: "التقييم غير متوفر الآن" ^^.

- ال Observability: وبكل تأكيد يجب أن يكون لدينا وسيلة لضمان مراقبة ومتابعة أي فشل يحدث على مستوى النظام! وهذا التتبع يجب أن يكون فوريا، فلا يعقل أن تفقد الخدمة لمدة معينة وأنت لا تعلم أن الخدمة قد توقفت عن العمل! وهذا يقودنا

للمبدأ السابع:

٧. ال Highly Observable:

ويقصد بهذا المبدأ أنه إذا حدث شيء غير طبيعي في النظام فستتمكن من معرفة "ماذا حدث؟ وأين؟ ولماذا؟ ومتى؟" بأسرع وقت، بعبارة أخرى يجب أن تكون لديك القدرة على رصد وتحليل مخرجات النظام بوضوح! وذلك يكون من خلال:

- تتبع ما يحدث داخله.
- معرفة الأخطاء والمشاكل فورًا.
- فهم سلوك كل خدمة وكيف تتفاعل مع غيرها.

وهذه العملية مهمة جدا بسبب:

- أن الأنظمة المبنية بوجود microservice تتكون من عشرات الخدمات أو أكثر.
- ولأن كل خدمة منفصلة فسيصعب تتبع الأخطاء خصوصا إذا كانت كل خدمة تعمل في container أو server مختلف.
- هناك مشاكل قد تحدث بسبب الشبكة أو الاعتماد المتبادل بين ال services.

ويمكن تحقيق هذه الغاية من خلال عدة طرق منها:

- ال Logs وذلك من خلال تسجيل الأخطاء والتفاصيل المهمة للتتبع...إلخ
- ال Metrics وذلك من خلال قياس الأداء وزمن الاستجابة واستخدام الموارد...إلخ

- ال Tracing وذلك من خلال قدرة النظام على تتبع الطلبات بين الخدمات المختلفة.
- ال Dashboards والتي من خلالها يمكن عرض حالة النظام بشكل فوري أو لحظي.
- ال Alerting وهي الخاصية المهمة التي تحذرنا الآن من وجود مشكلة تستلزم الحل عند تجاوز حدود معينة مثل نسبة الفشل أو انخفاض المستخدمين...إلخ.

والآن، بعد أن تحدثنا عن مبادئ ال Microservice وجب علينا الانتقال للخطوة التالية، وهي، كيف يمكننا تطبيق هذه المبادئ على ال Frontend ليصبح لدينا micro-frontend؟

والجواب ببساطة هو ملخص ما ذكرناه سابقا مع صياغة بسيطة تجعلها للفرونت اند! وسنحاول تلخيص ما ذكرناه بشكل سريع كما يلي:-

١. ال Modeled Around Business Domains: تقسيم العمل وفقا لمجالات عمل كل جزء، واعتمادا على مبادئ ال DDD، ومن ذلك تقسيم العمل الخاص بالفرونت اند على شكل Domains وإعطاء الصلاحيات للفرق.

٢. ال Culture of Automation: بغض النظر عن عدد المكتبات التي تستخدمها وبغض النظر عن الآلية التي قمت ببناء الواجهات الخاصة بالفرونت اند بها؛ يجب أن تخضع للأتمتة بشكل كامل في المراحل المختلفة.

٣. ال Hide Implementation Details:

ينبغي لكل micro-frontend أن يخفي تفاصيل تنفيذه الداخلية.

4. ال Decentralize Governance

تمكين الفرق من اتخاذ قراراتها التقنية بشكل مستقل ضمن أطر أو قواعد محددة مثل استخدام أحد الفرق axios والآخر fetch!

5. ال Deploy Independently

يفضل أن يكون لكل micro-frontend القدرة على النشر بشكل مستقل (تحديث التنسيق الخاص بالتوصيات ونشرها دون انتظار تحديث شاشة الطلبات والدفع).

6. ال Isolate Failure

يجب تصميم micro-frontend بحيث لا يؤثر فشل واحدة من الخدمات على النظام (مثال تعطل التوصيات في صفحة المنتجات).

7. ال Highly Observable

ينبغي توفير أدوات ووسائل لمراقبة أداء وسلوك كل micro-fronted.

ومن هنا نعود للنقطة الأولى وأول عنوان في هذه السلسلة وهي: "لا تستخدم المدفع الرشاش لتقتل نملة"، وعلى نفس النمط يقول الكاتب: "Micro-Frontend Are Not a Silver Bullet"، أي أن ال micro-frontend ليس حلا سحريا للمشاكل التي تواجهها، وهي ليست الحل الأمثل في كل الحالات! بل إن استخدامها في المكان الخاطئ هو أحد أكبر

الأخطاء التي يمكن أن تركتبها، لذلك يأتي السؤال المهم، متى من المناسب استخدام هذه المعمارية؟ ومتى من غير المناسب استخدامها؟

من المناسب استخدام هذه المعمارية إذا كان مشروعك:

- للمشاريع طويلة الأمد والتي تحتاج إلى صيانة مستمرة وتحديثات دورية.
- عند وجود عدة فرق تعمل على نفس التطبيق وفي الوقت نفسه.
- عند حاجتك إلى تحديث تدريجي لتطبيق قديم (Legacy)، ودون إعادة بنائه من الصفر.

ومن غير المناسب استخدامها إذا كان مشروعك:

- صغيرا ثابتا لا يتغير كثيرا
- صفحات html بسيطة وعددها قليل
- مواقع ال SPAs
- مواقع ال server-side rendering

لحظة لحظة... توقف هنا!!! كيف تقول أن مواقع ال SPAs وال SSR ليس من المناسب تحويلها إلى معمارية micro-frontend مع أن أغلب المواقع التي نرغب بتحويلها هي من هذا النوع؟!

سؤال ذكي ومنطقي، وهنا اللفتة التقنية المهمة، أن الأصل في هذه المواقع عدم حاجتها للتحويل إلى معمارية تفصل مكوناتها وتفككها إلى micro-frontend افتراضيا!، وإنما نحتاج إلى تحويلها إذا انطبقت عليها الشروط، مثل وجود عدة فرق تعمل في نفس المشروع وpain point ظاهرة وتحتاج إلى حل مثل البطئ وزيادة التعقيد..إلخ، وهنا ننتقل من غير المناسب إلى المناسب!

خلاصة ما تم ذكره حتى هذه اللحظة :

- الانتقال يكون عند وجود حاجة حقيقية لذلك! ومن الآن سنبدأ فعليا بالتعمق أكثر داخل ال micro-frontend.

- ال Micro-frontends مستوحاة من مبادئ microservices، ولكنها تُطبَّق في الواجهة الأمامية، وهذا يعني أن Micro-frontends = Microservices من أحد الوجوه وينبثق منها.

- هذه المعمارية تهدف إلى تفكيك واجهة المستخدم إلى وحدات مستقلة، بحيث يكون لكل فريق مجال ومسؤولية واضحة.

- السبعة مبادئ المستفادة من microservices يمكن تطبيقها في بناء واجهات أمامية أكثر مرونة واستقلالية وقابلية للتوسع والتطوير...

فائدة

هل استحضار الموت واليوم الآخر يصرف الإنسان عن العمل وبناء الحضارة؟
الحقيقة أن هذا فهم مغلوط كلياً، ولا يقول هذا الكلام رجل قرأ كتاب الله وأيقن صدقاً بمعانيه، فإن استحضار الموت واليوم الآخر هو الذي يدفع فعلاً للعمل الصالح النافع المثمر طبقاً لمراد الله.

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

الفصل الثاني: معمارية ال micro-frontend وتحدياتها

لقد تحدثنا عن الكثير من المبادئ الأساسية المتعلقة بال microservice من قبل، وقلنا كيف يمكن لهذه المبادئ بمجرد تغيير بسيط أن تصبح micro-frontend، لكن عند الانتقال من المستوى المفاهيمي للمستوى التطبيقي يختلف الأمر قليلا، فنحن نحتاج حينها إلى الخوض في غمار وتفاصيل هذه المبادئ، وكيف يمكننا تطبيقها وتنفيذها، وكيف يمكننا ربط المكونات فيما بينها، وكيف يمكننا إيجاد حلول للمشاكل والتحديات التي تواجهنا حتى نتجاوزها...

في هذا الجزء سنبدأ في التركيز على أربعة محاور أساسية، وهي:

- ال Business domain representation
- ال Autonomous codebase
- ال Independent deployment
- ال Single-team ownership

إن ال Micro-frontend يقدم العديد من المزايا والخيارات، لذلك فاختيار النهج المناسب والصحيح يعتمد بشكل كبير على خبرة المطورين أنفسهم في بناء هذه المعمارية، كما تعتمد بشكل رئيسي على متطلبات المشروع والهيكل التنظيمي للشركة خصوصا فيما يتعلق بالفرق ولغة التواصل فيما بينها.

هناك العديد من المشاكل التي يمكن تواجهنا أثناء بناءنا لل micro-frontend، نذكر منها على سبيل المثال:

- كيف تتواصل هذه الأجزاء مع بعضها؟
- كيف سننتقل من واجهة لأخرى بسلاسة؟
- ما هو الحد المناسب لحجم كل جزء؟

لذلك هناك قرارات كثيرة متعلقة بالمعمارية يجب أن يتم اتخاذها من البداية لمواجهة مثل هذه التحديثات أو الأسئلة وقبل الشروع في العمل؛ لأن تغييرها لاحقًا قد يكون مكلفا وصعبا. وذلك مثل تحديد آلية الاتصال والتواصل بين ال services منذ البداية... ومن هنا نذكر نقطة مهمة تذكرها جيدا: "كلما زادت استقلالية ال services، زادت صعوبة التكامل فيما بينها، والعكس صحيح"، أي أنك دوما عند رسم نقاط وحدود ال service يجب عليك المفاضلة بين استقلالية ال service وتكاملها مع ال services الأخرى، وهذا ما سنتحدث عنه بالتفصيل بالأجزاء القادمة إن شاء الله...

والآن لنذهب لأول نقطة مهمة وهي: Micro-Frontends Decisions Framework، ويقصد بها مجموعة الأفكار والأطر التي ستساعدنا في اتخاذ القرارات الصحيحة وضبطها حتى تناسب المشروع الخاص بنا، وحتى تتم هذه العملية بشكل صحيح علينا أن نفهم السياق (context) الخاص بنا وما سنعمل عليه! فإذا لم تفهم ما ستقوم ببناءه ومكوناته فكيف ستقوم ببناء معمارية له؟!

بناء على ما ذكرناه، فيمكن تقسيم محاور اتخاذ القرارات لأربعة محاور وهي:

1. **التعريف (Defining)**: ما هو الجزء الذي يُعتبر micro-frontend؟ هل هو صفحة؟ مكون؟ وحدة وظيفية؟ وما هي حدوده؟

2. **التجميع (Composing)**: كيف ستجتمع هذه الأجزاء الصغيرة لتكوين الواجهة الكاملة التي يراها المستخدم؟ (مثل آلية جمع صفحة المنتجات وصفحة الدفع).

3. **التوجيه (Routing)**: كيف سيتم الانتقال بين هذه الأجزاء؟ (مثل طريقة الانتقال من صفحة المنتجات إلى صفحة الدفع).

4. **التواصل (Communicating)**: كيف ستتبادل هذه الأجزاء البيانات؟ (مثل: مشاركة حالة تسجيل الدخول ومعلومات المستخدم).

والآن لنبدأ بشرح كل محور بشكل تفصيلي، وتذكر أن هذه المعلومات النظرية هي معلومات مهمة ستكون تكلفة تجاهلها عالية ومكلفة عليك وعلى المؤسسة، فاحرص على الانتفاع من هذه التفاصيل قدر الإمكان.

أول محور: Define Micro-Frontends

هذا المحور يدور حول كيفية وآلية تحديد (تعريف) الجزء الذي سيعتبر ك Micro-Frontend في البنية المعمارية للتطبيق الخاص بنا، وهو أول قرار أساسي يمكنك اتخاذه وله تأثير كبير على باقي القرارات.

من الناحية التقنية هناك خيارين اثنين يمكننا الاختيار بينهما لتعريف الوحدة أو المكون لل micro-frontend، وهما:

- أن تحتوي نفس الواجهة (view) على عدة micro-frontends، وهذا ما يطلق عليه: Horizontal split، وخصائص هذا الاختيار هي:
  - أكثر من micro frontend في نفس الواجهة
  - كل فريق سيكون مسؤولا عن جزء معين من ال view، ويجب على الفرق تنسيق العمل فيما بينهم.
  - هذه الطريقة مرنة لأنها تسمح للفرق في إعادة استخدام بعض الأجزاء مرة أخرى في أماكن أخرى.
  - تتطلب انضباط كبير وتنظيم عالي المستوى حتى لا تصبح الكبسة الواحدة عبارة عن micro frontend ^^.
  - وجود تعقيد كبير في التنسيق بين الفرق.
- أن تحتوي كل واجهة (view) على micro-frontend واحد فقط وهذا ما يطلق عليه Vertical split، وخصائصه هي:

- كل فريق يكون مسؤولا عن نطاق عمل محدد (business domain) مثل ال Authentication وفريق آخر على Cert... إلخ
- مرونة أقل
- استقلالية أعلى وأفضل
- صعوبة في دمج ال UX بين المكونات المختلفة بسلاسة.
- يظهر هنا سبب قوي يدفعنا لاستخدام ال DDD في هذه المعمارية.

شاهد هذه الصورة F3-1 حتى تتضح لك الفكرة:

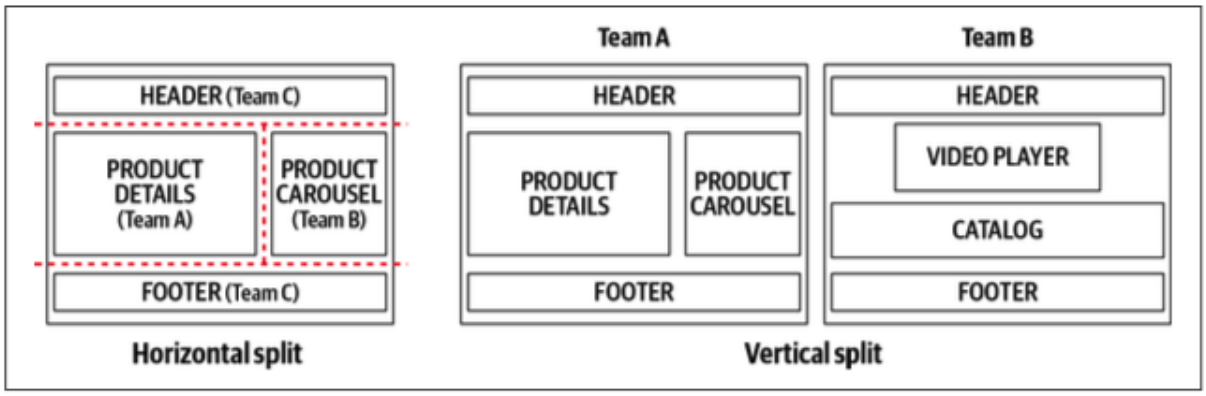


Figure 3-1. Horizontal versus vertical split

قلنا أن دور ال DDD يظهر هنا بشكل قوي، وهذا ما يدعونا للتعرف عليه قليلا، ويمكن القول أنه أسلوب لتطوير البرمجيات منطلقا من تركيزه على بناء نموذج (Model) ضمن مجال (Domain) معين، وهذا يتطلب فهما عميقا للقواعد والعمليات التي تحكم هذا ال domain، وتطبيق ال DDD في ال Frontend يختلف قليلا عن ال Backend في بعض المفاهيم،

لكن أفكاره الأساسية ومبادئه مهمة جدا لنا لبناء micro-frontend رائع ويحقق الغاية التي نصبو إليها.

وليتضح المفهوم لنأخذ مثالا، ولنفترض أنه لديك تطبيق لبث (stream) فيديوهات تعلم القرآن الكريم، وهنا لدينا التقسيم التالي:

- ال Core Domain: ويمثل في حالتنا هذه ال video stream، ويقصد به المجال الأساسي (جوهر النظام) والسبب الرئيسي لوجود هذا التطبيق.
- ال Subdomains: ويقصد بها المجالات الموجودة ضمن ال Core Domain الذي قمنا بتحديده وسنقوم ببناء ال micro-frontend لخدمتها، وهي ثلاثة أقسام، ويمثل في حالتنا -على سبيل المثال- ال catalog الخاص بفيديوهات القراء وال authentication مثل عملية التسجيل في الموقع، ومشغل الفيديو...، أما أقسام ال subdomains فهي:
  - ال Core Subdomains: تمثل السبب الأساسي لوجود التطبيق ضمن ال Core Domain، ويمكن أن يكون هناك أكثر من Core Subdomain...، عند تعريفك ميزة ما داخل هذا القسم فهذا يعني بالضرورة أنك يجب أن تتعامل معها على أنها الأهم بالشركة، ولها الأولية في الخدمة، مثل: الكاتلوج في حالتنا هذه.
  - ال Supporting Subdomains: ويقصد بها النطاقات الداعمة والمساندة في التطبيق والتي تخدم التطبيق ببعض الجوانب المتعلقة أو المترابطة مع ال core subdomains، لكنها بنفس الوقت ليست ما يميز التطبيق! ما يتم تصنيفه هنا

يعتبر مهما لكنه ليس الأهم أو ليس مركز الاهتمام، وغالبا ما يتم تطويرها أيضا داخل الشركة أو المؤسسة، ومن الأمثلة على ذلك: وجود نظام للتصويت على الفيديوهات.

- ال Generic Subdomains: هو جزء من النظام لا يميزك عن غيرك، ويمكن تنفيذه باستخدام حلول جاهزة، لأنه ليس جزءا خاصا بنشاطك أنت وحدك، بل هو "مجال عام" موجود في معظم الأنظمة، ويمكنك الاستعانة فيه بمكتبات أو خدمات خارجية بدلًا من بنائه بنفسك، ومن الأمثلة على ذلك ال Auth أو عملية إرسال الإيميلات...، لذلك ما يقبع تحت هذا التصنيف لا يحتاج إلى تصميم معقد أو استثمار كبير، ويمكن شراؤه أو الاستعانة بفريق خارجي لبناءه أو استخدام مكتبة جاهزة.

والآن شاهد الصورة F-table 3-1:

Table 3-1. Subdomains examples

Subdomain type	Example
Core subdomain	Catalog
Supportive subdomain	Voting system
Generic subdomain	Sign-in or sign-up

ثانيا: Domain-Driven Design with Micro-Frontends

ننتقل الآن للحديث عن ثاني نقطة مهمة في موضوع معمارية ال Micro-Frontend وتحدياته، وهو ال Domain-Driven Design with Micro-Frontends، وهنا نعود للحديث عن مصطلح مهم ذكرناه سابقا وهو ال Bounded Context، ويمكنك الرجوع لتعريفه الذي ذكرناه... لكن باختصار يقصد بها الحدود المنطقية الخاصة بكل مجال (domain) داخل النظام والتي تجعله مستقلا بذاته عن غيره، ومن المزايا المهمة التي نستفيدها عند تطبيق هذا المفهوم أن كل سياق سيضم ال Model الخاص به والشيفرة البرمجية وطرق التواصل بين ال context المختلفة، والفرق وحدود هذه الفرق والتي تعمل على ما أنيط إليها من خدمات.

هناك علاقة وثيقة بين ال Domains وال Bounded Contexts، ففي المشاريع الجديدة التي بينت باعتبار أن معماريتها ستكون micro-frontend وستطبق مبادئ ال DDD؛ ستكون جيدة ومترابطة ومتوافقة فيما بين مكوناتها، فالمكونات سيتم توزيعها على مجالات واضحة منذ البداية، وبالتالي سيكون توزيع المهام واضح وبالتبعية توزيع الفرق على هذه المهام، لكن في الأنظمة القديمة (legacy) قد تحتوي ال Bounded context على أكثر من subdomain لأن النظام القديم لم يكن قائما أو مبنيا بالأساس على فكرة أو مبادئ ال DDD.

ولتنفيذ ال micro-frontend هناك عدة طرق تقنية تمكننا من ذلك، وهي:

- ال ifreams
- ال web components
- ال component library

والآن تخيل معي هذا السيناريو ^^، لدينا مؤسسة تنوي تحويل النظام القديم لمعمارية تطبق ال micro frontend، ويوجد لدينا ثلاثة فرق يعملون في مناطق جغرافية مختلفة على نفس ال code base، هنا لدينا حالتين:

1. اعتماد نهج ال horizontal split، ولتنفيذ ذلك سيختارون إما ifreams أو web component في ال micro-frontend التي سيقومون ببنائها، لكن بعد فترة قليلة من العمل ستكتشف الفرق الحاجة لطريقة لتواصل هذه الواجهات فيما بينها (حتى يتم تناقل البيانات مثلا) ولأجل ذلك ولحل هذه المشكلة سيضطر أحد الفرق بتحمل مسؤولية إنشاء صفحة ال View لعرض وتجميع هذه الأجزاء معا، وهذا الفريق سيتحمل ضغط كبير في تنسيق هذه الواجهات معا وضبطها وحل المشكلات التي تظهر... فوق كل هذا تخيل وجود مؤثرات وتحديات تعيق العمل مثل اختلاف المناطق الزمنية لأن كل فريق في منطقة جغرافية معينة، أو وجود cross dependencies بين الفرق مثل أن الفريق A يعمل على User Form وفريق B يعمل على Email Verification System، لكن فريق A لا يستطيع إتمام عمله إلا بعد أن ينهي B ال API الخاصة بالتحقق... هذا كله سيؤدي إلى شعور عام سيء (إحباط عام) مما سيعود بنتائج سلبية على المؤسسة والمنتج والعميل...

ملاحظة مهمة: لاحظ هنا أن التركيز منصب على جزئية محددة في هذا النهج، وهي الجانب التقني فقط!

2. اعتماد نهج ال vertical split: عودة لنفس السيناريو الذي قمنا بطرحه، لنفكر الآن بتقسيم العمل انطلاقا من نظرة تجارية ^^، أي لننظر إلى مكونات النظام الذي نرغب بفصله بعين ال business، في هذه الحالة يمكننا استخدام ال SPA أو Single Pages لتمثل context واضح للعمل، وهذا يعيدنا للدائرة الأولى والتي حوت الأسئلة التي نريد الإجابات عنها، أهمها: "ما هي ال business value وال core domain لدينا؟"... في هذه الحالة كل فريق سيعمل على تصميم صفحاته وال API الخاصة به مع إمكانية التحكم الكامل فيما يحتاج دون مشاكل خارجية... وهذا كله يوفر إمكانية تسلم ال service بشكل مستقل دون وجود مخاطر كبيرة على مستوى النظام ككل.

ملاحظة مهمة: لاحظ هنا أن التركيز منصب على أكثر من جانب، أول جانب كان جانب ال business، ولدينا الجانب التقني...

ومن هنا يظهر لدينا مفهوم جديد وهو ال Microarchitectures، وهو ببساطة يمثل النهج العام في تصميم معمارية الأنظمة الخاصة بنا، وهذا يحتوي بداخله ال "Microservice + Micro-Frontend"، وهنا إشارة من الكاتب أن تحقيق أفضل استفادة ممكنة تبدأ من كون معمارية النظام الذي نعمل عليه معمارية مجزأة، فيها ال backend عبارة عن microservice تقدم خدماتها بشكل مستقل، وواجهات ال frontend عبارة عن micro-frontend تقوم بالاتصال مع ال backend service

المناسبة لها، وكل فريق مستقل ضمن طوابط معينة..إلخ، وبذلك سنحصل على نظام مرن وقابل للتوسع وقابل للتسليم ويقلل من المخاطر من المحتملة...
ملاحظة: تذكر أننا نتحدث هنا عن نظام كان من المناسب تقسيمه وتجزئته، وليس كل نظام! وتذكر: "لا تستخدم المدفع الرشاش لتقتل نملة"!.

والآن نعود لنقول، إن وجود bounded context يساعدنا على تصميم النظام الخاص بنا، ومن خلال ذلك نستطيع رسم ال domains وال sub domains الخاص بالمشروع، وبعد تعريف جميع هذه الحدود؛ سنتمكن من رسم خريطة تمثل حدود ومكونات النظام وطريقة تواصل المكونات فيما بينها... وهذا أيضا سيقدم لنا معلومات مهمة عن آلية التواصل من خلال ال API's، حتى يتم ربط ال backend مع ال micro-frontend.

شاهد الصورة F3-2 والتي تمثل ال catalog bounded context map:

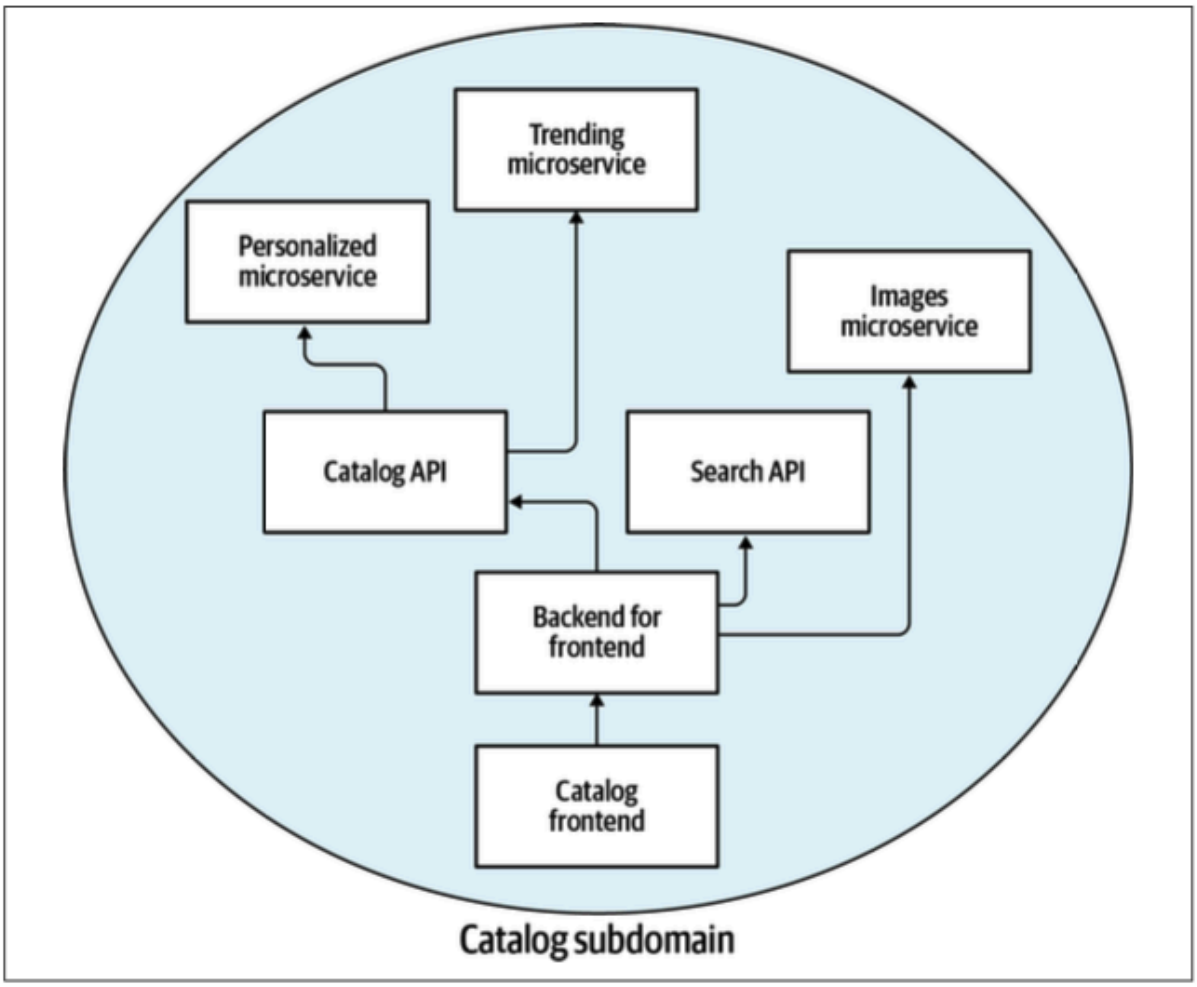


Figure 3-2. A representation of bounded context

ملاحظة: ال Backend for Frontend واختصارا ال BFF: هي طبقة وسيطة مصممة أو مخصصة لكل micro-frontend يتم بناؤها للتواصل مع ال microservices لجلب البيانات وتجهيزها بما يتناسب مع احتياجات ال FE، وسبب وجود هذه الطبقة أننا عادة -كما في الصورة- يوجد لدينا العديد من ال microservices الموزعة، ونحن نحتاج إلى بيانات مجمعة وبنسق معين وبأداء سريع، وهذا يمكن توفيره من خلال ال BFF، وحتى تتخيل أهميتها تخيل أنك ستقوم بضرب خسمة من ال API على خمسة micro services لتحقيق نفس

الغرض! عدا عن المزايا الأخرى المهمة مثل وجود مستوى أعلى بالأمان وأسهل بالضبط والتحكم والمصادقة وتحسين الأداء وال caching وتيسير عملية الصيانة والتطوير..إلخ.

قانون Conway ومناورة Conway العكسية

لقد أشرنا إلى أن تنظيم الفرق والتواصل فيما بينها من الأهمية بمكان، لذلك الشركات في محاولتها لتحسين التواصل بين الفرق تصمم أنظمتها بما يتوافق مع قانون Conway والذي ينص على أن: "الأنظمة التي تصممها المؤسسات تعكس بنية التواصل وكفاءة التواصل بين الفرق داخل تلك المؤسسات"، وبهذا يعكس التصميم طريقة التواصل ولا يعكس أفضل تصميم منطقي يمكن أن يصمم عليه النظام، فالقانون هنا يمثل الجانب التقني للنظام = لأن نظرتنا ستكون كالتالي:

لنفرض أن لدينا ثلاث أفرقة، الأول FE والثاني BE والثالث DB، كل فريق من هؤلاء أصدر نظاما معزولا ويحتاج إلى جهد حتى يتم ربط المكونات مع بعضها = وهذا يعكس أن الفرق لم تتواصل مع بعضها بشكل جيد ولا توجد هناك آلية واضحة لتنظيم هذه الفرق، وهذا التشوه في النتائج حصل أو ظهر لأننا اعتمدنا على الفرق الحالية بوضعها وطرق اتصالها لتصميم النظام مهتمين بالجانب التقني فقط، وبنفس المفهوم لو كان لدينا فريق API قام بتصميم كل ال Apis لكل ال service وفريق UI صمم كل واجهات النظام... إلخ.

ومن هنا ظهرت مناورة Conway العكسية، وهي تعتمد على تعديل هيكلية الفرق بالمؤسسة حتى تتناسب مع التصميم الذي نرغب بالحصول عليه ^^... فقط وببساطة ^^.

يعني باختصار، في القانون الأصلي النظام هو من يعكس آلية التواصل في المؤسسة، بينما في المناورة نحن من يقوم بتغيير هيكلية الفرق لتخرج لنا النظام الذي نرغب.

فائدة

فلْتعلمْ أن المؤمن لا يحسد! فلا يتمنى الاستئثار بالنعمة التي رزقها الله لغيره، ومن ثَم يتمنى زوال هذه النعمة عن صاحبها! بل المؤمن من يدعو لأخيه باليُمن والبركة فيما رزقه الله -سبحانه وتعالى-، ويدعو الله -تعالى- لنفسه بأن يرزقه ما رزقه مع تمني دوام الخير لمن رُزق هذه النعمة.

- كتاب إلى الجنة زمرا، الصفحة ٩٤

كيف يمكننا تعريف / تحديد ال Bounded Context بشكل صحيح؟

هناك بعض المبادئ التي يمكن أن تساعدك في تحديد النطاقات أو حدود الخدمات لديك... نذكر منها:

1. لا تتعجّل في التقسيم:

   التحسين المبكر قد يقود إلى تقسيم النظام إلى Bounded صغيرة جدا بناء على افتراضات خيالية لا حقائق موجودة، لذلك احذر من تقسيم النظام إلى Bounded Contexts صغيرة قبل أن تتوفر معلومات كافية لديك.. ومن ذلك قد تظن أن نظامك سيحتاج إلى تكامل معين مع service أخرى، فتبني لذلك معمارية معقدة ثم تكتشف فيما بعد أن ذلك لم يكن ضروريا...

2. النطاقات المحدودة تتغير مع تطور العمل:

   قد تبدأ ب Bounded واحد كبير، لكن مع الوقت يصبح معقدا جدا، فتحتاج إلى تقسيمه لاحقا، وهذا أمر طبيعي ومنطقي، فلماذا أقسم الكتلة الكبيرة إن لم يكن هناك داع لذلك؟! أما إذا اتسع العمل وازداد التعقيد، فحينها سيكون من الأفضل تقسيمه لأجزاء أصغر.

3. اجمع معلومات قبل التقسيم:

   القرار الأفضل عند رسم الحدود يُتخذ بعد توفر البيانات الواضحة، وهذه البيانات تشمل معلومات المستخدمين ومعلومات الفرق التي ستعمل على هذه services ومعلومات

الفريق نفسه الذي سيعمل على الخدمة! وإياك والتخمين ^^.

4. تعاون مع زملائك من الفرق الأخرى:

تعاون مع زملائك من الفرق الأخرى والذين لديهم خبرة حقيقية حول المنتج وآلية عمله وأهم خصائصه وأهم ما يهتمون به في خدمة عملائهم أو المزايا التي يعملون عليها ونحو ذلك، فالمطور لوحده لا يستطيع تحديد كل الخصائص والمزايا خصوصا في جانب ال business، وذلك حتى يتم التقسيم بطريقة صحيحة وشمولية.

5. ابدأ بالبيانات وسلوك المستخدم:

من الأمور المهمة استخدم بيانات التفاعل الحقيقية لتحديد ما يحتاجه المستخدمون، وتحديد آلية وسلوك المستخدمين في التطبيق، فعلى سبيل المثال: إذا كان 70% من الزوار يذهبون من الصفحة الرئيسية إلى صفحة تسجيل الدخول، و40% فقط يكملون، فيمكن هنا تحسين تجربة المستخدم من خلال جعله يتعامل فقط مع ما يحتاجه من الكود في كل خطوة مثل إضافة(lazy render)، أو تغير النسق أو الألوان أو موضع ال buttons حسب سلوك المستخدمين... وهذا يقودنا للنقطة التالي ^^

6. تحسين الأداء عن طريق تحميل الأجزاء التي يحتاجها المستخدم فقط:

هناك العديد من الطرق لتحسين أداء التطبيقات، واحدة من أهم النقاط في تحسين الأداء والتي لها تأثير مهم على ما نفكر فيه، عملية تحسين الأداء من خلال تحميل فقط ال service التي سنعمل عليها في الوقت الحالي... فمثلا لن أحتاج لتحميل صفحة تقيم

المنتجات داخل صفحة تسجيل الدخول! وهذا الأمر يشمل الصفحة الرئيسية باعتبارها من أهم الصفحات التي تعطي انطباعا مهما عن الموقع...

7. الاختيار بين ال Horizontal Split وال Vertical Split:

تحديد طبيعة المشروع وآلية العمل المناسبة لكل مشروع ستحدد الآلية المناسبة لاختيار أسلوب التقسيم المناسب... وعادة ما يكون ال Vertical Split هو الأسلوب المناسب لتطبيقات ال Micro frontend.

8. مراعاة مهارات الفرق لديك وإمكانيات الشركة

عادة ما تناسب ال Vertical Split الفرق التقليدية بينما تكون التقسيمات ال Horizontal أصعب لحاجتها لأدوات أكثر وأتمتة أقوى.

الآن يمكننا صياغة النقاط السابقة على الشكل التالي:

علينا كمطورين أن نفهم ال Domain Model للنظام الخاص بنا ومن ثم محاولة تقسيمه إلى Domains و Subdomains تقدم معنى حقيقي، وكل جزء من النظام يجب أن يرتبط بواحدة من هذه المجالات التي قمنا بتحديدها، ويجب أن تكون هذه ال micro مفصولة عن غيرها و مستقلة ولها سلوكها الخاص، كما لكل micro قمنا بتحديدها مجموعة المصطلحات التي ينبغي استخدامها، كما ينبغي علينا تحديد حدود التواصل وآليته بين هذه الأجزاء، وسيكون كل فريق مسؤول عن جزء من هذه الأجزاء... هذا كله مما يجب عن السؤال الذي طرحناه أول هذه الكلمات: "كيف يمكننا تعريف / تحديد ال Bounded Context بشكل صحيح؟".

شاهد هذا المثال البسيط:

لدينا نموذج لموقع تجارة إلكترونية، وبناء على دراستنا كان لدينا Cert, Checkout, Products, Users، Warehouse... وكل مجال من هذه المجالات لن يعرف عن الآخر إلا من خلال ال API، فال cert لن تعرف تفاصيل عملية الدفع (checkout)، وإنما سيتم التواصل من خلال واجهة صممت لذلك API، وسيقوم كل فريق بتصميم جميع الأجزاء المرتبطة بخدمة معينة وتقديم ال API حتى تتمكن من استخدامها الفرق الأخرى، وسنقوم جلب (download/render) الصفحات عند الحاجة إليها، لذلك لن يتم تحميل صفحة ال checkout والكود الخاص بها إلى بعد أن ينتهي المستخدم من التسوق، وقام الفرق الأول بتعريف أن كلمة "العميل" في سياق ال cert تشير إلى "الزبون" في حين قام الفريق الثاني إلى تعريف ال "العميل" في ال Warehouse في حدوده على أنها "الشركات".

والآن بعد تحديد أو تعريف الحدود بين ال Context المختلفة، كيف يمكننا تجميع مكونات ال Micro-frontend التي تم بناؤها؟ وهذا ما سنتطرق إليه الآن بإذن الله تحت عنوان: Micro-Frontends Composition.

ثالثا: ال Micro-Frontends Composition:

لتجميع ال Micro Frontend التي قمنا ببنائها داخل تطبيق هناك ثلاثة طرق وهي:

1. ال Client-side composition

2. ال Edge-side composition

3. ال Server-side composition

شاهد الصورة F3-3:

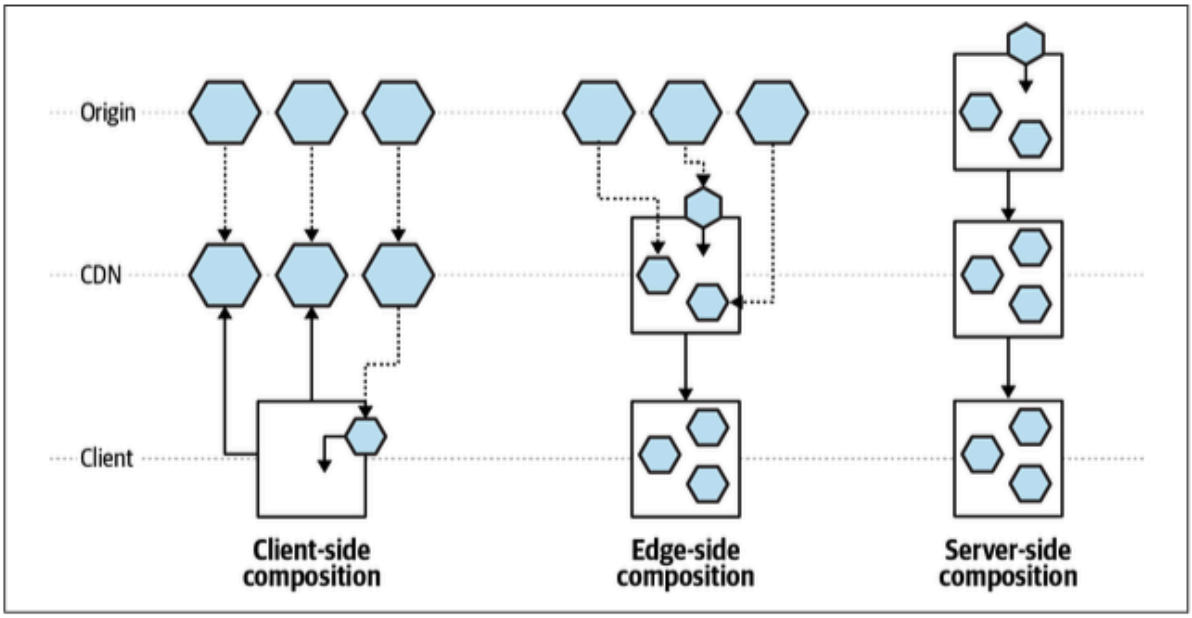


Figure 3-3. Micro-frontends composition diagram

تبسيط الصورة: تشير الأشكال السداسية لل micro-frontend، وتشير المستطيلات إلى أماكن التجميع، وتشير الأسهم المتصلة إلى عملية نقل المحتوى مباشرة إلى المرحلة التالية وتشير الأسهم المتقطعة لعملية التخزين المؤقت لكل micro-frontend في ال CDN.

بناء على ذلك ستجد أن أول مخطط على اليسار يمثل عملية جلب ال micro frontend من ال origin وعمل cache لها وجلب ال micro التي نحتاجها عند ال client... أما في المخطط الأوسط يتم جلب ال micros من ال origin وتجميعها عند ال cdn مع عمل

cache لها ومن ثم إرسال الصفحة كاملة لل client، في حين يمثل آخر مخطط عملية تجميع ال micros على مستوى ال origin ومن ثم إرسالها مجمعة كاملة إلى ال cdn, client...

ملاحظة: لاحظ في أن ال server-side composition لا يوجد فيه خطوط متقطعة لأن الصفحة كاملة يتم بناؤها على السيرفر، فيكون دور ال cdn هو حفظها مؤقتا وإرسالها لل client عند الطلب وليس تجميعها أو حفظ ال micros عنده مستقلة ليتم تجميعها...

والآن لنذهب لشرح أساليب التجميع:

أولا: Client-side composition:

ملاحظة قبل أن نبدأ: هناك مفهوم ستراه في هذه النقطة وهو ال Application Shell، وهذا المفهوم يشير لنقطة مهمة يدور عليها هذا الجزء من الشرح، وباختصار ال Application Shell هو ال Main Layout أو الإطار الرئيسي أو الهيكل الكبير الذي يحتوي عناصر (component / sections) لا تتغير بين الصفحات، ولا يحتاج لتحميل الصفحة كاملة إذا حصل navigation، ويتم تحميله مرة واحدة عند فتح التطبيق... هذا الهيكل أو الإطار يحتوي على هذه الأجزاء:

- ال Header
- ال Footer
- ال Sidebar / Navigation Bar

- مكان لتجميع ال Micro-frontend، بحيث يتم مثلا عرض صفحة الدخول في هذا المكان عندما نكون في صفحة الدخول، ويتم جلب صفحة المنتجات عند الدخول لصفحة المنتجات مكان الصفحة القديمة... وهكذا.

في هذا الأسلوب -Client-side composition- يتم تجميع ال Micro Frontend من خلال ما يسمى بال Application Shell، بحيث يتم تحميل ال Micros Frontend وعرضها في المكان المناسب اعتمادا على واحدة من الأساليب المخصصة لذلك، ويشترط وجود Entry point يتم من خلالها تحميل ملف ال HTML أو ال JS الذي يسمح لل Application Shell بالتفاعل معها وبشكل Dynamic وعرضها في المكان المخصص، ومن الأساليب المستخدمة لذلك:

- ال Iframes: وهي ال Html iframe التي نعرفها جميعا، فمن خلالها يمكننا رفع هذه ال service وأخذ ال URL وإضافته ك src داخل ال iframe، وهذه الطريقة تعزل وتمتع ال micro-frontend المختلفة من التداخل فيما بينها وتقلل من نسبة الخطأ... لكنها تجعل التواصل بين المكونات ومشاركة البيانات والتفاعلات فيما بينها أمرا معقدا، وكلما زاد عدد ال micro-service زاد التعقيد...
- ال Client Side Include: يتم هنا الاعتماد على أي من الوسائل التي تقوم باستبدال component / tags الوهمية أو تحميلها وإضافتها عند الحاجة وبشكل غير متزامن أو lazy loading، فمثلا يمكن أن يتم جلب الصفحة بناء على Ajax request / JS Fetch أو أي وسيلة أخرى مثل جلب صفحة التقييمات بعد الضغط على رؤية التقييمات... وعادة في هذه الحالة يتم وضع placeholders مكان المحتوى الذي

سيظهر ثم يتم استبداله عندما يتم إدراج ال DOM الخاص بال micro-frontend مكان ال Placeholder... وتتميز هذه الوسيلة بأنها مرنة أكثر بحيث يتم تحميل الأجزاء بشكل مستقل ودون الحاجة لتحميل الصفحة كاملة كما يمكن أن تحسن في بعض الحالات من الأداء بسبب جلب المحتويات بشكل غير متزامن، لكنها أيضا قد تبطئ من عملية التحميل في البداية وسبب ذلك الحاجة لتحميل كل جزء على حدا، كما أن عملية التفاعل بين الأجزاء المختلفة يصبح أكثر تعقيدا... هل تشعر أن ال SPA يعمل بنفس هذا المبدأ؟ وهل تشعر أن react أو anguler أو vue تقوم بتنفيذ هذا الأسلوب بشكل ما؟

أما مزايا هذا الأسلوب وعيوبه بشكل عام -والتي يمكن استنباطها مما ذكرناه فهي-:

- ميزة: سرعة ممتازة في تحميل الأجزاء خصوصا لإمكانية وضعها داخل ال CDN بسهولة.
- ميزة: يمكن تحميل الأجزاء بشكل مستقل ودون الحاجة لإعادة تحميل كل الصفحة.
- عيب: وجود تعقيد في نقل التفاعل / التواصل بين المكونات.
- عيب: في حالة ال client قد يحدث تكرار في تحميل ملفات نفس المكتبة عند الاختلاف بين الإصدارات.

طبعا، ويضاف إلى هذه العيوب والميزات ما قمنا بذكره أو الإشارة إليه أو ما هو معروف بالضرورة -بالصفحات السابقة والمتعلقة بالمبادئ ونحو ذلك- مثل مشاكل ال security المحتملة أو مشكلة bundling وكثرة الطلبات على ال network أو مزية فصل المكونات وال deploy المستقلة والقدرة على إضافة المزايا وتطويرها بشكل تدريجي... إلخ.

ثانيا: Edge-side Composition:

هذا الأسلوب قائم على مبدأ استغلال ال CDN لبناء وتجميع ال View الخاصة بنا بكافة مكوناتها ومن ثم إرسالها للمستخدم، وحتى تتم هذه العملية يتم استخدام Markup language مبنية على XML تسمى ب Edge Side Include واختصارا ESI، وبالتأكيد بناء على ذكرناه فهي ليست لغة جديدة وإنما آلية أو معيار لوصف كيف ومن أين سيتم جلب المكونات والبيانات الخاصة بهذه الصفحة...

يتيح ال ESI عمل Scale كبير في بنية الويب حتى يستفيد المطور من أكبر عدد ممكن من ال PoPs (Points of Presence) المنتشرة حول العالم والتي توفرها ال CDN مقارنة بعدد ال Data Center المحدود حول العالم.

والآن لنرى كيف يعمل هذا الأسلوب:

عند استخدام ال Edge-side Composition، يتم تقسيم الصفحة إلى مكونات (Header, footer, content...إلخ)، وكل مكون يتم تخزينه بشكل منفصل في الـ CDN. وعندما يقوم المستخدم بطلب الصفحة فإن الـ CDN يقوم بتجميع المكونات المطلوبة منه في الوقت الفعلي وقبل إرسالها للمستخدم، وحسب ال ESI فهناك مكونات يتم تعريفها على أنها ثابتة وهذه يقوم بجلبها ال CDN لمرة واحدة ثم يقوم بعمل Cache لها، وهناك نوع مغير فيقوم حينها ال CDN بجلب المحتوى من ال server الحقيقي... ومن ثم يقوم ال CDN بإرجاع البيانات مجمعة مع المكونات على شكل صفحة كاملة للمستخدم.

أما مزايا هذا الأسلوب وعيوبه بشكل عام -والتي يمكن استنباطها مما ذكرناه فهي-:

- ميزة: أداء سريع لأن التجميع يتم عند أقرب نقطة للمستخدم (أقرب Edge)
- ميزة: يقل الحمل على ال server الخاص بنا لأننا اعتمدنا على cdn.
- ميزة: بسبب وجود cdn edge كثيرة حول العالم فهذا يعني انتشار واسع ويعني قدرة على ال scale أكبر.
- عيب: لن يخدم المواقع التي تحتاج لتحديث لحظي بشكل كامل لأن ال ESI يتطلب بعض الوقت للقيام بالتحديثات.
- عيب: لا يتم تنفيد ال ESI في كل مزودي ال CDN بنفس الطريقة، وهذا يعني الحاجة إلى عمل Refactor للكود أو جزء منه عن الانتقال من مزود خدمة لآخر.

ثالثا: Server-side Composition:

هذا الأسلوب قائم على مبدأ تجميع ال micro frontend على ال server، وليس عند ال client أو ال cdn، وعملية التجميع هذه إما أن تتم على مستوى:

- ال Compile time: في هذه الحالة، يتم بناء الصفحة وتجميعها أثناء عملية ال Build، ويتم تخزينها في الـ CDN أو ال server كصفحة static... لذلك فهذا النوع مناسب للصفحات التي لا تتغير كثيرا أو لا تحتوي معلومات متغيرة بناء على المستخدم...
- ال Runtime: هنا يتم تجميع الصفحة عند كل HTTP request من المستخدم، ويتم دمج الـ Micro-Frontends مباشرة أثناء معالجة ال request قبل إرسال الصفحة

لهذا المستخدم، وهذا النوع مناسب للصفحات التفاعلية أو التي تحتوي معلومات متغيرة بناء على المستخدم مثل dashboard أو ال user profile... إلخ.

أما مزايا هذا الأسلوب وعيوبه بشكل عام -والتي يمكن استنباطها مما ذكرناه فهي-:

- ميزة: التحكم يكون مركزي في كيفية تكوين الصفحة من على السيرفر.
- ميزة: يقدم مستوى ممتاز وأداء أفضل في ال SEO.
- ميزة: تحميل الصفحة بشكل أسرع وتقليل وقت ال TTFB لأن المتصفح سيقوم بجلب الصفحة مرة واحدة ولن يحتاج لانتظار وقت لتجميع المكونات عنده...
- عيب: يعتمد بشكل كبير على ال server مما يزيد من العبء عليه.
- عيب: قد تكون تجربة المستخدم في التعامل معه أبطأ لأن كل action قد يعني جلب صفحة جديدة أو يمكل request جديد إلى السيرفر.
- عيب: وجود بعض التعقيد في التجميع مع صعوبة ال code splitting لتحميل الأجزاء المطلوبة فقط.
- عيب: نحتاج لضمان تحديث البيانات دون تأخير.

ملاحظة: يمكن الجمع بين أكثر من أسلوب حسب طبيعة ال micros التي نعمل عليها...

والآن ننتقل إلى جزئية مهمة أخرى، وهي ال Routing Micro-Frontends... فمن المهم معرفة آلية التنقل بين الصفحات لأن ذلك يؤثر بشكل مباشر على طريقة جلب وبناء / تجميع ال micro frontends.

رابعا: ال Routing Micro-Frontends:

عملية التوجيه وإدارتها تعتبر الخطوة التالي بعد نقطة التجميع التي تحدثنا عنها، لأنها ستتأثر بشكل كبير بناء على اختيارنا لآلية الجمع ومكانها... ويمكن القيام بذلك من خلال ثلاثة طرق أيضا وهي:

- ال Origin Routing
- ال Edge Routing
- ال Client Routing

شاهد الصورة F3-4

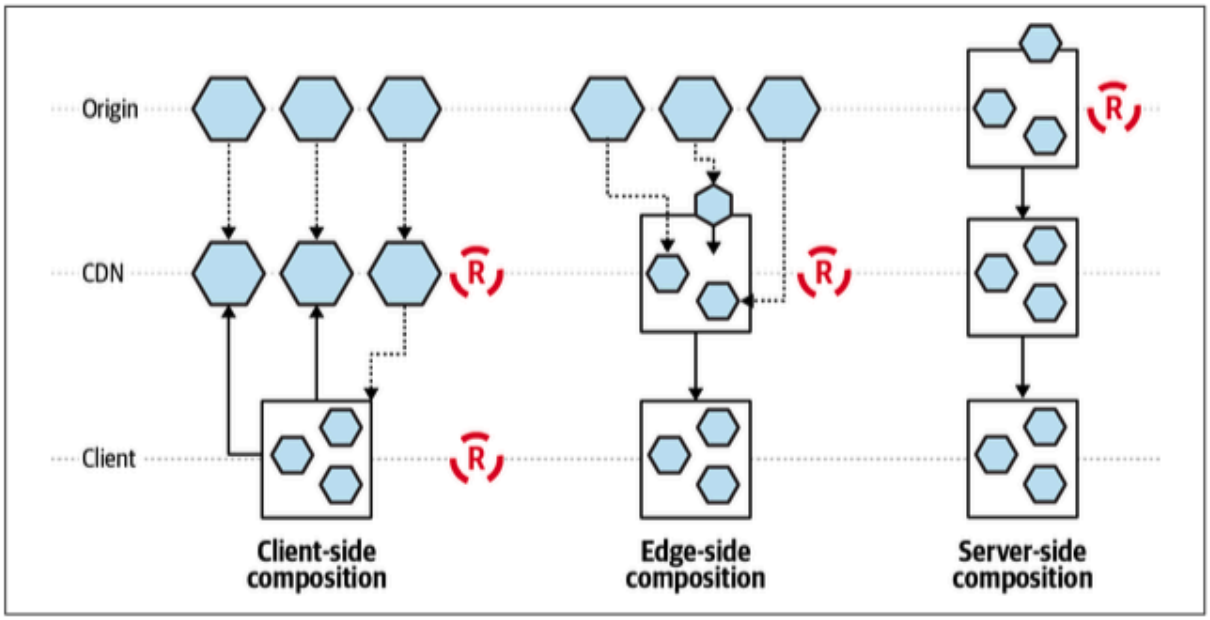


Figure 3-4. Micro-frontends routing diagram

أولا: ال routing على origin:

إذا قررنا تجميع الـ Micro-Frontends على السيرفر (Server-side Composition) فإننا مضطرون حينها لعمل ال route على ال origin server، لأن application logic بالكامل موجود على origin، وهذا كلام منطقي جدا!

وهذا الأمر أيضا سيخلق لنا مجموعة من التحديات التي يجب أن نأخذها بعين الاعتبار وهي:

- scaling an infrastructure: عند التعامل مع كمية request كبيرة في الثانية الواحدة RPS أو في حالة وجود Burst Traffic فإن إدارة عملية التوسع -Scaling- ليس أمرا سهلا، لذلك يلزم وجود آلية تضمن عمل ال horizontal scale بشكل سريع إذا ازداد الضغط على ال origin server.
- يجب أن نضمن أن جميع origin server قادرة على بناء وتجميع واسترجاع ال micro frontend للمستخدم.

قد يقول قائل، لكن يمكننا استخدام ال CDN للتقليل من هذه المشكلة، وهذا كلام صحيح لكنه ليس كافيا، والسبب في ذلك يعود لما ذكرناه من عيوب في موضوع التجميع إن كنت تذكر... فمثلا وجود بيانات dynamic تخص مستخدم معين أو يتم جلبها حسب المستخدم الذي يقوم بعمل ال request ستحد من آلية ال cache كما ستحتاج لضبط متى سيتم تحديث البيانات وضمان أن تصل المستخدم محدثة... تخيل مليون مستخدم طلب عرض صفحة ال dashboard ولكل واحد منهم بياناته الخاصة!

ملاحظة: يقصد بال Burst Traffic هو وجود زيادة مفاجئة وكبيرة في كمية البيانات المرسلة أو المستقبلة خلال فترة زمنية قصيرة وخارجة عن نطاق المألوف، لذلك عادة ما تحدث عند وجود سبب ما أدى لزيادة الضغط على الشبكة، وكثير من الأحيان ما تكون هذه الزيادة مؤقتة ضمن فترة زمنية قصيرة، أو مفاجئة وغير منتظمة أي تحدث دون سابق إنذار... ومن الأمثلة على ذلك حصول حدث معين صار (Trend) فأدى ذلك لطلب صفحة معينة من موقعك! أو أنت قمت بعمل إعلان أو مسابقة لمدة ساعتين... ومن أمثلة ذلك مثلا بث المباريات خلال ساعة معينة...

ملاحظة ٢: ال RPS هي اختصار ل Requests Per Second، وهي مقياس يعبر عن عدد الطلبات التي تأتينا في الثانية الواحدة والتي يمكن للنظام / التطبيق أن يتعامل معها! وهذا يشمل ال server, api, db... إلخ، وهذا الأمر مهم لتحديد ظروف العمل المناسبة للنظام ومتى سنحتاج إلى تحسين ال resource وما هي ال resource المناسبة للنظام الخاص بنا ومتى سيبدأ النظام عندنا بالتباطؤ ونحو ذلك... وهناك علاقة بين ال RPS وال Burst Traffic، فعدد ال RPS يرتفع بشكل مفاجئ عند حدوث Burst Traffic.

ثانيا: ال routing على ال edge:

عند اختيار ال composition على مستوى الـ Edge، فإن الـ routing سيعتمد على ال Page URL ببساطة، هذه الروابط ستكون static، وسيقوم الـ CDN بإرجاع الصفحة

المطلوبة عن طريق تجميع الـ Micro-Frontends باستخدام تقنية Transclusion على مستوى الـ Edge، وال Transclusion هو مفهوم يشير إلى إدراج محتوى أو جزء من محتوى عند وصول request من المستخدم، وهو ما ذكرناه عندما تحدثنا عن edge composition من أن عملية تجميع الأجزاء المطلوبة تتم بالوقت الفعلي على CDN عند ال request.

ببساطة، سيقوم المستخدم بطلب الصفحة التي اسمها example.com/about وسيقوم ال cdn بجمع الأجزاء الخاصة بهذه الصفحة عنده ومن ثم إرجاعها للمستخدم.

وهذا ينشىء لنا مجموعة من التحديات أيضا أهمها صعوبة وتعقيد إنشاء Smart Routing لأن عملية ال routing تعتمد على static url.

ثالثا: Client-side Routing:

في هذا الأسلوب فإن عملية ال routing تتم داخل المتصفح مباشرة، ويقوم المتصفح بجلب ال micro frontend المناسبة عند تغيير ال url، وهذا يعني اعتماد هذا الأسلوب على ال User State، فإذا كان المستخدم قد قام بتسجيل الدخول فسنقوم بجلب الأجزاء المناسبة له، وإذا كان مستخدم جديد فسنقوم مثلا بتحميل ال homepage...

إن ال Application shell هو من سيتولى هذه العملية، فمثلا لو اعتمدنا أن ال application shell سيكون SPA مثل react فإنه من خلال ال react router سنقوم بجلب ال micro frontend بناء على ال route الذي وصلنا له... ويعتبر هذا الأسلوب هو الأسلوب الأفضل والأمثل عند وجود complex routing مثل اعتمادية ال micro frontend على ال authentication وال Geolocation أو أي logic معقد... أما في حالة ال multi pages website فيمكننا تحميل ال micro frontend من خلال ال Client-side Transclusion، وهو يشبه تقريبا الآلية التي ذكرناها بال edge (ESI).

والسؤال الآن، متى نستخدم هذا الأسلوب؟

- إذا كان لدينا فريق FE ذو مهارات عالية لأن الاعتماد سيكون على ال FE في عملية ال routing ودون الرجوع لل server.
- إذا أصابنا الخوف من مشاكل ال scalability، لأن ال routing هنا يتم على المتصفح.

ملاحظة: جميع ما ذكرناه من طرق ليست متضادة، بل يمكن الجمع بينها أو دمجها للحصول على أفضل نتيجة ممكنة وكل ذلك حسب الحاجة، وهذ ما سنراه لاحقا بإذن الله تعالى... لكن المهم الآن أن تحدد آلية ال routing داخل النظام الخاص بك لأن ذلك سيؤثر بشكل كبير على طريقة تطويرك لل micro frontend. (لاحظ أهمية التسلسل بالشرح والمفاهيم).

فائدة

فلْتعلمْ أن ذِكْر الله -سبحانه وتعالى- في جميع الأحوال والأوقات فيه من الخير ما فيه، وأعظم ما يكون للقلب من الذكر هو تقوية القلب! فإن قوي القلب قوي البدن معه وسانده، ولْتعلمْ أن هذا الأمر أعظم وأوجب عند الخوف! ولْتعلمْ أن من يتحرك لسانه وقلبه بذكر الله في جميع الأحوال والأوقات، أثناء عمله وراحته، يُغبط على ذلك! ومما يتمناه المرء ويدخل السرور على القلب أن يجد لسانه ذاكرًا لله -سبحانه وتعالى- حتى وهو سارح، فوالله إن فيها فرحة لما يدل عليه حال الفؤاد واللسان وما يسمعه الإنسان...

- كتاب إلى الجنة زمرا، الصفحة ١١٦

خامسا: ال Micro-Frontends Communication

في عالم مثالي، لن تحتاج الـ Micro-Frontends إلى التواصل مع بعضها البعض؛ لأنها ستكون مستقلة بشكل كامل... لكن الواقع غير ذلك! خصوصا عند وجود أكثر من micro-frontend داخل نفس الصفحة، والسبب في ذلك يعود لحاجتنا في بعض الأحيان إلى إشعار الـ Micro-Frontends الأخرى بحصول حدث معين أو تفاعل معين، أو نحتاج إلى إرسال معلومة معينة إلى ال micros الأخرى، وهذا كله يدعونا لأن نفكر كيف يمكننا إرسال البيانات / التفاعلات بين هذه ال micros، وزد على ذلك أن وجود أكثر من micro-frontend في النفس الصفحة سيزيد من صعوبة تجميع هذه ال micros وجعلها وحدة واحدة متجانسة = فهذا يعني الحاجة إلى قدرة على إدارة هذه المكونات بشكل احترافي، وبما أننا لا نريد أن ننتهك المبادئ التي تكلمنا عنها سابقا؛ فلا يجب أن تكون هذه ال micros مدركة لل micros الأخرى... أي لا نريد كسر مبدأ الـ Independent Deployment ولا نريد كسر مبدأ العزل بينها (Isolation)...

من الأمثلة على ذلك:

- إذا قام المستخدم بتسجيل الدخول فكيف ستخبر ال micros المهتمة بأن المستخدم الآن صار auth... فمثلا ال header سيعرض اسمه وال main سيعرض ال dashboard... إلخ.

- إذا وصلك إشعار فتريد تغير رقم الإشعارات على ال icon الخاصة به وتريد عرض الرسالة الجديدة في ال chat و popup ستظهر في الصفحة الرئيسية تخبرك بوصول رسالة جديدة فقم بفتح الشات...

هذه الحالات تقودنا للبحث عن خيارات ال communication الممكنة والتي ستساعدنا في الربط بين هذه ال micros، وهي كثيرة نذكر منها ^^

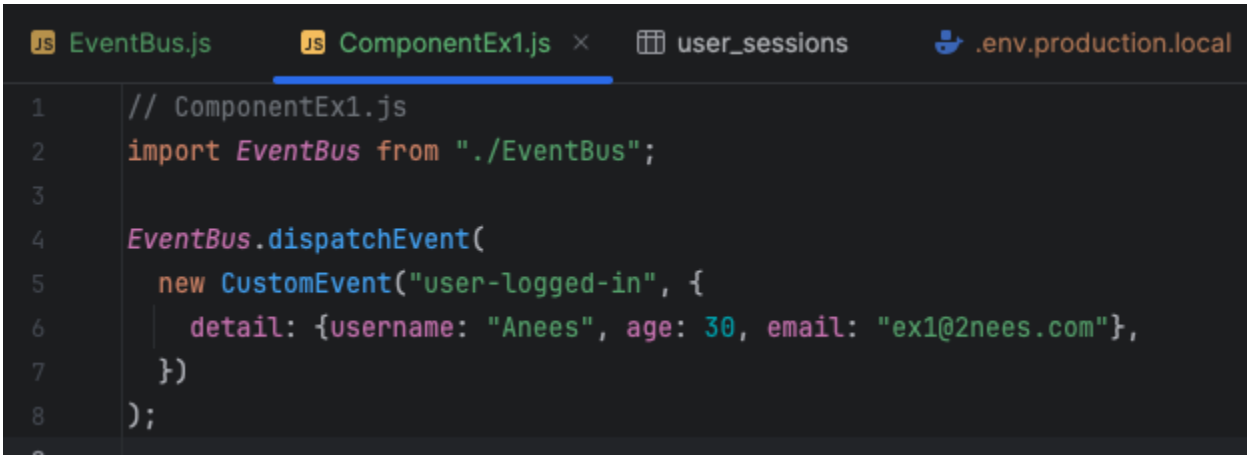
1. ال EventBus:

هو نمط تصميم (Design Pattern) يستخدم للتواصل بين مكونات مختلفة في التطبيق بشكل غير متزامن (Asynchronous) ودون وجود ترابط مباشر بينها (Decoupled Communication)، أو بعبارة أخرى هو communication channel بين ااال component المختلفة أو نمط يقوم على تطبيق مفاهيم ومبادئ ال publisher/subscriber pattern... هذا المفهوم الجميل يسهل عملية إرسال واستقبال الـ events بين المكونات المختلفة، ودون الحاجة لتمرير البيانات بشكل مباشر مثل ال Props في react، ودون الحاجة لتغليفها ب Context مثال ContextApi في ال React.

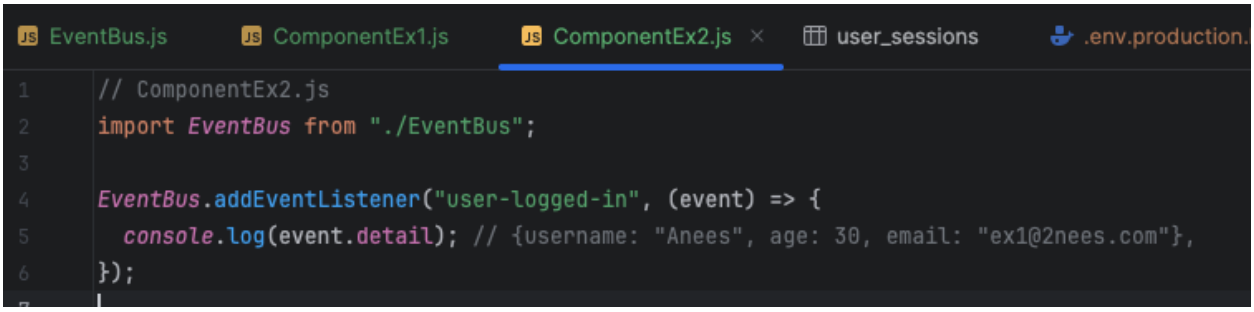
مثال عملي: لو حدث تسجيل دخول فإننا سنقوم بإشعار جميع ال microfrontend بذلك عبر ال EventBus، ويمكن تطبيق ذلك ببساطة من خلال الجافا سكربت بالطريقة التالية -مثال مختصر-:

أولا نقوم بإنشاء ال EventBus باستخدام ال Event

ثم نقوم بإرسال الحدث من إحدى ال micros مثل هذه الصورة، والتي قمنا بإرسال معلومات المستخدم الذي قام بتسجيل الدخول.



ثم ستجد أن ال micros الأخرى المهتمة بها جالسة ومنتظرة بفارغ الصبر هذه الرسالة الكريمة كما في الصورة التالي:



ملاحظة: في الـ EventBus يتم عمل ال inject له من خلال Application shell إلى ال micros مما يسمح لها بال listen وال emit، شاهد الصورة F3-5.

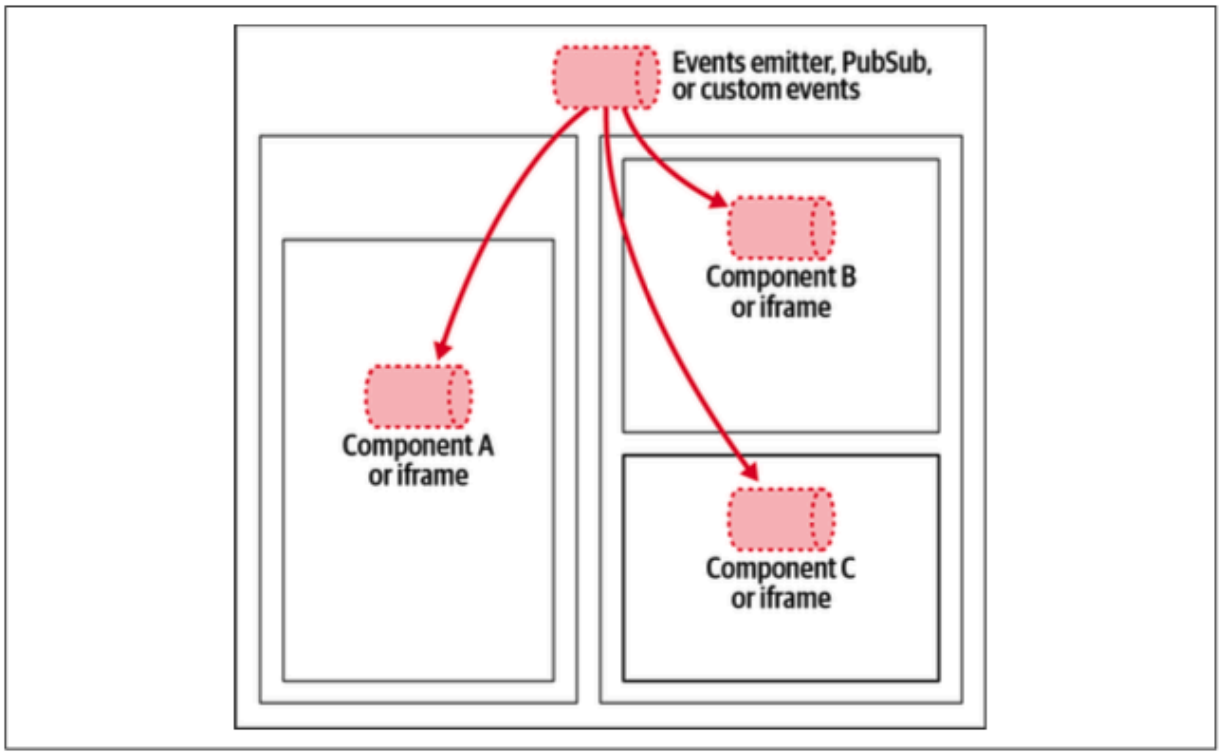


Figure 3-5. Event emitter and custom events diagram

2. ال Custom Event:

وذلك من خلال استخدام ال CustomEvent الموجودة بالجافا سكربت مباشرة والتي تستخدم للتفاعل مع Event مخصصة اعتمادا على ال Dom أو EventTarget لإرساله، في حالتنا هذه سنحتاج إلى استخدام ال window حتى تتمكن بقية ال micros من الوصول إليها... لكن هذا الخيار لن يعمل مع ال iframes لأن كل iframe لديه ال window object الخاص به.

```
// ComponentEx3.js

const myCustomLogEvent = new CustomEvent("user-logged-in", {
  detail: {username: "Anees", age: 30, email: "ex1@2nees.com"},
});

window.dispatchEvent(myCustomLogEvent);
```

3. ال Web Storage:

يمكننا استخدام ال Web Storage لحفظ البيانات وتمريرها بين ال micros المختلفة، فهناك أنواع من البيانات قد نحتاجها دوما حتى لو قمنا بتحديث الصفحة وهناك معلومات نرغب بالحفاظ عليها حتى تنتهي ال session الخاصة بالمستخدم، لذلك يمكن استخدام ما يناسب المشروع من ال storage مثل ال:

a. ال LocalStorage

b. ال SessionStorage

c. ال Cookie

وهذا جميل طالما أننا نتعامل مع هذه المكونات ضمن نفس ال subdomain، لأنه سنتمكن من جلب المعلومات كما نحب في أي micros نريدها...

شاهد الصورة F3-6:

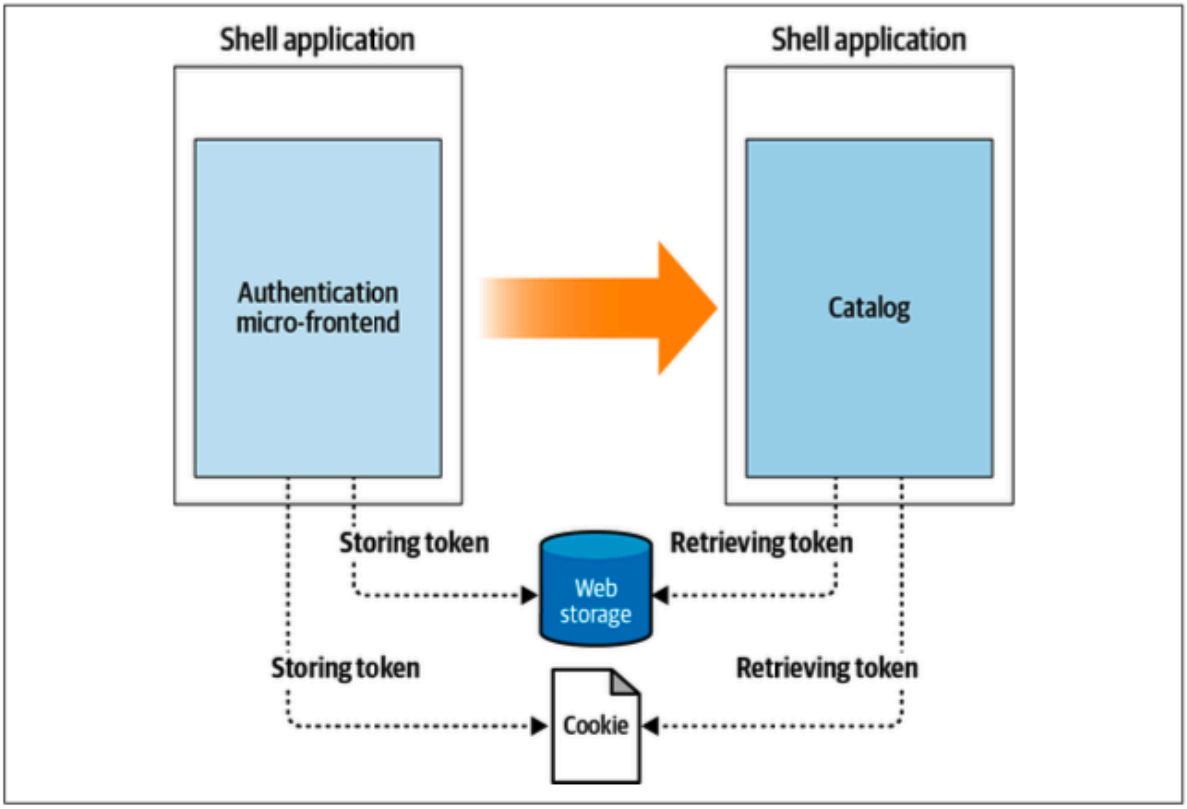


Figure 3-6. Sharing data between micro-frontends in different views

4. ال Query Strings:

ببساطة يمكننا أيضا استخدام ال Query String لإرسال المعلومات من ال url والذي ستتمكن من خلال قراءته كل ال micros من الوصول للمعلومات... لكن يجب أن نأخذ بعين الاعتبار القواعد والقيود الخاصة بال URL وال GET Reqeust حتى لا نستخدم التكنولوجيا بطريقة خاطئة ^^... "مش أشوفك حاط كلمة السر مثلا بال url" ^_*

مثال: https://2nees.com/listing-courses?v=1... ال v1 تمثل ال Query

String هنا.

والآن نأتي إلى الجزئية المهمة أو الأهم بعد ذكرنا لآليات التجميع والتقسيم، وهي تلخيص ما سبق بشكل واضح حتى نستطيع اتخاذ القرار الصحيح لتقسيم التطبيق إلى Micro Frontend، شاهد الجدول F-table 3-2:

Table 3-2. Micro-frontends decisions framework summary

Micro-frontends definition	Composition	Routing	Communication
Horizontal	Client side Server side Edge side	Client side Server side Edge side	Event emitter Custom events Web storage Query strings
Vertical	Client side Server side	Client side Server side Edge side	Web storage Query strings

هذا الجدول يمثل خلاصة ما تعلمناه هنا حول مفاتيح اختيار اتخاذ القرار، والتي تشمل ال identifying (المحور الأول) وال composing (المحور الثاني) وال routing (المحور الثالث) وال communication (المحور الرابع).

في هذا الفصل -بفضل الله تعالى- أنهينا المفاهيم النظرية وال High-Level Architectures المتعلقة بتطوير وتصميم ال Micro-frontend، هذه المعمارية ليست حكرا على مجال الويب فقط، بل تستخدمها المؤسسات في تطبيقات Desktop وال Smart TV وغيرها... والآن يمكننا أن ننتقل بسلام من التطبيق النظري إلى التطبيق العملي... توكلنا على الله.

الفصل الثالث: Discovering Micro-Frontend Architectures

في الفصل السابق، تعلمنا عن اتخاذ القرارات (Decisions Framework) آلية تحديد وتعريف الأجزاء التي ستكون Micro-Frontend وآلية التواصل والتجميع والتوجيه... وفي هذا الفصل، سنستعرض الخيارات المختلفة للمعماريات (Architecture Choices) والتي تحدثنا عنها سريعا في الفصل السابق لكن بشكل أعمق ومع تطبيق ما تعلمناه حتى الآن بشكل تقني واضح ومدعوم بأمثلة عملية.

والآن، لتطبيق إطار اتخاذ القرار الذي تعلمناه وبناء على الخصائص الخاصة بالمشروع، سيكون قرارنا الأول باعتماد ال Vertical Split أو ال Horizontal Split، شاهد الصورة F4-1:

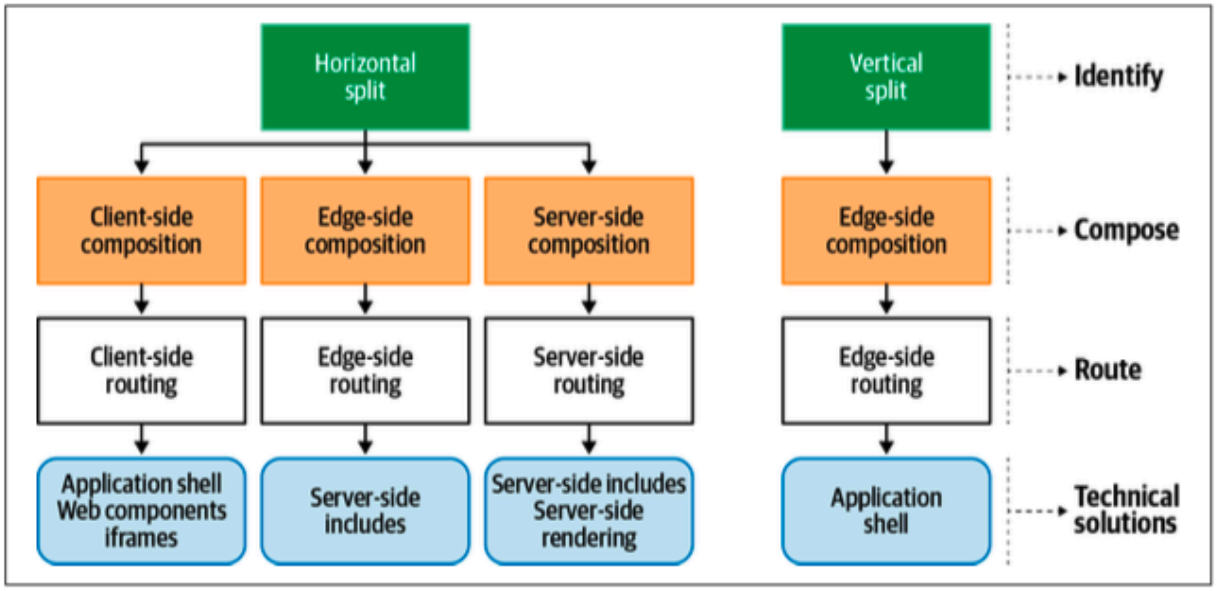


Figure 4-1. The micro-frontends decisions framework

إذا لنتخذ القرار الخرافي ^^ الأول باعتماد ال Vertical Split لمعمارية ال micro-frontend الخاصة بنا، وعليه:

١. ال Vertical Split:

لقد تحدثنا عن هذا النوع من التقسيم في الأجزاء السابقة، لكن سندخل ببعض التفاصيل التقنية المتعلقة بخصائص هذا النوع من التقسيم مع طريقة تطبيقه المتوقعة، ونبدأ مع أول جزء من الصورة F4-1 والتي يظهر فيها بشكل جلي مسار واحد فقط لل Vertical Split، وهو مسار ال Client Side، وهو الخيار الذي اعتدنا عليه كمطورين FE عند برمجتنا صفحات SPA.

ولأننا نستخدم هذا النوع من التقسيم فإننا يمكننا استخدام الأدوات التي اعتدنا عليها والخروج بنتائج اعتاد عليها المستخدمون أيضا في مواقع ال SPA، وهذا يعود بشكل أساسي لتقديم تجربة متسقة للمستخدمين بين الصفحات... لكن كيف يمكننا القيام بجمع هذه ال Micros ونحن في Vertical Split؟

لأن جميع التجارب التي تقبع تحت هذا التقسيم تندرج تحت ال Client Side فإن ال Application Shell سيكون هو الحل المباشر لهذه المشكلة -يمكنك الرجوع للأجزاء السابقة إذا لم تقرأ ما كتبناه عنها-...

سيقوم ال Application Shell هنا بتجميع ال Micro Frontend عند الطلب، أي عندما يقوم المستخدم بالتنقل بين الصفحات، الصفحة المطلوبة (route) سيحدد من هو ال micro-frontend الذي سيتم جلبه، وهذا ما يطلق عليه ال Global Route، لأنه يكون على مستوى ال App Shell، ولأننا نتحدث عن Vertical Split فالعلاقة لن تكون سوى One to One بين ال App Shell وال Micro Frontend، أي micro واحدة فقط في كل صفحة، أما عمليات التنقل بين الصفحات والتي لا تغير ال micro التي تم تحميلها فهذه يطلق عليها ال Local Route، وهي تكون مبنية وموجودة داخل ال micro نفسها...

شاهد الصورة F4-2 والتي تعرض ال Application Shell مع ال Global Route

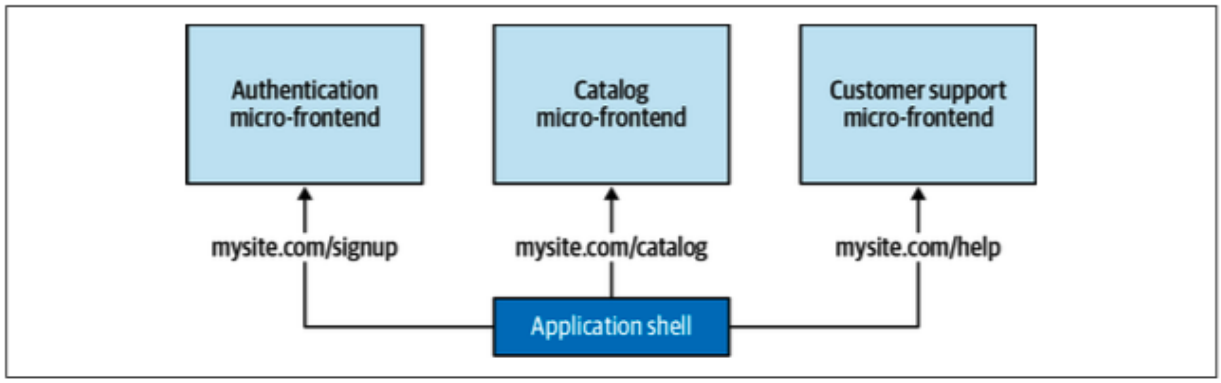


Figure 4-2. The application shell is responsible for global routing between micro-frontends

وشاهد الصورة F4-3 والتي تعرض ال Local route داخل Micro frontend تم تحميلها داخل ال Shell.

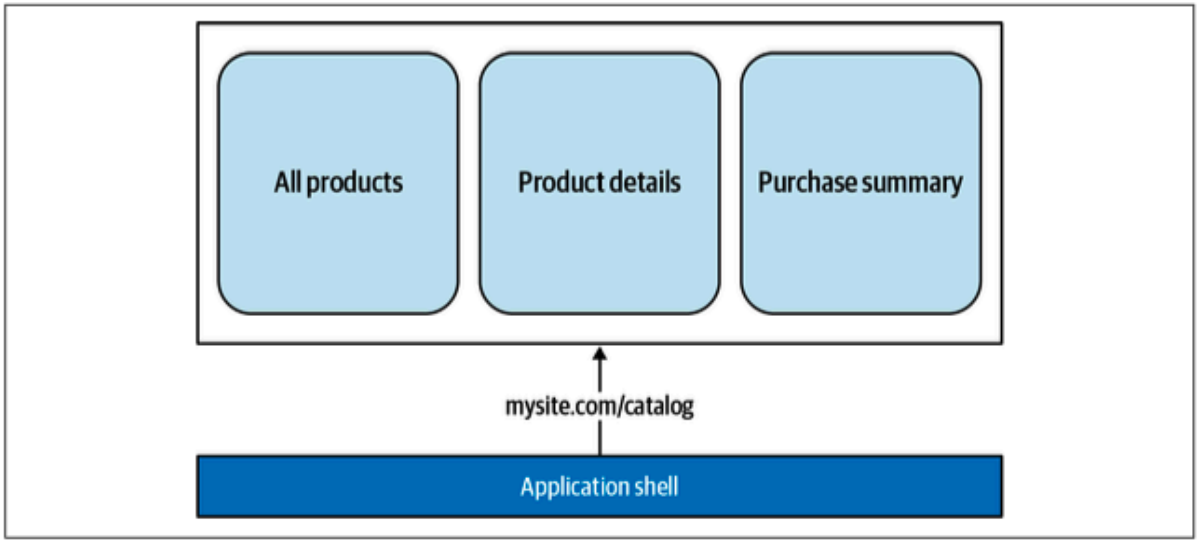


Figure 4-3. A micro-frontend is responsible for routing between views available inside the micro-frontend itself

يبدو أن الفكرة أصبحت سهلة وجميلة بعد أن اتضحت المفاهيم ^^... لنتابع ولننظر تسلسل التنفيذ أو التطبيق في هذا التقسيم:

لتنفيذ معمارية ال Micro-Frontend باستخدام الفصل العمودي سيقوم Application Shell بتحميل ملف HTML أو JavaScript كنقطة بداية (Entry Point) وفي نفس الوقت يجب ألا يقوم ال Application Shell بمشاركة أي logic متعلق بال Business Domain مع أي Micro-Frontend آخر -وهذا تكلمنا عنه سابقا وأشرنا لخطورته وأهميته-، ومن ثم ستتم مشاركة البيانات بين ال micro service من خلال الطرق التي تكلمنا عنها سابقا مثل ال Web Storage وال Query String، وبهذا تصبح الفكرة العامة (المجملة) شبيهة من ناحية التطبيق مع ال Horizontal Split...

ملاحظة مهمة: يجب أن يكون ال Application Shell محايدا من حيث التقنية، ويقصد بذلك ألا يتم استخدام تقنية معينة لتطوير وبناء ال App Shell، وذلك ليدعم التطور المستقبلي للنظام بسهولة ودون تعقيد ودون محددات تحد من التطوير أو قد تسبب مشاكل معينة لارتباطها بتقنية معينة، وهذا يعني تفضيل استخدام ال Vanilla JavaScript في ال App Shell في حالتنا كمطورين ويب ^^.

ملاحظة ٢: ال Vertical split حسب الوصف الحالي سيقدم لنا خدمة جميلة جدا بأن يخلصنا من مشاكل ال Isolation، فمثلا لن نخاف من حدوث مشكلة تداخل أو تعارض ال css بين ال micros المختلفة ^^.

أما إذا كنت تريد اتخاذ القرار الثاني الخرافي ^^، وستذهب لاعتماد ال Horizontal Split فعليك اللحاق بي ...

فائدة

إياك يا أخي أن تقف مع الظالمين والمفسدين في الأرض، وإياك أن تدافع عنهم فيما يقومون به مِن ظُلم وإفساد، بل عليك أن تمنع الظالمين عن ظلمهم، وأن تنصر المظلوم، وأن تُظهر للناس فساد المفسدين لا أن تدافع عنهم!

- كتاب إلى الجنة زمرا، الصفحة ١١٦

٢. ال Horizontal Split:

لقد تحدثنا عن هذا النوع أيضا من التقسيم في الأجزاء السابقة، لكن سندخل ببعض التفاصيل التقنية المتعلقة بخصائص هذا النوع من التقسيم مع طريقة تطبيقه المتوقعة، ونبدأ مع ثاني جزء من الصورة F4-1.

هذا النوع من التقسيم وجوده قد يكون ضروريا إذا كان لدينا Business Subdomain سيتم استخدامه أكثر من مرة في أكثر من مشهد (مكان) في واجهات متعددة، وبذلك تصبح لدينا حاجة ملحة لإعادة استخدام هذا النطاق داخل المشروع باعتباره جزء أساسي ومحوري...، كما أن هذا النوع من التقسيم مناسب عند وجود حاجة لتحسين ال SEO وتريد استخدام ال SSR، أو إذا كان لديك الكثير من المطورين الذين يعملون معا وتحتاج لتقسيم المهام اعتمادا على جزئيات محددة ودقيقة من ال subdomain، أو إذا كان المشروع الخاص بك MultiTenant وهناك حاجة لتخصيص بعض الواجهات لبعض ال client...

بناء على ذلك، ولأننا اتخذنا قرار التقسيم من خلال ال Horizontal فإننا سنعود لواحدة من ثلاثة أفكار لتجميع هذه المكونات -والتي تحدثنا عنها- سابقا، لكن نذكرها هنا مجددا مع إيضاح وقت الاستخدام المناسب لكل واحد منهم أثناء التقسيم:

- Client-side Composition:

- هذا الأسلوب مفيد إذا كان فريقك لديه خبرة جيدة في بيئة تطوير ال frontend ecosystem.
- ومناسب في حالة ارتفاع عدد الزيارات، لأنه يسمح لك بتخزين ال Micro-Frontends مؤقتا (Caching) بسهولة، ويتيح لك نقل المحتوى من خلال استخدام ال CDN بسهولة أيضا، وهذا كله مما يجنبك تحديات ال scaling.

- Edge-side Composition (على مستوى الـ CDN):
  - هذا الأسلوب مناسب عندما يكون لديك محتوى static وعليه الكثير من الزيارات.
  - ومفيد في حال رغبتك بتفويض ال CDN بإدارة عملية ال Scaling والتخلص من هذا العبء.

لكن احذر، فلهذا الأسلوب تحديات خطيرة مثل عملية التطوير المعقدة، كما أن هناك limitation من بعض مزودي CDN ولا يدعم كل مزودوا ال CDN هذه الخدمة! وعادة ما يُستخدم هذا الأسلوب في المشاريع التي لا تحتوي على محتوى شخصي مثل الكتالوجات [٨٨].

- Server-side Composition:

- هذا الأسلوب مناسب إذا كنت تريد أعلى درجة تحكم ممكنة في ال output، وهو مثالي للمواقع التي تحتاج للفهرسة بشكل سريع وواسع (highly indexed websites) مثل مواقع الأخبار أو التجارة الإلكترونية.
- كما أن هذا الأسلوب مناسب للمواقع التي تحتاج إلى أداء عالي وجيد، ولعل من أشهر الأمثلة على مواقع استخدمت مثل هذا النوع من التقسيم PayPal و American Express.

والآن كما تعلمنا سابقا، سننتقل لاختيار آلية ال Routing، ومع أنه نظريا يمكنك استخدام أي نوع من أنواع ال Routing التي تحدثنا عنها مع آلية التجميع التي اخترتها، إلا أنه عمليا أو من الشائع استخدام وربط استراتيجية ال routing بحسب نوع ال composition المختار، فمثلا لو قمنا باستخدام ال Client-side Composition فإن ال Routing غالبا ما سيكون عند ال client، وفي هذه الحالة يمكننا أن نفصل logic معين على ال edge حتى وإن كنا نستخدم ال client routing وذلك لحل مشاكل مهمة مثل الحاجة لوجود Canary Release أو لتحسين محركات البحث! أما إذا استخدمت ال Edge-side Composition فسيكون لديك Html page مرتبطة بكل view، وبناء على ذلك سيتم بناء هذه الصفحة وتجميعها على ال cdn من عدة micros، إما إذا استخدمت ال Server-side Composition فإنك بالتالي ستقوم باستخدام ال Server-side routing والذي يقوم ببساطة على تحديد ال html template لكل route وبالتالي يتم جلب الصفحة وتجميعها على ال server ومن ثم إرسالها لل client، وكما قلنا عند ال client، يمكن هنا استخدام ال edge لتحسين الأداء وحل بعض التحديات مثل ال Canary Release.

ملاحظة: يقصد بال Canary Release هي عملية إطلاق لميزة معينة بشكل تجريبي لعدد قليل من المستخدمين أو لفئة محددة لدراستها والتأكد منها واكتشاف الأخطاء الممكنة قبل تطبيقها ونشرها على جميع المستخدمين، وسبب ارتباط هذه الملاحظة في نقطة ال client routing أن هذه العملية لو كانت على مستوى Application Shell نفسه قد تكون عملية صعبة ومعقدة أو ستقوم بجعل الكود "ملوثا *ـ* Code pollution"... لذلك، يمكننا مثلا ببساطة أن نقوم من خلال ال edge بتحميل الإصدار 1.4.0 أو 2.0.0 لنسبة معينة من المستخدمين...

ملاحظة ٢: ال Code pollution هو مصطلح شائع يشير لوجود تلوث في الكود نابع من تعقيد أكثر من اللازم ولا داع له أو وجود logic ليس له علاقة بالوظيفة الأساسية لما نعمل عليه، أو أنه مليئ بالشروط الخاصة والاستثناءات وال temp code لتغطية مواقف معينة فقط! وكل هذا التلوث سيزيد من صعوبة فهم الشيفرة البرمجية وبالتالي صعوبة صيانتها مع إمكانية خطأ أكبر.

ملاحظة ٣: قد تتعجب من طرح نقطة تحسين ال SEO إذا قمنا باستخدام ال edge في بعض الحالات مع ال client، مع أن المواقع الخاصة بنا غالبا هي SPA... فكيف ذلك؟! والجواب ببساطة يكمن في جمال وجود بعض الآليات أو الأدوات أو الطرق التي تضمن ذلك من خلال النظر إلى طبيعة من يطلب الصفحة هل هو مستخدم أو bot -مثل جوجل-؟ إذا كان bot سيقوم ال Edge بتحميل الصفحة عنده وإعطاء ال googlebot

هذه الصفحة مع محتواها مما يحد من مشكلة صفحة ال Html الفارغة (أي بدلا من أن ينتظر ال bot وقت تحميل صفحة ال spa حتى يتم وضع المحتوى بصفحة ال html يقوم بذلك ال cdn مجهزا إياها للفهرسة... وهناك أكثر من آلية للقيام بذلك)، ومن الأدوات المستخدمة لذلك Cloudflare Workers.

والآن، ما هي الخيارات المناسبة لتنفيذ ما ذكرناه من تفاصيل تقنية؟
والجواب يمكن إيجازه بما يلي:

- إذا كان الخيار Client Composition & Routing فإن ال Application shell سيقوم بتحميل عدة micros في نفس ال html view، وبالتالي يمكننا استخدام ال Webpack module federation أو ال iframe أو ال Web Components لتغطية هذه المتطلبات.

- في حالة استخدام ال Edge Composition and routing، فهذا يعني أننا سنستخدم ال ESI.

- في حالة استخدام ال Server side composition & routing فهذا يعني إمكانية استخدام ال SSR لهذا الغرض والذي سيعطي أيضا قدرة كبيرة وعالية على التحكم.

والآن وصلنا لآخر نقطة ومن أكثرها أهمية ^^ لأن خطأك هنا قد يدمر ما بنيناه حتى هذه اللحظة وهي آلية التواصل بين ال micros في هذا التقسيم... فإذا قلت يمكننا استخدام ال

state لمشاركة المعلومات بين ال micros لأننا horizontal فهنا سأقول لك انتظر لأن هذا الذي سيدمرنا -قد تكون هناك حالات مختلفة تسمح بذلك، لكن الحديث هنا بشكل عام عن الشائع-^^... إن استخدام ال state في مثل هذه الحالة يعد antipattern، والحل يكون من خلال استخدام إحدى الطرق التي تحدثنا عنها سابقا، فمثلا يمكننا استخدام ال Event أو ال Custom Event لإرسال البيانات أو التواصل، أو يمكن استخدام ال query string في بعض الحالات للبيانات المؤقتة مثل عرض صفحة منتج معين أو ال Web Storage لتفضيلات المستخدم...

والآن، ننتقل لجزئية مهمة وهي: Architecture Analysis

:Architecture Analysis

لقد تحدثنا عن المعماريات المستخدمة أو المتبعة بال micro frontend، وقد أشرنا لبعضها إشارة سريعة... لكن كيف يمكننا القيام بتحليل حقيقي يقيم الخصائص الموجودة عندنا في النظام وطبيعة المؤسسة مع المعمارية التي سنختارها؟ والجواب ببساطة من خلال فهم هذه النقاط ومحاولة إسقاطها على المعمارية الموجودة ومن ثم اختيار أقل المعماريات سوءا.

لحظة لحظة! "اختيار أقل المعماريات سوءا"!!!

لا تتعجب يا صديقي، نعم، أقل المعماريات سوءا، فالمعمارية المثالية (perfect architecture) لا وجود لها إلا في أحلامنا الوردية ^^ أما على أرض الواقع فإننا نقايض عوامل متعددة التقنية + والاجتماعية (Sociotechnical) مع الخصائص في كل معمارية والتي ستقدم لنا أفضل الحلول الممكنة مقارنة بغيرها من المعماريات، كما أن فكرة نسخ معمارية معينة من مشروع آخر أو من منشور على قبيلة ^^ كما هي ثم تطبيقها على مشروعك الخاص قد تكون حركة لها عواقب وخيمة، وإنما يجب النظر هل فعلا هذه المعمارية والتي تمت مشاركتها أو نرغب في نسخها مناسبة لما نحتاجه؟ وهل هي أفضل خيار لنا اعتمادا على العوامل المتعددة التي نعيش فيها ضمن المؤسسة؟!

ولقد ذُكر ذلك في كتاب: "Fundamentals of Software Architecture" للمؤلفين "Neal Ford" و "Mark Richards": "لا تسع أبدا لأفضل معمارية، بل لأقل معمارية سوءا"! واعتمد على المقايضات (Trade Off) للوصول لمعماريتك! وهذا يذكرنا مجددا في فكرة جميلة نعيشها مع المشاريع الضخمة والتي أشرنا إليها سابقا في إحدى المقالات وهي: CAP Theory، والتي تبرز فيها دور المقايضة للمفاضلة بين الخصائص المختلفة... ولذلك يقول Luca Mezzalira -مؤلف الكتاب-: "لا يوجد شيء اسمه صحيح أو خاطئ في المعمارية هنا، بل يوجد أفضل مقايضة تقوم بها لتناسب ال context الخاص بك".

وخلاصة هذا الاستطراد هي:

- لا توجد معمارية مثالية يمكنك تطبيقها، فلكل خيار إيجابياته وسلبياته، وهنا تظهر براعتك في المقايضة لتحقيق أفضل توازن ممكن بين الخصائص... وتذكر الحل الأفضل هو الذي يلبي احتياجاتك بأقل ضرر ممكن، وليس المثالي نظريا.
- ال context هو أساس الانطلاق لمعماريتك، وال context يشمل المؤسسة والفرق التي تعمل في هذه المؤسسة.
- لا تقم بتقليد ونسخ كل شيء أو تطبيق ما رأيته منشورا هنا أو هناك حول معمارية معينة دون دراستها دراسة حقيقة والتحقق من كونها خيارا مناسب لك أم لا!

ملاحظة: ال Sociotechnical هي كلمة مكونة من شطرين، الأول هو Socio وتعني اجتماعي وفيها كل ما يتعلق بالبشر وعلاقاتهم (أي الأشخاص أنفسهم وثقافتهم وقيمهم وأدوارهم وعلاقاتهم...) والثاني هو Technical وتعني تقني وفيها كل ما يتعلق بالأدوات التي يستخدمها البشر في هذه المنظومة (أي التكنولوجيا المستخدمة والبرمجيات والعمليات...)، أما المعنى المراد من هذا المصطلح والذي يمكن فهمه من الجمع بين الشطرين فهو: أننا لا يمكن أن ننظر إلى التكنولوجيا فقط بمعزل عن الناس الذين يستخدمون هذه التكنولوجيا أو بمعزل عن البيئة التي يعملون فيها. فالنظام يكون ناجحا إذا كان هناك توازن وتكامل بين الجانبين الاجتماعي والتقني، وكمثال عملي: إذا كان لديك تطبيق لتوصيل الطعام فالجانب التقني يتكون من التطبيق ذاته وآلية الدفع ونظام إدارة الطلبات وسيكون الجانب البشري هم مستخدمو التطبيق والسائقون وخدمة العملاء والهيكل الإداري... إلخ.

والآن نعود بعد هذا الاستطراد لموضوعنا، وهو كيف يمكننا تحليل النظام حتى نختار المعمارية المناسبة؟ والجواب هو من خلال وضع تقييم من ١ إلى ٥، بحيث يمثل الرقم ١ أن هذه النقطة غير مدعومة بشكل جيد أو تشير إلى اعتمادية قليلة في هذه المعمارية، في حين يشير الرقم ٥ إلى أنها تعد من أقوى الجوانب في هذه المعمارية أو بحاجة إلى اعتمادية عالية...

أما الجوانب التي سيتم وضع التقييم لها فهي:

- ال Deployability: سهولة نشر ال micro-frontend وموثوقيتها (استمرار عمل النظام دون فشل) على environment معينة.
- ال Modularity: وهذه تشير إلى مدى سهولة إضافة أو إزالة واحدة من ال micro-frontend ومدى سهولة تكاملها مع المكونات المشتركة من قبل ال micro-frotnends الأخرى.
- ال Simplicity: مدى سهولة فهم النظام والتعامل معه = وهذا يعني عملية برمجية أسهل = وهذا يعني أنها أسهل في الفهم والتعامل، وبذلك يكون التقيم أعلى اعتمادا على مدى سهولة الفهم والتعامل ومنها سهولة تطبيق الأفكار برمجيا.
- ال Testability: مدى دعم قابلية الاختبار، هل هي قابلة للاختبار بدرجة عالية ضمن نفس السياق بحيث يصبح اكتشاف الأخطاء أسهل من خلالها أم لا، إذا كان أسهل أو يحقق هذه القابلية بشكل قوي فهذا يشير لجانب قوة.

- ال Performance: مدى قدرة ال Micro-Frontend على تحقيق أفضل تجربة المستخدم مستخدم كما توصفها مؤشرات ال Web Vitals.

  ملاحظة: ال Web Vitals هي مجموعة من المقاييس (metric) التي قامت جوجل بتطويرها لقياس جودة ال UX على صفحات الويب، وخصوصا من خلال اختبار ثلاثة معايير وهي LCP وال INP وال CLS... شاهد الصورة أدناه:

- ال Developer Experience: مدى سلاسة العمل من قبل المطورين على هذه المعمارية من حيث الأدوات المستخدمة مثل ال client library التي يمكن استخدامها وال SDK وال frameworks وال open source code وال API وال Technology التي يمكن استخدامها...
- ال Scalability: القدرة على التوسع كلما زاد عدد الطلبات أو احتيج ذلك.
- ال Coordination: مدى سهولة التعاون بين الفرق المختلفة وتوحيد جهودها لتحقيق الأهداف المشتركة فيما بينهم.

يعني باختصار ستقوم بسؤال نفسك هذه الأسئلة:

- هل من السهل نشر ال Micro-Frontend؟
- هل من السهل تعديل أجزاء النظام؟
- هل يستطيع المطورون فهم المعمارية بسهولة وتطويرها؟
- هل من السهل كتابة اختبارات قوية وفعالة؟
- هل تقدم المعمارية أداء ممتاز يحقق أفضل تجربة مستخدم نطمح لها؟
- هل يتمتع المطورون بتجربة تطوير مرنة أم مقيدة؟
- هل يمكن عمل scale للنظام مستقبلا؟
- هل تحتاج فرق العمل لتنسيق كبير ومعقد أم تنسيق عادي كفيل بذلك؟

والسؤال الآن كيف يمكننا استخدام هذه التحليلات أو الأجابات لاختيار المعمارية؟
والجواب يكون من خلال بناء جدول للمعماريات الممكنة مع إعطاء الدرجة المناسبة لكل خاصية من ١ إلى ٥ حسب احتياجاتك الحقيقية...
وهنا يظهر سؤال آخر، ماذا لو كان لدي معمارية حصلت على ٥ في كل نقاطها؟
والجواب هنا يكون: راجع نفسك يا باشا :P لأن هذا غير ممكن!، بل هذا يشير لوجود خطأ خطير قد ارتكبته في مكان ما، والسبب يعود في ذلك إلى أن بعض الخصائص تتعارض فيما بينها، مثلا لا يمكن أن يكون بسيطا جدا وبنفس الوقت لديه قابلية للتوسع عالية جدا! ولماذا تحدثنا أصلا عن مفهوم المقايضات (trade off)؟!

مثال توضيحي -مبسط-: عودة لمثال موقع التجارة الإلكترونية، لو افترضنا أن تقسيم الموقع بشكل horizontal يحتوي على header فيه عربة التسوق ومحرك بحث، ومكان لعرض محتوى مختلف مثل المنتجات مع خيارات ال filter ولدينا footer في روابط متعددة، في حين لو تم تقسيم الموقع بشكل vertical فسيكون لدينا صفحة المنتجات وصفحة الدفع والصفحة الرئيسية... مع افتراض أن كل فريق ضمن هذه المؤسسة يمتلك FE, BE فيمكنه تطوير المزايا بشكل مستقل، وأن عدد المكونات المشتركة قليل.

إذا بناء على هذه المعطيات، سنخرج بهذا الجدول (القيم الافتراضية سيتم مناقشتها ووضعها لاحقا عندما نتحدث عن خصائص كل واحدة من هذه المعماريات بشكل مخصص وليس ضمن نطاق مشروع أو context محدد):

المعيار	Horizontal Split	Vertical Split	التعارضات الملحوظة
Deployability	٢	٥	التقسيم الأفقي يتطلب تنسيقًا عاليًا للنشر بسبب الاعتماد على المكونات المشتركة.
Modularity	٣	٥	التقسيم الرأسي أكثر استقلالية، لكنه قد يزيد من تعقيد الربط بين الميزات.
Simplicity	٣	٤	التقسيم الرأسي يبسط حدود الفرق، لكن التكرار قد يقلل البساطة.

Testability	٣	٤	الاختبار في التقسيم الأفقي يتطلب اختبارات تكاملية أكثر.
Performance	٤	٣	التقسيم الأفقي يُحسّن الأداء عبر مشاركة الموارد، بينما الرأسي قد يُسبب تكرار للشيفرة البرمجية.
Developer Experience	٢	٤	التقسيم الأفقي يزيد ال dep بين الفرق، مما يقلل تجربة المطور.
Scalability	٢	٥	التقسيم الرأسي يدعم توسع الفرق دون تنسيق مكثف.
Coordination	٥ (تنسيق عالي)	١ (تنسيق منخفض)	التنسيق العالي في الأفقي يعيق المرونة والاستقلالية.

بناء على هذا الجدول، لدينا هذه المعايير التي تتعارض فيما بينها:

في ال Horizontal split يقلل ال Coordination العالي (٥) من قدرة الفرق على ال Deployability (٢) في حين وجود قدرة عالية على ال Deployability في ال vertical (٥) يقلل من الحاجة للتنسيق (١). (أول مثال عملي على فكرة المقايضات).

كما أن ال Horizontal سيحقق أداء (٤) أفضل من ال vertical بسبب وجود مكونات مشتركة بين ال micros، لكنه سيقلل من الاستقلالية (٣)، في حين يزيد ال Vertical من الاستقلالية (٤) لكن الأداء (٣) يضعف بسبب تكرار الشيفرة البرمجية...

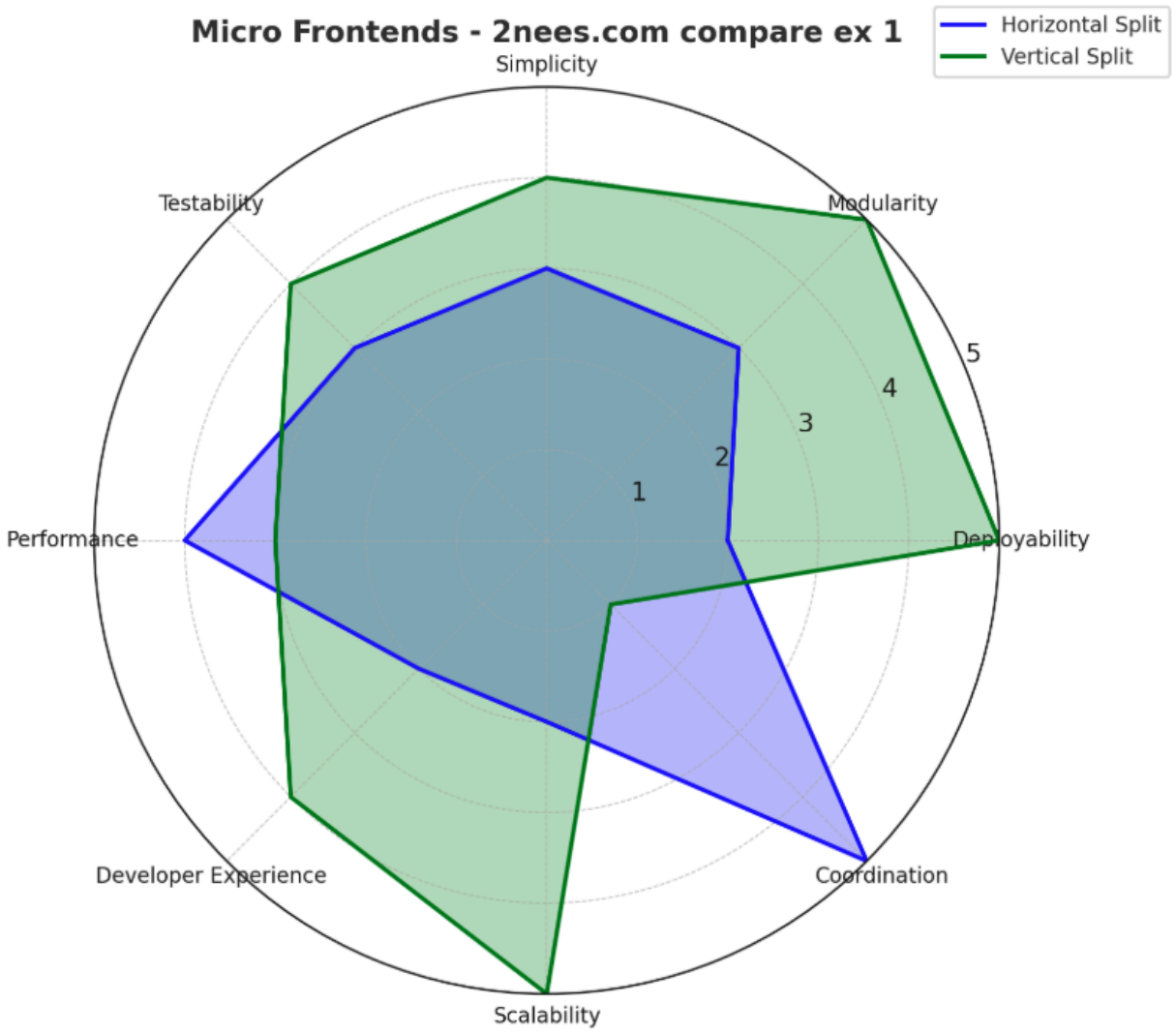
كما أن التقسيم ال Vertical هنا سيقوم بتحسين تجربة التطوير وتسهيلها (٤) وسيساهم في التوسع (٥)، بينما يعيق ال Horizontal كلا الأمرين (٢).

كما أن ال Vertical سيقوم بتبسيط ال testing لأن الاختبار سيكون معزولا عن غيره (٤) لكن التكرار قد يقلل البساطة (٤).

بناء على هذا التحليل نستنتج الآتي:
سيكون استخدام ال Horizontal هنا مناسبا إذا كان مشروع التجارة الإلكترونية الخاص بنا يحتوي على الكثير من المكونات المشتركة لذلك فهو يتطلب تنسيقا عاليا بين الفرق والمكونات، وهذه الحالة ليست لدينا حسب مقتضيات السؤال -عدد المكونات المشتركة قليل-.
وسيكون استخدام ال Vertical أفضل في حالة المشاريع الكبيرة ذات الفرق المستقلة، لكن قد نواجه مشاكل تتعلق بالأداء...
إذا المفاضلة بين هاتين المعماريتن سيكون بين التنسيق العالي بال Horizontal والاستقلالية بال Vertical.

ملاحظة:
خسرنا بعض الأداء المحتمل بسبب تكرار الشيفرة البرمجية
وقد نحتاج إلى آلية مركزية لتوحيد تجربة المستخدم (design tokens / style guide) لتجنب تباين واجهة المستخدم.

شاهد الصورة لتمثيل المقارنة بالجدول على شكل Radar لكن احذر أن ال Coordination = ٥ تشير لنقطة سلبية وليست إيجابية.



والآن، لقد بدأ اللعب الحقيقي ^^، وسنبدأ بالغوص في المعماريات بشكل أكبر تفصيلا وأكثر دقة... ولنبدأ بأول معمارية:

ال Vertical-Split Architectures:

إن استخدام الـ client-side composition مع ال client-side routing مع ال Application shell يعد من أكثر الخيارات سهولة ومناسبة لمبرمجين ال FE الذين تعاملوا تحديدا مع تطبيقات ال SPA، لذلك تعد هذه المعمارية من أسهل المعماريات أو أول المعماريات التي يمكن تطبيقها أو تعلمها من قبل مبرمجين ال FE... وهذا هو سبب بدايتنا في هذه المعمارية ^^.

والآن سنذهب إلى أول جزء من هذه المعمارية وهو ال Application Shell.

فائدة

تذكر دائمًا أن ما بك من هداية وتوفيق هي من فضل الله -سبحانه وتعالى- عليك، ولْتعلمْ أن ما لديك من علم ومعرفة هو مما تفضّل الله تعالى به عليك، فلا تغتر بما لديك من علم وتوفيق، فإنها ما صارت إليك إلا بفضل الله -سبحانه وتعالى- عليك، فأدِ حقها! وحقها أن تتيقن الصواب فيما لديك من علم، وألا تظلم بعلمك غيرك عن جهل يقتضيه نقص العلم لديك أو بعملٍ يُخالف العلم الذي علمت!

- كتاب إلى الجنة زمرا، الصفحة ١١٧

ال Application Shell:

لقد قمنا بتعريفه سابقا، لكن نعيد التذكير به باختصار: "ال Application Shell هو ال Main Layout أو الإطار الرئيسي أو الهيكل الكبير الذي يحتوي عناصر (/ component sections) لا تتغير بين الصفحات، ولا يحتاج لتحميل الصفحة كاملة إذا حصل navigation، ويتم تحميله مرة واحدة عند فتح التطبيق"، أي بمعنى آخر وأكثر دقة هو ذلك الجزء الذي يبقى موجودا كحاوية لجميع ال micro-frontends التي يمكن تطبيقها، ولذلك فهو أول جزء يتم تحميله في التطبيق الخاص بك، وهو الجزء المسؤول عن جلب وتحميل وحذف ال micros بحسب الروابط التي قام المستخدم بالاستعلام عنها... والسؤال الآن، ما هو الهدف أو السبب الرئيسي لوجود ال Application shell؟

والجواب يكمن لعدة أسباب وهي:

- ال Handling the initial user state:

  ويقصد بها إدارة حالة المستخدم عند الدخول لأول مرة على التطبيق الخاص بنا، فمثلا إذا حاول المستخدم الوصول إلى صفحة تحتاج لتسجيل الدخول مثل الصفحة الشخصية الخاصة بك، وكان ال Token غير موجود أو منتهي أو غير صحيح؛ فإن App Shell سيعيد توجيه المستخدم إلى صفحة تسجيل الدخول أو الصفحة الرئيسية، وانتبه أن هذا يتم تنفيذه فقط عند أول load للموقع،أما العمليات اللاحقة فسيتم التعامل معها من خلال ال micro-frontend نفسها لأنها المسؤول الحقيقي عن إدارة ال auth أو التحقق داخلها...

- ال Retrieving global configurations:

  ويقصد بها جلب أي معلومات تخص الإعدادات العاملة للمتسخدم والمفيدة في تحسين تجربة المستخدم له، وذلك يشمل بلد المستخدم واللغة المفضلة والسمات مثل dark/light mode... إلخ، لكن كيف ستكون هذه مفيدة؟ والجواب قد تكون هناك تجربة مستخدم مختلفة أو بيزنس مختلف باختلاف الدولة، بمعرفة الدولة سيقودك لمعرفة ماذا ستعرض... وكذلك جلب السمة dark / light بناء على اختيار المستخدم الذي قام باختياره أو إرجاع الخيار الافتراضي في حال عدم وجود تفضيل...

- ال Fetching the available routes and associated micro-frontends to load:

  ويقصد بها عملية جلب ال routes المتاحة والمرتبطة بكل micro-frontend، وهذا الأمر يجب أن يتم في ال run time وليس ال build time، والسبب في ذلك يعود لأننا نتعامل مع micro frontend، ونريد خاصية النشر المستقل التي تحدثنا عنها، فكيف سنتمكن من عمل deploy مثلا لصفحة ال product دون الحاجة لعمل deploy لل application shell كاملا؟! وتخيل أننا قمنا بإضافة micros جديدة، ماذا سنفعل؟ لذلك، نقوم هنا بجلب ال routes المرتبطة بكل ال micros من

خلال ال BE، شاهد الصورة أدناه:

```
async function loadRoutes() {
  const routes = await fetch('/api/routes').then(res => res.json());

  /**
   * [
   *   {
   *     "path": "/home",
   *     "microFrontendUrl": "https://cdn.2nees.com/home-app.js"
   *   },
   *   {
   *     "path": "/product",
   *     "microFrontendUrl": "https://cdn.2nees.com/product-app.js"
   *   },
   *   {
   *     "path": "/cart",
   *     "microFrontendUrl": "https://cdn.2nees.com/cart-app.js"
   *   }
   * ]
   */

  // Register Routes and handling import...
}
```

- ال Setting logging, observability, or marketing libraries:

  ويقصد بها عملية تحميل المكتبات المستخدمة لمراقبة النظام أو التتبع ونحو ذلك على مستوى ال Application shell، لأنها ستكون مستخدمة ومهمة في جميع ال micros المختلفة.

- ال Handling errors if a micro-frontend cannot be loaded:

  ويقصد بها إدارة ومعالجة الأخطاء الممكنة إذا حصلت في حال فشل تحميل ال

micro-frontend نفسها... مثلا يمكننا عرض 404 عند فشل التحميل أو أي أسلوب أو طريقة تكون مناسبة لنا...

قد يتسائل البعض، أن بعض هذه النقاط والتي صنفت على أنها من الأسباب الرئيسية لجعلنا نستخدم ال App shell يمكن فعلا وضعها داخل ال micro مثل ال logging... وكلامك صحيح، لكن هذا الأمر سيخلق لنا عدة مشاكل منها زيادة التعقيد وزيادة الحاجة للتنسيق بين الفرق لضمان التزامن بين المكتبات وإصدارها وزيادة ال risk عند حدوث deploy... لكن هذا أيضا يقودنا لنقطة مهمة وهي: لا تستخدم ال Application shell لتكون ك layer تتواصل وتتفاعل باستمرار مع ال micros أثناء ال session، وهذه نقطة مهمة وخطيرة جدا، ووقوعك بها سينتج مشكلة خطيرة وهي ال Distributed monolith، وهو من أسوأ الكوابيس التي يمكن أن تعيشها كمطور FE داخل مشروع فيه معمارية كهذه! وذلك يعود لأن ال micro frontend ستفقد الاستقلالية وذلك سيؤدي لوجود test و deploy معتمد على ال micros الأخرى أو ال app shell نفسه... وهذا كله سيعيدنا للنقطة الأولى أن أي عملية تحديث ستجعلنا نقوم بعمل build و deploy لل app shell كاملا... وهنا سنقول: ما الفائدة من ال micros إذا! وكمثل عملي: تخيل أن product تقوم بالاتصال بقاعدة البيانات الخاصة بال cart مباشرة وليس من خلال API!

ملاحظة: هناك مصطلح مهم وهو ال Logical Coupling، ويقصد بهذا المصطلح وجود اعتمادية غير مرئية أو ضمنية بين جزأين يفترض أنهما مستقلان، لكن أحدهما يعتمد ضمنيا على تفاصيل الآخر أو سلوكه، ومن هذا المعنى نكتشف أن وجود ال Logical Coupling

هو المسبب الأساسي في ظهور ال Distributed monolith... وحسب حالتنا فإن أي اعتماد منطقي ضمني بين أي من ال micro frontend مع ال application shell أو العكس = مشكلة Logical Coupling... وكمثال عملي، لو افترضنا أننا انتهكنا ال ":Fetching the available routes and associated micro-frontends to load"، وقمنا بوضع ال routes مباشرة داخل ال App shell بدلا من جلبها من ال BE... فهنا ارتكبنا هذا الخطأ وأصبح لدينا اعتماد ضمني على ال App shell بحيث لو حصل مثلا أي تغير على ال url الخاص بال micro فسنتحاج لتحديث ونشر ال App shell نفسه...

ملاحظة ٢: هناك مصطلح آخر وهو ال Physical Coupling ولست بحاجة للقول أنك إذا وقعت بهذا فقد قمت بقتلي وقتل كل من يعمل معك ومن سيعمل بعدك على المشروع :P، وذلك لأن المشكلة هنا ظاهرة ولها وجود حقيقي وليست شيء أو خطأ منطقي يمكن أن نقع به عند خلل بالتفكير -وهو وارد جدا-، والترابط الفعلي بين المكونات مثلا يكون من خلال وضع ملفات مثل ال home-app.js -شاهد الصورة في الأعلى- داخل ال shell مباشرة ^^... فحدوث أي تغيير على هذا الملف شخصيا أو ما يعتمد عليه = الحاجة لعملية deploy جديدة لل App shell.

أما السلوك المتوقع من ال App shell في هذه المعمارية هو:

- يتم تحميل أو جلب micro frontend واحدة في كل مرة

- لا حاجة للخوف من تعارض المكتبات أو ملفات ال css وال js عند التنقل بين ال micros لأن كل واحدة سيتم تحميلها بشكل مستقل وسيتم إزالة المحتوى الخاص بها عند الانتقال لغيرها.
- ال Application shell يجب أن يكون في أبسط صورة ممكنة، ويفضل أن يكون عبارة عن صفحة HTML و vanilla js و css بكل بساطة.

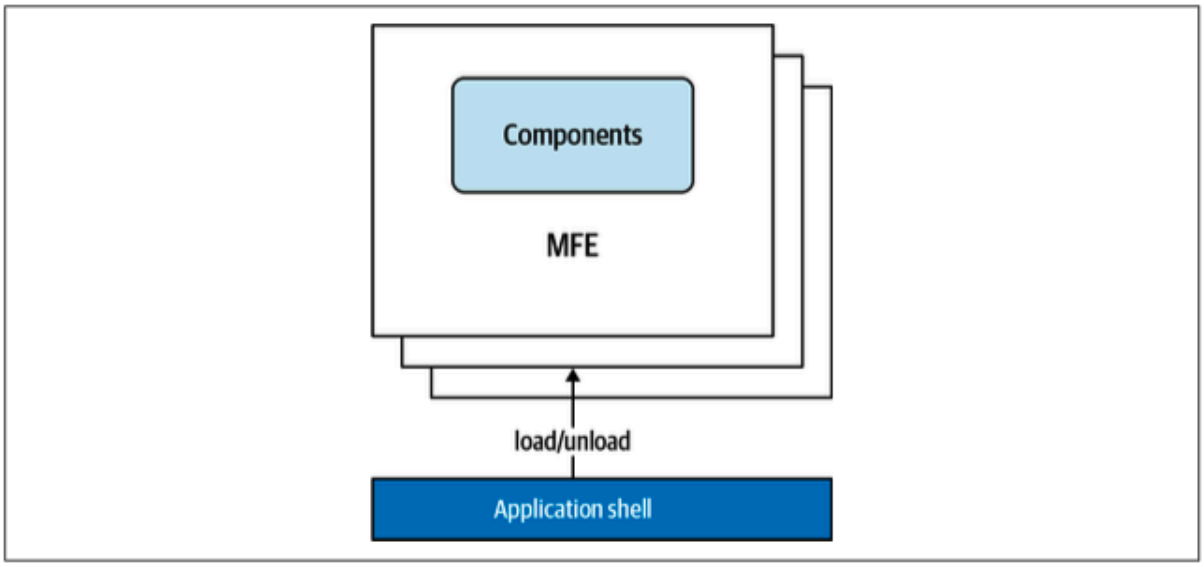


Figure 4-4. Vertical-split architecture with client-side composition and routing using the application shell

بناء على ما تحدثنا عنه، نستنتج أن هذه المعمارية مناسبة جدا عند وجود Business domain واضح ومنفصل ولا يوجد تداخل كبير ولا يوجد حاجة لاستخدام ال micros ضمن أكثر من صفحة في أكثر من مجال، مع وجود فرق مستقلة لهم كامل الحرية في اتخاذ القرارات المتعلقة بهم، لكن يجب أن نفهم كيف يعمل التطبيق بشكل حقيقي من خلال جمع معلومات المستخدمين أثناء استخدام التطبيق من خلال أي أداة مثل Google

Analytics، ويمكنك وضع أطر أو بناء component مشتركة بين كل ال components ليتم استخدامها في جميع ال micros للحفاظ على نسق واحد في التصميم... ومن أهم مزايا هذا النوع من المعماريات إمكانية استخدام هذه ال micros في مشاريع مختلفة ومتنوعة بسهولة باعتبارها وحدة واحدة متكاملة فمثلا حسب مثالنا الخاص بالتجارة الإلكترونية يمكننا الاستفادة من ال Micro الخاصة بعملية الدفع بأكثر من مشروع وكذلك الحال لصفحة ال cart مثلا...

والآن، بكل تأكيد الحياة ليست وردية إلى هذا الحد، لذلك، لنذهب وننظر إلى التحديات التي تواجهنا وعلينا حلها... هذه التحديات تقسم إلى جزئين وهما:
الأول: تحديات مرتبطة بسياق المشروع لا يمكن الإجابة عليها مباشرة.
الثاني: تحديات شائعة بعضها يمكن الإجابة عليه بشكل مباشر.

والآن لنلقي نظرة على أربعة تحديات رئيسية تواجهها:

١. ال Sharing state
٢. ال micro frontend composition
٣. ال multi-framework approach
٤. ال Architecture Evolution

التحديات - Challenges:

أول تحدي: ال Sharing State:
إن أول تحدي ستواجهه في عملك على ال micros هو كيفية أو آلية مشاركة ال state بين ال micros المختلفة، وعلى الرغم من أننا في هذه المعمارية لا نحتاج لمشاركة الكثير من المعلومات إلا أنه ما زال هناك داع لذلك!

ولحل هذه المشكلة يمكن تقسيم الحل لثلاثة أجزاء بحسب طبيعة البيانات وآلية أو مكان التخزين...، فمثلا يمكن استخدام ال Web Storage كأول حل لهذه المشكلة وفيه يتم حفظ أي معلومات غير حساسة مثل site mode هل هو dark أم light، أما الحل الثاني فيظهر عند وجود بيانات حساسة نرغب بحفظها والتعامل معها مثل ال token مع ربطها بال timestamp، فمثلا لو قامت إحدى ال micros بجلب معلومات المستخدم وتم حفظها لمدة ٥ دقائق وانطلقنا ل micro أخرى فسيتم جلب البيانات من مكان التخزين، في حين لو انتهى وقت التخزين فستقوم هذه ال micro بجلب البيانات مجددا من ال API وتخزينها مع الوقت الجديد...، لكن إذا صار هناك اعتماد كبير على ال Web Storage ونحن بحاجة للقيام ببعض التحقيقات أو العمليات الأمنية على البيانات المحفوظة؛ ففي هذه الحالة يمكننا الانتقال للحل الثالث وفيه يتم إنشاء Abstraction layer داخل ال app shell، وفيه ستكون API مسؤولة عن إرجاع وحفظ المعلومات، وستمثل المكان المركزي لعملية ال validation على هذه البيانات مع إرجاع رسائل واضحة ومفهومة لل Errors في حال وجود خطأ بأحد ال validations.

ومن المهم العناية في الحلول المقدمة لهذا التحدي، فلو حدث خطأ معين فقد ينتج عنه تضارب في البيانات بين ال micros المختلفة (ومشكلة عدم تزامنية البيانات)، وهذا سيسبب مشاكل في تجربة المستخدم، كما أن التعامل الخاطئ مع البيانات الحساسة قد ينتج لنا مشاكل أمنية خطيرة...

ثاني تحدي: ال micro frontend composition:

هناك العديد من الخيارات لمعالجة هذا التحدي، ولأننا اتجهنا لأن يكون التجميع وال routing على مستوى ال client side، فالخيارات ستكون محصورة بما يسمح به المتصفح ^^، ولدينا هنا أربعة طرق لل composition داخل ال application shell، وهي:

1. ال ES module: هذه الطريقة التي نستخدمها اليوم بشكل افتراضي مع الكثير من مشاريعنا تم تقديمها بشكل رسمي في ES6 (ECMAScript 2015) وهذا الأسلوب واحد من أفضل الأساليب المستخدمة لهذا الغرض، وهي الآن مدعومة من جميع المتصفحات الحديثة وبيئات تشغيل الجافا سكربت... كما أن هذا الأسلوب يمكن استخدامه عند ال Compile time أو عند ال Runtime، وفي حالتنا بوجود ال micros وال app shell فغالبا ما يكون خيارنا هو ال Runtime، ومن النقاط المفيدة التي يقدمها لنا أن ال module الذي يتم تضمينه من خلال هذا الأسلوب يتم تحميله عند الحاجة، ويتم تحميل الأجزاء أو الوحدات المطلوبة من هذا ال module عند الاستخدام، كما يتكفل بشكل تلقائي بإدارة عمليات ترتيب ال import ونحو

ذلك... لذلك لو قمت مثلا بتضمين ال module داخل ال htnl: <script type="module" src="catalogMFE.js"></script> فإنه سيقوم باعتباره module وسيتم تأجيل تحميل هذا ال module تلقائيا (defer).

2. ال SystemJS: هو Loader تم تصميمه وتطويره لتحميل ال modules في الجافا سكربت، ويدعم عدة أنواع من ال modules التي يمكنه التعامل معها، وكان هذا الأسلوب أحد أشهر وأقوى الطرق المستخدمة مع ال micro frontend، وإلى الآن قد يكون أحد الخيارات المستخدمة... لكن مع إصدار ال ES6 واعتماد المتصفحات وتطبيقها لل ES module، لم يعد هناك داع كما كان سابقا لاستخدام مثل هذه الأداة... لكن من الحالات التي تجعلنا نستخدمه إذا كانت المتصفحات القديمة أو المستخدمين الذين نتعامل معهم ما زالوا على متصفحات قديمة لا تدعم ال ES module ونحتاج ل polyfill لضمان دعم المتصفحات...

3. ال Module Federation: وهي Plugin تمت إضافتها لل Webpack 5 تستخدم لتحميل modules أو library أو app كامل داخل app آخر، وكما أن لديها القدرة على إدارة الإصدارات المشتركة لمكتبة ما داخل جميع ال micros، ويعامل كل micros وكأنه ك component داخل المشروع... وتتم هذه العمليات أيضا أثناء ال runtime لذلك فهي مناسبة لنا، ومن أكثر النقاط المفيدة التي قد تدعوك لاستخدام هذا الأسلوب هو إدارة المكاتب أو المكونات المشتركة بين ال micros، فمثلا لو كنا نستخدم react فإن ال module federation يضمن لنا مشاركة نسخة react مع جميع ال micros، ويتم تحميل ما نحتاجه فقط... لكن احذر أن جميع ال micros يجب أن تستخدم نفس النسخة من المكتبة التي تمت مشاركتها... فلا يصح وجود

react 12 و react 16 معا لتجنب أي تضارب ممكن! كما أن عملية الإعداد قد تكون معقدة (كتابة ال config) لها أو لبعض المكتبات التي تحتاج حلول خاصة بها مثل vite، وهذا كله مما قد يزيد صعوبة ال testing وال debugging.

4. ال HTML parsing: إذا تم اعتماد ال entry page داخل ال micro frontend لتكون عبارة عن صفحة html، فإننا يمكن أن نستخدم ال javascript لقراءة الصفحة ومن ثم إدراج ال dom المطلوب، فبالنهاية هنا أنت تتعامل مع xml! ويتم ذلك باستخدام DomParser وال adoptNode وال cloneNode... وهو مناسب في حال كانت ال micros مبنية بالكامل من خلال ال html ..

ملاحظة: من الشائع استخدام ال ES Module أو ال Module Federation لتقديم حلول لهذا التحدي... ومع ذلك فلديك مجموعة الحلول المقترحة الآن أمام هذا التحدي ويمكنك دراسة الأنسب لك حسب سياق المشروع.

ثالث تحدي: ال Multi Framework approach:

يعد استخدام أكثر من إطار عمل (framework) في نفس النظام والذي يطلق عليه ال Multi-Framework Approach، من أكثر التحديات حساسية في عالم ال micro frontend، ويقصد بذلك استخدام أكثر من مكتبة أو framework لبناء ال UI الخاص بال frontend مثل استخدام ال React لبناء صفحة المنتجات وال vue لصفحة المشتريات

و angular للصفحة الرئيسية! ومع أن هذا ممكن من الناحية التقنية، إلا أنه غير محبذ وجوده واستخدامه إلا ضمن ظروف معينة، وعادة ما تكون هذه الظروف هي عملية انتقال سلسة من نظام قديم إلى نظام جديد، في هذه الحالة ستكون هذه العملية ممتازة لأننا لن نحتاج إلى الانتظار لأشهر طويلة حتى نخرج بأول إصدار، وكذلك سيكون لدينا تغذية راجعة مفيدة عن سلوك المستخدمين وقتها... لكن إذا لم يكن هناك داع لذلك فأنت قد تسببت لنفسك في مشاكل تقنية ستحتاج إلى حلها وستتفاقم بشكل دوري مع كل إصدار أو framwork جديد يتم إضافته...

إن المشاكل المتعلقة بهذا السلوك يمكن تلخيصها بما يلي:

1. مشاكل تتعلق بالأداء، فلكل framework مجموعة كبيرة من المكتبات التي يحتاجها وهذا يعني بالضرورة bundle كبيرة = وهذا سيضاعف من كمية البيانات التي سيحتاج المستخدم لتحميلها وقد يجبر المستخدم في جلسة واحدة على تحميل أكثر من نسخة من كل framework أيضا!
2. ال Dependency Clashes: وهي المشاكل المتعلقة بوجود إصدارين مختلفين من نفس المكتبة أو ال framework مثل react 16 و react 19، وهذا قد يؤدي أخطاء أثناء ال runtime أو مشاكل clashes بين المتغيرات التي يتم تعريفها (مشكلة global namespace clashes).
3. صعوبة الصيانة والتحديث لأن كل مكتبة أو framework سيحتاج إلى طريقة فحص أو build أو deploy مختلفة، وسيصعب توحيد هذا بين ال teams المختلفة.

والآن بعد أن تعرفنا على هذه المشاكل التي يمكن أن يتسبب بها ال Multiframe work، يمكننا إيجاد الحلول التقنية الممكنة -وتذكر أنها ليست مثالية-:

1. تقليل الاعتماد على المكتبات الخارجية قدر الإمكان وعمل import لما نحتاجه فعلا داخل كل micro.
2. تقليل عدد ال framework أو library المستخدمة قدر الإمكان -إذا كان لدينا قدرة على ذلك-.
3. توحيد الإصدارات المختلفة لنفس المكتبة بين ال micros -إن أمكن ذلك-.
4. يمكن استخدام ال iframes للخروج من المشاكل المتعلقة بالتضاربات المختلفة، لكنها ستخلق لنا مجموعة من التحديات خصوصا في موضوع ال communication بين ال micros.
5. يمكن استخدام ال Web Components واستغلال ال Shadow DOM لحمايتنا من بعض التداخلات.
6. يمكن استخدام ال Module Federation إذا توحدت الإصدارات بين ال micros.

وغيرها من الحلول الممكنة بحسب سياق المشروع... لكن تذكر أن التوصية النهائية بأن لا يكون خيارك للتطبيق الخاص بك هو اعتماد وجود أكثر من framwork إلا إن دعتك الضرورة لذلك. (هذه التوصيات أو الملاحظات يمكنك تخيلها أو إسقاطها على جميع

المعماريات التي سنتحدث عنها، فكثير منها ستكون مشاكل مشتركة، نتقبل بعضها في ظروف معينة ولا نتقبلها في الظروف المعيارية...).

فائدة

فلْتعلمْ أن الكلام بين الناس يُقسم لثلاث فئات، الأول: أن يكون الكلام مفيدًا وفيه خير وصلاح، والثاني: كلام مباح لا خير فيه ولا منفعة، ولا ضرر فيه ولا إثم، والثالث: أن يكون الكلام كلامًا محرمًا فيه إثم ومفسدة، لذلك احرص دومًا أن تكون ممن يقول أحسن القول، فيكون القول فيه حث على الخير والإنفاق في المال والعلم، والإصلاح بين المتخاصمين ونحو ذلك من الكلام الطيب، واحذر من كل كلام مفسد للقلب ومفسد للناس، واحذر من كل قول يحث على المنكر ونشر البغضاء مثل الغيبة والنميمة، والله المستعان.

- كتاب إلى الجنة زمرا، الصفحة ١١٧

التحدي الرابع والأخير: Architecture evolution and code encapsulation:

من غير الممكن دائما تحديد ال subdomains بدقة من المحاولة الأولى ^^، وهذه نقطة مهمة جدا عليك أن تأخذها بالحسبان، فجميع ما تحدثنا عنه من مبادئ من قبل وآليات للتحليل وطرق للقياس كانت وسيلة لأن نصل لأعلى دقة ممكنة بحيث يتم تحديد ال subdomains بدقة من المرة الأولى، لكن ليس هذا ما يكون دائما في الحياة العملية خصوصا مع وجود بعض المؤثرات التي ستؤثر على المشروع لا محالة...

عند استخدامك لل vertical-split فإن ذلك قد يؤدي إلى إنشاء micro-frontends واسعة النطاق (coarse-grained)، والتي ستصبح معقدة مع مرور الوقت بسبب توسع نطاق المشروع وزيادة الضغط على ال teams مع هذا التوسع لإنجاز الأعمال وربطها ونشرها ونحو ذلك، كما أنه من الممكن أن تظهر رؤى جديدة بشأن الافتراضات التي تم اتخاذها عند بداية المشروع مما قد يغير على طبيعة ال micros التي تم اعتمادها سابقا... لكن مع ذلك، فهذا الأمر يمكن حله وتجاوزه بسهولة بإذن الله، فطبيعة هذه المعمارية تمكننا من البرمجة أو التطوير من خلال تطبيق مبدأ أو فكرة ال modular، وبهذا يمكن تطوير هذه ال module وتطوير الأعمال معا بالتوازي دون مشكلة...

ملاحظة ١: Coarse-grained: يشير هذا المصطلح لوجود شيء كبير الحجم أو واسع النطاق يقوم بأكثر من مهمة أو وظيفة بدلا من القيام بوظيفة محددة، مثلا إذا قمت بتعريف method واحدة تقوم بالتحقق من معلومات المستخدم وصلاحية البيانات ومن ثم حفظها

وإرسال إشعار للمستخدم وتصدير ملف له... ولهذا الأمر تبعات كثيرة من أهمها تأثر المطورين بذلك ووقوعهم تحت ال Cognitive load.

ملاحظة ٢: الCognitive load: يقصد به العبء العقلي، وهو يشير إلى مقدار الجهد العقلي اللازم من شخص ما لفهم شيء ما... فلو عدنا للمثال السابق في الملاحظة ١ سنجد أن المطور إذا جاء ليقوم بتعديل بسيط فإنه سيحتاج إلى جهد عقلي كبير، وهناك علاقة طردية بين زيادة الحمل العقلي و زيادة فرصة ارتكاب الأخطاء.

هناك عدة أساليب للتخص من هذه المشكلة، واحدة من هذه الأساليب وأشهرها ال Encapsulation ^^، ولا تتعجب يا صديقي من هذا، فهذا المفهوم هو نفسه الذي تعلمناه في ال OOP، وفكرته ببساطة أن عمليات تعديل أو معالجة البيانات يجب أن تتم من داخل ال method الموجودة داخل ال class والتي تستخدم لتعديل أو استرجاع القيم الموجودة في ال attribute، لذلك يكون هذا ال class محمي من العالم الخارجي ويعمل كما تم تصميمه، وتقريبا بنفس هذا المفهوم سنقوم بتطبيقه على ال micros، لكن لماذا سنقوم بذلك؟!

انظر لهذا المثال الذي قد يصادفك: لو افترضنا وجود micro اسمها Authentication وفيها ال views التالية: Payment, Signup, Signin, Forget email or password... حسب طبيعة المستخدمين فإن المستخدمين سيقومون بتسجيل الدخول أو استرجاع كلمة السر وهذا السلوك الأول، أما المستخدمين الجدد فسيقومون بالتسجيل أو إجراء عملية الدفع... وهنا يظهر لدينا مشكلة Cognitive load على حدود مسؤولية الفريق، فسيكون على الفريق نفسه ضمن

نفس المعمارية الاهتمام بنوعين مختلفين من المستخدمين لكل منهم مساره الخاص ضمن السلوك الغالب أو الشائع، وفي نفس الوقت يجبر المستخدم على تحميل كل ما يحتاجه وما لا يحتاجه... وهنا يظهر جمال الفصل، فلو كانت آلية تطويرنا moduler وبكل بساطة قمنا بنزع ال Payment وال Signup من ال Auth ووضعناها في micro جديدة أسميناها Subscription ستحل هذه المشاكل ^^، شاهد الصورة F4-5 و F4-6.

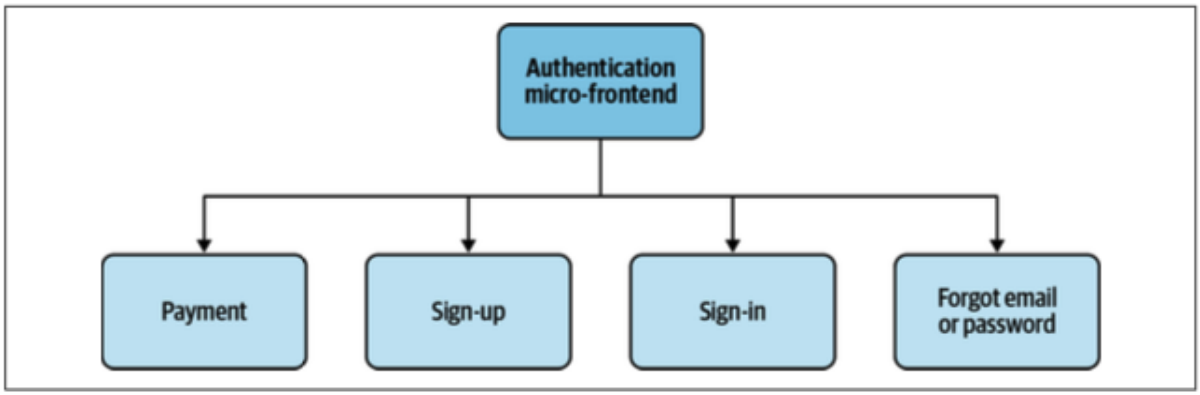


Figure 4-5. Authentication micro-frontend composed of several views that may create a high cognitive load for the team responsible for this micro-frontend

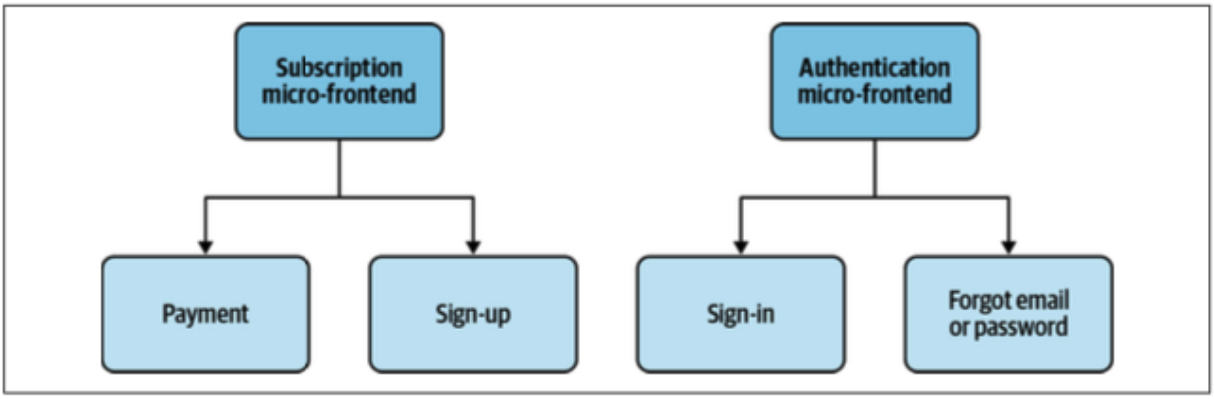


Figure 4-6. Splitting the authentication micro-frontend to reduce the cognitive load, following customer experience more than technical constraints

وهنا نعود لنقطة مهمة جدا ذكرناها فيما سبق... وهو أن الفصل ليس بالضرورة أن يكون ناتجا عن سبب تقني، فهنا الفصل كان نابعا من المنطق التجاري (Business Logic)، ولذلك هذه الطريقة هي ليست الطريقة الوحيدة في التقسيم، يمكنك استخدام الأسلوب الذي تجده مناسبا للقيام بذلك، لكنك يجب أن تتذكر أن التركيز يجب أن يكون على نتيجة مخرجات هذا التقسيم من ناحية ال business وليس من الناحية التقنية كخيار مجرد.

وهنا يظهر دول ال encapsulation، ويقصد بها في حالتنا تجنب استخدام ال state واحدة تمثل ال micro كاملة، وإنما يفضل استخدام state محصورة داخل حدود واضحة في جزء محدد من التطبيق، وهذه النقطة مفيدة جديا في تقليل العبء العقلي وتسهيل فصل المكونات... لكن ماذا لو كانت لدينا state / library / logic ممكن أن يستخدم ضمن أكثر من domain، فماذا سنفعل؟ والحل بواحدة من هذه الطرق:

- ال Duplicate the code: نعم يا صديقي، كما قرأت، تكرار الشيفرة البرمجية ^^، فتكرار الشيفرة البرمجية لا يعد دوما فكرة سيئة أو عملا سيئا كما يعتقد الكثيرون، فهناك حالات يعد تكرار الشيفرة البرمجية فيها أمرا مستحسنا لأنه يقلل من تعقيد لا داع له! لذلك يمكنك القول أن تكرار الشيفرة البرمجية لا يعد بذاته أمرا سيئا إلا إذا كان الأثر الإجمالي لهذا العمل سيئا أو أن الذي نحاول تحسينه لا يحتاج لهذه الطريقة وهناك طرق أخرى أفضل لأجل ذلك، ومن الأمثلة على ذلك -مثال توضيحي أو تقريبي- لو افترضنا وجود component مثل header menu مستخدمة عندك في أكثر من domain، ثم في ال micro التي أنت فيها كان يلزمك القيام بتعديلات

متعددة للوصول لما تريده، مثل إضافة profile menu إليها ووضع شريط الإشعارات والتحكم بعدد الروابط... جميع هذه الأمور يمكن كتابتها ووضعها من خلال if/else على ال header menu الأساسية، لكن الأفضل هنا هو تكرار الشيفرة البرمجية باستخدام ال new header menu داخل ال micro frontend، وهذا سيخلصنا من التعقيد ويقلل من العبء العقلي ويحافظ على النشر المستقل... حسنا، لكن ماذا لو تغير ذلك في المستقبل وأصبحنا نحتاجها في مكان آخر؟ والجواب هنا يكون بسؤال آخر، هل سألت ال business عن الخطة وأن هناك فعلا خطة لاستخدامها في أكثر من مكان أم لا؟ أم أن هذا الأمر خوف من المستقبل فقط؟! وهل هذا التغيير سيكون قريبا أم بعيدا؟! بناء على الأسئلة من هذا النوع ستجد الإجابة الصحيحة ^^... لكن لو افترضنا الحالة الطبيعية أنها مصممة لخدمة هذه ال micro بسبب محددات معينة، فالتكرار هو الخيار الأفضل، وإذا كان ذلك نظرة بعيدة الأمد فلا داع للقيام بتغييرات كبيرة من الآن -وبكل تأكيد نحن لسنا خائفين من الوقت البعيد لأن الـ component بنيت moduler، وعملية الفصل يمكن أن تتم بسهولة وفي أي وقت!-... ولذلك وبناء على هذه المعطيات يمكن استخدام هذا النهج إذا كان حجم التكرار محدودا فقط!

ملاحظة مهمة: هذه القاعدة تختلف باختلاف حجم التغيير وخطورته وخطورة انعكاسه، أي لا يمكن إسقاطها على كل شيء وفي كل مكان، فاحذر... فقد يكون العكس هو الصحيح، وعمل abstraction هو الصحيح، في النهاية ما يحكمك هو السياق الذي أنت بداخله، وأن تعلم أن بعض التكرار قد يكون مفيدا أو أحسن من

غيره من الحلول.

- ال Abstract your code into a shared library: هناك بعض الحالات التي تجعل ال centralize لل business logic أمرا مهما وضروريا ولا استغناء عنه، وهذا لتجبر جميع ال micros على استخدام ال logic / code الموجود بال shared library، ومن أشهر الأمثلة على ذلك ال checkout، فمثلا عملية الدفع والتحقق منها وما يلزم من عمليات لا يمكن أن نقبل أن تكون مبنية بشكل أحادى من كل فريق في كل micros، لأن هذا سيزيد من المشاكل الأمنية والتقنية المحتملة ويعقد الاختبار ويصعب من عملية إضافة أو تعديل هذه الخدمة لأن ذلك سيتم على مستوى جميع ال micros... إلخ من المشاكل الخطيرة، لذلك وجود مكان يشارك شيفرة برمجية معينة أو library معينة بين ال micro frontend هو خيار جيد في بعض المواقف... لكن عليك أن تحرص على بناء atomization pipeline للتحقق من إصدارات هذه المكاتب عند كل micro موجودة، بحيث لو تغير إصدار ال checkout من v1 إلى v2، سيتم معرفة ذلك بشكل تلقائي واتخاذ الإجراء المناسب... واعلم أن هذه العملية تتطلب فرض بعض الممارسات على الفرق لصالح المنظومة ككل.

- ال Delegate to a backend API: في بعض الحالات قد يكون من المناسب تفويض ال Backend ليقوم هو بإرجاع الأجزاء المشتركة أو التفاصيل المشتركة لكل micro-frontend موجودة، ومن الأمثلة التوضيحية لو افترضنا وجود validation

معين لتحقق ما يمثل regExp، هذا ال validation سيتم استخدامه في كل ال micros، فهنا يمكننا بناء shared lib لذلك، لكن لماذا قد نقوم بذلك ولدينا خيار آخر بسيط وهو إرجاعها من خلال ال API على شكل configuration وحفظها داخل ال web storage ومن ثم استخدامها من جميع ال micros، وهذا يقدم لنا عدة فوائد أهمها أننا لن نحتاج لعمل deploy إذا قمنا بتغيير ال validation، ولن نحتاج لتتبع الإصدارات على صعيد كل micro...

ملاحظة مهمة: تذكر أنه لا يوجد هناك حل مثالي أو fit لكل الحالات، فكل سياق الحل المناسب له، فلا تنسى هذه القاعدة أبدا!

ملاحظة ٢: DRY: إن المعنى الشائع لهذا المعنى هو لا تقم بتكرار الشيفرة البرمجية مرتين، وهذا معنى صحيح لكنه سطحي جدا! لكن المعنى الصحيح لهذا المصطلح هو لا تكرر نفسك! أي لا تقم بتكرار الفكرة أو المنطق أو القاعدة في أكثر من مكان في شيفرتك البرمجية!
ولتتضح الفكرة أكثر لنأخذ أمثلة:
في المعنى المشهور أو السطحي لدينا:

```
// DRY - 1

let radius : number  = 5;
const circle_area1 : number  = 3.14 * radius * radius;
console.log(circle_area1);

radius = 10;
const circle_area2 : number  = 3.14 * radius * radius;
console.log(circle_area2);
```

في المعنى الثاني وهو تكرار في المنطق:

```
// DRY - 2

// 1 ملف
if (employeeType === "manager") {
  let bonus = salary * 0.2;
}

// 2 ملف
if (role === "manager") {
  let extra = baseSalary * 0.2;
}
```

في المعنى الثالث: وهو تكرار في نفس المعرفة -وسنأخذ مثالا غير الكود حتى يتضح المعنى-:

```
// Config file:
// be_url: api.example.com/v1
// fe_url: v1.example.com
// test_server: test-1.example.com/v1
```

في هذا المثال المشكلة هي بال v1، أي أننا قمنا بتعريفها بشكل يدوي مع كل رابط، مع أنها يفترض أن ترتبط بقيمة dynamic تتغير بناء على ال config الذي تقرأه وترتبط به الأنظمة.

```
// Backend
if (totalORder < 10) {
  showError();
}

// Frontend
if (totalORder < 10) {
  raiseValidationError();
}
```

في هذا المثال المشكلة تكمن أن القيمة ١٠ والمرتبطة بال Business Logic تم وضعها بشكل static في كل من ال FE/BE، وهذا يعني أن القيمة لو تغيرت سيتوجب علينا تغييرها في مكانين... والحل يكون بإنشاء ملف config لها أو حفظها داخل قاعدة بيانات.. إلخ.
والكثير من الأمثلة، والقصد من كل ذلك هو أن تنظر إلي DRY باعتباره منظومة كبيرة وليس مجرد تكرار حرفي الأسطر البرمجية.

والآن شاهد مثال بسيط يطبق هذه المعمارية:

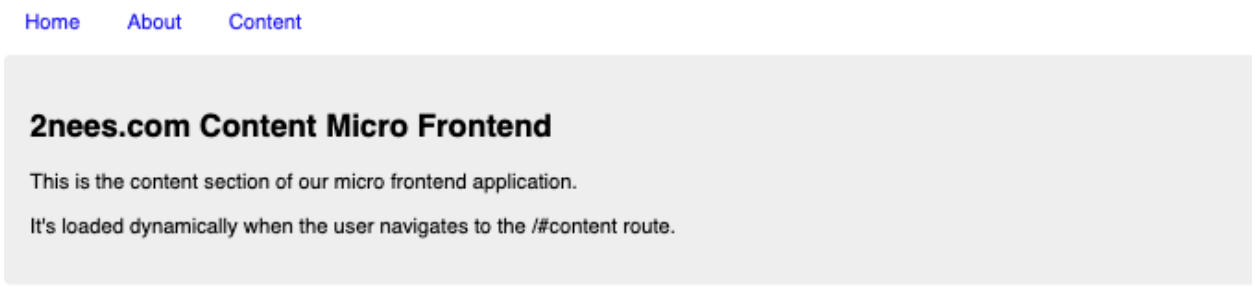
```html
<body>
<nav>
  <a href="/" data-link>Home</a>
  <a href="/#about" data-link>About</a>
  <a href="/#content" data-link>Content</a>
</nav>
<div id="app"></div>

<script type="module" src="app.js"></script>
</body>
```

```js
class About extends HTMLElement {
  connectedCallback() {
    this.innerHTML = `
      <div style="background: #333; color: white;
        <h1>About MFE - 2nees.com</h1>
      </div>
    `;
  }
}
customElements.define('mfe-about', About);
```

Home About Content

Welcome to the App Shell

Use the navigation above to load micro frontends.



ويمكنك الدخول إلى هذا المثال من خلال هذا الرابط.

ال Implementing a Design System:

عندما نعمل على معمارية فيها العديد من ال micros فإننا سنحتاج لآلية ما لتنفيذ التصميم وتوحيد ال UX، وهذا الأمر قد يظنه البعض صعبا، لكنه على النقيض من ذلك، فهو يشبه من الناحية التقنية ما نعمل عليه عادة عند تصميمنا لمواقع SPA...

لبناء تصميم يخدم هذه ال micros يمكننا أن نتخيل نظاما مكون من ٤ طبقات، وهذه الطبقات هي:

1. ال Design token: وهذه تشمل التفاصيل الدقيقة في أي تصميم مثل حجم الخط ونوعه، الألوان المستخدمة ونحو ذلك، وهذه عادة ما يتم وضعها في config file ولا يتم استخدامها مباشرة وإنما تورث أو تستخدم داخل ال Basic component، أما في حال كنت ستكتفي بوجود هذه الطبقة ثم ستذهب للطبقة الرابعة مباشرة -الصورة بالأسفل- فهنا يجب أن تضع في ذهنك أن هناك وقت وجهد وتكرار في التطوير

سيكون موجودا على مستوى كل فريق لكل micro مستخدمة، وهو أحد المسارات المستخدمة في بعض الحالات.

2. ال Basic Component: وهذه تشمل المكونات الأساسية في أي موقع إلكتروني مثل ال button وال label... وبكل تأكيد يجب ألا تحتوي أي business logic!
3. ال UI Library: وهذه عادة مجموعة من المكونات التي تحتوي على business logic معين، وهذه لا يفضل استخدامها وبنائها إلا إذا كان لدينا logic معقد أو يحتاج للكثير من العمل فهنا يكون دورها منع التكرار وإضاعة الوقت.
4. ال Micro frontend: وهذه ال micro نفسها والتي ستحتوي على التصميم والمكونات المشتركة حتى تظهر لنا النتيجة النهائية، وهنا يجب الحذر من عدد ال dependency المرتبطة بال micro frontend من ال UI library... فإذا زاد عددها عن ٣ أو ٤ فهذا ناقوس خطر يشير إلى بدء تحول النظام إلى distributed monolith.

شاهد الصورة التالية:

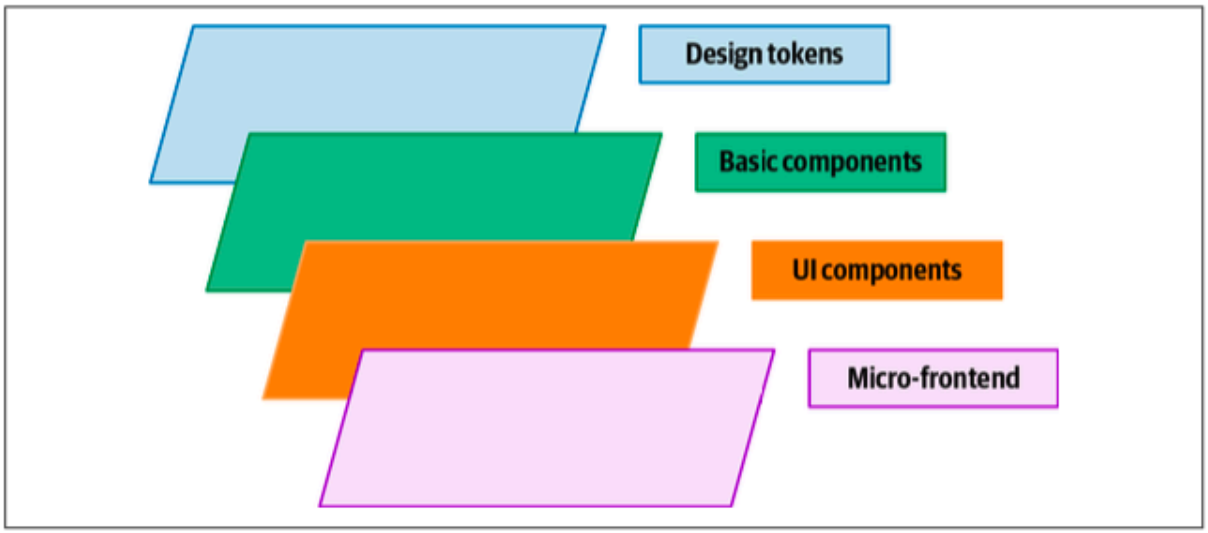


Figure 4-7. How a design system fits inside a micro-frontends architecture

ملاحظة مهمة: إذا كانت ال UI library هي مجرد visual layout لمجموعة من الخصائص ولا تعرف هذه ال component ما الذي يحدث بالضبط بداخلها لو مثلا ضغط المستخدم على button معين -لأنها مرتبطة ب action من ال micro- ولا تحتوي على state مشترك أو business logic فهي بذلك لن تعطل عملية ال deploy المستقلة ولن يشكل إصدار version جديد منها مشكلة... فبذلك لن تشكل هذه الاعتمادية خطر، بل ستكون نقطة جيدة للتصميم، وهي بذلك أقرب لل Basic component من ناحية المعنى وإن صنفت تحت ال UI Library، الخطورة تكمن في وجود ال business logic وتبعاته داخل مكون ما... لأن التغيير هنا يعني أن أي micro مرتبطة به تحتاج إلى إعادة اختبار، وتذكر أن بعض الأحيان تكرار الشيفرة البرمجية أفضل بحسب الحالة من جعلها shared كما تكلمنا سابقا.

بناء على المعطيات السابقة نجد أننا بحاجة إلى نظام أتمتة قوي لضمان الحفاظ على تجربة المستخدم في جميع ال micros، ويمكن ذلك من خلال العديد من الطرق منها:

- التحديث التلقائي لجميع ال micros إذا حدث تغير بالإصدار من نوع معين مثل minor أو patch.
- إجبار المطورين على تحديث ال micro الخاصة بهم إذا كان هنا اختلاف بالإصدارات.. مثل اشتراط ألا يكون الإصدار متقدما بأكثر من one main version.
- بناء dashboard لهذا الغرض يوضح المكتبات والإصدارات الخاصة بها ومن من ال micros لم يتم تحديثها حتى يتم اتخاذ الإجراء المناسب معها.
- يمكن إرسال أو إظهار warning عند صدور إصدار جديد من إحدى مكونات التصميم.

وبهذا نكون شملنا جميع الجوانب في خطة بناء التصميم وتجربة المستخدم في ال micros، ويبقى لدينا جانب مهم وهو كيفية هيكلة الفريق / الفرق لهذا الغرض؟

الشركات في هذا الباب نوعان:

1. شركات تجعل فريق التصميم فريق مركزي بحيث يقوم بتجهيز وتوفير المكتبة للمطورين، ويستلم التصميم من الفكرة وحتى التنفيذ.
2. شركات تعتمد نموذجا موزعا وأقل مركزية بحث يكون فريق التصميم هو المرجع المركزي لكنه يوفر فقط ال Basic component والتوجه العام لتجربة المستخدم، في

حين تقوم الفرق بناء على هذا التوجيه ببناء مكوناتها أو تطويرها... وهذا يخلصنا من ال bottlenecks الممكنة في النموذج المركزي لكنه يحتاج إلى اجتماعات منتظمة بين المصممين والمطورين، وساعات محجوزة لمساعدة الفرق على التصميم وهذا يعني collaborative session أعلى.

ال Developer Experience:

من النقاط المهمة في عالم ال micro frontend والتي تأخذ اهتماما أساسيا أثناء اختيار المعمارية وتؤثر على طرق اتخاذ القرار هي تجربة المطور في المعمارية التي سيتم اختيارها، وتعد تجربة المطور في معمارية ال Vertical Split واحدة من أسلس أو أسهل الطرق المستخدمة لبناء ال micro من قبل مطورين ال fe، وذلك لأن المطورين يمكنهم استخدام الأدوات التي تعلموها ويعملون عليها، وهي تشبه بشكل كبير تجربة تطوير مواقع ال SPA، ومن نقاط القوة كذلك هي سهولة الاختبار وال isolation على مستوى كل micro frontend، لكن هناك حاجة للتحقق من أن ال app shell يضمن الانتقال بين ال micro frontend المختلفة، ويفضل أن يكون هذا الأمر منوطا إلى فريق تطوير ال app shell ومن خلال وضع end to end testing.

ومع ذلك هناك بعض الاقتراحات التي يفضل أن تبقيها في ذهنك وأن تفكر فيها مسبقا، مثل:

- إنشاء CLI تستخدم لإنشاء micro frontend جديد تلقائيا مع وجود المكتبات المشتركة التي يجب أن يتم استخدامها في كل ال micro frontend التي لدينا.
- إنشاء لوحة تحكم (Dashboard) تعرض ملخصا لإصدارات الـ micro-frontends المختلفة في ال environments التي لدينا.
- الكثير من الأدوات التي كنت تستخدمها مع ال SPA ستتمكن هنا من استخدامها دون الحاجة لجهد يذكر، وستكون صالحة بشكل مباشر غالبا مثل ال testing library!

سنتحدث بمزيد من التفصيل حول ذلك وال Automation Pipelines في الفصول التالية بإذن الله تعالى.

ال Search Engine Optimization:

يعد ال SEO واحدا من أهم المواضيع التي ينبغي علينا الاعتناء بها في عالم Micro frontend، خصوصا إذا تعلق الموضوع بالمواقع ذات المحتوى التي تعتمد في انتشارها على محركات البحث مثل مواقع التجارة الإلكترونية أو الصفحة الرئيسية لموقعك ونحو ذلك، ولأننا الآن في معمارية Vertical split فإننا أمام تحد مهم، فكل جزء من التطبيق الخاص بنا يمثل SPA مستقل، ولأن هذه ال SPA غالبا ما يتم اختيار ال render الخاص بها ليكون عند ال client side، فإن ال crawlers الخاصة بمحركات البحث قد تعاني أو تفشل في قراءة صفحاتك، وذلك لفشلها في تحميل وتشغيل ملفات الجافا سكربت، فال crawlers غالبا

لن تنتظر وقتا طويلا حتى يتم عمل render لل SPA! ولذلك هناك العديد من الحلول المميزة لهذه المشكلة ^^.

الحلول الممكنة:

- إن أول حل قد يخطر على بالنا هو تقليل الوقت المستغرق لتحميل ال SPA، وذلك من خلال عمل Client-Side optimization مثل ال chunking ونحو ذلك! ولتحقيق هذا الأمر ينبغي علينا ضمان تحميل كامل ال DOM الخاص بنا خلال ٥ ثواني كحد أقصى -والأصل أقل من ذلك بكثير-! كما ينبغي علينا عرض جميع المعلومات المهمة وتجهيزها مباشرة في ال DOM دون الاعتماد على actions أو events من المستخدم (مثل الضغط على كبسة حتى يظهر هذا المحتوى)...
- الخيار الثاني هو Static Html Markup، وفيه يتم بناء صفحة Html لكل صفحة موجودة عنا بال SPA قابلة للفهرسة، وبهذا تتمكن محركات البحث من قراءة وفهرسة الصفحات دون الحاجة لتنفيذ ال js code، وهذا يشبه ما يقوم به ال nextjs من عمل static html pages عند ال build لل Static Generation، ويمكن بالقيام بهذه العملية يدويا أو من خلال أي أداة مناسبة لمشروعك...
- الخيار الثالث هو ال Dynamic Rendering -لا ينصح بها الآن، لكن استمر للنهاية-، وهذه الطريقة تجمع بين أمرين وهما الحفاظ على تجربة المستخدم للموقع أو النظام الخاص بنا مع تحسين الفهرسة للموقع ^^، وفكرة هذا الخيار هي اكتشاف الـ user-agent ثم إعادة توجيهه بناء على نوعه، فإذا كان ال user-agent عبارة عن مستخدم طبيعي فسنقوم بتوجيهه إلى الـ micro frontend الخاصة بنا بشكل طبيعي،

أما إذا كان bot مثل googlebot فإننا نقوم بتوجيهه إلى prerendered page، ويقصد بها تلك الصفحة التي تم تجهيزها مسبقا لترجع لك نفس البيانات التي يراها المستخدمون، ولدينا هنا خيارين إثنين لتطبيق هذه الآلية وهما:

- أثناء ال Compile time وذلك من خلال استخدام ال SSR عن طريق بناء template ثم استخدامه لحفظ ال static html output الخاص بها على ال object storage مع الحفاظ على نفس ال url structure التي يتعامل معها المستخدم. (أي تقوم بجلب template ال html وقت ال compile time ثم يكون فيه كود يجلب البيانات الممكنة لهذه الصفحة ثم يقوم بحفظها ومشاركتها)، وهي غير مناسبة مع المحتوى الذي يجب عكس تعديلاته بشكل فوري ولدينا قدرة على بناء pipeline يخدم هذه العملية.
- أثناء ال Runtime وذلك من خلال توليد صفحة ال html على ال server عندما يقوم ال bot بطلب هذه الصفحة فورا، وهذا الحل نقطة قوته أننا لسنا بحاجة لحفظ أو تخزين صفحات ال html الناتجة من هذه العملية، بل نقوم بإرسال هذه النسخة مباشرة لل bot الذي طلبها، إلا أنه يحتاج لجهد أكبر بالتنفيذ، وهو من أكثر الخيارات مرونة وكفاءة من حيث تخصيص وتحسين نتائج البحث لموقعك -قديما، والآن لا ينصح به-، ويمكن استخدام ال Puppeteer أو Rendertron لهذا الغرض!

ملاحظة مهمة: إن مع التحسن الملحوظ الذي حصل على ال bots الخاصة بمحركات البحث جعل من عملية الفهرسة للصفحات أمرا أكثر سهولة وأكثر فاعلية، فقد صارت صفحات ال

SPA التي تحتاج إلى تنفيذ سكربت قبل فهرستها يتم تتبعها ولم يعد يتجاهلها ال bot كما كان من قبل -ضمن قواعد معينة-، لذلك أطلقت جوجل تحذيرا من استخدام ال Dynamic rendering ك workaround لهذه المشكلة وذلك لأنه يضيف تعقيدا إضافيا، ويتطلب موارد أكثر، ونصحت باستخدام ال SSR أو Static Rendering أو ال Hydration... ومع ذلك ما زال بإمكانك استخدامه إذا كنت ترغب بتحمل هذه التكلفة كحل مؤقت لتحسين محرك البحث عندك!

فائدة

فلْتعلمْ أن الأوثان التي كان يعبدها المشركون ما زالت موجودة حتى وقتنا الحاضر! لكن اختلفت الطرق والأساليب وتعددت! فتجد كثيرًا من شباب اليوم مدمنين على الألعاب والأفلام والمسلسلات الهابطة وأفلام الزنا عاكفين عليها -والعياذ بالله-، لا يتركون هواتفهم المحمولة أو أجهزة الحاسوب إلا لغرض من أغراض الدنيا، ومنهم من وثنه المال وجمعه، ومنهم من وثنه الجاه وسلطته...إلى آخره، لذلك احذر من أن تكون من أصحاب الأوثان، فتُلهيك عما خُلقت لأجله، وتلهيك عما ينفعك من التعلم والعمل الصالح النافع، وما إلى ذلك من الأعمال المفيدة للإنسان...، لذلك نظمْ وقتك واصرفه فيما ينفعك، واجعل من عملك وراحتك وقعودك وحركتك بابًا لتطور به من نفسك على صعيد الروح والبدن، فتكسب الدنيا والآخرة -بإذن الله-.

- كتاب إلى الجنة زمرا، الصفحة ١١٧ - ١١٨

ال Performance and Micro-Frontends:

من التحديات التي نواجهها جميعا عند تصميم المواقع الإلكتروني هي الحصول على أفضل أداء ممكن، لأن ذلك عامل مهم ومؤثر على تجربة المستخدم والذي بدوره سينعكس مباشرة على كمية المستخدمين الذين يستخدمون الموقع، وقد يكون قرار استخدام موقع ما من عدمه مناطا بأدائه وقبل القيام بأي مهمة!

إن ال micro frontend حالها كحال أي موقع تقوم بتصميمه، فيجب الاعتناء بالأداء الخاص بها، ومن خلال بنية ال vertical split يمكننا الحصول على ميزة جميلة وهي تحقيق أداء جيد بفضل تقسيم الموقع إلى عدة أجزاء بحيث يمثل كل جزء micro مستقل يتم تحميله عند الطلب، وهذا من الناحية النظرية سيحقق لنا أداء أفضل من تحميل موقع SPA فيه نفس الخصائص لأن المستخدم ملزم بتحميل كامل الأجزاء مرة واحدة، وحتى لو لم يكن يحتاجها، وهذا يقودنا لمفهوم مهم، وهو **أن تحسين تجربة الأداء ليست مرتبطة بالجانب التقني فقط! وإنما ترتبط ارتباطا وثيقا بسلوك المستخدم على الموقع!** وحتى نفهم هذه العبارة بشكل أوضح تخيل الآتي: لو كان لديك صفحات مثل ال signup وال login وال home page وال product for auth user... وكانت تجربة المستخدم هي الدخول للصفحة الرئيسية ثم التسجيل أو تسجيل الدخول، وإذا كان يحتاج ال product سيقوم من الرئيسية بالانتقال لل login ثم ال product list... من خلال فهم هذا السلوك يمكننا أن نقوم ببناء أداء يفصل ال bundle بحيث يتم تحميل ال home page لوحدها وستكون بال init bundle، وال product لوحدها ولا يتم تحميلها إلا إذا كان المستخدم auth، وصفحة ال

login/signup في chunk ويتم تحميلها lazyload عند الطلب ... وبذلك ارتبط تحسين الأداء بما يقوم به المستخدم أو سلوكه، وهذا ما يطلق عليه: Behavior-Based Chunking.

والآن، كيف هذا سيؤثر كله على الأداء؟

لفهم ذلك تخيل الآتي، لدينا موقع إلكتروني حجم ال bundle هي ٥٠٠ كيلو بايت، مقسمة على جزئين، ٢٥٠ كيلو بايت لل vendor و٢٥٠ كيلوبايت للصفحات الموجودة في الموقع، وهذه الصفحات ١٠٠ كيلوبايت منها للصفحات التي لا تحتاج إلى تسجيل دخول و ١٥٠ كيلوبايت للصفحات التي تحتاج لتسجيل الدخول... انظر الصورة F4-9:

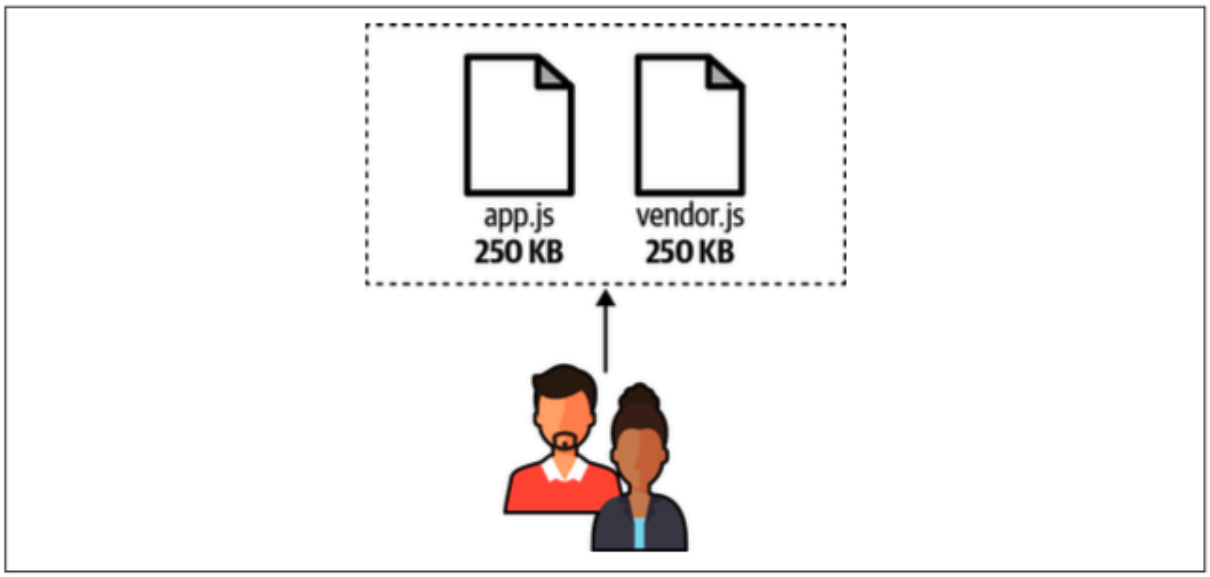


Figure 4-9. Any user of an SPA has to download the entire application regardless of the action they intend to perform in the application

من خلال هذا الكلام نفهم أن موقع ال SPA التقليدي سيقوم بتحميل ال 500 كيلوبايت هذه بمجرد دخول المستخدم للموقع وبغض النظر عن احتياجه الفعلي، بينما هذه المشكلة لن

تكون موجودة بال micro frontend لأن كل micro تمثل جزء معين من النظام، وبهذا سيتم تحميل فقط الأجزاء المطلوبة، فلو افترضنا أني مستخدم غير مسجل الدخول وقمت زيارة صفحة ال help فإن ما أحتاجه هو application shell وهو ملف صغير الحجم لن يؤثر ولنفرض أنه ١٥ كيلوبايت وصفحة ال help تساوي ٣٠ كيلو بايت فيصبح لدينا الآتي:
٢٥٠ + ٣٠ + ١٥ = ٢٩٥ كيلو بايت من أصل ٥٠٠ كان سيتم تحميلها! شاهد الصورة F4-10:

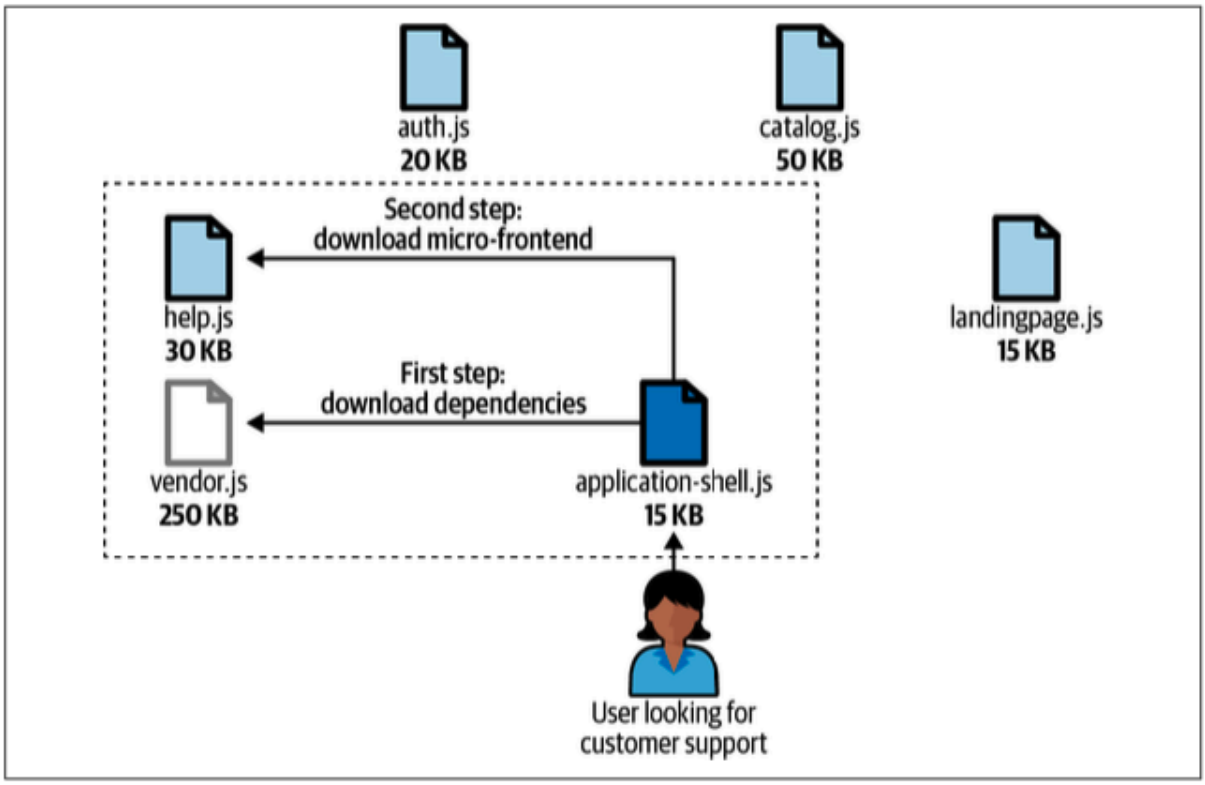


Figure 4-10. A vertical-split micro-frontend enables the user to download only the application code needed to accomplish the action the user is looking for

وقد يتساءل البعض: ما هي الفائدة العملية من كل هذا إذا كان المستخدم سيتنقل بين الصفحات؟ والجواب يكمن ببساطة أن المستخدم سيقوم بتحميل الملفات المطلوبة بالتدريج

وحسب الحاجة ^^، وليس دفعة واحدة، وهذا يعني أداء أفضل لأنه سينعكس بالإيجاب على مقاييس الأداء Web Vital Metrics.

والآن، ما هي التحديات التي تواجهنا في مثل هذه الحالة؟ أكيد الحياة مش وردية ^^ أول تحدي أمامنا هو وضع مجموعة من القواعد أو المحددات لكل micro frontend بحيث نحافظ على الأداء الجيد لها، وهذا ما يطلق عليه Performance Budget. كما علينا معاملة كل micro frontend في المشروع على أنها موقع SPA مستقل نريد تحسين أداءه والوصول لأفضل أداء ممكن له، وهذه نقطة مهمة وخطيرة.

ملاحظة: ال Performance Budget هي إستراتيجية تستخدم أثناء تطوير المواقع لتحديد الحدود القصوى والمسموحة لمقاييس الأداء، بحيث لا يتم يتجاوزها، والهدف الأساسي منها هو ضمان أن تبقى تجربة المستخدم سريعة وسلسة على جميع الأجهزة والشبكات -قدر الإمكان-، وخاصة تلك الأجهزة أو الشبكات ذات الاتصال البطي أو القدرات المحدودة، ومن الأمثلة على ذلك: ال final bundle size وال total css size وأكبر عدد من ال requests في ثانية واحدة...، ويمكن استخدام أداة مثل ال Lighthouse لأجل التحقق.

ومن التحديات الخطيرة والمهمة هي كيف سنتعامل مع ال shared library بال vertical split، ولدينا خيارين هنا، هما:

1. تجميع جميع ال library المشتركة في مكان واحد تستخدمه جميع ال micros، هذا الخيار يقدم أداء ممتاز لأن المستخدم حتى لو تنقل بين ال micros لن يتم تحميل

المكتبات المشتركة لأنه قد تم تحميلها بالفعل، لكن هذا الأمر يتطلب تنسيق كبير بين الفرق المختلفة + إذا كان هناك حاجة لتطوير الإصدار الخاص بمكتبة ما -major change- فعلى جميع الفرق تحديث ال micros الخاصة بهم حتى يتم عمل ال release.

2. تجميع جميع ال library في كل micro على حدا: في هذا الخيار كل micro يدير نفسه بنفسه بشكل مستقل، فلا اعتمادية بينه وبين ال miros الأخرى، وهذا يقلل بدوره من الحاجة للتنسيق بين الفرق أيضا، لكنه يزيد من حجم البيانات المحملة للمستخدمين.

وهنا نذكر مجددا بمجموعة من القواعد المهمة في هذا الباب، وهي:

- لا يوجد قرار خاطئ أو صحيح هنا في اعتماد الآلية التي سيتم تحسين الأداء من خلالها بشكل مطلق، لأن كل أسلوب لديه مميزاته وسلبياته.
- لا تخف من اتخاذ القرار الذي تراه مناسبا لتحسين الأداء، فإن اكتشفت أن الخيار الأنسب للمشروع -رغم عيوبه- هو ما اخترته فعلا فهذا ممتاز، وإلا فيمكنك تعديل هذا الخيار بسهولة!
- من المهم مراقبة المستخدمين وطريقة تفاعلهم من الموقع الخاص بنا، لأن ذلك عامل مهم ومؤثر على الأداء.
- قد يكون القرار الذي اتخذته لتحسين الأداء هو الأفضل في مرحلة ما، لكن عليك مراقبة التغييرات التي قد تطرأ مع المتطلبات الجديدة والخصائص الحديثة، فكن مرنا بالتعديل إن حصل هذا.

ال Available Frameworks:

يمكننا كمطورين FE أن نقوم ببناء ال Application Shell بأنفسنا، وكما ذكرنا فإن العملية سهلة، ولن تحتاج إلى استخدام أي framework خارجي، على أن تحافظ على الفصل بين ال App shell وال micro frontend، ومع ذلك هناك بعض ال Framework المتوفرة لدعم ال micro frontend، ومن الأمثلة على هذه ال frameworks:

- ال single-spa
- ال qiankun

يعد ال single-spa واحدا من أطر العمل ال lightweight، لكنها تتكفل بال undifferentiated heavy lifting، أي أنها ستنزع عنك عبء الإعداد المتكرر من خلالها، والحلول الأساسية التي تقدمها القدرة على عمل register ل micro frontend جديد، وعمل bootstrap له (تنفذ عند اول تحميل) وعمل Mount (تشغيل أو عرض التطبيق في الصفحة) و UnMount (إزالة التطبيق أو العرض من الصفحة) عند الانتقال لصفحة أخرى... مع خيارات أخرى، أي أنه باختصار يزيح عنك عبء ال Routing وال Lifecycle والفصل بين ال micros.

أما متى سترغب باستخدام framework ومتى سترغب في بنائه بيدك فهذا القرار يعود لك بناء على معطيات العمل عندك وعدد المطورين والنظر لمستوى التعقيد الذي سيضاف والمحددات من استخدام framework من عدمه والخبرة السابقة في ذلك...

أيضا من الطرق التي يمكنك استخدامها لهذا الغرض ال Module Federation -كما ذكرنا سابقا-، وسنتحدث عن الآلية في الأجزاء القادمة -بإذن الله تعالى-، وعادة ما يستخدم هذا الأسلوب مع ال Horizontal split، إلا أنه لا يوجد مانع من استخدامه مع ال Vertical split وقد يكون خيارا جيدا...

إذا، بعد حديثنا هذا:

متى يكون من المناسب استخدام Vertical Split؟

1. إذا كان مطورو ال FE لديك معتادين على بناء ال SPAs، فإن Vertical Split هو خيار ممتاز لهم لأنه:
    a. يُحافظ على نفس المفاهيم (routing, component state, rendering...).
    b. سهل التعلم ولا يتطلب خبرة متقدمة لبدء استخدامه
    c. كل Micro-Frontend يُعامل كـ SPA مستقلة.
2. إذا كان لديك فرق صغيرة أو متوسطة فيمكن استخدام هذه المعمارية للتوسع (Scalability)، لكن إذا زاد عدد مطوري ال FE وصار ضخما بالمئات، فيفضل

الانتقال من ال Vertical إلى ال Horizontal لأن ذلك يسمح بتقسيم أدق داخل كل صفحة ويقلل من التداخل بين الفرق ويتيح قابلية أعلى لإعادة الاستخدام داخل نفس الصفحة...

3. إذا كنت تحتاج إلى اتساق في تجربة المستخدم (UI/UX Consistency) ، فكل فريق لديه مجال معين ومحدد يعمل عليه، فمثلا لدينا فريق يعمل على ال Product وآخر على ال Payment، فهذا يساعد في الحفاظ على اتساق العمليات داخل ال micro frontend، لأنه يتيح تجربة تطوير كاملة end-to-end.
4. إذا كانت إعادة الاستخدام محصورة في ال UI Components أو Libraries، فكل micro frontend يستخدم مكتبات مشتركة مثل ال logging أو ال payment gateway، وتقوم باعادة استخدام ال UI component فقط وليس shared logic component! أما إذا كانت الأجزاء الموجودة في نفس ال micro ستتكرر في أكثر من صفحة فهنا قد تفكر بالانتقال إلى ال horizontal، مثل لو كانت صفحة المنتجات تظهر لديك في أكثر من micro أو في أكثر من صفحة...
5. أن يكون هو الخيار الأول أو المبدئي لك عند تبني نهج ال Micro frontend، فكما ذكرنا سابقا انطلاقة المشاريع بهذا النوع لا يزيد التعقيد ولا يحتاج لخبرة كبيرة، كما أنه يقدم تقسيم واضح لل Business domain، ولا يتطلب أدوات متقدمة لتطبيقه، كما أنه يسمح بتوزيع العمل على عشرات المبرمجين بسهولة [٨].

والآن، إلى الجزء الأخير في هذه المعمارية وهو:

ال Architecture Characteristics:

إن لكل معمارية خصائص تتميز بها عن غيرها، هذه الخصائص مما يساعدك على اتخاذ القرار، ولقد ضربنا سابقا مثالا فيه مجموعة من القيم للمقارنة بين خصائص معمارية ال Vertical وال Horizontal لموقع تجارة إلكتروني، لكن الآن لنأخذ الأرقام بناء على معطيات تفصيلية.

1. ال Deployability (سهولة النشر) - العلامة ٥/٥ ^^:

   لأن كل Micro-Frontend هو عبارة عن SPA أو صفحة HTML مستقلة فيمكن عمل deploy بسهولة على أي App Server أو Cloud Storage، كما يمكن استخدام ال CDN لتحسين الأداء وزيادة السرعة ^^، بل ويمكن استخدام أكثر من CDN عند الحاجة لتفادي أي فشل ممكن وضمان وصول المحتوى للشباب الطيبة :P.

2. ال Modularity - العلامة ٢/٥ ^^:

   البنية التركيبية هنا ليست modular في أغلبها، أي أن إمكانية مشاركة feature كاملة بين هذه ال micro وال micro الأخرى ستكون عملية صعبة، كما أنك لو احتجت لفصل ال micro frontend إلى جزئين بسبب مزية جديدة فإن ذلك يحتاج منك جهدا ووقتا لأن المكونات وال state ونحوها مترابطة بشكل وثيق.

3. ال Simplicity - العلامة ٤/٥ ^^:

   كمت تحدثنا فإن هذه المعمارية لا تتطلب جهدا كبيرا في تعلم تقنيات جديدة، فتعلم

framework مثل single-apa أو تعلم الـ module federation لن يكون صعبا أو يشكل عبء عقليا على المبرمجين، كما أن الأدوات المستخدمة هي ما اعتاد عليه المبرمجين، لذا بساطة هذا الأسلوب وسهولته هي من أبرز نقاط قوته.

4. ال Testability (قابلية الاختبار والفحص) - العلامة ٤/٥ ^^:

مقارنة بال SPA فالفارق هنا هو ظهور بعض التحديات لعمل test على مستوى App Shell للتأكد من أن كل شيء يعمل على ما يرام (مثلا ومن أهمها أن جميع ال micro frontend يتم تحميلها عند التنقل بين ال global route)، وهنا يظهر دور ال End to End testing، أما على مستوى كل micro frontend فهذا ليس مشكلة أساسا لأنه لا يوجد اختلاف فيما كنت تعمل عليه من طرق الاختبار ونحوها...

5. ال Performance (الأداء) - العلامة ٤/٥ ^^:

الأداء في هذه المعمارية ممتاز أيضا، ويمكن مشاركة ال Common Library بين ال Micros المختلفة، لكن هذا سيتطلب تنسيقا بسيطا بين الفرق، والتأخير الحقيقي يحدث هنا إذا كانت ال Micro معقدة وتحتاج إلى الكثير من ال request بين ال client وال server، ومما يؤثر أيضا هنا أن في هذه المعمارية لن يكون هناك المئات من ال micros، لذلك يسهل ضبط الأداء.

6. ال Developer Experience - العلامة ٤/٥ ^^:

إذا كان الفريق معتادا على تطوير مواقع ال SPA، فهذه المعمارية ستكون مألوفة لهم، ومعظم الأدوات يعرفونها، لكن قد تظهر هناك حاجة لتعديل أو تطوير بعض الأدوات للتوافق مع هذه المعمارية، لكن الأدوات الشائعة أو الافتراضية ستكون كافية في البداية، ويمكن تأجيل التعديل أو التطوير للأدوات الإضافية لوقت آخر ضمن حياة المشروع أو خطة العمل...

7. ال Scalability (القابلية للتوسع) - العلامة ٥/٥ ^^:

تتميز هذه البنية بقابلية للتوسع بشكل ممتاز، لدرجة أنه يمكننا تجاهلها وكأنها شيء افتراضي لدينا ^^، فمثلا يمكنك ضبط مدة ال TTL حسب نوع ال Assets، فالملفات التي نادرًا ما تتغير مثل الخطوط نعطيها وقت TTL أطول، أما الملفات التي تتغير بشكل متكرر والتي تحتوي على business logic فنعطيها وقت TTL أقصر، كما أن هذه البنية يمكنها أن تتوسع بشكل شبه غير محدود حسب قدرة الـ CDN، والتي غالبًا ما تكون كافية لخدمة مليارات المستخدمين في نفس الوقت...

8. ال Coordination (التنسيق) - العلامة ٤/٥ ^^:

هذه البنية، مقارنة بغيرها، تُتيح لا مركزية قوية في اتخاذ القرار واستقلالية كبيرة لكل فريق، لكن ذلك يرتبط بوضع ال bounded context بشكل جيد، وهو ما يتطلب التنسيق الأولي عند بناء ال shell لضمان عدم حدوث أي تماس بين ال micros أو ارتباط بين ال shell وال micros.

والآن، شاهد الجدول T4-1:

Table 4-1. Architecture characteristics summary for developing a micro-frontends architecture using vertical split and application shell as composition and orchestrator

Architecture characteristics	Score (1 = lowest, 5 = highest)
Deployability	5/5
Modularity	2/5
Simplicity	4/5
Testability	4/5
Performance	4/5
Developer experience	4/5
Scalability	5/5
Coordination	4/5

ملاحظة: تذكر أن هذه القيم تتأثر بال context الخاص بالمشروع، لكن يمكنك اعتبارها نقطة الأساس أو الانطلاق عند التفكير أو البناء أو اعتمادها في الظروف المعيارية عند البدايات.

فائدة

فلْتعلمْ أن الأماني وحدها لن تجعل منك عالمًا، أو تاجرًا، أو صاحب حرفة ومهارة، أو رجلًا صالحًا! بل ستجعلك تعيش أحلامًا وردية حتى تستيقظ، فتجد نفسك متأخرًا متراجعًا إلى الوراء، فالوقت يمضي، ونحن نمضي، فإن لم تُتابع تقدمك فإنك بالتأكيد ستتراجع! لذلك الأماني يجب أن يرافقها العمل، وكل ذلك وقلبك وعقلك متكلٌ على الله -سبحانه وتعالى-.

- كتاب إلى الجنة زمرا، الصفحة ١١٩

والآن ننتقل إلى ثاني معمارية لدينا وهي:

ال Horizontal-Split Architectures:

في هذه المعمارية يمكننا تقسيم المشروع على شكل مكونات وظيفية داخل نفس الصفحة، بحيث يمكن أن يتواجد أكثر من micro frontend في نفس الصفحة، لذلك فهذه المعمارية تقدم حلولا تلبي احتياجات أغلب تطبيقات ال micro frontend، لكنها في ذات الوقت تحتاج إلى تنسيق عالي بين الفرق ويجب أن تكون الحدود المرسومة واضحة، كما أن الخطأ هنا في التقسيم سيصنع لدينا مشاكل متعددة من أهمها التفصيل الكبير بحيث تصبح ال micro frontend هي component! لذلك من المهم القيام بمراجعات دورية للمخرجات وحدود التقسيم وآلية التواصل وهيكلية الفرق بشكل دوري، وتجنب الاعتماد المفرط على الفرق الأخرى -على كل فريق أن يعرف واجباته ومسؤولياته-، وكل ذلك مع الحفاظ على مستوى معين من الحرية في التطوير.

ولأن هذه المعمارية تتميز بقدرة عالية على تجزئة الصفحات وتركيبها فهذا يجعلها مناسبة للمشاريع الكبيرة أو المشاريع التي تحتاج لمشاركة أجزاء معينة من الكود بكثرة، كما أنها مناسبة للشركات التي تمتلك فريق كبير من المطورين.

ومن الأمثلة على هذا النوع من التقسيم صفحة منتج ما في موقع تجارة إلكتروني، فمثلا يمكن تقسيم هذه الصفحة إلى:

- ال MicroFE-ImageGallary وستكون مسؤولة عن عرض صور المنتج.
- ال MicroFE-ProductDetails والمسؤولة عن عرض معلومات المنتج وسعره.
- ال MicroFE-Recommendation والمسؤولة عن التوصيات المرتبطة بهذا المنتج.
- ال MicroFE-Reviews وفيها مراجعات المستخدمين لهذا المنتج...

كل واحدة من هذه ال micros يتم تحميلها في الصفحة كمكون منفصل، وقد يكون أكثر من فريق قد قام بتصميمها.

وهنا نأتي لسؤال مهم ونقطة مهمة قبل الانتقال والتعمق في جزئيات هذه البنية، وهو الفرق بين الـ Micro Frontend في هذه المعمارية وال Component ^^، وحقيقة هذا التساؤل مهم جدا هنا، فهيا بنا لنجيب عن هذا السؤال ونوضح الفرق:
إن أجمل قاعدة توضح الفرق بين ال Component وال Micro-FE هي أن نسأل أنفسنا: هل ما نقوم بتصميمه سيتم استخدامه في أكثر من مكان بأكثر من سلوك، أي سلوك مختلف حسب المكان الذي سيقوم باستخدامه؟ وهل سيتم استخدام ال props لتغطية هذه العملية؟ أم أن ال logic سيتم تضمنيه بداخل هذا الجزء وسيتم ال communication من خلال ال events؟
إذا كان جواب السؤال الأول والثاني نعم فهذا دليل على أنها Component قابلة لإعادة الاستخدام، وهذا يشبه ال Buttons، فمثلا يمكن إرسال onClick لل button الأولى لجمع رقمين والثانية لإظهار modal... أما إذا كان جواب السؤال الثالث هو نعم، فهذا دليل قوي

على أنها Micro-FE، لأننا نحتاج إلى إدارة جميع العمليات في داخلها، وأن يتم نشره وتحديثه بشكل مستقل وبال logic الخاص به... وهذا يشبه مثال عربة التسوق [٨٨].

والآن سنذهب إلى أول تحديات هذه المعمارية، وهي آلية التجميع، وكما تحدثنا سابقا فلدينا ثلاثة خيارات لتجميع ال micros في ال Horizontal split وهي:

- ال Client side
- ال Edge Side
- ال Server Side

ال Client Side:

إن تجميع مكونات الصفحة هنا يجب أن يتم أيضا من خلال وجود Application Shell، لكن الفرق الجوهري هنا عن ال Vertical Split أننا سنقوم بتجميع أكثر من micro في نفس الصفحة، وهذه الأجزاء يمكن أن يبنيها فريق واحد أو أكثر من فريق، ويتم تجميعها بشكل dynamic على المتصفح... لكن نجدد التذكير على أهمية ألا نقع ضحية للتفكير الزائد فيصير كل component هو micro frontend، بل علينا الحفاظ على نظرتنا من ناحية شمولية تبدأ من الـ business perspective.

والآن لنشاهد مثالا عمليا يوضح الفكرة، شاهد الصورة: 4-11:

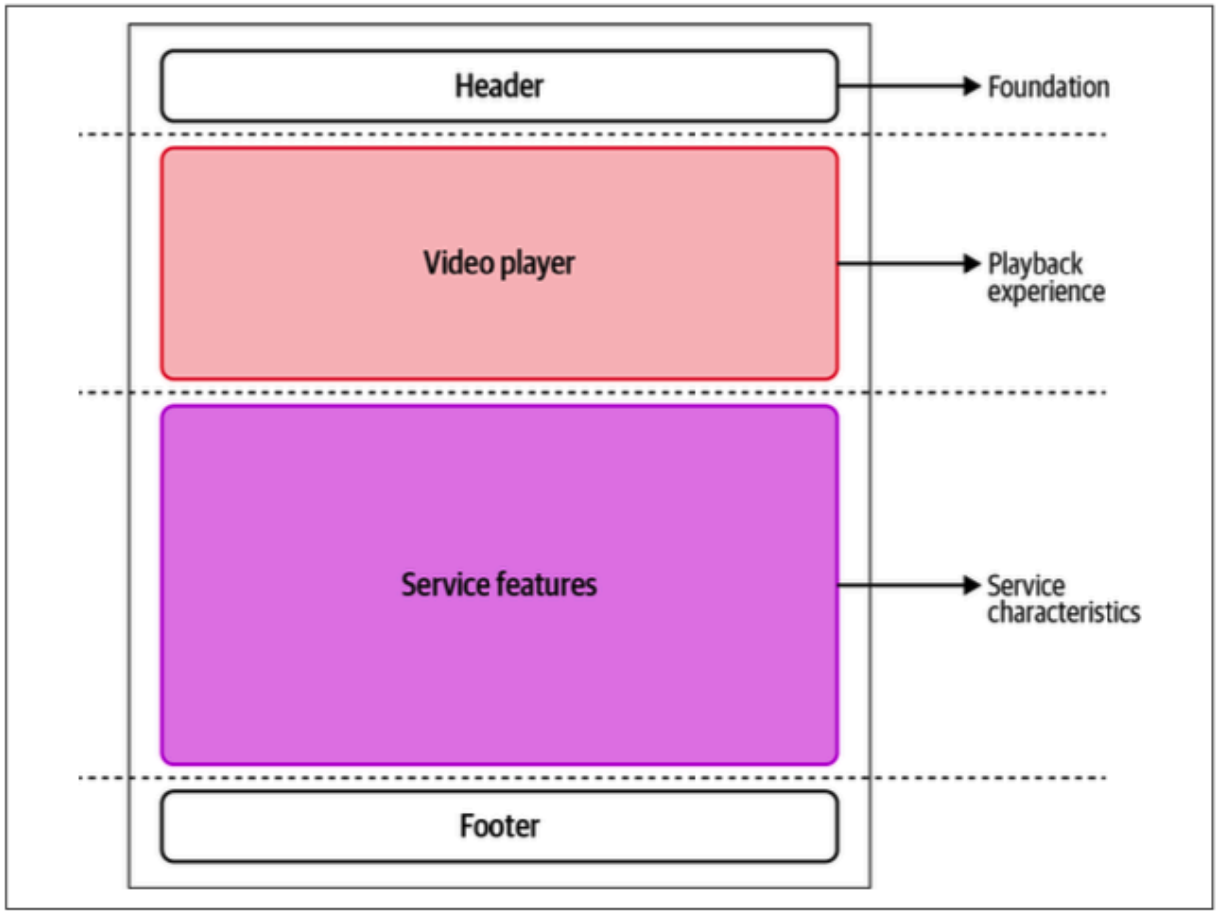


Figure 4-11. The landing page view is composed by the application shell, which loads two micro-frontends: the service characteristics and the playback experience

هذه الصورة عبارة عن مثال توضيحي لصفحة تم تصميمها اعتمادا على horizontal split، وهو يمثل موقع لتشغيل الفيديو ستريم، ولدينا هنا ٤ فرق ستعمل على هذه الصفحة وهي:

1. ال Foundation team: وهو يمثل الفريق الأساس والذي يتمحور دوره على بناء App shell وربط ال micros داخله، والعمل مع ال UX من ناحية تقنية لتطبيق ال system design.

2. ال Landing page team: وهو الفريق المسؤول عن إنشاء صفحات ال landing page (لترويج المنتج وعروض المنتج وخطط الاشتراك...) المختلفة حسب الطلب والحاجة، ويعمل على دعم فريق ال marketing.
3. ال Catalog Team: وهو الفريق المسؤول عن عرض المحتوى المتاح للبث للمستخدمين بعد تسجيل الدخول، والتعاون مع الفرق الأخرى لتحقيق هذه الغاية.
4. ال Playback Experience Team: وهو الفريق المسؤول عن تطوير مشغل للفيديو لأن مشغلات الفيديو الموجودة لا تلبي الاحتياجات التي نريدها، لأننا نحتاج إلى وسيلة أمان ومنع للاستخدام الغير مرخص مع إدارة تحليلات للفيديوهات...

إن آلية العمل حسب التسلسل والفرق التي ذكرناها ستكون كما يلي:
سيقوم ال Foundation team ببناء ال App shell وال header وال footer، وسيقوم ال Landing page team بعرض محتوى ال stream بالإضافة لبعض التفاصيل حول المنصة، وسيقوم فريق ال Playback بتوفير مشغل الفيديو مع إمكانية عرض الإعلانات وحماية الفيديو ونحو ذلك، أما في المكان المخصص لل Service feature سيتم عرض micro frontend معينة حسب الصفحة، وهنا سيكون دور ال catalog team مع ال Foundation team، فعند تسجيل الدخول من قبل المستخدم سيتم جلب ال micro frontend التي تم تصميمها من قبل ال catalog team ليتم عرضها في هذا ال block، شاهد الصورة: F4-12.

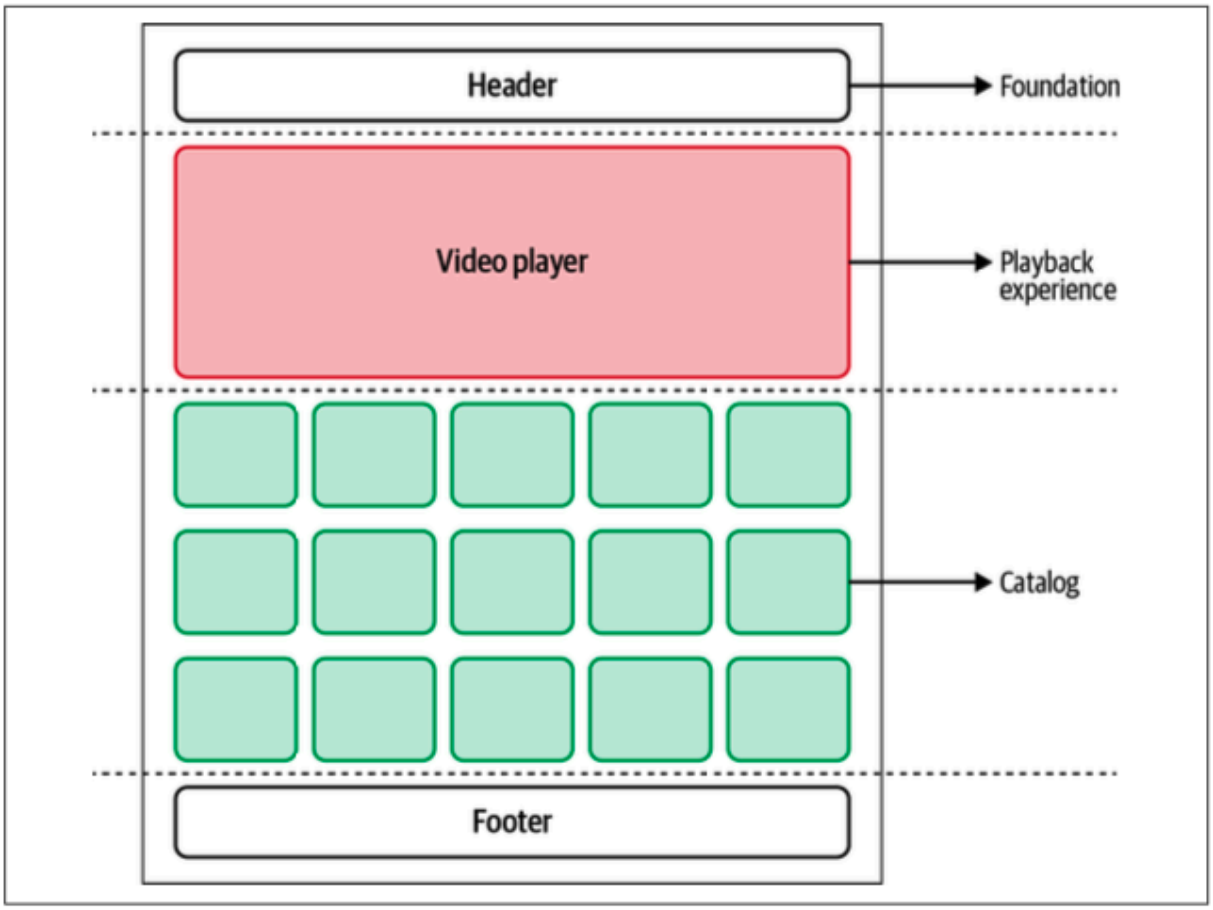


Figure 4-12. The catalog view is composed by the application shell, which loads two micro-frontends: the playback experience and the catalog itself

والآن يمكننا الانتقال للتحديات المناطة بهذا الجزء ^^، فكما هو الحال مع أي بنية معمارية فإن ال Horizontal split له مزايا جميلة وتحديات من المهم التعرف عليها؛ لضمان مدى هذا العمل معك، كما أن تقييم المزايا والعيوب قبل الشروع في عملية التطوير يقرّبك خطوة نحو النجاح.

تحديات ال Horizontal client-side split:

- ال Micro-frontend communication:

إن اعتماد معمارية ال Horizontal split يعني الحاجة لوضع خطة جيدة حول كيفية التواصل بين ال Micros المختلفة في نفس الصفحة وبين الفرق المختلفة، ومن أخطر وأشنع الأخطاء التي يمكن أن تراها في هذا الباب هو استخدام ال State للقيام في هذه المهمة! والسبب يعود في ذلك لتسبب ال state في العديد من المشاكل الخطيرة، انظر للصورة F-14 أولا ثم تابع معي...

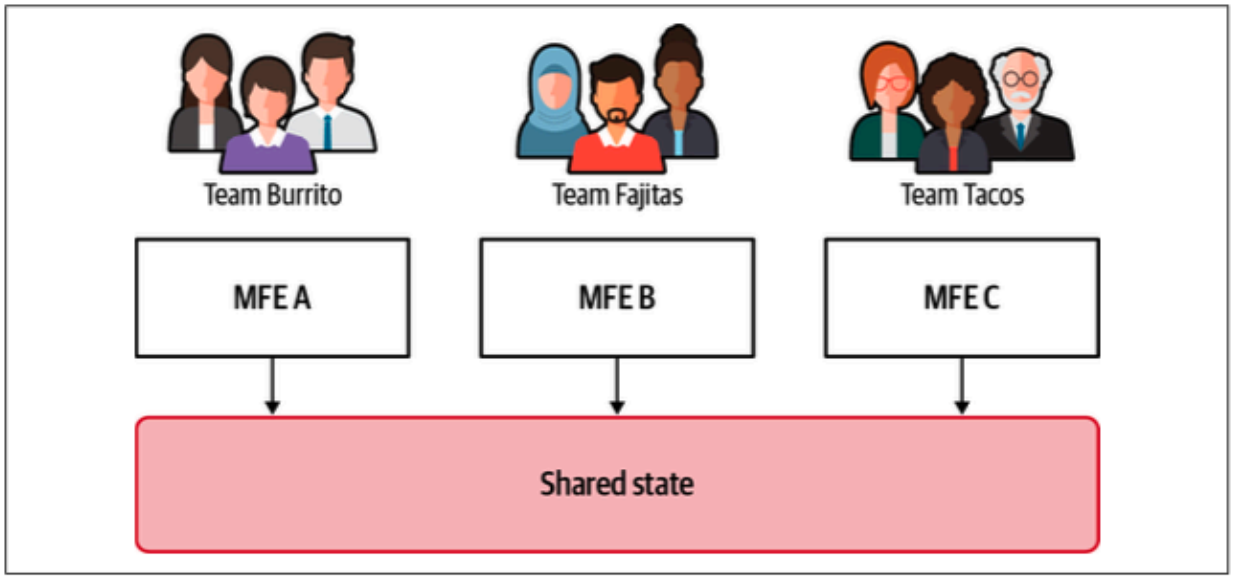


Figure 4-14. Shared state between multiple micro-frontends represents an antipattern

في هذه الصورة لدينا ثلاثة فرق، هذه الفرق تحتاج للتواصل فيما بينها لمشاركة البيانات أو ال event أو أي شيء مشترك نحتاج معرفته، في حالة وجود ال Shared state واستخدام هذه الفرق لهذا الأسلوب فإننا نتجه مباشرة من ال micro frontend إلى ال distributed monolith، أي أننا قمنا ببناء نظام ظاهريا هو micro وموزع

ومقسم لكنه في الحقيقة نظام monolith! وهذا بدوره يعني اختفاء الحدود بين ال micros وزوال الاستقلالية بين ال micros وفقدان للمرونة وتباطؤ في سرعة التطوير، وكل ذلك يعود لأن أي عملية تغيير هنا على ال state ستجعل جميع الفرق تذهب لتتأكد من أن هذا التعديل لم يدمرها ^^، وإذا تم استخدام micro معينة في أكثر من view فإن ال Foundation team سيكون مسؤولا عن متابعة هذه ال state في أكثر من view مما يصعب عملية الصيانة، كما أن التغيير هنا سيزيد من ال Team coupling وعليه زيادة التنسيق للحفاظ على كود يعمل! وهذه المشاكل كلها جزء صغير من المصائب التي يمكن أن تترتب على هذا الجزء! وزد على ذلك مشاكل ال test الممكنة وال deploy...إلخ.

إن إحدى أهم المميزات التي ينبغي علينا الحفاظ عليها عند تعاملنا مع ال micro frontend هو وجود حدود واضحة لل micros والفرق التي تعمل عليها، هذه الحدود تخلق استقلالية واضحة بين الفرق مما يزيد من سرعة التطوير ويقلل من خطورة التعديلات مع القدرة على التحكم الممتاز على Micro التي نقوم بتطويرها... ولتحقيق هذه الغاية في هذه البنية المعمارية يمكننا استخدام ال choreography pattern، هذا ال pattern ممتاز في هذه الحالة لأنه يحقق التكامل بين ال micros دون وجود ترابط مباشر بين ال micros، ويعمل ببساطة من خلال وجود Event Broker أو Asynchronous Communication... فتقوم إحدى ال micros مثلا بإرسال event معين وتقوم ال micros الأخرى والمهتمة بهذا ال event بعمل listen لها والقيام بما يتناسب مع طبيعة عمل هذه ال micro، ومن الأمثلة على ذلك لو قام مستخدم بشراء منتج معين فيتم إرسال event لعملية الشراء وهنا سيقوم ال chip

micro بالاستماع لهذا ال event وتحويل الحالة لجاري نقل البضاعة أما ال micro غير المهتمة فستتجاهل هذا ال event ببساطة! والآن شاهد الصورة F4-15:

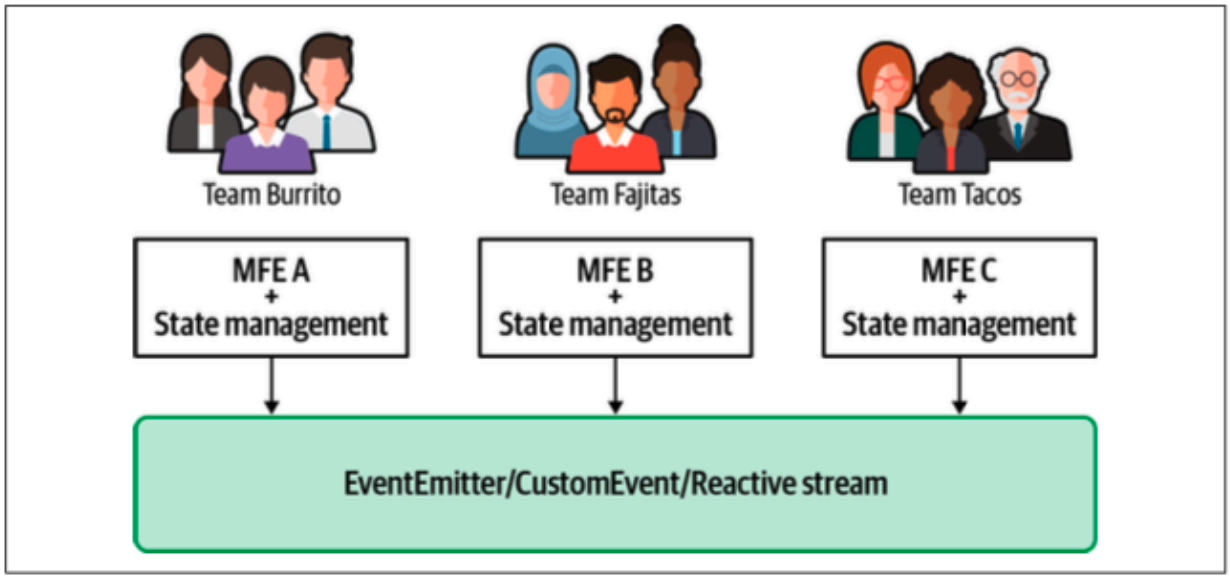


Figure 4-15. Every micro-frontend in the same view should own its own state and should communicate changes via asynchronous communication using an event emitter or CustomEvent or reactive streams

من خلال تنفيذ هذه الاستراتيجية فإننا نحصل على micros يمكنها التفاعل أو عدم التفاعل مع ال events الخارجية بما تقتضيها طبيعة العمل التي صممت لأجلها، كما أن حدود هذه ال micro ستكون واضحة ومستقلة وذلك يقلل من عبء التواصل بين الفرق ويعطي سرعة ومرونة في التطوير ^^، وهذا أيضا يظهر أهمية التوثيق المرتبط بكل micro هنا! فكل فريق عليه كتابة وتوثيق جميع ال event التي يتلقاها ويرسلها حتى يسهل التواصل بين الفرق ^^.

ملاحظة: ال Asynchronous Communication هي عملية الاتصال بين مجموعة من الخدمات بحيث لا يحتاج ال sender إلى انتظار رد فوري من ال Receiver

ليتابع عمله...

- ال Clashes with CSS classes and how to avoid them:

من المشاكل التي يمكن أن تراها بشكل سريع عند عملك على هذه البنية المعمارية هو وجود تضارب في التنسيق بسبب وجود بعض ال css selector المتشابهة في أكثر من micro، فمثلا في microA كان هناك class اسمه avatar. فيه ال color=red، وال microB كان هناك أيضا class اسمه avatar. فيه ال color=blue ^^...
فالنتيجة ستكون تضارب هذه التنسيقات لأننا نتحدث عن أكثر من micro في نفس الصفحة! ولتجنب هذه المشكلة ببساطة يمكننا اضافة prefix قبل ال selector مثل: micro-a-avatar و micro-b-avatar، وبهذا نضمن أن جميع التنسيقات الموجودة في كل micro لن تتعارض أو تتضارب مع micros أخرى ^^.

ملاحظة: هذه الحالة ستواجهها في حال كنت تستخدم ملفات css عامة ولا يوجد build لل style، فمثلا لو كنت تستخدم css-in-js فلن تحدث هذه المشكلة بسبب إنشاء unique class name لكل تنسيق، لكن تبقى احتمالية التضارب موجودة في حالات نادرة أو إذا قام أحد المطورين بكتابة بعض ال classes بشكل يدوي، ومن الطرق أيضا التي تساعد على التخلص من هذه المشكلة استخدام ال css module وال BEM style مع ال prefix وال build time healing...إلخ، لذلك، ولتريح نفسك وغيرك، اجعل وضع ال prefix قاعدة عندك عند كتابة أي class ^^.

- ال Multiframework approach:

استخدام أكثر من framwork أو أكثر من إصدار لنفس ال framework تعد هنا عملية سيئة جدا، فإذا كانت في ال vertical split سيئة وتتسبب في بعض المشكلات بالأداء، فهنا المشكلة أكثر خطورة، لأنها قد تتسبب بأخطاء خطيرة أثناء ال runtime، لذلك عند الحاجة لاستخدام أكثر من framework في هذه البنية المعمارية فهذا يحتاج لتخطيط دقيق أثناء مرحلة التصميم! ومن المشاكل الموجودة أو المحتملة هي تعارض بعض المتغيرات بين إصدار قديم وإصدار جديد في ال micro-frontend، مثلا react 10 وال react 19! ومع أن هناك العديد من الطرق للحد من هذه المشكلة مثل استخدام ال iframe أو بناء web component، إلا أنه لا ينصح بذلك إلا في حالة واحدة، وهي عند الحاجة للانتقال من تطبيق أو نظام قديم لنظام جديد بحيث يتم إصدار ال micros بشكل تدريجي ^^.

- ال Authentication:

هذا التحدي من التحديثات المميزة في هذه المعمارية ^^، وذلك يعود لأن عملية التحقق يمكن أن يتم تنفيذها أكثر من مرة في نفس الصفحة لو تركت هكذا دون إدارة! فكيف يمكننا إذا معالجة هذه المشكلة؟!

إذا تخيلنا المشكلة فلدينا مثلا ثلاثة micros تحتاج لأن يكون المستخدم قد قام بتسجيل الدخول فعلا، وللقيام بهذه العملية بالشكل التقليدي ستقوم كل micro frontend بعمل request يحتوي على JWT token إلى ال Backend ومن ثم اتمام عملية التحقق والقيام ب action ما... شاهد الصورة F4-18:

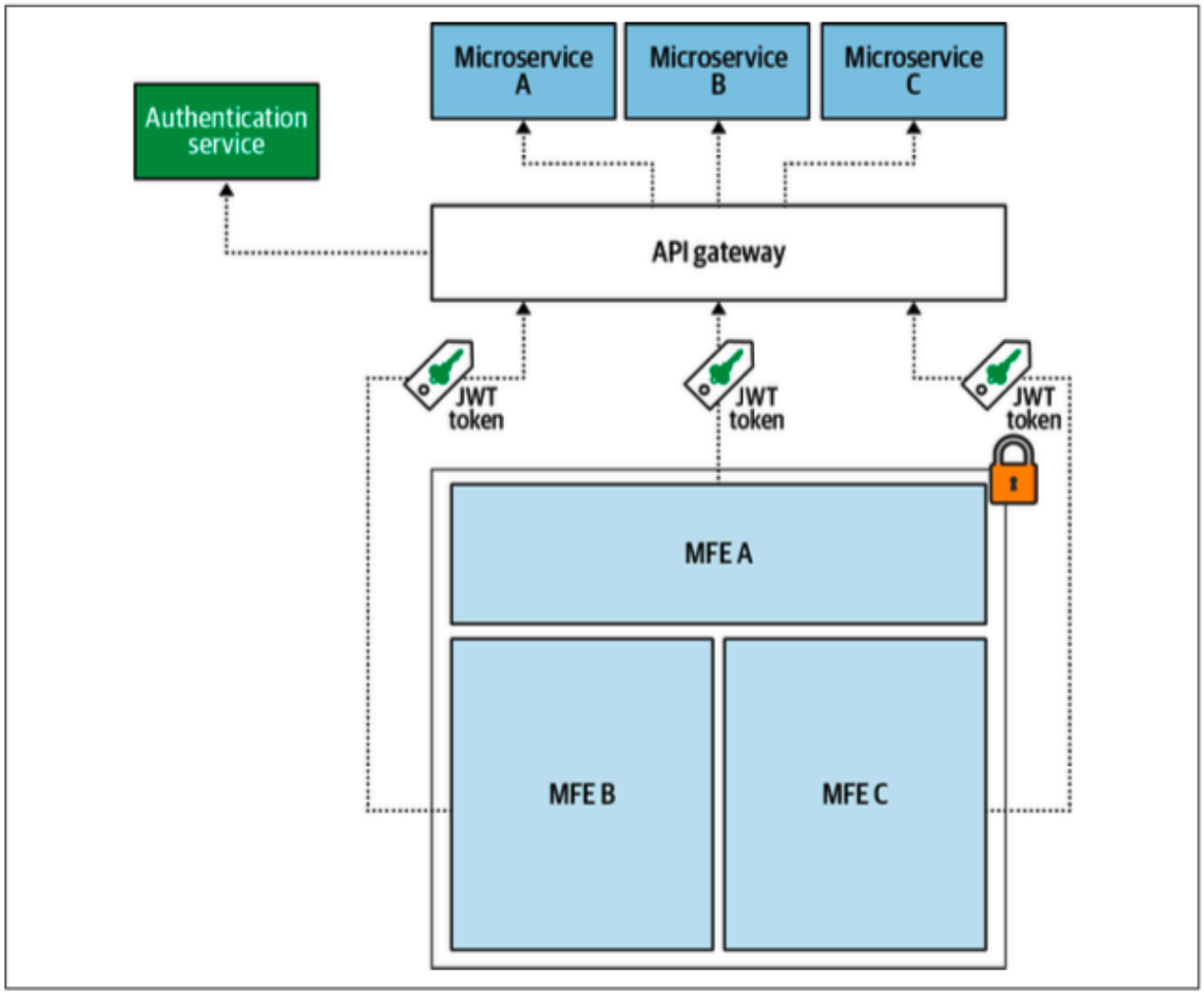


Figure 4-18. Every micro-frontend in a horizontal-split architecture has to fetch data from an API passing a JWT token to the backend to validate that the user is entitled to retrieve the data requested

إذا ما الحل؟ الجواب ببساطة من خلال استخدام واحدة من آليات التخزين التي نعرفها جميعا مثل ال localStorage أو ال sessionStorage أو ال cookie، لكن عليك الانتباه للقيود والمشاكل الأمنية المحتملة، فمثلا يمكن استخدام ال cookie على أكثر من subdomain ضمن نفس ال domain، ولديها مستوى أمان أعلى، لذلك قد يكون هذا خيارا مناسبا لحفظ ال access token!

وهذا يقودنا لنقطة مهمة أخرى، وهي أن وجود أكثر من micro frontend تقوم بالاتصال بنفس ال API وتتعامل مع نفس ال response = أن هذه ال micros غالبا ما يمكن دمجها معا، لكن هذا يتطلب إعادة دراسة ال bounded domain (الحدود) حتى يتم معالجة المشكلة مبكرا بدلا من تجاهلها... وهذا أيضا يعود بنا للوراء عندما تحدثنا عن أهمية وجود مراجعة دورية وإعادة تقييم لمثل هذه المعماريات بين الحين والآخر بناء على مجموعة من القواعد التي يتم اعتبارها كمعايير: زمانية أو تقنية أم اجتماعية أو إدارية وأعمال!

- ال Micro-frontends refactoring:

هذه النقطة واحدة من النقاط المميزة لهذا النوع من التقسيم، وبذلك فهو يتفوق على ال vertical split من هذا الباب، فقرار إعادة هيكلة إحدى ال micro-frontend هنا أو كتابة شيفرة برمجية جديدة بدل القديمة لتحسين الأداء وتقليل الأخطاء أمر أسهل بكثير، فلهذه ال micro وظيفة محددة ولها فريق عمل لديه خبرة كاملة في البنية الوظيفية لها وبالتالي فهم أفضل للمتطلبات والمهام التي تقوم بها وبالتالي قدرة على إعادة كتابة وصيانة أفضل! كما أن هذه العملية يمكن البدء بها فورا عند اتخاذ القرار فلا يوجد هناك داع للانتظار عدة شهور حتى يتم تصليح كامل التطبيق، فال micro هي جزء من عدة أجزاء في هذا التطبيق ^^، كما أن عملية ال revamp لهذه ال micro ستكون بالضرورة أسهل!

لكن احذر، لا يعني سهولة هذه العملية هنا أننا بحاجة لإعادة كتابة الكود فقط لجعله أسهل! بل يجب أن تكون عملية ال refactor نابعة من سبب حقيقي، مثلا تم بناء

هذه ال micro بسرعة بسبب ضيق الوقت أو هناك كمية bugs كبيرة تتسبب بها أو هناك حاجة لإعادة صياغتها لتسريع عملية إضافة أو تطوير هذه ال micro... ونحو ذلك.

- ال Search Engine Optimization: ما تحدثنا عنه سابقا في ال Vertical Split ينطبق هنا باعتبار أننا أيضا على مستوى ال client-side، كما أن ال Dynamic rendering هنا قد يكون أسلوب valid لمواجهة التحديثات في هذه البنية إذا تم استخدام ال ifream لاحتواء ال micros المختلفة بالصفحة.

- ال Developer Experience:

بالنسبة لتجربة المطور هنا، فلدينا العديد من التحديات الحقيقية التي سنواجهها أثناء عملنا، ويمكن تقسيم هذه التحديات لجزئين، الأول إذا كان فريق التطوير مسؤولا عن تطوير micro-frontend واحدة خاصة به ودون ارتباط بغيرها من ال micros، فهنا تكون العملية سهلة ومشابهة لما تحدثنا عنه سابقا في ال vertical split، أما الجزء الثاني في حال كان الفريق مسؤولا عن تطوير micro مرتبطة بمجموعة أخرى من ال micros من فرق مختلفة بنفس الصفحة، ويحتاج إلى عمل test لل micro الخاص به داخل هذه الواجهة، هنا ستظهر تعقيدات في التنفيذ من ناحية توافق الإصدارات أو اختلاف بيئات العمل أو صعوبة إنتاج حالة اختبار كاملة بشكل locally ونحو ذلك، ومع ذلك: فالتحدي الأكبر هنا هو القدرة على تتبع الإصدارات المختلفة من ال micro-frontend وتركيب ال view الخاصة بهذه ال micros على جهاز ال

developer (locally) بأسرع وقت ممكن حتى يتمكن من اختبار التفاعل بين الأجزاء المختلفة بسهولة.

هناك حلول مؤقتة لهذا التحدي سنتحدث عنها لاحقا، لكن يمكن استخدام ال Webpack DevServer Proxy لاختبار ال micro frontend بشكل locally مع القدرة على جلب ال micros الأخرى من ال environments المختلفة مثل ال testing أو ال staging أو ال production... كما أن هناك حلول أخرى مناسبة بحسب طبيعة المشروع وحجمه وآلية تطويره.

هذا التحدي تقوم الشركات بمجابهته من خلال بناء tools لديها واجهة CLI لتساعدها في التعامل مع ال micro frontend وتحقيق أفضل تغذية راجعة ممكنة.

ملاحظة ١: من المهم ملاحظة أن هذه المعمارية قد تتطلب بعض الاستثمار من الشركة أو المؤسسة لإنشاء تجربة developer قوية، ويمكن استخدام ال Webpack Module Federation لتقديم نهج ونموذج تطوير opinionated approach، كما أن هذه الشركات قد تحتاج لبذل جهد إضافي إذا كانت ترغب في تخصيص نموذج التطوير الخاص بها واعتماد نهج غير ال Opinionated، وأي طريق من هذه الطرق لا يعد طريقا سلبيا إن تم اعتماده بناء على احتياجات الشركة أو ما يتعلق بها من خبرات اجتماعية وتقنية.

ملاحظة ٢: ال Webpack DevServer Proxy هو إحدى المزايا الجميلة في ال webpack، وهو option في ال Webpack DevServer يسمح لك بتحويل أو

توجيه ال requests التي تذهب إلى سيرفر ال Backend أو ال API إلى server آخر، وذلك يتم من خلال Webpack أثناء ال development mode، من الأمثلة الشائعة FE: http://localhost:3000 يريد التواصل مع ال BE من خلال http://localhost:5000، وهنا في هذه الحالة عادة ما يحدث عندنا مشكلة ال CORS، وهنا يأتي دور ال proxy بحيث يقوم باعتراض ال fe request ومن ثم يعيد توجيهها لل BE، ويمكن ضبطها بناء على قاعدة ما مثل أن أي fe request كان بداخله "/api" فهذا يجب أن يتم إعادة توجيهه إلى ال localhost:5000/api ^^... لذلك يعد هذا ال Proxy هو أحد الوسائل المستخدمة أثناء التطوير لل micro frontend، باعتباره وسيلة يمكنها أن تصل لأكثر من micro ومن ثم تجمعها وكأنها موقع production ^^، وذلك من خلال المحاكاة التي تقوم بها نتيجة ضبط ال requests والقدرة على إعادة توجيهها والحد من مشاكل ال CORS...

ملاحظة ٣: ال opinionated approach هو مصطلح معناه الحرفي: "النهج القائم على الرأي"، وهو مصطلح كثيرا ما يستخدم في عالم التقنية للإشارة إلي أن هذه التقنية أو هذا ال framework أو هذه المكتبة تعتمد نهجا محددا ومعروف مسبقا في كيفية إنجاز مهمة معينة، وفلسفة هذه الفكرة نابعة من أن الحد من المرونة في بعض الأحيان قد يقدم فائدة لا تقدمها المرونة غير المقيدة، وهذا يساعد على تسريع عملية التطور وتقليل الأخطاء من خلال توجيه المطورين لاتباع النهج الذي تم اعتماده، كما يجعل هذا الأسلوب التركيز منصبا على ال business logic وحل المشكلات الفعلية أكثر من الانشغال بالتفاصيل التقنية الأساسية، ومن الأمثلة على ذلك طريقة تسمية

الملفات في مشروع ما أو طريقة التفاعل ومشاركة البيانات داخل التطبيق أو طريقة كتابة الشيفرة البرمجية ونحو ذلك، ومن أشهر الأمثلة على ذلك Laravel باعتباره framework يجب أن تخضع لقواعده، أما مشاكل هذا الأسلوب تتمحور حول نقطة قوته! فانعدام المرونة قد يشكل حاجزا أمام استخدام هذه التقنية، وبهذا لا يعد هذا الأسلوب مناسبا لكل الحالات، كما أن هناك منحنى تعلم (Learning Curve) تحتاجه لفهم النهج المتبع والسير عليه حتى يعمل النظام كما هو متوقع...

- ال Maintaining control with effective communication:

يتمركز هذا التحدي حول كيفية الحفاظ على التحكم وعدم فقدان السيطرة على ال micros من خلال بناء تواصل فعال، وهذا من المركزيات المهمة في ال Horizontal split.

إن التواصل الفعال والتوثيق الجيد وتقليل الاعتماديات، وتبني عقلية التحسين المستمر هي مفاتيح النجاح عند بناء وإدارة الأنظمة المعقدة، مثل تلك الأنظمة التي تعتمد على ال micro frontend، وكل ذلك لضمان تقديم أفضل تجربة للمستخدم بشكل سلس وخال من المشاكل، وهنا يبرز هذا التحدي في تحديد آلية أو كيفية تحقيق هذا الغرض -أي كيفية الخروج بأفضل output ممكن للمستخدمين-! وهذا الأمر ليس بالسهل الهين هنا ^^.

إن هذه المعمارية -ال Horizontal Split- تعد من أكثر المعماريات مرونة إلا أنها

تقدم هذا التحدي كواحد من أبرز التحديات التي يمكن أن تواجهها مؤسسة ما أو مطور يعمل في هذه المؤسسة، وهنا يظهر الدور الحقيقي للتواصل بين الفرق وأهميته! فإن كنا ننوي حقا تفادي المشاكل المتوقع حدوثها عند اختلاف الإصدارات لمكتبة ما بين ال micros المختلفة أو عند حصول Css Clashes -كما تحدثنا سابقا- فعلينا ببناء تواصل فعال بين الفرق، كما أن هناك حاجة ملحة لإضافة أدوات لمراقبة النظام لتحديد أي فشل أو خطأ قد يحصل أو حصل على production وإرساله مباشرة للفريق المعني، وهنا لدينا مجموعة من السلوكيات أو الممارسات التي يمكنك اعتمادها للحصول على أفضل تواصل ممكن، وهي:

١. يجب أن تكون هناك قنوات اتصال مفتوحة بين الفرق بحيث يمكن الحصول على أي feedback ومعالجته بشكل سريع، مع الحفاظ على أكبر تزامن ممكن ببين الفرق، ويمكن تحقيق ذلك من خلال القيام باجتماعات دورية كل أسبوع أو نصف أسبوع يشارك فيها عضو من كل فريق، أو من خلال استخدام البريد الإلكتروني أو تطبيقات التواصل المباشر (ال chat application) لتحقيق هذا الغرض.

٢. نوصي بشدة بتقليل عدد الفرق المسؤولة عن العمل على صفحة واحدة، فالصفحة التي تعمل عليها ثلاثة فرق أفضل من التي تعمل عليها عشرة فرق -إن كان ذلك صحيحا أو ممكنا-، لأن هذا يقلل من الأخطار المحتملة التي ذكرناها سابقا مع زيادة فعالية التواصل وسهولة تحقيقه مقارنة لو كان عدد الفرق أكبر.

٣. في كل صفحة يجب أن يتم اختيار فريق يكون مسؤولا عن قيادة هذه الصفحة

بحيث يضمن وصول أفضل output للمستخدم، وهذا بكل تأكيد لا يعني أن هذا الفريق سيأخذ على عاتقه كل المهام في هذه الصفحة، وإنما هي محاولة لجعل المسؤوليات المشتركة في يد أمينة ^^ بدلا من جعلها مفتوحة دون ضبط حتى تتعارك الفرق فيما بينها :P... ومن الأمثلة الجميلة على ذلك تخيل أن فريق ال Catalog وفريق ال Video Player يعملون على نفس الصفحة فهنا يمكننا إعطاء مسؤولية إدارة هذه الصفحة لفريق ال Catalog، وبهذا يتعين على فريق ال Catalog التنسيق مع فريق ال Video Player للحصول على أفضل تجربة مستخدم، ومن دور هذا الفريق المسؤول التحقق من أن هذه ال micros تعمل بشكل صحيح بعد زيارة الصفحة...

٤. يجب أن تكون هناك مسؤولية منوطة بال Engineering Manager أو ال Team Lead، وهي عملية التقليل أو الحد من ال external dependencies، كما ينبغي علينا عدم تقبل الوضع الراهن لل micros إلا إذا كانت فعلا جيدة، وقد أشرنا لذلك مرارا في هذه المعمارية لأهميتها...

٥. يجب توثيق ال input وال output وال events لل micro frontend من قبل الفريق الذي يعمل عليها، وهذه عملية مهمة تجعل من الفريق والأفرقة الأخرى على علم بما يجري حولهم وما هو متوقع من كل micro، كما تتيح مناقشة أي break change ممكن أن يحصل في حال الحاجة لتغيير شيء ما... وهنا يظهر جمالية وأهمية ال RFCs، وينصح باستخدامها بشدة في ال horizontal split لضمان التنسيق

والاتساق وتجنب المشاكل التي قد تنشأ بسبب التعارضات المحتملة!

ملاحظة: ال RFCs هو اختصار ل Request for Comments، وهو عبارة عن وثيقة (ملف) يتم إنشاؤه لوصف معايير معينة أو سلوك معين أو مشكلة معينة أو بحث معين أو ابتكار معين لتسهيل التواصل والتعاون بين الفرق المختلفة، وتوثيق ما دار بينهم لحظة اعتماد ما دار في الوثيقة، ويعمل هذا الأسلوب من خلال إنشاء ملف ويمثل المقترح الذي نرغب بتقديمه (ال Proposal)، وفيه يتم تحديد المشكلة التي نرغب في حلها والحل المقترح لهذه المشكلة بوصف تفصيلي يشمل آلية التنفيذ والتصميم والآثار المحتملة أو الناتجة من هذا المقترح، كما يتم طرح البدائل الممكنة ولماذا تم اختيار هذا الحل بالتحديد، ثم الحديث عن المخاطر والآثار الناتجة عن هذا التغيير المقصودة أو المحتملة، والوقت المقدر لتنفيذ هذا المقترح -وهذا خيار اختياري وقد يرتبط بالخطوة التالية-... الآن بعد خطوة إنشاء المقترح -الآن هي مسودة- نقوم بعمل Request for comments (أي طلب تعليق من الأشخاص المستهدفين)، ويتم فيها جمع الملاحظات وطرح الأسئلة وتحديد أي مشكلة لم يتم أخذها بعين الاعتبار، وهذا كله بشكل غير متزامن -ولذلك هو مفيد جدا خصوصا إذا كانت الفرق في أكثر من مكان أو زمان-، بعد ذلك ننتقل لخطوة المراجعة والمناقشة بحيث تتم مراجعة ما تم كتابته بالمقترح ويتم دمج التعليقات والمقترحات في الملف الأصلي مع التعديل على الملف بناء على الملاحظات... ثم يلي ذلك إما قبول أو رفض أو طلب تعديلات كبيرة تجعلنا نعود للبداية مجددا [٨٨].

فائدة

فلْتعلمْ أن الإنسان ما ترك شيئًا لله إلا عوضه الله -سبحانه وتعالى- أفضل مما ترك، وهذا العوض قد يكون من جنس المتروك، وقد يكون من غير جنسه، ولْتعلمْ أن أعظم ما يعوض به الإنسان هو الأنس بالله -سبحانه وتعالى- ومحبته، وطمأنينة القلب والنشاط في طاعته، ورفعة الدرجات في جنات النعيم والتكفير عن الذنوب، وهذا لا ينفرد بل قد يجتمع كله بفضل الله -سبحانه وتعالى- فيكون العوض ظاهراً مثل البركة وزيادة المال، أو العلم أو الصحة أو القوة إلى آخره، فالحمد لله على كرمه.

- كتاب إلى الجنة زمرا، الصفحة ١٢٠

والأن: متى يكون من المناسب استخدام ال Horizontal Split؟

يعد استخدام هذه المعمارية -رغم كل التحديات التي ذكرناها- خيارا ممتازا وجذابا للمطورين والشركات في كثير من الحالات، ومن هذه الحالات التي تدعونا لاستخدام هذه المعمارية:

1. عند الحاجة لوجود قابلية لإعادة استخدام ال micros التي سيتم تصميمها وتطويرها في أكثر من مكان، أكان ذلك داخل نفس التطبيق أو خارج التطبيق، وهو ما يطلق عليه: Micro-frontend Reusability...

2. عند بناء Enterprise Applications and Dashboards، فهذه المعمارية مناسبة للتعامل مع كميات كبيرة ومتنوعة من البيانات التي نحتاج لعرضها في ال dashboards وفي أكثر من view لأكثر من هدف، فهذه تجمع وتعرض معلومات المالية أو تعرض وتجمع معلومات ناتجة عن مراقبة النظام للمستخدمين ونحو ذلك، وهنا يمكنك مشاهدة مثال عملي لما قامت به شركة New Relic عند بنائها وتصميمها لل micro-frontend الخاص بها حتى تضمن تقديم خدمة مميزة وتحصل على أفضل أداء وقدرة على التطوير والتوسع؛ حيث قاموا بتوزيع عدد فرق قليل يمكنه العمل ضمن نفس ال view -وتحدثنا عن أهمية وفوائد ذلك سابقا- ثم سمحوا لهذه الفرق بالمساهمة في بناء تمثيلات مختلفة من البيانات ثم جمعها كلها داخل لوحة تحكم موحدة، ويتم جلب ال micro frontend المناسبة لها بناء على اختيار المستخدم... شاهد الصورة F4-19:

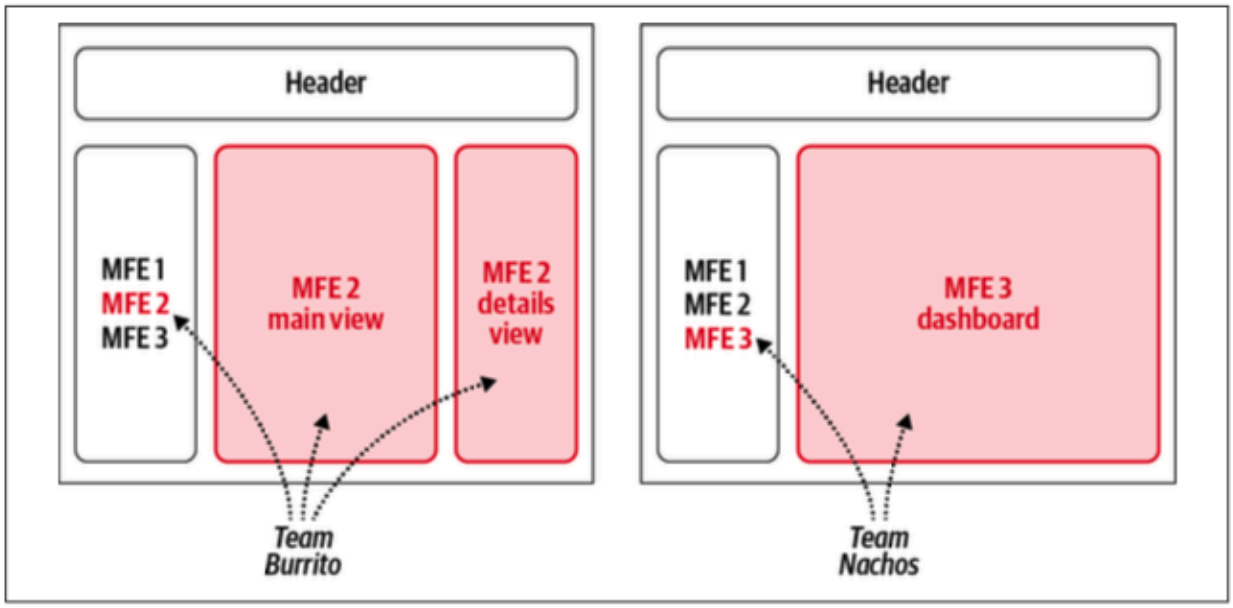


Figure 4-19. New Relic micro-frontend implementation. Every team is responsible for their own domain, and when a user selects a dashboard, the related micro-frontend is lazy-loaded inside the application shell.

لاحظ جمال هذه الصورة وكيف يظهر دور هذه المعمارية هنا، ولاحظ أن هناك عدد قليل من الفرق التي تعمل في نفس الواجهة، مما يقلل من الحاجة إلى التواصل المفرط لتجميع ال view النهائية، وفي نفس الوقت فهو يسمح لكل فريق أن يكون على دراية بال business domain داخل ال micro التي يعمل عليها ودون معرفة بقية ال micros عنها -لا يوجد state shared-.

3. عند بناء Multi Tenant Application، فهذه المعمارية أيضا تعد مناسبة ومثالية لهذا النوع من التطبيقات، بحيث تكون غالبية الواجهات متماثلة ومتشابهة لأغلب العملاء، لكن قد يكون لديك عملاء مميزون يرغبون بإضافة بعض الإحتياجات الخاصة بهم على هذه الواجهة أو التعديل على ما فيها، وهنا يظهر دور ال micro

frontend الذي سيسمح لك بإضافة أو استبدال ال micros بناء على tenant الخاصة بهذا العميل ^^ وببساطة ^^.

ال Module Federation:

لقد تحدثنا سابقا بشكل سريع عن ال Module Federation، لكن سنقوم بالدخول ببعض التفاصيل التقنية هنا لأهمية هذه الميزة في هذه المعمارية ^^...

تعد هذه المزية واحدة من الهدايا الكبيرة التي قدمها ال Webpack للمطورين في إصداره الخامس، وهي plugin وظيفتها الأساسية السماح لمجموعة من ال javascript chunks من التحميل بشكل متزامن أو غير متزامن، كما يتيح لنا التحكم بالإصدارات المستخدمة في ال micros لتوحيدها أو عمل wrap لها في حال الحاجة لاستخدام أكثر من إصدار -لا ينصح بذلك إلا للضرورة-، وهذا كله يتيح للفرق أن يعملوا بشكل مستقل على ال micro الخاصة بهم مع تولي أحد الفرق مسؤولية الجمع (Application Composition)، كما يتم تحميل ال chunks المختلفة من الجافا سكربت خلف الكواليس أثناء ال runtime من خلال ال Lazy-loading، وهذا يعني أن الكود الذي سنقوم بتحميله هو ما نحتاجه فقط، مما يؤدي

إلى تحسين أداء التطبيق بشكل ممتاز، شاهد الصورة الخرافية F4-20:

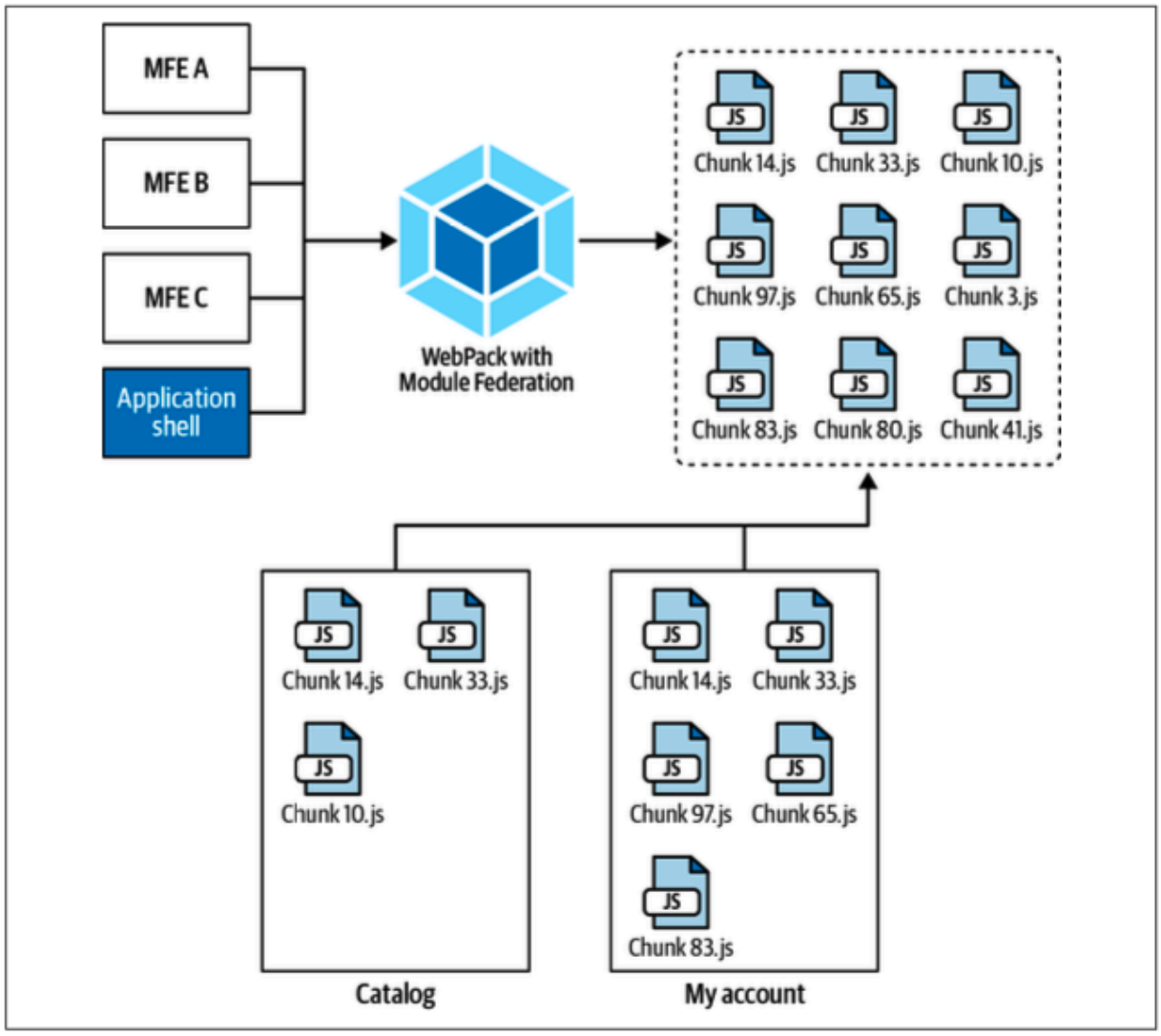


Figure 4-20. Module Federation allows multiple micro-frontends to be loaded asynchronously, providing the user with a seamless experience

والآن ننتقل إلى مكونات هذا ال Plugin، وهو يتكون من جزأين رئيسيين وهما:

1. ال The Host: وهو يمثل ال container الذي سيقوم بتحميل واحد أو أكثر من ال micros أو ال library المستخدمة، ويمكن اعتباره التطبيق الأساسي التي تستضيف المكونات الأخرى (App Shell or App Root) - لاحظ أن مهمة ال Host هي **جلب** ما نحتاجه، والذي سيكون عبارة عن Remote، لذلك يتم استخدام ال remote key داخل ال config للإشارة إلى host سيقوم بجلب شيء من الخارج، شاهد الصورة أدناه:

```
plugins: [
  new ModuleFederationPlugin({
    name: "host",
    remotes: {
      header: "header@http://localhost:3001/remoteEntry.js",
      footer: "footer@http://localhost:3002/remoteEntry.js",
      dashboard: "dashboard@http://localhost:3003/remoteEntry.js",
    },
  }),
],
```

2. ال The Remote: وهو يمثل ال micro frontend أو ال library نفسها والتي سيتم تصديرها أو تحميلها داخل ال Host أثناء ال runtime، وآلية العمل تكون من خلال عمل expose لهذه ال micro أو الـ library حتى يتمكن ال Host من استخدامها... ولهذا يتم استخدام ال expose key داخل ال config للإشارة إلى أن

هذه المكونات قابلة للاستضافة ^^، شاهد الصورة أدناه:

```
plugins: [
  new ModuleFederationPlugin({
    name: "header",
    filename: "remoteEntry.js",
    exposes: {
      "./Header": "./src/Header",
    },
  }),
],
```

والآن بعد هذا الحديث الجميل، هل أدركت ما هو أهم ما يميز ال Module Federation؟ نعم يا صديقي، أحسنت: بساطة عمل ال expose وجلب ال micro frontend المختلفة أو المكتبات المشتركة! ولذلك فهو يشعر المطور ألا شيء جديد قد حصل على طريقة البرمجة التي يعمل عليها، بحيث يشعر بأنه قريب من العمل على مواقع ال monolithic التي اعتاد عليها، كما يمكن إنشاء environment configuration تخدم ال testing أو ال staging أو ال production مع وجود إعدادات مشتركة للجميع...

كما أن فكرة مشاركة ال Shared Library بين عدة micros دون الخوف من التعارضات المستقبلية فيما بينهما مع ضمان الاتساق واحدة من أجمل المزايا، فتخيل أنك تستخدم عدة micros ثلاثة منها تستخدم axios، فال module federation سيقوم بتحميل إصدار واحد من ال axios ولمرة واحدة ومشاركة هذه المكتبة مع ال micros التي تحتاجها ^^ جمال وروعة خرافية ^^ كما يمكن السماح بوجود أكثر من إصدار لنفس المكتبة عند الحاجة -للضرورة-... لكن احذر فالكمال لله -سبحانه وتعالى- وحده، فهناك بعض التحديات التي

ستواجهنا عند استخدام هذه ال plugin مثل الحاجة لانضباط وقواعد تضمن سير العمل بالشكل المتوقع حتى لا نصل لمرحلة فيها من التعقيد ما يصعب عملية الصيانة، خصوصا فيما يتعلق بموضوع إدارة ال shared library والإصدارات ونحو ذلك، كما أن هناك بعض المشاكل التي تنتج في حال لو كان ال remote كبير ويتم التحميل بشكل dynamically أثناء ال runtime، فهذا قد يؤثر على ال performance من خلال التأثير على TTI.

وسيتم الحديث عن هذا الموضوع بشكل أكثر تفصيلا في الأجزاء القادمة بإذن الله... تحديدا كيف يمكننا عمل composition على مستوى ال vertical أو ال horizontal مع أمثلة جميلة إن شاء الله.

ال Shared code:

بوجود ال Module federation فعملية مشاركة الكود والتحكم في شكل وآلية تدفق البيانات عملية سهلة جدا، وهذه النقطة هي إحدى أهم نقاط قوة ال module federation، كما أنه يوفر أسلوبين لنقل ومشاركة الكود وهما ال bidirectional وال Unidirectional، لكن نقطة القوة هذه يجب أن نحذر عند استخدامها حتى لا تصبح نقطة قاتلة وسيئة في تصميمنا...

- يمثل ال bidirectional آلية لمشاركة الكود ثنائية الاتجاه بين ال micro-frontend، وهذا يعني أن ال remote يمكنه مشاركة الكود مع ال host والعكس صحيح، وهذه العملية توفر مزية مهمة وهي إلغاء الهرمية التقليدية في التطبيق

الخاص بنا... ولكن لهذا الأسلوب مشاكل خطيرة يجب أن نحذر منها مثل ال Tight coupling، فعند هذه اللحظة سيصبح لدينا اعتمادية متبادلة بين مكونين مختلفين وهذا ينتهك ما نريد تصميمه وفعله من خلال انتقالنا لل micro frontend، كما أنه يزيد من صعوبة ال scaling وال maintenance وال debugging، ويزيد من المخاطر المحتملة لحدوث Circular Dependencies!، لذلك يفضل استخدام هذا الأسلوب عند بناء نظام يحتاج إلى تزامن أو تبادل فعال ودائم للبيانات مع استجابة سريعة ومباشرة... من الأمثلة على هذا الأسلوب: أن تشارك ال micro A كود ال header مع micro B وتقوم micro B بمشاركة كود ال sidebar مع micro A.

- يمثل ال Unidirectional آلية لمشاركة الكود أحادي الاتجاه، وهذا يعني أن ال Parent سيقوم بإرسال الكود إلى ال Child من اتجاه واحد، من الأعلى إلى الأسفل، وهو بسيط وسهل لكن أكبر مشاكله قد تكون في فقدان جزء من المرونة التي تخص التفاعل المباشر والتبادلي، كما أن سوء التنسيق به قد يؤدي إلى تكرار إرسال البيانات...

بناء على هذه النبذة السريعة يمكننا أن ننصح باستخدام ال Unidirectional في معماريتنا وعند تعاملنا مع ال micro-frontend، وهذا الأمر مهم ومفيد جدا لنا للتخلص من العديد من المشاكل واكتساب العديد من المزايا التي انطلقنا لعالم ال frontend لأجلها! علما أن اعتمادنا على ال Unidirectional سيقدم عدة فوائد مهمة وهي:

- سهولة اكتشاف الأخطاء وتصحيحها، فأنت تعرف اتجاه البيانات وشكل البيانات في كل مرحلة، وبالتالي سهولة كشف الخطأ ومن ثم معالجته.
- أقل عرضة للأخطاء لأن عملية التفاعل بين المكونات ليست معقدة... فهي باتجاه واحد!
- أكثر كفاءة: لأن كل micro ستعرف حدودها بدقة ولن يحدث هناك تداخل أو تصادم في المهام فكل micro تعرف حدودها بدقة... مما يجعل النظام أكثر كفاءة ودقة في التعامل مع ال events المختلفة...

هذه النظرة ظهرت أهميتها لدى مطوري ال frontend بشكل جلي منذ سنوات قليلة، ولعل الانتقال الذي حصل من ال bidirectional لل Unidirectional من الأدلة القوية على فاعلية هذا الأسلوب، ولعل من الأمثلة الجميلة على هذا التغيير هي ال react كواحدة من هذه النماذج التي أظهرت فاعلية هذا النهج من خلال استخدام ال props وكيف تنتقل بين المكونات المختلفة...

ال Use cases:

يوفر ال Webpack مع ال Module Federation مرونة عالية تدعم مختلف أنماط ال micro-frontends، أكان ذلك من خلال التقسيم الأفقي أو العمودي، ويمكن عمل composition للتطبيق على جانب ال client أو server مع دعم ال routes بسهولة، والتواصل بين الأجزاء المختلفة من خلال استخدام ال events... وهذا يغطي معظم حالات

استخدام ال micro frontend ويوفر تجربة تطوير ممتازة للمطورين المعتادين على استخدام ال Webpack.

ال Architecture characteristics:

وكما تحدثنا سابقا، لكل معمارية أو نمط خصائص تميزه عن الآخر، واستخدام تقنيات مختلفة أو أسلوب ما قد يؤثر على الخصائص المعمارية التي تعمل عليها... والآن لنأخذ نظرة مع هذه الخصائص التي تتعلق بوجود Module federation ضمن معماريتنا:

١. ال Deployability (سهولة النشر) - العلامة ٤/٥ ^^:
يقوم ال Webpack بتقسيم ال micro-frontend إلى Chunks من ال JavaScript، مما يجعل من السهل نشرها على أي خدمة سحابية ومن خلال أي CI/CD، وبما أن جميع هذه الملفات static، فمن الممكن عمل cache بسهولة وبكفاءة عالية، لكننا سنحتاج إلى التعامل مع قابلية التوسع هنا في حالة استخدام ال SSR، لكن يمكن تخطي هذه المشكلة من خلال تعويضها بسهولة ال integration وسرعة تحصيل ال feedback.

٢. ال Modularity - العلامة ٤/٥ ^^:
من خلال هذه المعمارية وهذا الأسلوب يمكننا تحصيل قدرة عالية على تقسيم العمل لأجزاء صغيرة أكانت component أو module، هذه الأجزاء مستقلة وصغيرة وقابلة للاختبار والصيانة بشكل منفصل عن باقي الأجزاء، مما يقلل الاعتمادية بين ال micros بشكل كبير

جدا ويعطي قابلية عالية لإعادة الاستخدام، لكن هذه المرونة والميزة قد تصبح لعنة على المشروع إن أسيء استخدامها، والسبب في ذلك يعود لإمكانية حدوث أو إنشاء الكثير من الاعتماديات (ال dependency) بين ال micros دون قصد! وبهذا تصبح الفرق معتمدة على فرق أخرى أو أجزاء أخرى تمنعها من متابعة العمل، وهذا ما يطلق عليه بال: interdependencies، ونتيجة ال interdependencies هي ال Organizational Friction، والتي بدورها ستؤثر على عمل الفرق بشكل مستقل وستظهر مشاكل تحد من الاستقلالية لأن ال micros صارت متداخلة وتعديل إحداها قد يؤثر على الأخرى، وزيادة الحاجة للتنسيق المستمر...

٣. ال Simplicity - العلامة ٥/٥ ^^:

العمل من خلال هذه ال plugin سهل جدا وبسيط ويجعل من عملية ال integration بين ال micros سهلة ومشابهة لحد كبير لما اعتدنا عليه في مواقع ال SPA أو ال SSR.

٤. ال Testability (قابلية الاختبار والفحص) - العلامة ٤/٥ ^^:

يمكننا استخدام ال federated test من خلال ال jest داخل ال module federation، ومع ذلك لدينا القدرة على القيام بال unit testing وال end to end testing كما اعتدنا ^^.

ملاحظة: يقصد بال federated test القدرة على test بين مجموعة من ال micros المختلفة أكانت remote أو host للتأكد من أن ال micros المختلفة تعمل معا بشكل صحيح.

٥. ال Performance (الأداء) - العلامة ٤/٥ ^^:

لأن ال module federation يقدم لنا العديد من المزايا الجميلة التي تحدثنا عنها والتي تتعلق بآلية تنظيم ال shared library أو common library، فإن هذا سبب أساسي في تحسين الأداء، لكن هناك مشكلة صغيرة وهي أن مخرجات ال micro frontend قد تكون ملف جافا سكربت واحد أو أكثر، وهذا يزيد عدد ال Roundtrips (الطلبات بين ال client وال cdn) في حالة وجود أكثر من ملف جافا سكربت سيتم تحميله لهذه ال micro.

٦. ال Developer Experience - العلامة ٥/٥ ^^:

يقدم تجربة تطوير سهلة وممتازة، مع قدرة عالية على التحكم والإعداد ودون الدخول في تعقيدات عملية ال composition بين ال micros.

٧. ال Scalability (القابلية للتوسع) - العلامة ٥/٥ ^^:

عملية التوسع هنا سهل، فكما رأيت يمكن جلب الملفات التي نحتاجها بناء على ال micros التي نحتاجها في هذه الصفحة، وهذه الملفات هي عبارة عن static js file يمكن تحميلها من ال cdn مباشرة، وهذا يبرز بشكل جلي في معمارية ال Vertical.

٨. ال Coordination (التنسيق) - العلامة ٣/٥ ^^:

إن استخدام ال module federation بالتزامن مع ما تحدثنا عنه سابقا في موضوع ال decisions framework (آلية اتخاذ القرار) = عملية تطوير أسهل حتى في المؤسسات الكبيرة والمشاريع الكبيرة مقارنة بالطرق الأخرى، فهو يقلل الاعتمادية ويسرع التطوير

ويحسن من قابلية التوسع ^^، لكن هذا الأمر أيضا قد يقلب المزايا لعيوب فورا عند اساءة الاستخدام كما ذكرنا سابقا، وسنحتاج مع كل خطأ إلى زيادة في التنسيق بدلا من التخفيف منه! باختصار فإن هذه القدرة ستمكننا من تحسين التنسيق من خلال تمكين الاستقلالية لكنه في نفس الوقت يحمل خطرا كبير عند إساءة الاستخدام يؤدي لعكس النتائج المتوقعة.

شاهد الصورة Table 4-2:

Table 4-2. Architecture characteristics summary for developing a micro-frontend architecture using webpack with Module Federation

Architecture characteristics	Score (1 = lowest, 5 = highest)
Deployability	4/5
Modularity	4/5
Simplicity	5/5
Testability	4/5
Performance	4/5
Developer experience	5/5
Scalability	5/5
Coordination	3/5

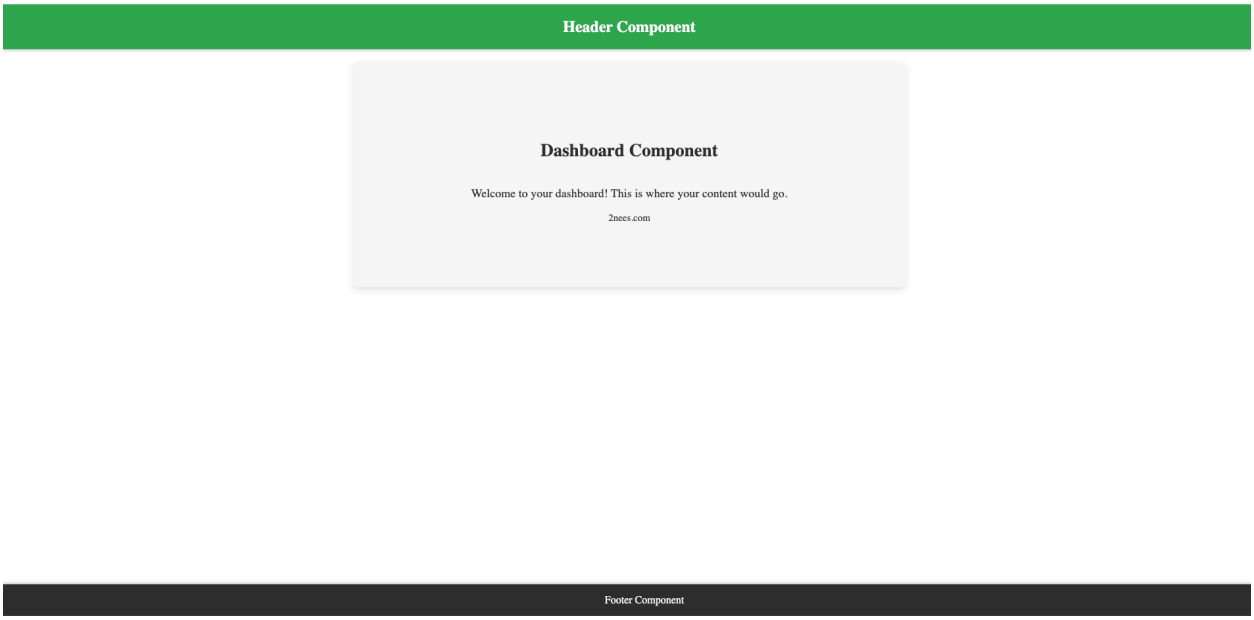
والآن شاهد مثالا عمليا على ال module-federation: ويمكنك الدخول إلى هذا المثال من خلال هذا الرابط.

```
new ModuleFederationPlugin({
  name: "header",
  filename: "remoteEntry.js",
  exposes: {
    "./Header": "./src/Header",
  },
}),
```

```jsx
import React from "react";

const headerStyle = {
  backgroundColor: "#4CAF50",
  color: "white",
  padding: "20px",
  textAlign: "center",
  fontSize: "24px",
  fontWeight: "bold",
  boxShadow: "0 2px 4px rgba(0,0,0,0.2)"
};

const Header = () => <header style={headerStyle}>Header Component</header>;

export default Header;
```

```jsx
import React, { Suspense } from "react";

const Header = React.lazy(() => import("header/Header"));
const Footer = React.lazy(() => import("footer/Footer"));
const Dashboard = React.lazy(() => import("dashboard/Dashboard"));

const App = () => (
  <div>
    <Suspense fallback={<div>Loading Header...</div>}>
      <Header />
    </Suspense>
    <Suspense fallback={<div>Loading Dashboard...</div>}>
      <Dashboard />
    </Suspense>
    <Suspense fallback={<div>Loading Footer...</div>}>
      <Footer />
    </Suspense>
  </div>
);

export default App;
```

```
new ModuleFederationPlugin({
  name: "host",
  remotes: {
    header: "header@http://localhost:3001/remoteEntry.js",
    footer: "footer@http://localhost:3002/remoteEntry.js",
    dashboard: "dashboard@http://localhost:3003/remoteEntry.js",
  },
}),
```



فائدة

فلْتعلمْ أن أعداء المسلمين وإن تفرقوا فيما بينهم، فإنهم يرون الإسلام عدوهم، يجتمعون ويساندون بعضهم البعض ضده! لذلك، كن أنت وإخوتك يدًا واحدة أمام طغيانهم، وانصر أخاك عند حاجته، ولا توالِ أعداء الإسلام على أخيك مهما حصل!

- كتاب إلى الجنة زمرا، الصفحة ١٢٩

ال Iframes:

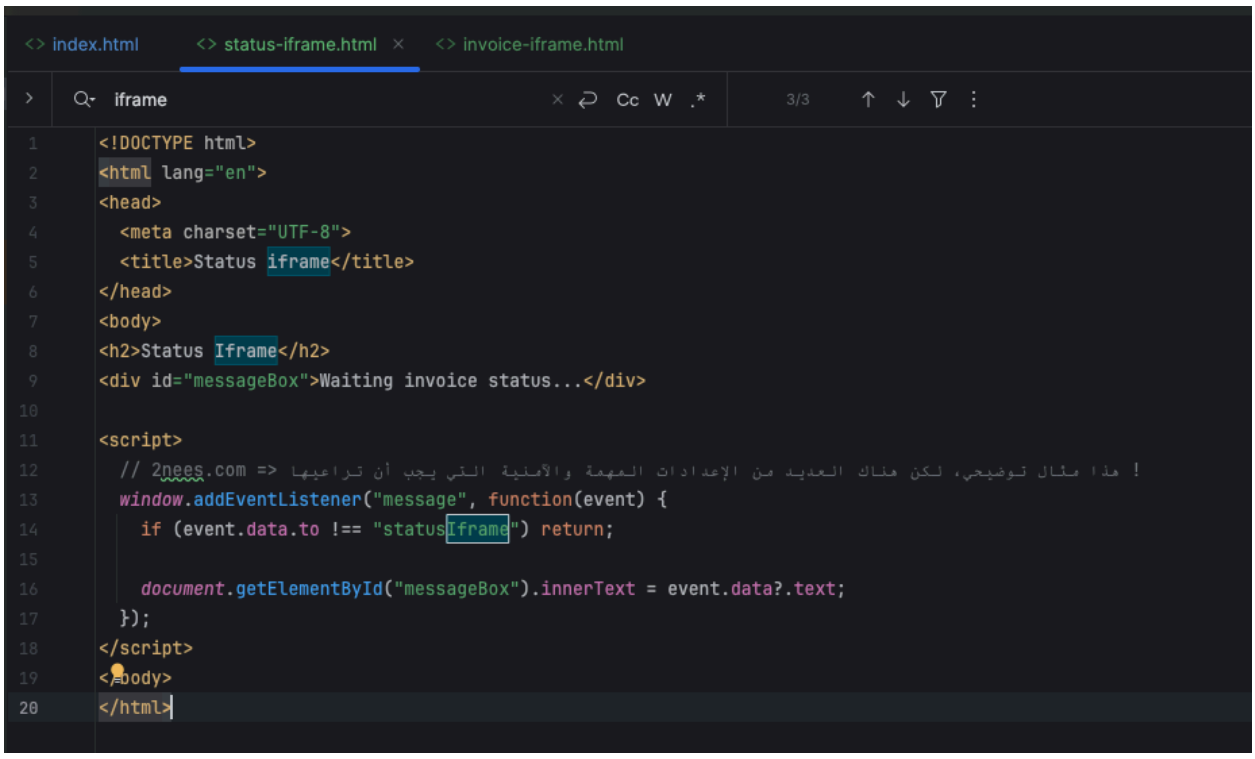
جميعنا تعلم أو قرأ عن ال iframe في بداية رحلته في تعلم تصميم تطبيقات الويب، وغالبا ما يعد استخدام الـ iframe هو آخر خيار قد يفكر فيه المطور للقيام بأمر ما أو مهمة ما، لكنها في بعض المواقف قد تكون هي الخيار الأفضل الذي سيحل لك مشاكل عويصة ^^، إذ أن ال iframe يقدم مزية لا يمكن أن تتوفر بالحلول الأخرى بنفس قوة ال iframe، هذه المزية هي: ال isolation (العزل)، وحتى نفهم الموضوع بشكل صحيح لنرجع للبداية لنتعرف على ال iframe.

ال iframe هي html tag تعتبر اختصارا ل inline frame، وهي إطار يستخدم دخل صفحة ويب لتحميل html file آخر بداخل هذه الصفحة، وبهذا يمكنك أن ترى أكثر من نافذة تعرض محتوى مختلفا تماما عن الصفحة ال host، كما يوفر ال iframe مجموعة من الخيارات المهمة التي تتعلق بآلية التحكم أو الصلاحيات لهذا ال iframe المستخدم داخل الصفحة، ومن ذلك منع أي js من أن يتم تنفيذه أو السماح بال form submission ونحو ذلك، وبما أن هذا الإطار يقوم بتحميل صفحة ال html الخاصة به فهو معزول عما يحدث بالصفحة ال host وبهذا لن يحصل تعارض بين المكونات والعناصر، بل إن كل واحدة من ال iframe ستأخذ مساحة خاصة لها داخل ال cpu باعتبارها صفحة مستقلة، وهذا من المآخذ على هذا الأسلوب لأنها تستهلك المعالج وتؤثر على الأداء! لكن هذه الأمور يمكن تجاوزها عند وجود حاجة للحصول على مزية العزل، ومن الحالات المفيدة هنا استخدامها في تطبيقات سطح المكتب أو تطبيقات ال B2B، مثال: لو افترضنا أن لديك نظام CRM

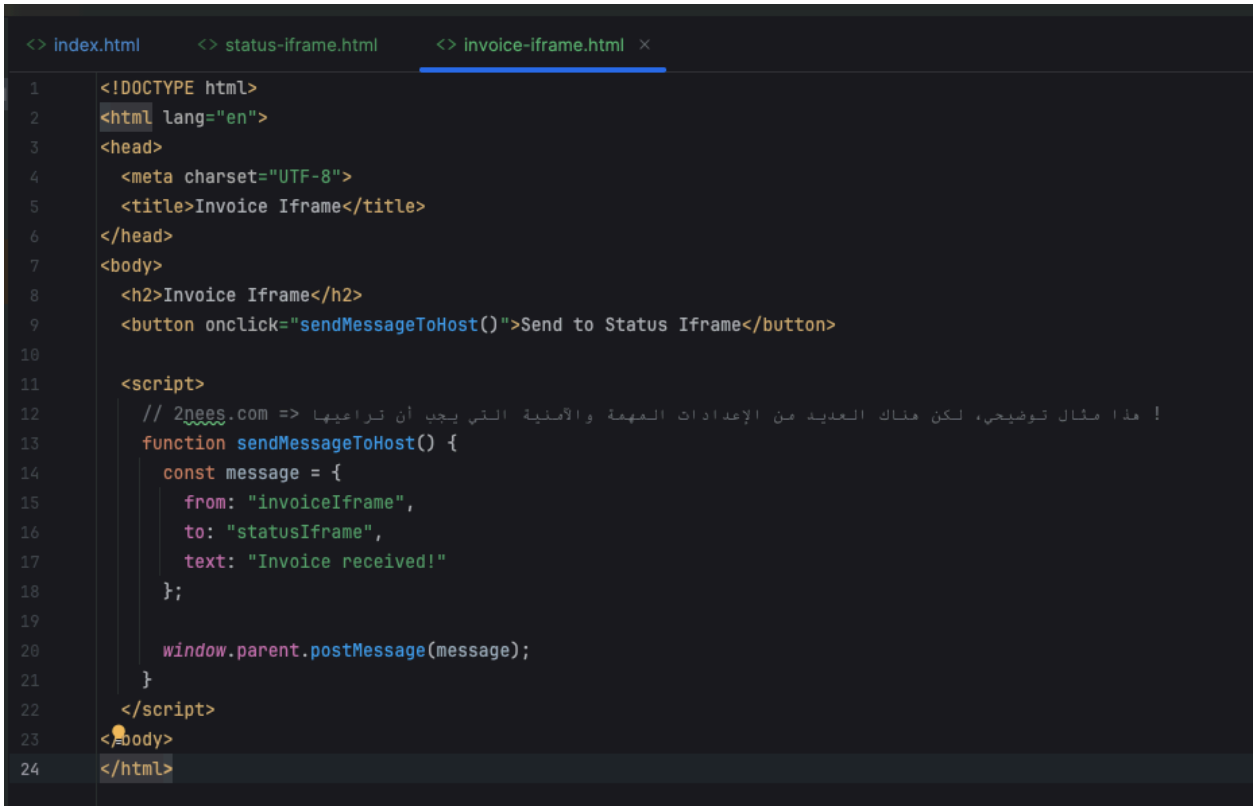
لشركة كبيرة نرغب بتحويله ل micro frontend، ويتم استخدامه من قبل المبيعات والعملاء والدعم الفني، ولدينا هذه الأجزاء فيه: ال Dashboard وال Customer Management وال Ticketing System وال Invoicing، نظام ال ticketing هنا نظام قديم أو نظام مصمم بلغة مختلفة كليا مثل PHP أو ASP أو له تفاصيل تقنية مختلفة، وعملية نقله ليصبح جزءا من النظام عملية مكلفة وتحتاج إلى وقت، وهي ليس ذا أولوية لأنها ليست من ال core domain للبيزنس الخاص بنا! بكل بساطة في هذه الحالة يمكننا الحفاظ على ال url الخاص بال ticketing مثل https://tickets.2nees.com واستخدام ال iframe لتحميل هذه الصفحة داخل ال app shell عند الضغط على رابط ال ticketing! فقط وببساطة، ولاحظ هنا الفائدة الكبيرة التي نلناها من هذه الحركة، فقد وفرنا الوقت والمال وجعلناه لما هو أكثر أهمية، واستطعنا تجاوز المشاكل التقنية المتعلقة باختلاف اللغات المستخدمة، ولا خوف هناك من حصول تعارض بين المكتبات... باختصار استخدام ال iframe لدمج الأنظمة القديمة أو الأنظمة الخارجية كجزء مستقل مع ال micro frontend يعد خيارا ممتازا.

والسؤال المهم الآن، كيف يمكننا أن نرسل أو نعلم ال host إذا حصل تفاعل معين على ال ifram؟ أي باختصار كيف يمكن تحقيق التواصل بين ال iframe وال host page؟ والجواب من خلال ال postMessage ^^، شاهد المثال التالي:

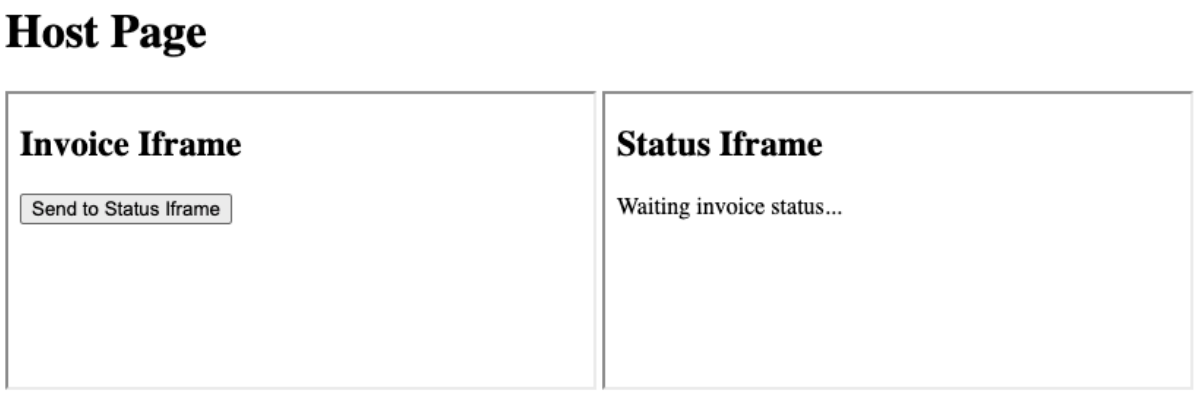
```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Host Page</title>
</head>
<body>
<h1>Host Page</h1>
<iframe id="invoice-iframe" src="invoice-iframe.html" width="400" height="200"></iframe>
<iframe id="status-iframe" src="status-iframe.html" width="400" height="200"></iframe>

<h4>Thank you for your invoice!</h4>
<div id="messageBox"></div>

<script>
  // 2nees.com => هذا مثال توضيحي، لكن هناك العديد من الإعدادات المهمة والأمنية التي يجب أن تراعيها !
  window.addEventListener("message", function(event) {
    if (event.data.to === "statusIframe"){
      document.getElementById("messageBox").innerText = `Message: '${event.data?.text}' was received from '${event.data?.from}' and sent to '${event.data?.to}'`;

      const statusIframe = document.getElementById("status-iframe");
      statusIframe.contentWindow.postMessage(event.data);
    }
  });
</script>
</body>
</html>
```

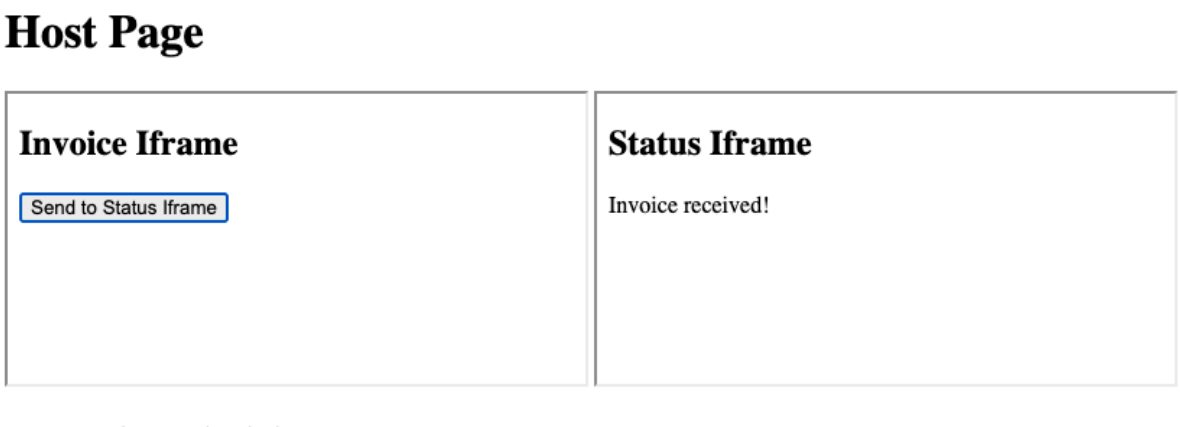


```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Status iframe</title>
</head>
<body>
<h2>Status Iframe</h2>
<div id="messageBox">Waiting invoice status...</div>

<script>
  // 2nees.com => هذا مثال توضيحي، لكن هناك العديد من الإعدادات المهمة والأمنية التي يجب أن تراعيها !
  window.addEventListener("message", function(event) {
    if (event.data.to !== "statusIframe") return;

    document.getElementById("messageBox").innerText = event.data?.text;
  });
</script>
</body>
</html>
```



وهذه النتيجة النهائية:





ويمكنك الدخول إلى هذا المثال من خلال هذا الرابط.

معلومة مفيدة: هناك مقترح قيد الدراسة الآن من لجنة TC39 لإضافة مفهوم جديد يدعى: "ShadowRealm"، وهو عبارة عن sandbox مثل ال iframe، لكنه متوافق مع ال modern web apis وأقل استهلاكا لموارد الجهاز، باختصار هي وسيلة أو آلية جديدة

لتحقيق مفهوم العزل (ال isolation) بطريقة أفضل من ال iframe ومتوافقة مع ال modern api، ويركز هذا المقترح على آلية دمج هذا ال sandbox مع ال ECMAScript، لكن هذا المقترح ما زال (Stage 2.7 draft)، ويمكنك متابعة آخر الآخبار أو الاطلاع على تفاصيل المقترح من خلال هذا **الرابط**.

ملاحظة ١: TC39 هي اختصار ل Technical Committee 39، وهي واحدة من اللجان التقنية التابعة لل ECMA International ومسؤولة عن تحديث مواصفات وخصائص الجافا سكربت، والتي تعرف رسميا ب ECMAScript، وللمضي بأي مقترح هناك عدة مراحل وهي:

- ال stage 0: وهي مرحلة الفكرة.
- ال stage 1: وهي مرحلة الاقتراح والمناقشة الأولية لهذه الفكرة لما تم اعتباره ذو قيمة.
- ال stage 2: وضع المواصفات الأولية بشكل رسمي لهذا المقترح مع التفاصيل التقنية
- ال stage 2.7: اعتماد المواصفات الأولية باعتبارها نسخة نهائية قابلة الآن للاختبار والتنفيذ.
- ال stage 3: اقتراحات جاهزة وقابلة للتنفيذ من قبل ال engine المختلفة، والاختبارات قد تمت.
- ال stage 4: تم إدراجها رسميا كمعيار معتمد، وتم اختبارها وتوثيقها وتطبيقها...

ملاحظة ٢: ال ECMA International هي منظمة غير ربحية تضع معايير لغات البرمجة والتقنيات المرتبطة بها، وفيها العديد من اللجان واحدة منها ال TC39.

والآن، ما هي ال Best practices and drawbacks لاستخدام الـ iframe في معمارية ال horizontal؟
من السلوكيات أو الممارسات الجيدة لك كمطور عند اعتمادك لل iframe في مكان ما:

١. اعتماد Template لل iframe داخل ال layout الخاص بك، وبهذا يمكن للفريق الخاص بك أو الفرق الأخرى معرفة مكان ال iframe الذي سيتم استخدامها ومنها فهم كيفية تنفيذهم للمتطلبات الخاص بهم.

شاهد الصورة F4-21:

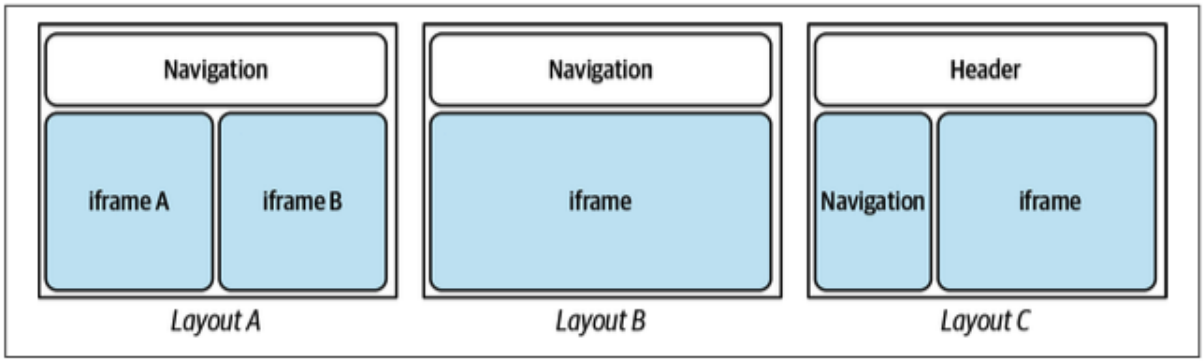


Figure 4-21. Different layouts for composing micro-frontends with iframes. Minimizing the number of iframes in a page would result in better performance despite there being an intrinsic performance overhead when we integrate one or more iframes into a view.

٢. تجنب عدد التفاعلات الكبير بين ال micros، لأن كثرة التفاعلات (interactions) بين هذه ال micros سيزيد من التعقيد الخاص بالشيفرة البرمجية مما سيزيد من الأخطاء وصعوبة الصيانة، لذلك عليك تقليل هذا التفاعل قدر الإمكان، فإن لم يكن الأمر ممكنًا فعليك اختيار نمط أو أسلوب آخر، لأن ال iframe ممتاز في حالة العزل، لكنه سيء في حالة الحاجة للتواصل المكثف!

٣. يجب أن يكون ال UI متجانس في جميع ال micro frontend، لذلك ينصح بمشاركة ال system design أثناء ال build time لكل micro، وتذكر أننا نتحث هنا عن iframe موجودة فلا يمكن أن تأخذ التنسيق الخاص بها من ملف css موجود عندك، لذلك فنحن بحاجة لضمان حصولها وحصول جميع ال micros على نفس نظام التصميم، لذلك يكون الحل بتضمين ال system design عند ال build لكل micro.

٤. في حالة ال iframe يتعين عليك حفظ أو تخزين البيانات على webstorage من خلال ال app shell، وذلك لتجنب أي مشاكل متعلقة بعملية استرجاع هذه البيانات، فكما تعلم كل iframe هو صندوق منعزل ^^، أما إذا كانت هذه البيانات غير مهمة لأي micro أو صفحة أخرى... فلا مشكلة ^^.

٥. في حالة استخدام ال Pub/Sub للتواصل بين ال iframe، علينا مشاركة نسخة من ال Event Emitter instance مع العناصر الأساسية في هذه الصفحة... إذا كان التواصل

فقط سيتم من خلال ال iframe مع ال host فلا داعي لمشاركة ال event emitter، فال postMessage كافي...

شاهد المثال:

README.md | shared-event-emitter.js | index.html | invoice-iframe.html | status-iframe.html | RE

```
// 2nees.com => هذا مثال توضيحي، لكن هناك العديد من الإعدادات المهمة والأمنية التي يجب أن تراعيها !
class SharedEventEmitter {
  constructor() {
    this.events = {};
  }

  on(eventName, listener) {
    if (!this.events[eventName]) {
      this.events[eventName] = [];
    }
    this.events[eventName].push(listener);
  }

  emit(eventName, data) {
    if (this.events[eventName]) {
      this.events[eventName].forEach(listener => listener(data));
    }
  }

  off(eventName, listenerToRemove) {
    if (!this.events[eventName]) {
      return;
    }
    this.events[eventName] = this.events[eventName].filter(listener => listener !== listenerToRemove);
  }
}

const sharedEventEmitter = new SharedEventEmitter();
```

README.md | shared-event-emitter.js | index.html | invoice-iframe.html | status-iframe.html | README.en.md

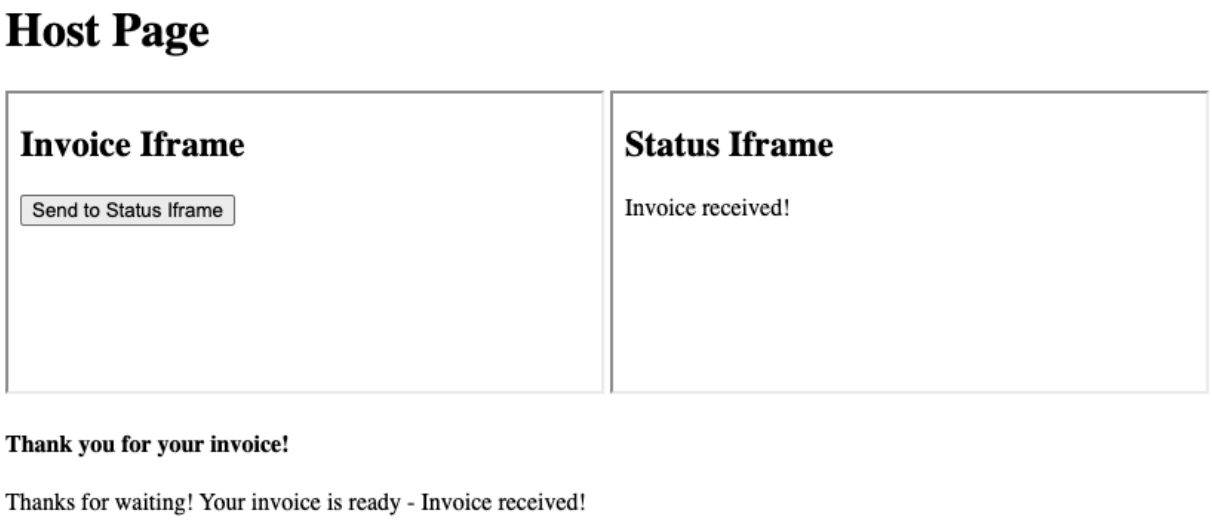
```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Host Page</title>
</head>
<body>
<h1>Host Page</h1>
<iframe id="invoice-iframe" src="invoice-iframe.html" width="400" height="200"></iframe>
<iframe id="status-iframe" src="status-iframe.html" width="400" height="200"></iframe>

<h4>Thank you for your invoice!</h4>
<div id="messageBox"></div>

<script src="shared-event-emitter.js"></script>
<script>
  // 2nees.com => هذا مثال توضيحي، لكن هناك العديد من الإعدادات المهمة والأمنية التي يجب أن تراعيها !
  const invoiceIframe = document.getElementById("invoice-iframe");
  const statusIframe = document.getElementById("status-iframe");

  invoiceIframe.contentWindow.appEventEmitter = sharedEventEmitter;
  statusIframe.contentWindow.appEventEmitter = sharedEventEmitter;

  sharedEventEmitter.on('invoiceReady', (data) => {
    document.getElementById("messageBox").innerText = `Thanks for waiting! Your invoice is ready - ${data}`;
  });
</script>
</body>
</html>
```

README.md | shared-event-emitter.js | index.html | invoice-iframe.html | status-iframe.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Invoice Iframe</title>
</head>
<body>
  <h2>Invoice Iframe</h2>
  <button onclick="sendMessageToHost()">Send to Status Iframe</button>

  <script>
    // 2nees.com => هذا مثال توضيحي، لكن هناك العديد من الإعدادات المهمة والأمنية التي يجب أن تراعيها
    function sendMessageToHost() {
      window.appEventEmitter.emit("invoiceReady", "Invoice received!");
    }
  </script>
</body>
</html>
```

README.md | shared-event-emitter.js | index.html | invoice-iframe.html | status-iframe.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Status iframe</title>
</head>
<body>
<h2>Status Iframe</h2>
<div id="messageBox">Waiting invoice status...</div>

<script>
  // 2nees.com => هذا مثال توضيحي، لكن هناك العديد من الإعدادات المهمة والأمنية التي يجب أن تراعيها !
  window.onload = function () {
    const appEventEmitter = window.appEventEmitter;

    if (appEventEmitter){
      appEventEmitter.on('invoiceReady', (data) => {
        document.getElementById("messageBox").innerText = data;
      });
    }
  };
</script>
</body>
</html>
```



ويمكنك الدخول إلى هذا المثال من خلال هذا الرابط.

ال Architecture characteristics:

وكما تحدثنا سابقا، لكل معمارية أو نمط خصائص تميزه عن الآخر، واستخدام تقنيات مختلفة أو أسلوب ما قد يؤثر على الخصائص المعمارية التي تعمل عليها... والآن لنأخذ نظرة مع هذه الخصائص التي تتعلق بوجود Iframe ضمن معماريتنا:

١. ال Deployability (سهولة النشر) - العلامة ٥/٥ ^^:

عملية ال deploy هنا سهلة جدا، وهي مشابهة لل vertical split، فكل iframe يمثل نطاق كامل بذاته، ومع أننا في معمارية ال horizontal split نقوم بعرض أكثر من view داخل نفس الصفحة يمكن أن تحتوي أكثر من iframe، إلا أن هذا لن يسبب لنا أي

مشكلة عن ال deploy، فبالنهاية هي html tag معزول ومفصول ما بداخله ^^، وهذا يعني أن كل micro frontend تمثل iframe قابلة لل deploy بشكل مستقل ودون أن ترتبط بغيرها، ولا يوجد تداخل بال library أو الملفات ويمكن بناء ci/cd لكل واحدة بشكل منفصل ^^.

٢. ال Modularity - العلامة ٣/٥ ^^:

من خلال هذه المعمارية وهذا الأسلوب يمكننا تحصيل مستوى جيد من تقسيم العمل لأجزاء صغيرة، هذه الأجزاء مستقلة وصغيرة وقابلة للاختبار والصيانة بشكل منفصل عن باقي الأجزاء، مما يقلل الاعتمادية بين ال micros بشكل كبير جدا ويعطي قابلية عالية لإعادة الاستخدام، لكن هذه المرونة والميزة قد تصبح لعنة على المشروع إن أسيء استخدامها، وقد ناقشنا نفس هذه الفكرة في ال module federation، فالسهولة قد تقودنا الإساءة ^^، ومن أخطر المشاكل في حالة ال iframe التقسيم الكثير مما يعني وجود عدد كبير من ال iframe أو وجود تفاعل كبير... وهذا هو السبب في كونها ٣ من أصل ٥... لأن هذا الأمر احتمالية عالية جدا إذا لم يكن هناك تخطيط جيد والتزام بمبادئ التصميم.

٣. ال Simplicity - العلامة ٣/٥ ^^:

العمل من خلال ال iframe تعتبر عملية سهلة، فيمكن للمطورين التركيز على ما بيدهم من أعمال دون الخوف أو القلق من التعارض مع ال micros الأخرى، لكن هناك تحديات تقلل من هذه البساطة أهمها إقامة نظام تواصل بين ال iframes المختلفة، كما أن القيام بإنشاء

iframe بشكل متجاوب مع أحجام الشاشات المختلفة (Responsive Design) عملية صعبة وتحتاج لجهد حقيقي لضبطها...

ملاحظة: إن ال iframe ليس حلا مناسبا عند الحاجة ل SEO ممتاز أو عند الحاجة إلا وجود Accessibility قوية تدعم ال screen reader، وسبب ذلك عدم قدرة كلاهما على فهم ما في داخل ال iframe أو التفاعل معها...

٤. ال Testability (قابلية الاختبار والفحص) - العلامة ٣/٥ ^^:
عملية ال unit testing هنا هي عملية سهلة، لكن الصعوبة والتعقيد يكون عند ال E2E testing، فلدينا Dom كثيرة ومتداخلة.

٥. ال Performance (الأداء) - العلامة ٢/٥ ^^:
تعد مشكلة الأداء أسوأ سمة لهذه البنية، وإذا كان التنفيذ فيه مشاكل وغير صحيح فهذا يعني أنك ستكون أبعد ما يكون عن الأداء الجيد *_*، خصوصا فيما يتعلق باستهلاك الموارد الخاصة بالجهاز.

٦. ال Developer Experience - العلامة ٣/٥ ^^:
يقدم تجربة تطوير سهلة وجيدة، مع قدرة عالية على التحكم والإعداد، لكن هناك تحديات مهمة على مستوى ال developer من أهمها بناء Solid Client Side Composion، بحيث يمكن لكل فريق أن يعمل على الجزء الخاص به وفحصه مع الأجزاء الأخرى، وتجميعها بشكل

صحيح عند ال deploy على ال production، كما أن كتابة ال test كما ذكرنا عملية معقدة وستؤثر على تجربة المطور هنا.

٧. ال Scalability (القابلية للتوسع) - العلامة ٥/٥ ^^:
عملية التوسع هنا سهلة، فكما رأيت يمكن جلب الملفات التي نحتاجها بناء على ال micros التي نحتاجها في هذه الصفحة، وهذه الملفات هي عبارة عن static js file يمكن تحميلها من ال cdn مباشرة.

٨. ال Coordination (التنسيق) - العلامة ٣/٥ ^^:
التنسيق سيكون معقدا بحسب الظروف الخاصة بالنظام، فكلما زاد عدد التفاعلات والتأثيرات بين الأجزاء المختلفة زادت الحاجة للتنسيق، لذلك كما ذكرنا سابقا وهذا الأسلوب ممتاز في حالة كانت التفاعلات على مستوى ال micro نفسها أو مع القليل من التواصل الخارجي...

شاهد الصورة Table 4-3:

Table 4-3. Architecture characteristics summary for developing a micro-frontend architecture using horizontal split and iframes

Architecture characteristics	Score (1 = lowest, 5 = highest)
Deployability	5/5
Modularity	3/5
Simplicity	3/5
Testability	3/5
Performance	2/5
Developer experience	3/5
Scalability	5/5
Coordination	3/5

فائدة

فلْتعلمْ أن الإله المستحق للعبودية والمحبة هو الإله الواحد الأحد، الخالق، الذي ليس كمثله شيء، الرحمن الرحيم بعباده، الذي ليس له زوجة ولا ولد، الغني عن عباده، تتجه له كل المخلوقات في الرخاء والشدة، طوعًا وكرهًا، وهذا لا يكون إلا لإله واحد، وهو الله العظيم -جل في علاه- لا لأحد سواه.

- كتاب إلى الجنة زمرا، الصفحة ١٤٩

ال Web Components:

يعد استخدام ال Web Component واحدة من الخيارات الممتازة عند تفكيرك في بناء وتصميم ال micro frontend، فهي تمتلك خصائص مميزة قد تجعلها فعلا الحل المناسب للكثير من الحالات ^^، والسبب في ذلك يعود إلى:

- يمكن عمل Customization لها وإعادة استخدامها في أكثر من مكان وأكثر من مشروع، ويضاف إلى ذلك أن المحتوى الخاص بهذه ال component سيكون Encapsulated مما يعطي ميزة ممتازة لإخفاء التفاصيل التقنية الخاصة بهذه ال component.
- أيضا يمكننا الاستفادة من ال Style Encapsulation، وهي من المزايا الممتازة، فيمكننا من خلالها منع أي تعارض بال css مع أي أجزاء أخرى بالمشروع ^^.
- يمكن أن تعمل هذه ال Web Component مع جميع ال frameworks أو library المشهورة، فهي بالنهاية تعتمد على ال Web Platform API، وهذا يعني مزية جميلة لو أردنا تصميم component يمكنها أن تعمل بنفس التصميم وبغض النظر عن التقنية المستخدمة ^^.
- يمكن استخدام ال Web Component لمشاركة ال component المختلفة مع ال micro frontend أو لعمل encapsulated لل micro frontend نفسها، وهذه العملية مفيدة في حال رغبتنا بالاستغناء عن ال iframe والحصول على شيء أداءه أفضل. -تذكر أن ال iframe يحجز engine وموارد خاصة به باعتباره سيقوم بتحميل doc كاملة-.

ولأجل تحقيق الحالات التي في الأعلى والحصول على هذه المزايا، فلدينا تقنيات رئيسية ينبغي علينا معرفتها، وهي:

- ال Custom Element: هي Html Extension يمكن استخدامها لإنشاء عناصر html جديدة من تعريفنا مع إمكانية التفاعل معها من خلال ال callbacks وال events، ويمكن أن تستخدم ك container لل micro frontend، وهذه ستتأثر من أي style خارجي أو ستؤثر على أي style خارجي في حال وجوده.

شاهد المثال:

```
class ComponentWithoutShadow extends HTMLElement {
  connectedCallback() {
    this.innerHTML = `
      <div style="background: #333; color: white; height: 100px; display: flex; align-items: center; justify-content: center;">
        <h1 class="heading-style">Custom Web Component Without Shadow</h1>
      </div>
    `;
  }
}
customElements.define('mfe-without-shadow', ComponentWithoutShadow);
```

- ال Shadow Dom: هي عبارة عن مجموعة من ال JS API التي تستخدم لعمل encapsulated لل DOM tree، ويتم عمل ال render لها بشكل مستقل عن ال main dom، وبهذا نحن نضمن ألا يحدث تداخل في ال script أو ال styles كما في ال custom element.

شاهد المثال:

```
class ComponentWithShadow extends HTMLElement {
  constructor() {
    super();
    this.shadow = this.attachShadow({ mode: 'open' });
  }

  connectedCallback() {
    this.shadow.innerHTML = `
      <div style="background: blueviolet; color: white; height: 100px; display: flex; align-items: center; justify-content: center;">
        <h1 class="heading-style">Custom Web Component With Shadow</h1>
      </div>
    `;
  }
}
customElements.define('mfe-with-shadow', ComponentWithShadow);
```

```
<body>

<mfe-with-shadow></mfe-with-shadow>
<mfe-without-shadow></mfe-without-shadow>

<script type="module" src="component-without-shadow.js"></script>
<script type="module" src="component-with-shadow.js"></script>
</body>
```

ويمكنك الدخول إلى هذا المثال من خلال هذا الرابط.

لكن علينا أن نحذر هنا من فخ خطير: وهو ال Domain Logic Leak، وهي من أشهر الأخطاء إذا كانت ال Web Component تمثل ال container لل Micro frontend، ويقصد بها وجود تداخل بحيث تعرف ال web component عن معلومات لا يجب عليها أن تعرفها... مثلا أن يتم ارسال ال API endpoints من خلال ال web component إلى ال micro frontend، مع أن هذه تعد من المعلومات الخاصة بال micro frontend! واعلم أن ال Web component يمكنها أن تكون Open to extension لكن ال micro frontend يجب أن تكون Closed to extension لكنها open to communication،

أي أننا نتوقع تخصيصها أو تعديل السمات الخاصة بها دون الحاجة لتعديل الكود الخاص بها نفسه، أما ال Micro فإننا لا نتوقع أن يتم تغيير منطق عملها الأساسي أو سلوكها الداخلي بشكل كبير من الخارج أكان عن طريق App shell أو Web Component أو غيرها، ومع ذلك فهي مفتوحة للتواصل من خلال ال Custom Events أو بعض ال props البسيطة...

كما أن من التحديات المهمة التأكد من طبيعة المستخدمين للتطبيق، فهذه الميزة غير مدعومة من المتصفحات القديمة وقد تحتاج لاستخدام ال polyfill، كما يجب أن تتأكد من الخصائص وسلوكها خصوصا على متصفح safari.

كما أن هناك تحدي مهم في حال كان التطبيق الخاص بك يحتاج إلى ال SEO بشكل قوي، فال Shadow Dom من ال Web Component قد لا تستطيع قراءته ال bot القديمة أو بعض ال bots الموجودة حاليا، في حين حتى لو كان ال googlebot الموجود متقدم ويمكنه قراءته، فقد تكون النتائج ليست مثالية في كل الأحوال، وهنا قد تلجأ لبعض الحلول للتخلص من هذه المشكلة إما من خلال ال pre-rendered أو تغيير في طريقة التصميم أو ضمان وجود هذا المحتوى في ال Light Dom (ال dom العادي الذي يكون بين html tag) دون أن يكون shadow.

بناء على هذا، فإن ال Use Cases لهذه البنية هي:

١. دعم ال Multitenant Environment: إذا كان لديك نظام واحد شمولي يمكنه خدمة أو دعم أكثر من client وكل واحد منهم لديه بياناته الخاصة ومجموعة ال component الخاصة به، لكنهم يشتركون جميعا في ال base code، فتعتبر ال Web Component هنا

خيارا ممتازا لإمكانية تحقيق التوافق بشكل ممتاز بين هذه ال Web Component مع أي framework أو library يمكن لل client أن يستخدمها، فلو كان يستخدم React أو Vue أو html لن يختلف شيء بالنسبة لنا لأن ال Web Component بذاتها هي Web Standard ^^، كما أن ال Web Component تقدم خدمة رائعة لنا وهي إمكانية إصدار عدة إصدارات منها تعمل أو تتوافق مع أي إصدار من التطبيق الخاص بنا، وكمثال عملي لو أننا نملك Chat لخدمة ال Clients، فإن بناء ال Chat هنا ك Web Component سيقدم لنا خدمة خرافية، فلو كان ال client A يستخدم React و Client B يستخدم Angular لن تكون لدينا أي مشكلة، كما لو أننا قمنا بترقية الإصدار الخاص بال chat فيمكن ببساطة الحفاظ على نفس ال component واستدعائها مع القدرة على تغيير ال version بسهولة، ويمكن تحقيق ذلك من خلال طريقتين، شاهد الصورة أدناه mfe-chat:

```
<mfe-chat-widget platform-version="v1" customer-id="123"></mfe-chat-widget>

<mfe-chat-widget platform-version="v2" customer-id="456"></mfe-chat-widget>

<!-- OR -->
<mfe-chat-widget customer-id="456"></mfe-chat-widget>
<script src="https://2nees.com/cdn/web-components/v2/bundle.js"></script>
```

٢. بناء Shared Library أو System Design، فإن كان لديك مجموعة من التصاميم والأجزاء التي قمت ببنائها من خلال Web Component فيمكنها أن تكون كقاعدة لك على مستوى جميع المشاريع الخاصة بك داخل المؤسسة ودون الخوف من مشاكل التوافقية أو التعارض أو الخوف من تغيير التقنية المستخدمة...

باختصار، إن أفضل وقت يمكنك أن تستخدم فيه ال Web Component عندما تكون التوافقية وقابلية إعادة الاستخدام من الأولويات الرئيسية في مشروعك.

ويمكنك الدخول إلى هذا المثال من خلال هذا الرابط.

ال Architecture characteristics:

وكما تحدثنا سابقا، لكل معمارية أو نمط خصائص تميزه عن الآخر، واستخدام تقنيات مختلفة أو أسلوب ما قد يؤثر على الخصائص المعمارية التي تعمل عليها... والآن لنأخذ نظرة مع هذه الخصائص التي تتعلق بوجود ال Web Component ضمن معماريتنا:

١. ال Deployability (سهولة النشر) - العلامة ٤/٥ ^^:
عملية ال deploy هنا سهلة جدا أثناء ال runtime أو أثناء ال compiling، ويمكننا رفع ما نحتاجه في حالة ال runtime على ال cdn ومن ثم استخدامها في أي مكان كما فعلنا في الصورة المعنونة ب mfe-chat! أما إذا كانت وقت ال compile time فيمكن تضمينها ك module بكل سهولة مثل import 'my-component'، كما يمكن استخدام ال SSR لخدمة ال web component، لكن هذه العملية ليست سهلة أو سلسلة لذلك فهذه النقطة سبب جعل العلامة هي ٤ من أصل ٥.

٢. ال Modularity - العلامة ٣/٥ ^^:

من خلال هذه المعمارية وهذا الأسلوب يمكننا تحصيل مستوى جيد من تقسيم العمل لأجزاء صغيرة، هذه الأجزاء مستقلة وصغيرة وقابلة للاختبار والصيانة بشكل منفصل عن باقي الأجزاء، لكن تكمن المشكلة الكبرى في تداخل المفاهيم عند العمل بين ما هو حقيقة Micro frontend وبين ما هو Component، فإذا كانت ال Web Component تم تصميمها لاحتواء تطبيق أو فكرة مستقلة مسؤولة عن نطاق معين فهذا استخدام صحيح لأنها في هذه الحالة تمثل micro، أما إذا كانت وحدات صغيرة مثل ال buttons أو texts فهذه component، فالمشكلة هنا لن يكون من السهل التمييز بين ال micro وبين ال component في المشروع الخاص بنا، وهذا هو السبب في كونها ٣ من أصل ٥... لذلك فهذا يتطلب آلية وتوجيه واضح لماهية ال micro frontend ومن هي وبين ال component وماهيتها... وأين سيتم وضع واستخدام كل واحدة منها...

٣. ال Simplicity - العلامة ٤/٥ ^^:

العمل من خلال ال web component تعتبر عملية سهلة، فيمكن للمطورين تقسيم ال micro frontend بشكل ممتاز، لكن كما ذكرنا في النقطة السابقة فإن هذا التقسيم قد يفرط في استخدامه المطورين فتتحول من micro frontend إلى nano frontend ^^، وهذا سيزيد من التعقيد بشكل كبير... والحل يكون بالعودة لما ذكرناه من تأسيس في أول الكتاب، وهي التركيز على ال Business side وليس الجانب التقني فقط، بل الأولوية للبيزنس ثم ننظر للتقنية المراد استخدامها لتغطي حاجات البيزنس...

٤. ال Testability (قابلية الاختبار والفحص) - العلامة ٤/٥ ^^:

عملية ال unit testing أو ال integration test هنا هي عملية سهلة نوعا ما، لكن التحدي الأكبر عندك هو بناء ال test مع فهم لل Web Component Apis حتى يمكنك محاكاة دورة الحياة الخاصة بال component التي قمت ببنائها... وهذا قد يعني الحاجة لتعلم شيء جديد خصوصا إذا كانت خبرتك في ال test مبنية على أدوات مثل React Testing Library...

٥. ال Performance (الأداء) - العلامة ٤/٥ ^^:

تعد هذه النقطة من المزايا المهمة لل Web Component، فأداؤها ممتاز مع وجود العزل الذي تحدثنا عنه، والسبب في ذلك يعود لكونها ترث ال Html Component...

أما إذا كنت تبني ال component باستخدام ال html css js دون أي مكاتب ونحو ذلك فعند إذن سيكون لديك صاروخ سريع ^^... وتعد هذه واحدة من أفضل الخيارات والمزايا إذا كان الخيار المعتمد لدينا هو ال client side rendering.

٦. ال Developer Experience - العلامة ٤/٥ ^^:

لا يوجد اختلافات كبيرة أو صعبة في التطوير، فد تضطر لتعلم بعض الأشياء الخفيفة لكنها في المجمل تشبه ما اعتدنا عليه من العمل...

٧. ال Scalability (القابلية للتوسع) - العلامة ٥/٥ ^^:

عملية التوسع هنا من أسهل ما يكون، فكما رأيت يمكن جلب الملفات التي نحتاجها بناء على ال micros التي نحتاجها، ويمكن وضعها في ال CDN بسهولة، والسبب في ذلك يعود لأن النتيجة النهائية هي static html, css, js ^^.

٨. ال Coordination (التنسيق) - العلامة ٣/٥ ^^:

التنسيق هنا فيه نوع من التعقيد، والسبب يعود لما ذكرناه في أول النقاط، أهمها الحاجة لضبط آلية التجزئة وإدارتها بحيث لا تكون صغيرة جدا فننتقل من ال micro إلى ال nano، ولا كبيرة جدا فنخسر خصائص مهمة مثل ال deploy المستقل وتقليل الترابط بين الفرق وال micro... وهذا يلزم وجود guideline واضحة وحدود واضحة لكل فريق وكل micro سيتم تصميمها...

شاهد الصورة Table 4-4:

Table 4-4. Architecture characteristics summary for developing a micro-frontends architecture using web components

Architecture characteristics	Score (1 = lowest, 5 = highest)
Deployability	4/5
Modularity	3/5
Simplicity	4/5
Testability	4/5
Performance	4/5
Developer experience	4/5
Scalability	5/5
Coordination	3/5

ال Server Side:

يعد استخدام ال Server Side مع ال Horizontal Split واحدة من أكثر الحلول قوة ومرونة في عالم ال micro frontend، وذلك يعود لعدة أسباب أهمها وجود ال Cloud service ^^، والتي من خلالها أصبح التركيز على Value Stream بدلا من ال infrastructure Operationalization، وبذلك يصبح لدينا مرونة في التشغيل وتعقيد أقل بال infracstrucute الخاصة بنا، مما يسمح لنا بالتركيز على ال Value Stream التي سنقدمها... وعادة ما يتم استخدام هذا الأسلوب عند الحاجة لوجود SEO قوي للموقع، ومما يعزز آلية الحصول على أفضل SEO هنا هو حصولنا على أفضل أداء -سرعة تحميل- ممكنة لأن الصفحات يتم تجميعها على ال server ومن ثم عمل Cache لها -وممكن على أكثر من مكان مثل ال CDN أو ال server أو ال memory)، كما أنه يتم عرض وجلب الصفحات بما تقتضيه متطلبات الصفحة من ملفات js فقط دون الحاجة لتحميل أشياء كثيرة لا داع لها...

وطبعا كما تعودنا، الحياة هنا ليست وردية ^^، وبناء على ذلك فإن جميع التحديات والمشاكل التي تكلمنا عنها سابقا في ال Client side ستجدها هنا أيضا مضافا لها بعض التحديات الإضافية...

ملاحظة ١: يقصد بال Value Stream: هو تركيز المطورين على إنشاء أو تطوير مزية أو خاصية جديدة للمستخدمين المنتفعين من التطبيق الخاص بنا، وذلك يشمل التحليل لمتطلبات

ال client وتصميم الحل وكتابة الكود واختبار النتائج والتحقق منها ومن ثم إصدارها وتسليمها... بحيث تمثل كل خطوة في هذه السلسلة قيمة حقيقة ستضاف عند الوصول لكل خطوة... وبهذا نتخلص من أي waste time يمكن أن يتواجد ليس له علاقة مباشرة بعملنا أو يمكن اختصاره مع ما نقوم به من أعمال، فمثلا بدلا من إضافة الوقت الكبير على إعداد بنية تحتية تشمل السيرفرات وأخذ نسخ احتياطية من قواعد البيانات وإعداد ال firewall ونحو ذلك، نجعل المطور يركز على كتابة الكود وحل المشكلات والابتكار لتحسين المنتج وأي عمل يضيف قيمة مباشرة للمشروع التجاري، ونترك الأمور الأخرى بعيدا عن هؤلاء المطورين.

ملاحظة ٢: يقصد بال infrastructure Operationalization هو تحويل البنية التحتية مثل الخوادم والشبكات وقواعد البيانات وغيرها من كونها مكونات يتم تجهيزها بشكل يدوي إلى مكونات جاهزة للعمل وتدار بشكل آلي ومنهجي ضمن العمليات التشغيلية... فمثلا بدلا من أن يقوم المطور بإعداد وإنشاء قاعدة بيانات بشكل يدوي وصيانتها وأخذ نسخ احتياطية منها فإنه يقوم باستخدام واحدة من الخدمات السحابية الموجودة من أي مقدم خدمة مثل AWS أو GOOGLE... إلخ، لذلك تجد كثير من ال devops وال developer يتوجهون للتعامل وبناء الإعدادات بما يتوافق مع مقدمي هذه الخدمات باعتبارها ركيزة موجودة ضمن حياة المشاريع التي تحت يدهم... وبهذا يتم تحقيق مزية ال Value stream.

ال Scalability and response time:

أول تحدي مضاف سنتعامل معه هو كيف يمكن أن نقوم بإعداد بنية صحيحة للتطبيق الخاص بنا ومتوافقة مع حجم ال traffic الممكن واختلافاته الخطيرة الممكنة!
جميعنا يعلم أن ال Auto Scaling التي يوفرها معظم ال Cloud يمكن أن تساعدنا في هذه العملية بشكل كبير، إلا أن اختيار آلية المعالجة المناسبة لتحقيق ال Auto Scale بأفضل شكل ممكن سيؤثر بشكل كبير على جاهزيتنا، وهنا لنفترض أن ال Compute Layer ستكون إحدى الخيارات التالية:

1. ال Virtual Machine: في حالة اختيارنا لل Auto Scale اعتمادا على ال Virtual Machine = فهذا يعني أننا بحاجة لوقت أطول لعملية ال scale، والسبب في ذلك يعود للحاجة لبناء نظام تشغيل متكامل داخلها حتى تنهض ومن ثم ربطها بال scales server الخاصة بنا...
2. ال Containers مثل docker: هنا تعمل عملية ال auto scaling أسرع لأنها أخف من تشغيل ال Virtual Machine كما أنها تشترك بنواة التشغيل مع ال os host.
3. ال Managed Containers مثل Serverless: تعد هذه أسرع طريقة لل Auto Scaling لأن ال Cloud Provider سيقوم بالكامل بإدارة ال infrastructure. وهي سريعة الإعداد وسريعة ال Auto scaling ومرنة...

ملاحظة: كما تلاحظ هنا أن الحل الأفضل أو اختيار الطريقة الأفضل يتبع ما نقوم ببنائه، فعملنا على micro frontend قد يدعونا لاستخدام الـ 2 أو 3... في حين لو أننا

monolithic أو سيرفر يحتاج لتشغيل نظام تشغيل مختلف أو يحتاج لعزل الخدمات عن بعضها.. فسيكون الخيار رقم ١ هو أول ما نفكر فيه...

والآن، هل اختيارنا وتحديدا لل Compute Layer كافي لمواجهة الطفرات المفاجئة في ال traffic الخاص بنا؟

في الواقع لا ^^، سيتبقى لنا هناك خطورة واردة في بعض الحالات، لأن هذه الزيادات المفاجئة تعني الحاجة لعمل Auto Scale، وهذه العملية تحتاج لوقت حتى لو كانت VM أو Container... فالثواني القليلة مثلا لإعداد ال docker قد تكون كفيلة بإرجاع أخطاء مثل ٥٠٣! ومن الأمثلة على ذلك ما يسمى بال Black Friday أو وجود إعلان تسويقي أو خبر ما أدى لهجوم المستخدمين على موقعك ^^، كما أن بعض مقدمي الخدمة يضعون شروطا للوقت اللازم لتنفيذ عملية ال auto scale، أو قد يحدث تأخر ما بين الإعلام عن وجود زيادة بال traffic مقارنة بالوقت الفعلي -وإن كان قصيرا-...، ومن الحلول المقترحة للتخلص من هذه المشكلة: ال Predictive Load، وهو مصطلح يشير إلى الحاجة لوضع تقدير مسبق لحجم الضغط الممكن أن يتواجد على النظام الخاص بنا في تاريخ ما في المستقبل بناء على بيانات تاريخية أو أنماط سابقة وتجارب مشابهة... وبناء على ذلك سيكون الحل لدينا بالقيام بزيادة البنية الأساسية بشكل مسبق لضمان القدرة على التعامل مع زيادة ال traffic التي ستحصل... فمثلا إذا كنت أعلم أن لدي حملة تسويقية غدا وستنهال علي الطلبات، فيمكنني بكل بساطة زيادة عدد ال container استباقيا...

ولدينا تحد أخر هنا، وهو كيف يمكن أن نسرع response time لل Service أو ال micro frontend؟ والجواب يمكن أن يكون من خلال إحدى هذه الحلول:

- إذا كان لدينا إمكانية لعمل cache لبعض الخدمات أو ال micro فيمكننا فعل ذلك... أي هل يمكننا أن نضحي ببعض ال consistency في مقابل تحسين الأداء؟ وماذا عن عمل cache لـ Composition Dom Tree الخاصة بال micro؟.. ماذا عن استخدام خدمة أو أداة مثل redis لهده العملية؟... بكل تأكيد فإن هذه العملية ستزيد من كفاءة النظام وسرعته ^^.
- ماذا عن استخدام ال CDN بدل ال Cache؟ فتقليل ال Latency سيقوم بتسريع عملية إيصال الصفحات المطلوبة للمستخدم...

إن الفكرة التي تقوم عليها هذه الحلول هي محاولة استدعاء ما يتم استيراده ومعالجته بكثرة في مكان ما دون الحاجة للقيام بتلك العمليات مرارا وتكرار... أو من خلال تسريع وقت الاستجابة بتقليل وقت ال latency من خلال شبكات ال CDN والتي ستكون قريبة من المستخدمين وموزعة حول العالم...

بناء على هذه المعطيات يصبح لدينا معيار لتقييم مقياس النجاح في هذه العملية مع معماريتنا، وهذه المقاييس هي:

1. ال Latency
2. ال Response Time

3. ال Cache Eviction: ويقصد بها متى سنقرر حذف أو إزالة ال cache للسماح بنسخ جديدة أو خدمات جديدة أكثر استخداما الآن مكان القديمة... وهذه العملية مهمة جدا.

لكن بكل تأكيد عملية إنشاء هذه البنية عملية مهمة وضرورية وليست سهلة!، وتحتاج لخبرة ومعرفة متخصصة فعلا... خصوصا إذا زاد التعقيد الخاص بالخدمات وال micros المقدمة.

ال Infrastructure ownership:

من التحديات الخطيرة في هذه البنية هي تحديد المسؤول عن تجميع ال layers المختلفة (ال micros المختلفة) من عدة فرق حتى تنتج لنا صفحة كاملة تعمل كما هو متوقع... لذلك، هناك احتمالان هنا:

الأول أن يكون كل من فريق ال Frontend وال Backend (تذكر أننا نتحدث عن بنية server side) يعملان معا على layer مشتركة، وبهذا فكلا الفريقين يمكنهم ضمان انتقال البيانات بسلاسة من ال server إلى client، مع مراقبة الأداء ومتابعة ال logging بشكل أسهل وأوضح لكلا الفريقين، وفهم أوسع وأشمل حول الآلية المناسبة لتجميع هذه المكونات معا، ولذلك يعد هذا الخيار هو الخيار الأكثر جودة...

الثاني: وهو السيناريو الأقل جودة، حيث تفضل بعض الشركات أن يكون كل فريق وكل layer مفصولة عن غيرها، لذلك يكون هذا التحدي صعبا هنا أمام فهم كيفية مشاركة البيانات بين الطبقات وآلية متابعة الأخطاء وإصلاحها ومنها صعوبة تجميع المكونات دون تنسيق عالي...

وكمثال عملي، تخيل أن لديك موقع تجارة إلكتروني فيه هذه المكونات:

الفريق المسؤول	المايكرو-فرونت إند	ال Backend Service
فريق A	HeaderApp	UserService
فريق B	SearchResultsApp	SearchService
فريق C	CartApp	CartService
فريق D	RecommendApp	RecommendService

في الاحتمال الأول، تتعاون الفرق جميعها في ال layer الخاصة بتجميع المكونات، وبهذا كل فريق يعلم كل جزء كيف سيتم تحميله، وكيف سيتم تمرير البيانات له، ويمكنهم بناء على ذلك القيام بالتحسينات اللازمة في هذه الطبقة لتسريع التحميل وتحسين الأداء... لكن هذه العملية تحتاج لتعاون وتنسيق وقدرة على التواصل عالية بكل تأكيد..

في الاحتمال الثاني: فريق ال CartApp لا يعرف أن ال micro الخاصة به ستقوم بإبطاء الصفحة، وفريق SearchResultApps لم يعرف أن طريقة إرساله للبيانات خاطئة... لأن كل فريق يمثل بيئة مغلقة لا يدرك بشكل فعال ما يدور بعد تسليمه للمكون الذي يعمل عليه... وهنا سيكون دور فريق التجميع -بعد تحديد من هو- دور صعب...

ولنتخيل مخطط سير العمل بشكل واضح، يمكننا تخيل العمل كما يلي:

سنقوم باستلام request للمستخدم مثل search?query=computer، وهنا سيذهب ال request لل Composition layer، بعد ذلك سيقوم ال server بالتواصل مع خدمات ال backend المختلفة لجلب البيانات الخاصة بالمستخدم، وجلب نتائج البحث المتعلقة بهذا المنتج، وجلب محتويات ال cart وجلب التوصيات التي قام بها الزبائن حول هذا المنتج... ثم يتم تمرير هذه البيانات إلى ال micros المختلفة بحسب الآلية التي صممت من خلالها، مثلا عن طريق ال props... فمثلا سيعرض حينها ال header اسم المستخدم وعدد العناصر بال cart... ثم سيتم تجميع جميع هذه ال micros في صفحة html واحدة وترسل للمستخدم على المتصفح كاملة ودون الحاجة لتجميعها عنده من خلال الجافا سكربت (تذكر أننا نتحدث عن server side composition)... فقط وببساطة، لهذا كان الخيار الأول في هذه البنية هو الأكثر جودة...

فائدة

فلْتعلمْ أن الله -سبحانه وتعالى- جعل الناس خلائف في الأرض، وجعلهم متفاوتون في الدرجات، فمنهم الغني ومنهم الفقير، ومنهم الصحيح ومنهم السقيم، ومنهم العالم ومنهم الجاهل وغيرهم، وفي نَفَس الإنسان وعمره نفسُه في تفاوت، بين الغنى والفقر، والصحة والمرض إلى آخره، هذا التفاوت يحقق التكامل الذي يجعل عمارة الأرض ممكنة لتنوع الناس وميولاتهم ووظائفهم وأعمالهم، وفي هذا التفاوت يقع البلاء لكل منا، فينظر الله -سبحانه وتعالى- لأعمالنا وفي جميع أحوالنا، فينظر إلى ما نقدم ونفعل، في شبابنا وهرمنا، وغنانا وفقرنا، وصحتنا وسقمنا، فيا رب ارحمنا برحمتك أنت، وعاملنا بما أنت أهل له، لا بما نحن أهل له، وأدخلنا برحمتك في عبادك الصالحين، والحمد لله رب العالمين.

- كتاب إلى الجنة زمرا، الصفحة ١٥٩

Composing micro-frontends ال:

قد يختلف تركيب ال micro-frontend في هذه البنية المعمارية المعتمدة على ال server عما رأيناه سابقا، ولكن هذه الاختلافات ستكون في التفاصيل الدقيقة لا في الجوهر العام، لاحظ الصورة F4-23 والتي تعرض البنية النموذجية لتجميع ال micros والمكونة من ثلاث طبقات:

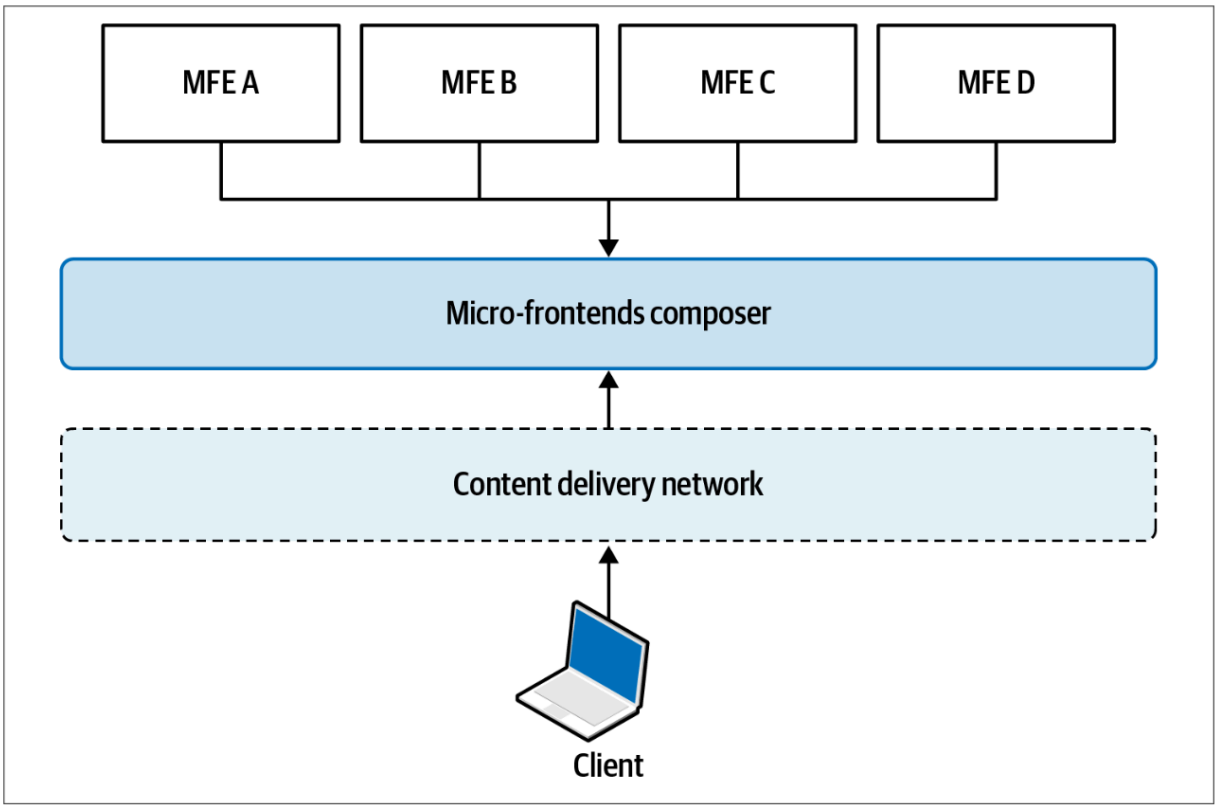


Figure 4-23. A typical high-level architecture for a server-side micro-frontend architecture, where a composer is responsible for stitching together the different micro-frontends at runtime. A CDN can be used for offloading traffic to the origin.

هذه الطبقات هي:

١. ال Micro-frontend: وهي تمثل الطبقة التي تحتوي ال micros المختلفة التي قمنا ببنائها، ويمكن عمل deploy لها على شكل static assets وقت ال compile time، أو يمكن أن يتم ذلك خلال ال runtime عن طريق server من خلال استخدام ال template لبناء dynamic assets...

مثال عملي: تخيل أن لديك متجر إلكتروني لبيع الكتب، وهناك صفحة لعرض معلومات الكتاب والتي تحتوي عنوان الكتاب، وصورة الغلاف ووصف عن الكتاب وسعر الكتاب... في هذه الحالة سنقوم بوضع Template يمثل هذه المكونات وأثناء ال runtime سيقوم ال server بإنشاء الصفحة اعتمادا على هذا ال template بعد استبدالها بالبيانات المطلوبة، مثلا book/view/1... شاهد الصور التالية:

```
<!-- Template -->
<div class="book-details">
  <h1>{{title}}</h1>
  <img src="{{coverImage}}" />
  <p>{{description}}</p>
  <p>{{price}} :السعر</p>
</div>
```

```
<!-- Render -->
<div class="book-details">
  <h1>إلى الجنة زمرا</h1>
  <img src="/covers/2nees.jpg" />
  <p>كتاب لكل بيت مسلم...</p>
  <p>السعر: مجاني</p>
</div>
```

٢. ال Composer: هنا سيتم تجميع جميع ال micros التي نحتاجها بالصفحة لإرجاعها للمستخدم، ويمكن استخدام ال nginx أو Apache HTTPd من خلال ال SSI لهذا

الغرض... ويمكن أن يكون الموضوع أكثر تعقيدا باستخدام ال kubernetes ربط كل الأجزاء معا...

إن آلية عمله ببساطة أن ال server يقوم بتوجيه ال request إلى ال composer، ويقوم ال composer بعمل request لكل ال micros التي نحتاجها في هذه الصفحة... مثلا https://mf-header.2nees.com و https://mf-content.2nees.com وهكذا... وبعد حصوله على ال html الخاص بكل micro يقوم بتجميعها داخل Main Html Template، شاهد الصور التالية:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>MF Example</title>
  </head>
  <body>
    <!-- Header -->
    {{HEADER}}

    <!-- CONTENT -->
    {{CONTENT}}

    <!-- Footer -->
    {{FOOTER}}
  </body>
</html>
```

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>MF Example</title>
  </head>
  <body>
    <header>...MFE Header من...</header>

    <div class="book-details">
      <h1>إلى الجنة زمرا</h1>
      <img src="/covers/2nees.jpg" />
      <p>...كتاب لكل بيت مسلم</p>
      <p>السعر: مجاني</p>
    </div>

    <footer>...MFE Footer من...</footer>
  </body>
</html>
```

٣. ال CDN: استخدام ال CDN كلما كان ذلك ممكنا يعد خيارا ممتازا عند تعاملنا مع هذه البنية، فإذا كان هناك إمكانية لعمل cache للصفحات ولو بشكل مؤقت للحد من الضغط

الكبير على ال composer، ستكون هذه العملية ممتازة وستحسن من الأداء ووقت الاستجابة بشكل ملحوظ...

ال Micro-frontend Communication:

في هذه البنية (ال Server-side approach)، الأصل عدم الحاجة للكثير من التواصل بين ال micros داخل نفس ال View، بل الغالب سيكون التواصل من خلال ال APIs، والسبب في ذلك يعود إلى أن الصفحة في النهاية سيعاد تحميلها (reload) بعد كل إجراء مهم يقوم به المستخدم عليها (للتقريب: السلوك يشبه ما اعتدنا عليه من مواقع لم تصمم ك SPA)، ومع ذلك فهناك حالات نحتاج فيها إلى إبلاغ إحدى ال micro بأن شيئا ما قد حدث من قبل المستخدم، مثل أن يقوم المستخدم بإضافة منتج إلى سلة التسوق، ففي هذه الحالة، تحتاج ال micro المسؤولة عن إضافة المنتج إلى إعلام ال header micro بوجود تغيير على عدد المنتجات، ومنه عرض عدد المنتجات الذي قام المستخدم بإضافتها... وذلك يتم من خلال ال Event Emitter كما تحدثنا سابقا...

شاهد الصورة F4-24 لترى كيف يتم الأمر بشكل عملي:

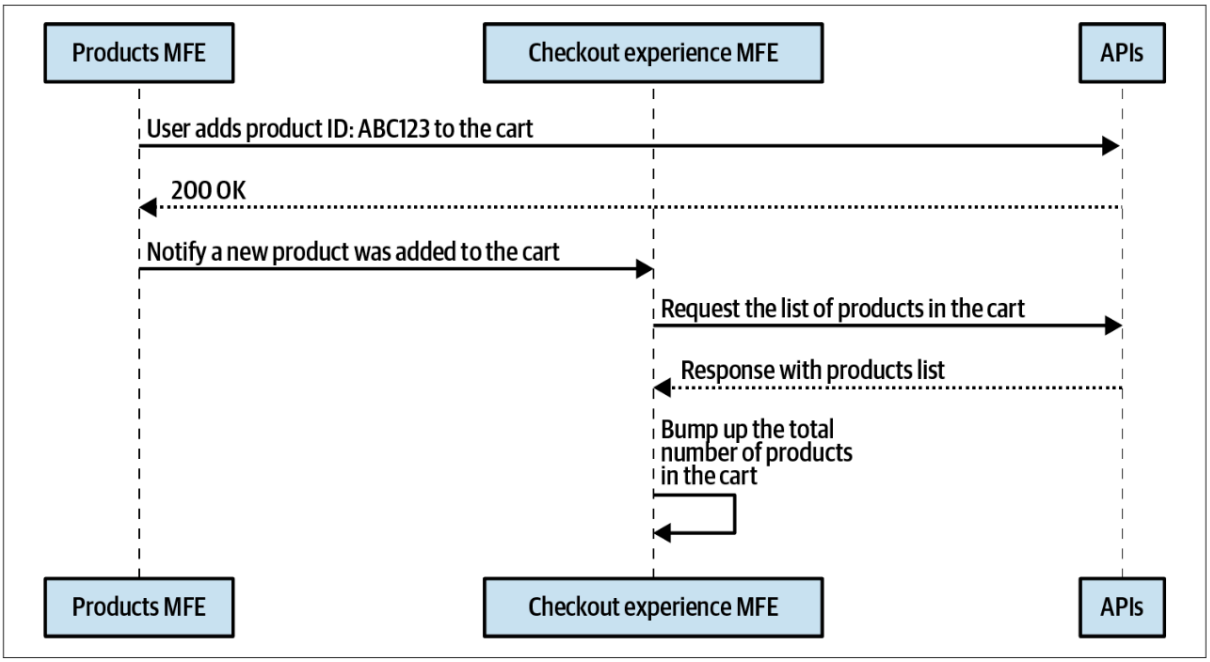


Figure 4-24. An example of how the product's micro-frontend notifies the checkout experience micro-frontend to refresh the cart interface when a user adds a product to the cart

في هذه الصورة:

١. **يقوم المستخدم بإضافة منتج إلى السلة،** ثم يتم إرسال هذا ال event إلى ال backend، والذي يقوم بدوره بتحديث ال session لتأكيد إضافة المنتج.

٢. **تقوم ال micro الخاصة بالمنتج بإشعار ال micro الخاصة بال checkout** بأن هناك منتجا جديدا قد أضيف إلى السلة.

٣. **تقوم ال micro الخاصة بال checkout بجلب قائمة المنتجات الجديدة من ال cart،** وتقوم بعرض المعلومات الجديدة في واجهة المستخدم.

فقط وببساطة ^^.... لكن تذكر أن الأصل في هذه المعمارية ألا يكون هناك العديد من هذه ال events بين ال micros المختلفة داخل ال view في معظم التطبيقات، ولذلك فإن هذا النوع من الكود **لا يؤثر سلبًا على الأداء أو قابلية الصيانة...** أما إذا زاد عدد الأحداث فهذا يدل على وجود مشكلة خطيرة تحتاج إلى حل مستعجل، والحل قد يبدأ من التحقق هل فعلا هذه المعمارية مناسبة لما نقوم به؟ وهل فعلا قمنا بإرسال ال events المهمة فقط أم أرسلنا كل ما هب ودب ^^... إلخ

ال Available frameworks:

هناك الكثير من الحلول والعديد من الطرق التي قامت الشركات بتطويرها أو تصميمها لبناء هذه المعمارية، ولكل أسلوب مزاياه الخاصة وعيوبه التي تنتج عنه... ويمكنك البحث عن ال framework المناسبة لك في حال رغبتك باستخدام إحداها... لكن سأنطلق مباشرة إلى ال SSI.

ال SSI هي اختصار ل Server-Side Includes، وهي تقنية يتم استخدامها لتقسيم ال View النهائية إلى عدة أجزاء، هذه الأجزاء تسمى Fragment، ويتم تجميع هذه الأجزاء من خلال ال server وقبل إرسال ال Static Page إلى ال Client... وقد كانت تستخدم قديما لفصل الأجزاء ال dynamic عن ال static في صفحات ال html، شاهد الصورة F4-27:

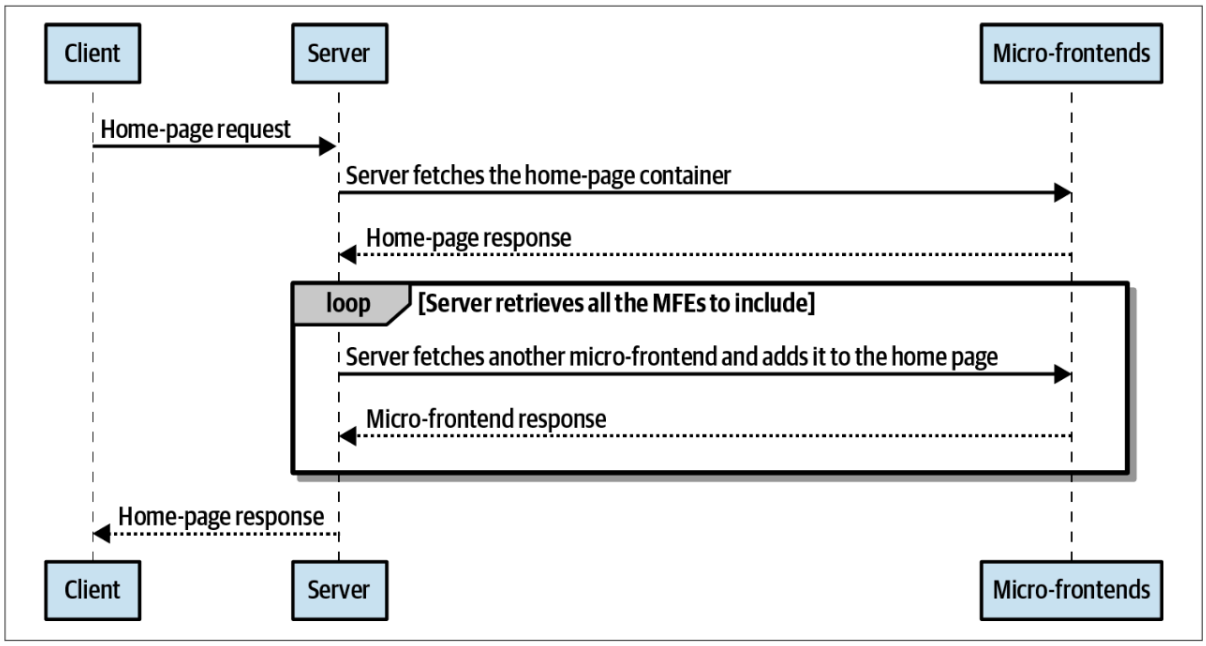
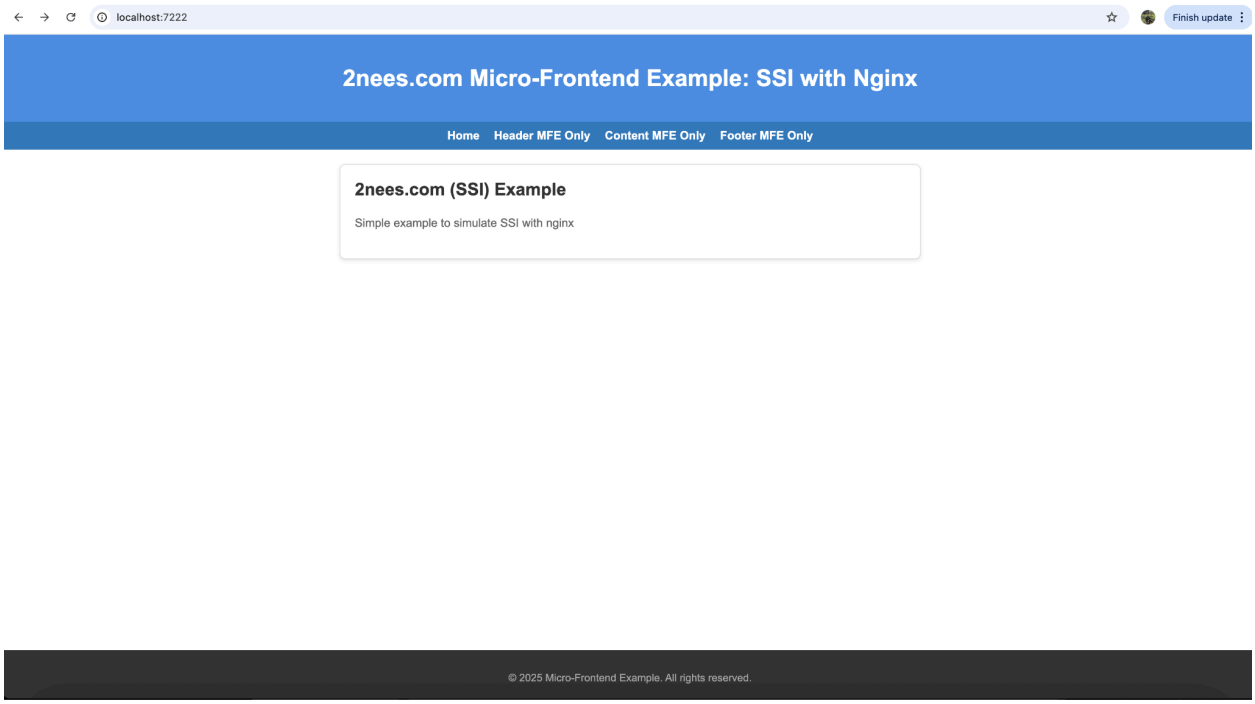


Figure 4-27. The sequence diagram shows how a client request is handled by a server when SSI are used

كما تشاهد في الصورة، فإن ال server قام بجلب الصفحة التي تحتوي على ال directives، ثم قام ال server بتحديد كل directive منها واستبداله (جلب) هذا الجزء ومن ثم تحميلها وتجميعها وإرسالها لل client، وهذه العملية تتم بشكل parallel... لكن هذا الأسلوب لديه تحديات مهمة -غير التي تحدثنا عنها سابقا مثل تضارب ال css- مثل مشكلة تأخر وصول ال TTFB إذا تأخر تحميل أي fragment... لذلك يجب الحرص على معالجة هذه المشاكل.

شاهد المثال:



ويمكنك الدخول إلى هذا المثال من خلال هذا الرابط.

والسؤال الآن، متى يكون من المناسب استخدام بنية ال server side؟

يمكننا القول أن هذا الأسلوب مناسب عند العمل على حلول ال B2B أو عند الحاجة للحصول على SEO قوي.
وفكرة التوصية بهذا الحل عند ال B2B نابعة من أن هذه الأنظمة تحتوي بداخلها العديد من ال modules والتي يعاد استخدامها في واجهات متعددة، وتكون ال layout الخاصة بها أيضا moduler، مثل ال dashboards...

لكن يجب علينا توفير Full stack developer للعمل من خلال هذا الأسلوب، أو Backend لديه المهارات المناسبة التي تساعده وتخوله للعمل على هذه المعمارية...

واحذر من استخدام هذا الأسلوب إذا كانت الواجهات الموجودة فيها تفاعل عالي وكبير بين مكوناتها المختلفة -كما أشرنا سابقا- أو أن الصفحات مصممة لتكون dynamic مثل اعتماد ال fluid layouts للتصميم، لأن ذلك يحتاج إلى تنسيق كبير بين الفرق المختلفة، ولأن ذلك سيزيد من صعوبة تتبع الأخطاء وإصلاحها...

ال Architecture characteristics:

وكما تحدثنا سابقا، لكل معمارية أو نمط خصائص تميزه عن الآخر، واستخدام تقنيات مختلفة أو أسلوب ما قد يؤثر على الخصائص المعمارية التي تعمل عليها... والآن لنأخذ نظرة مع هذه الخصائص التي تتعلق بال Server Side Composition ضمن معماريتنا:

١. ال Deployability (سهولة النشر) - العلامة ٤/٥ ^^:
عملية ال deploy هنا سهلة إذا تمت أتمتتها بالشكل الصحيح، فعادة ما يتم نشر API جديدة مع كل micro frontend جديدة، فيجب أن تتم إدارة هذه العملية بشكل تلقائي وهو ما يحتاج بعض الجهد لضبط القواعد وضمان الإصدار المتوافق...

٢. ال Modularity - العلامة ٥/٥ ^^:

من خلال هذه المعمارية وهذا الأسلوب يمكننا تحصيل مستوى عال من تقسيم العمل لأجزاء صغيرة، هذه الأجزاء مستقلة وصغيرة وقابلة للاختبار والصيانة بشكل منفصل عن باقي الأجزاء، كما أنها تتيح لنا إدارة ال Caching وتحسين النتائج النهائية مع القدرة على تغيير التقسيم بسهولة... وهذا نابع من أن كل frontend هو عبارة عن module يقوم بمهام محددة...

٣. ال Simplicity - العلامة ٣/٥ ^^:

لا تعتبر هذه المعمارية من المعماريات السهلة في التفنيذ، فهي تحتوي على العديد من العناصر المتداخلة والتي تحتاج إلى أدوات مراقبة وتحليل على مستوى ال BE & FE... كما أن ازدياد عدد المستخدمين بشكل مفاجئ قد يسبب مشكلة كبيرة خصوصا في المشاريع الكبيرة، ولذلك يجب الاعتناء بهذا الجانب أيضا... ومع أنها تعتبر معمارية قوية إلا أنها تعتبر الأكثر تحديا أيضا...

٤. ال Testability (قابلية الاختبار والفحص) - العلامة ٤/٥ ^^:

غالبا عملية الفحص هنا عملية سهلة، لأننا نقوم بفحص الصفحات النهائية وكأننا في Server side rendring التقليدي، لكن هناك تحدي عند وجود تفاعل بين مكونات الصفحة المختلفة على مستوى ال micro (تفاعل بين ال micros المختلفة)، فهنا قد نحتاج لجهد إضافي لفحص هذا الجزء...

٥. ال Performance (الأداء) - العلامة ٥/٥ ^^:

أداء ممتاز بسبب القدرة على التحكم الكامل في المخرجات النهائية التي سيتم إرسالها للمستخدم... فمثلا يمكننا حفظها بال cache قبل إرسال الصفحة، وتحميل الأجزاء المطلوبة فقط لهذه الصفحة...إلخ، وهذا لا يعني أن العملية بسيطة، بل تحتاج إلى جهد لتحقيق ذلك، لكن ذلك يقلل من البساطة لكن الأداء سيكون في أعلى مستوى...

٠٦. ال Developer Experience - العلامة ٣/٥ ^^:

ستحتاج المؤسسة إلى استثمار وقت في إنشاء أدوات مخصصة لتحسين إدارة المشروع، كما أن المطورين المناسبين لمثل هذه المعمارية هم ال Full Stack... أو FE لديه علم ومعرفة بال Backend و BE لديه فهم ومعرفة بال FE... حتى يكون الجميع مدركا لآلية التواصل بين المكونات وطريقة تجميعها...، لذلك يعتبر ال Full Stack هم الأكثر ميلا لمثل هذه المعمارية...

٠٧. ال Scalability (القابلية للتوسع) - العلامة ٣/٥ ^^:

عملية التوسع هنا مهمة غير بسيطة في المشاريع الكبيرة، فهي تحتاج للمتابعة والتحكم على عدة طبقات، فمثلا لدينا ال composition layer وتوسيعها... ال caching ب layers المختلفة التي يجب التحكم بها... ال API عددها والوقت الخاص بكل منها... إلخ

٠٨. ال Coordination (التنسيق) - العلامة ٣/٥ ^^:

التنسيق هنا فيه نوع من التعقيد، فيجب ضمان أن تعمل الفرق بشكل مستقل لكن يجب أن يحتفظ كل فريق بصورة كاملة صحيحة عن العمل وأجزائه... لذلك يجب أن يكون التنسيق هنا مصمم بعناية وحذر...

شاهد الصورة Table 4-5:

Table 4-5. Architecture characteristics summary for developing a micro-frontend architecture using horizontal split and server-side composition

Architecture characteristics	Score (1 = lowest, 5 = highest)
Deployability	4/5
Modularity	5/5
Simplicity	3/5
Testability	4/5
Performance	5/5
Developer experience	3/5
Scalability	3/5
Coordination	3/5

فائدة

فلْتعلمْ أن أهل الباطل والضلال، وأصحاب الاستبداد يبحثون دومًا عن مبررات لاستبدادهم وضلالهم، وأكثر المبررات التي يعتمدون عليها هي المبررات التي تمس عقائد الناس، ومن خلال أكثر الصور رمزية لهم، فتجدهم يتوددون لكل ضعيف النفس، ولكل رخيص تستهويه الدنيا، فيبحثون عن فُتيا رخيصة هنا أو هناك، ويسعون لإخراج النصوص عن معناها، حتى يتمكن لهم الفساد والإفساد بقوة السلاح والترهيب تارة، وبغطاء عقدي منتزع ممن باعوا أنفسهم ودينهم بثمن بخس تارة أخرى، ويصبح أهل الحق وأصحابه بعدها إما في السجون، وإما في القبور! وتختلف هذه الرواية باختلاف قوة الحق والباطل، فالتباين هنا يغير مجرى الرواية، ومن القصص على هذا أن بني إسرائيل بعد أن جاوزوا البحر -بفضل الله سبحانه وتعالى-، وبعد أن رأوا من المعجزات ما يوجب اليقين بقدرة الله -سبحانه وتعالى-، مروا على قوم يعبدون الأصنام، فقالوا لنبي الله موسى -عليه السلام- اجعل لنا آلهة -أصنامًا- كما لهم!! طلبت نفوسهم ممن معه الحق أن يرخص لهم في الباطل، أن يتنازل قليلًا عما أمرهم به، أن يبدل من أقواله، لكن حاشا لنبي الله -سبحانه وتعالى- موسى -عليه السلام- من الخروج أو الحياد عن أمر الله، وحاشاه من المداهنة والتنازل عن الحق، وهنا انتهى بني إسرائيل عما طلبوه...

- كتاب إلى الجنة زمرا، الصفحة ١٧٢

ال Edge Side:

لقد قمنا بذكر بعض التفاصيل المهمة عن هذا الأسلوب سابقا، يمكن مشاهدتها من خلال الضغط على هذا الرابط...
لكن سنزيد على ما ذكرناه نقطة مهمة، وهي أن مجتمع مطوري ال FE لم يتبنى هذا النمط بشكل كبير كما تبنى الطرق الأخرى، وذلك بسبب عدة عوامل أهمها:

١. ال Vendors Fragmentation: فالشركات التي تقدم دعما لل ESI لا تلتزم جميعها بنفس المبادئ والمعايير، وهذا يسبب مشكلة كبيرة من ناحية التوافقية (مشكلة Inconsistency)، مما يجعل الكود غير قابل للنقل بسهولة بين مزودي الخدمة المختلفين.

٢. ال Lack of Tools: عند استخدام تقنيات مثل React هناك نظام كامل (ecosystem) يدعمها من أدوات تطوير وأدوات اختبار وأدوات debugging... إلخ، أما مع ESI، فالأدوات محدودة، فلا توجد بيئة تطوير متكاملة لدعمها ، ولا توجد إضافات للمحررات أو أدوات مساعدة في التصحيح والتحقق، ولا يوجد مجتمع كبير يطور أدوات لدعمها، لذلك يشعر المطور هنا بأنه يعمل بيده فقط ودون وجود دعم كاف لذلك...

٣. ال Poor DX: بسبب ما ذكرناه في النقاط السابقة، فتعتبر تجربة التطوير غير سلسلة لذلك يفضل المطورون حلولا أخرى...

آلية التنفيذ:

إن آلية تطبيق هذا الأسلوب تشبه بشكل كبير ال SSI، لكن الفرق الجوهري عنا هو أن ال Markup سيتم ترجمتها وتفسيرها قبل إرسالها للمستخدم.

وال Main functionalities هنا هي:

١. ال Inclusion: ويقوم ال ESI بتجميع ودمج المحتويات من خلال استبدال ال placeholder tag بال micro frontend.

٢. ال Variable support: ال ESI يدعم ال header وال query params وغيرها من المتغيرات ال dynamic، ويمكن استخدام هذه المتغيرات داخل ال esi أو داخل ال html.

٣. ال Conditional processing: يمكن كتابة شروط يتم من خلالها التحكم في عرض أو إخفاء بعض الأجزاء.

٤. ال Exception and error handling: يتيح ال ESI التعامل مع الأخطاء لحظة حصولها لتحسين تجربة المستخدم في حال حدوث مشاكل عند تحميل أحد الأجزاء الموجودة بداخله.

شاهد المثال:

README.md index.html

```
<html>
<body>
<!-- تضمين header الصفحة -->
<esi:include src="https://cdn.2nees.com/fragments/header.html" />

<esi:choose>
  <esi:when test="'$(HTTP_HEADER_X_USER_NAME)' != ''">
    <p>مرحبًا، $(HTTP_HEADER_X_USER_NAME)! شكرًا لزيارتك.</p>
  </esi:when>
  <esi:otherwise>
    <p>مرحبًا بك زائرنا الكريم!</p>
  </esi:otherwise>
</esi:choose>

<!-- تضمين المحتوى الرئيسي -->
<esi:include src="https://cdn.2nees.com/fragments/product-list.html" />

<!-- تضمين footer الصفحة -->
<esi:include src="https://cdn.2nees.com/fragments/footer.html" />
</body>
</html>
```

ال Transclusion:

أذكر أنني أول ما بدأت بتعلم البرمجة تفاجأت بعد مدة من وجود آلية تسمح بمشاركة أجزاء من الشيفرة البرمجية في أكثر من مكان دون الحاجة لتكرارها في أكثر من مكان... تحديدا لقد كنت أتعلم لغة ال PHP حينها وتفاجأت بوجود ال include وال require والتي تسمح لي

بتضمين جزء من الشيفرة البرمجية ودون الحاجة لكتابة تلك الشيفرة كل مرة... وقتها شعرت بأقصى درجات الفرح وأدركت مقدار الجمال في عالم البرمجة ^^... وهذا السبب الذي أفرحني كان يطلق عليه Transclusion، وتم استخدامه لتقليل عمليات النسخ واللصق التي كان يعتمد عليها المطورين أثناء تصميمهم صفحات الويب ^^، وهذا كان في بداية العقد الأول من الألفية الثانية ^^... أما الآن فيمكن استخدام هذه التقنية لإعادة استخدام المحتوى وبناء views جديدة بناء على بنية تركيبة يتم تحديدها تعتمد على مجموعة من الشروط البسيطة أو المتغيرات القادمة من قبل ال client.

ما ذكرناه يقودنا للحديث عن مفهوم مهم وهو ال CSI، وهي اختصار ل Client-Side Includes، وتشير لعملية التضمين التي تتم على جهاز الـ Client، وتحديدا في حالتنا على المتصفح... وهي عملية قد نكون اعتدنا على استخدامها دون أن نعرف اسمها... فمثلا كم مرة قمت بال react بتضمين جزء معين في أكثر من صفحة؟ وعلى نفس هذا النمط يعمل هذا المفهوم... وهذا المفهوم هنا مهم لأننا سنحتاجه عند الحاجة لدمج ال ESI مع ال CSI لحل مشكلة المحتوى ال dynamic...

لكن ما هي التحديات التي سنواجهها مع ال ESI؟

١. أول وأهم تحدي هو أن ال ESI بمزاياها المختفة ليست مدعومة من جميع مزودي الخدمة بالشكل نفسه -إن كانت تدعمه أساسا-، وهذا يعني أنك قد تضطر لتغيير هذه البنية في المستقبل إذا واجهتك بعض التحديات الخطيرة... خصوصا إذا كان التطبيق الخاص بك يحتاج إلى مرونة عالية...

٢. مشكلة المحتوى ال dynamic، فوجود محتوى مخصص لكل مستخدم قد يعني إنشاء صفحة بيانات فريدة لكل مستخدم، وحتى إذا قمنا بتقسيم بعض الأجزاء بناء على شروط معينة فقد نجد أنفسنا حصلنا على مجموعات تحتوى شرائح كبيرة من المستخدمين... وهذا سيقودنا لتضمين هذه الأجزاء من خلال المتصفح عن طريق الدمج بين مفهوم ال ESI وال CSI...

٣. ال Developer Experience: تعتبر تجربة سيئة إلى حد ما ^^ حيث أنها أقل سلاسة مقارنة بالتقنيات الأخرى.

والسؤال الآن، متى من المناسب استخدام هذه المعمارية؟
من أبرز حالات استخدام هذه المعمارية هي إدارة المواقع الكبيرة ذات ال static content، فمثلا شركة لديها الكثير من المنتجات والعديد من التفاصيل المتعلقة بها يمكنها بناء Catalog لها باستخدام هذه المعمارية، ويمكن الدمج بين ال ESI وال CSI لتغطية جميع الحالات... كما يتم استخدامها في حال الرغبة في عمل Cache لبعض الأجزاء ال static في الموقع الخاص بنا والحفاظ على بعض ال micro frontend ليتم تنفيذها كما هي... وهو ما يطلق عليه micro caching... لكن يبقى هذا الأسلوب غير شائع حتى مع هذه الحالات التي ذكرناها، ويمكن تغطية هذه الحالات من خلال الطرق الأخرى والتي قد تكون أفضل من العديد من الجوانب...

ال Architecture characteristics:

وكما تحدثنا سابقا، لكل معمارية أو نمط خصائص تميزه عن الآخر، واستخدام تقنيات مختلفة أو أسلوب ما قد يؤثر على الخصائص المعمارية التي تعمل عليها... والآن لنأخذ نظرة مع هذه الخصائص التي تتعلق بال Edge Side Composition ضمن معماريتنا:

١. ال Deployability (سهولة النشر) - العلامة **٣/٥** ^^:
عملية ال deploy هنا سهلة نوعا ما لأنه يعتمد على ال CDN، لكن ذلك يحتاج إلى زيادة الجهد المتعلق بإدارة الأخطاء المحتملة والتي قد تمنع تجميع ال micros المختلفة... كما أن مقدمي خدمات ال CDN لا يشتركون في نفس المستوى من الدعم لهذا الأسلوب.

٢. ال Modularity - العلامة **٤/٥** ^^:
من خلال هذه المعمارية وهذا الأسلوب يمكننا تحصيل مستوى جيد من تقسيم العمل لأجزاء صغيرة بسبب الاعتماد على ال Transclusion، وتصبح أكثر فاعلية مع دمج ال ESI وال CSI معا... ومع ذلك هناك تحديات لضمان أن التقسيم لم يأكل بطريقه المحتوى ال dynamic.

٣. ال Simplicity - العلامة **٢/٥** ^^:

معمارية ال horizontal تعتبر بذاتها معقدة، فكيف بنا إذا زدنا التعقيد من خلال ال edge side ^^... هناك تعقيد عند التطوير وعند الاختبار وعند حدوث تغييرات تحتاج إلى مرونة...

٤. ال Testability (قابلية الاختبار والفحص) - العلامة ٣/٥ ^^:
عملية الفحص والاختبار تحتاج لمجهود ووقت خصوصا إذا تحدثنا عن end to end هنا... أما اختبار ال micros نفسها فسيكون كما اعتدنا... المشكلة تكون بعملية ال test التي تحتاج للتكامل بين المكونات المختلفة في نفس الصفحة...

٥. ال Performance (الأداء) - العلامة ٣/٥ ^^:
يعتبر الأداء هنا جيد بسبب التجميع على مستوى ال CDN، كما أنه افتراضيا يقدم سرعة ممتازة للمحتوى الثابت بسبب انتشار ال edge وتوزعها... لكن في حال حدوث أي مشكلة في الشبكة عند سحب ال micros فإن الصفحات لن يتم عرضها حتى انتهاء الوقت المخصص لهذا ال request - حتى يفقع بوجهنا timeout ^^-... وإدارة هذه المشكلة بهذه الطريقة لا يعد الخيار الأفضل بالنسبة للمستخدمين...

٦. ال Developer Experience - العلامة ٢/٥ ^^:
تجربة التطوير سيئة ومعقدة... والتغذية الراجعة ومعالجة الأخطاء تحتاج لمجهود كبير...

٧. ال Scalability (القابلية للتوسع) - العلامة ٤/٥ ^^:

إذا كان المشروع الخاص بنا يحتوي على محتوى static، فإن ال ESI يعد واحد من أفضل الحلول التي يمكنك استخدامها، بفضل التجميع على مستوى CDN والتخلص من تعقد ال scalability ونقل هذا الجهد ليكون على مستوى ال CDN.

٨. ال Coordination (التنسيق) - العلامة ٣/٥ ^^:

يمكن أن تعمل الفرق بشكل مستقل وعلى ملفات مستقلة، وبذلك فإن التنسيق هنا جيد، لكن بسبب تجربة التطوير المعقدة سنحتاج لزيادة مستوى التنسيق بين الفرق وبناء آلية تضمن ذلك.

شاهد الصورة Table 4-6:

Table 4-6. Architecture characteristics summary for developing a micro-frontends architecture using horizontal split and edge-side composition

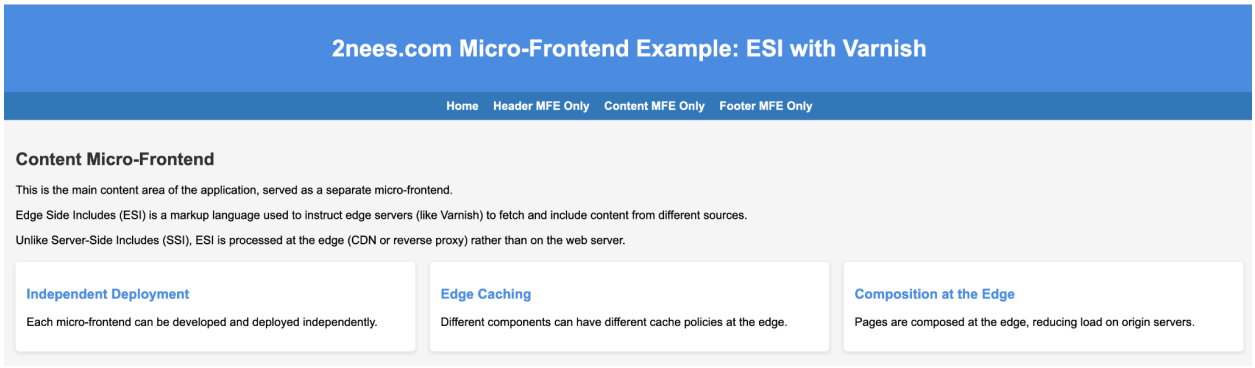
Architecture characteristics	Score (1 = lowest, 5 = highest)
Deployability	3/5
Modularity	4/5
Simplicity	2/5
Testability	3/5
Performance	3/5
Developer experience	2/5
Scalability	4/5
Coordination	3/5

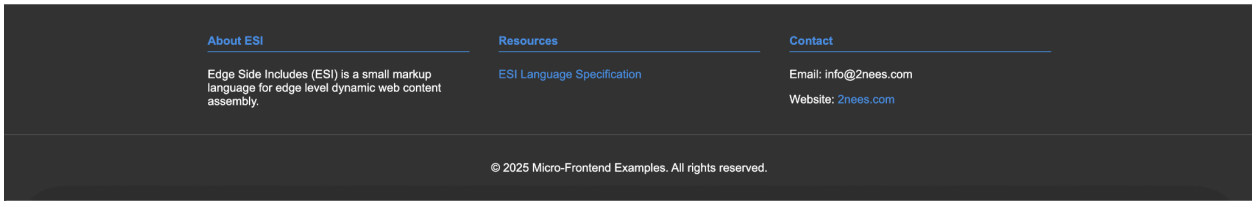
ملاحظة:

يمكن استخدام Varnish مع ال Nginx لتنفيذ عملية ال Caching لل Web Content. حيث يعمل Varnish كـ Reverse Proxy يدير طلبات ال HTTP ويسرع عملية تسليم المحتوى من خلال ال Caching. ويدعم ال Varnish ميزة ال ESI، والتي تتيح تقسيم الصفحة إلى أجزاء مستقلة بحيث يمكن عمل ال cache لكل جزء منها بشكل منفصل، لكن يشترط تفعيل خيار الدعم لل ESI في الإعدادات الخاصة به. ويوضع ال Varnish ك layer أمام ال Server لإدارة Request & Response بفعالية، مما يقلل من الضغط على Servers ويحسن من أداء الموقع... ويعتبر استخدام ال Varnish أحد الطرق البديلة لاستخدام وتفعيل ال ESI إذا لم تذهب لاستخدام ال CDN.

والآن شاهد مثالا عمليا على تطبيق ال ESI في هذه المعمارية:

```
sub vcl_backend_response {
    # Varnish received a response from the backend
    set beresp.ttl = 1h; # Cache for 1 hour by default

    # --- THIS IS THE KEY FOR ESI PROCESSING ---
    if (beresp.http.Content-Type ~ "(text/html|text/xml)") {
        set beresp.do_esi = true; # Instruct Varnish to process ESI includes
    }
    # ------------------------------------------

    return (deliver);
}
```





ويمكنك الدخول إلى هذا المثال من خلال هذا الرابط.

في هذا الفصل استعرضنا إطار اتخاذ قرارات لل Micro-Frontend وطبقنا ذلك على عدة أنماط معمارية مختلفة من خلال التركيز على الأعمدة الأربعة الأساسية: ال defining وال composing وال routing وال communicating، وتمكنا -بفضل الله- من تحليل كل خيار معماري بشكل منهجي ومقارنته وفقا لاحتياجات المشاريع المحتملة، كما قمنا بتقييم مزايا وتحديات كل بنية معمارية، ومنحناها درجات -تقييم- حتى نساعد في تسهيل اتخاذ القرار واختيار المعمارية الأنسب بناء على أولويات المشروع...كما أدركنا أن البنية المعمارية المثالية غير موجودة إلا في أحلامنا الوردية، وأن القرار الأفضل هو اختيار الحل الأنسب بناء على طبيعة المشروع والقيود المحيطة به، أي "أقل الحلول سوءا".

والحمد لله رب العالمين.

والآن سننتقل سننتقل من الجانب النظري إلى الجانب العملي، حيث سنستعرض تنفيذا تقنيا فعليا لل Micro-Frontend، مع التركيز على أبرز التحديات التي قد نواجهها أثناء التطبيق، أي أننا سننتقل إلى Micro-Frontend Technical Implementation ^^... نسأل الله تعالى التوفيق.

الفصل الرابع: Micro-Frontend Technical Implementation

بعد كل المقدمات التي ذكرناها، سنحاول معا بناء وتحليل مشروع بسيط بشكل واقعي بحيث تتمكن من عكس هذه التجربة على أي مشروع ستواجهه وتحديد متطلبات المشروع والخصائص المناسبة له والبنية المعمارية التي سنستخدمها لخدمة هذا المشروع... لذلك، اربط الأحزمة وهيا لننطلق سويا في هذه الرحلة الجميلة ^^.

لنفترض أننا مؤسسة كبيرة ونحتاج لبناء موقع تجارة إلكترونية خاص بهذه المؤسسة، لخدمة الموظفين الذين ينتمون لها.

بناء على هذه المعلومة لنفترض أن هذا الموقع مكون من عدة subdomains تشمل:

١. ال Login

٢. ال Payment

٣. ال Catalog

٤. ال Account Management

٥. ال Employee Support

٠٦. ال FAQ

فكرة مشروع مقترح للتنفيذ:

سيناريو مقترح لما سيدور في مؤسستك:

في مثالنا الحالي سنستخدم فقط ثلاثة من ال subdomains التي ذكرناها: وهي المتعلقة بال Authentication، والكتالوج، وإدارة الحساب (Account Management).

من القواعد المهمة التي يجب أن نراعيها في هذا المشروع: أن يتمتع هذا الموقع بواجهة مستخدم ذات نسق موحد، بحيث يحصل المستخدمون على تجربة تصفح سلسة ومنسجمة أثناء تسوقهم، وسيكون لدينا عدة فرق مسؤولة عن تنفيذ هذا المشروع، وللوصول إلى الموعد النهائي للمشروع قررنا إعادة استخدام نظام تم تطبيقه عمليا لحلول التجارة الإلكترونية الموجهة للمستهلكين (B2C)... هذا النظام عبارة عن Backend تم تصميمه بشكل Monolithic، وقد تم اختباره واستخدامه بشكل فعلي على ال Production ومنذ سنوات عديدة، مما يعطينا ثقة ممتازة به...

ومع ذلك، فإننا نرغب في الابتعاد عن فصل ال Experience الخاصة بمطوري ال Frontend و Backend، أي لا نريد أن يعمل FE لوحده وال BE لوحده، لذا سنقوم باعتماد الـ Micro-Frontends لتحقيق هذه الغاية، وسنقوم بتكوين فرق مستقلة لكل فريق منها مسؤولية محددة على subdomain معين. كما سيتم إشراك مطوري الـ Backend لإعادة

تصميم الخدمات باستخدام Microservices، من أجل إدخال المزيد من المرونة على المستوى التقني والتجاري لهذه الأجزاء المختارة.

لاحظ أن الفقرة السابقة فعليا خلاصة مجموعة من النقاشات والأحاديث التي يمكن أن نراها عادة في أي مؤسسة نعمل بها، ويمكن تلخيص ما ذكر أعلاه بما يلي:

١. لدينا ٣ subdomain أساسية للمشروع تم تحديدها للعمل في هذه المرحلة.

٢. هناك وقت محدد للتسليم وهو قريب نوعا ما.

٣. لدينا نظام قائم حاليا يخدم المستخدمين يمكننا نسخه واستخدامه للانطلاق من خلالها بدلا من البدء من الصفر في كل جزئية.

٤. نحتاج لتقسيم الفريق لعدة فرق بحيث نستطيع تسليم النظام بأسرع وقت ودون حدوث تعارض بين المكونات.

٥. نريد استغلال الخبرات الخاصة بالأفرقة المختلفة كل حسب الجزئية التي عمل عليها، واستغلال خبرات ال BE & FE.

٦. النظام الموجود مسبقا هو Monolithic ولأننا نحتاج لتقسيم واجهات الفرونت اند واستغلال الأجزاء الموجودة التي يمكن دمجها مسبقا واستغلال خبرات الفرق وخبرات المطورين ونحو ذلك، قمنا باستخدام ال micro frontend مع تحويل بعض الخدمات من ال Monolithic لتصبح micro service.

بعد هذه المقدمة، ستكون الخطوة التالية هي تقسيم الفرق، وستكون على الشكل التالي:

١. فريق علي: سيكون مسؤولا عن إدارة نظام ال Authentication، هذا الفريق سيكون مسؤولا عن بناء ال FE الخاصة بتسجيل الدخول، وإدارة عملية تسجيل الدخول والتحقق من ذلك، وسيكون هناك موظف واحد Full Stack Developer للقيام في هذه المهمة، فيما يقوم باقي الأعضاء بالعمل على التكامل ما بين ما يقوم به هذا الموظف وبين ال SSO الخاص بالشركة، كما ستكون من مسؤوليات هذا الفريق عرض معلومات المستخدمين داخل الصفحة المتعلقة بتفاصيل الحساب (ال Account details داخل ال Account Management).

٢. فريق سامح: سيقوم بالعمل على ال Core Domain وهو ال Catalog الخاص بالمنتجات التي يرغب الموظفين بالحصول عليها، وسيتولى مسؤولية تصميم وبناء ال UX الخاصة بهذا الجزء، وذلك يشمل خدمات مثل عرض المنتجات والبحث والتصفح... وسيتم تقسيم موظفي الفريق هنا FE للتعامل مع الواجهات وال BE للتعامل الخدمات والبيانات.

٣. فريق مهدي: وهو المسؤول عن ال Account Management، بما في ذلك إدارة نظام الدفع، وهذا يشمل إدارة طرق الدفع المختلفة من خلال ربطها مع مقدمي خدمات الدفع.

بناء على هذه التفاصيل هناك أكثر من خيار يمكن الذهاب إليه، لكن وجدنا أن خيار التقسيم المناسب يكون كما يلي:

سيتم بناء Application Shell فيه ال Header وال Navigation باعتبارها وحدات مشتركة بين جميع الأقسام، وسيحتوي ال Navigation على ال Global Routes، وسيتم اعتماد ال Vertical Split لكل من صفحات ال SignIn وصفحات ال Catalog، أما ال Account Management فسيتم اعتماد ال Horizontal Split فيها وسنأخذ ال Account details من خلال فريق علي، وال Payment Details من فريق مهدي.

شاهد الصورة التقريبية لما نرغب ببنائه F5-1:

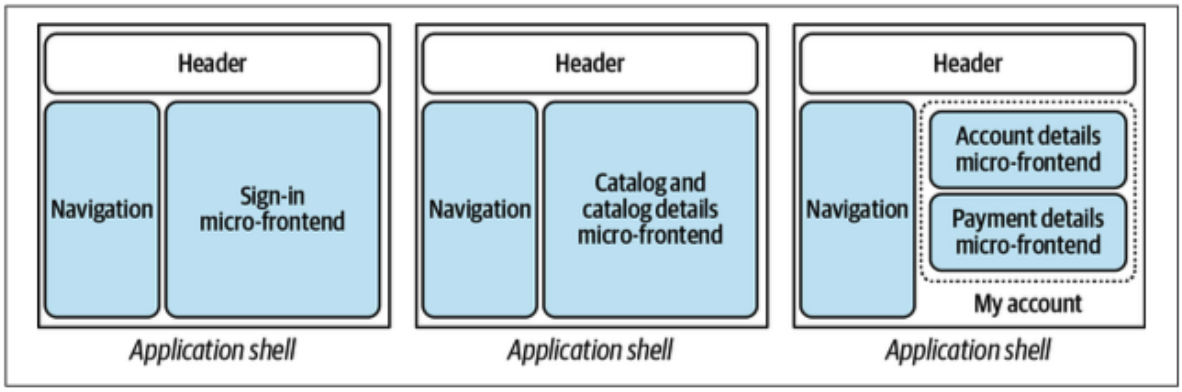


Figure 5-1. The swag ecommerce sections: sign-in, catalog with product details, and account management

أما آلية توزيع الموظفين فكان موظف واحد من فريق علي (ال Auth) كافي لبناء ال FE الخاص بهذه الصفحات، فهي لا تحتاج لجهد كبير، أما الجهد هنا فهو يقع على عاتق فريق ال BE، لذلك كان يكفي أن يعمل مطور واحد على هذه الجزئية.

أما فريق سامح (ال Catalog) فهذا الفريق مهم جدا ويقع عليه أكبر مجهود، لذلك يجب أن يكون الفريق الأكثر خبرة خصوصا في Frontend، لأننا سنتعامل مع Rich UX.

أما فريق مهدي (ال Account Management) فإنه يمثل تقاطع بين أكثر من subdomain، وهما ال Account Details وال Payment details، فسيكون هناك حاجة للتعاون بين فريق مهدي وفريق علي لاستمرار عملية التطوير بشكل فعال.

أما لماذا ذهبنا في هذا الطريق، فذلك لأننا اعتمدنا على ما تعلمناه سابقا في ال micro-frontend decisions framework ^^...

حيث قمنا باعتماد نظام Hybrid يتكون من Vertical و Horizontal حسب الصفحات بدلا من اعتماد نظام واحد، وذلك بسبب الحاجة لوجود صفحة في أقسام مشتركة، فبدلا من الذهاب ليكون النظام كاملا horizontal. نكتفي بالصفحة تلك ^^، وهذه أول حركة جميلة هنا، قلنا سابقا أن اتخاذ القرار المناسب يتبع طبيعة المشروع وقيوده والخبرات الموجودة ونحو ذلك، وقلنا أنه لا يوجد حل مثالي واحد مرسوم لكل مشروع... وهذا ما ظهر هنا، فلاحظ أننا تعلمنا سابقا استخدام نوع واحد فقط مع الإشارة لإمكانية الدمج بينهم... وهنا قمنا بالدمج ^^.

وقمنا أيضا باعتماد ال Client Side Composition، لأن هذا النظام سيكون نظاما داخليا وسهل للتوسع عند الحاجة ومناسب لخبرات المطورين الذين قاموا بتطوير المشاريع سابقا بنفس الطريقة، مع سهولة ربطه وإدارة ال routing...

وقمنا أيضا باعتماد ال Web Storage كوسيلة لتخزين ال JWT وال Event Emitter للتعامل مع الأحداث والتواصل بين ال micros.

وتم اعتماد ال WebPack مع ال Module Federation لتحقيق هذه الغاية، لأنها أداة ممتازة، سهلة، والمجتمع الخاص بها كبير، وخبرة الفريق مع ال WebPack قديمة... وبذلك لن يحتاج الفريق لتعلم شيء جديد، كما أنه يعتبر خيارا مثاليا لل Client Side Composition، ولو قمنا بتغيير قرارنا لاحقا وانتقلنا من ال Client Side إلى ال Server Side فلن تكون هذه مشكلة لدينا ^^... (تذكر أن هذا بناء على مفاضلة وليست هي الخيار الأفضل دائما).

والآن قبل المتابعة، لدي سؤال جميل لك ^^، لقد ذكرنا "سيتم بناء Application Shell فيه ال Header وال Navigation باعتبارها وحدات مشتركة بين جميع الأقسام، وسيحتوي ال Navigation على ال Global Routes"... ما رأيك بهذا الأسلوب؟ كيف يمكننا تحقيق استقلالية في النشر ودون الحاجة لنشر ال App Shell مجددا حتى لو قمنا بإصدار نسخة جديدة من ال header مثلا؟ ما هي الطرق الممكنة أو الأساليب التي يمكنك اتخاذها أو

التفكير فيها كحلول بديلة لو كنت بحاجة لذلك؟ أرنا الآن مهاراتك ومتابعتك لما ذكرناه سابقا ^^.

والآن، بعد هذه المقدمة، لنبدأ ببناء المشروع، وسنقوم باستخدام ال React لهذا الغرض، علما أننا سنقوم ببناءه بأبسط طريقة ممكنة حتى يكون واضحا قدر الإمكان، ولن نقوم ببناء BE لأن الغاية هنا هي تصميم ال micro frontend، كما أن هذا التبسيط قد يخل ببعض القواعد المتعلقة بالأمان، فيرجى مراعاة ذلك.

ال Project Structure:

أول خطوة لدينا هي تحديد ال Project Structure ولدينا خيارين أساسيين هما:

١. ال Monorepo: ببساطة هي Repo أو مكان يحتوي بداخله جميع المشاريع والخدمات التي نقدمها، أي أنها جميعا متواجدة في نفس المكان، وهناك أدوات كثيرة تساعد المطورين إذا ذهبوا في هذا الاختيار، وأرى أن ال NX tool من أجمل هذه الأدوات، وسنقوم باعتماد ال Monorepo لهذا المشروع، ولن نستخدم ال NX فيه.

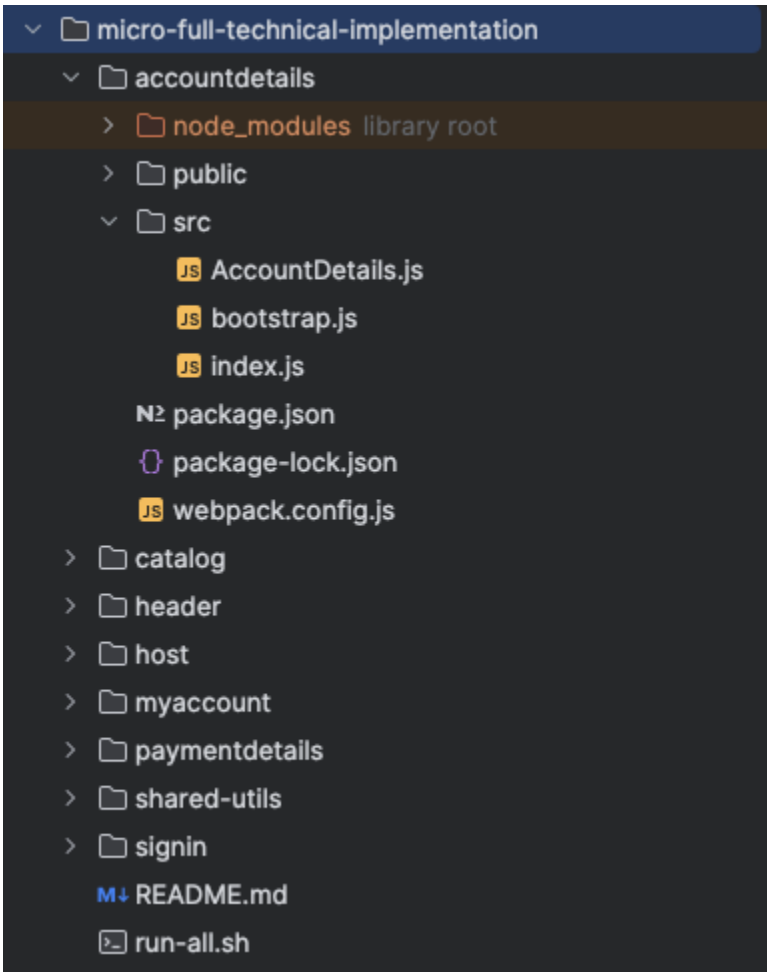
٢. ال Polyrepo: هي النموذج التقليدي الذي اعتدنا عليه، بحيث يكون كل مشروع أو خدمة في repo مستقلة.

ولعلنا سنتحدث عن هذه الاستراتيجيات وتأثيرها على سير العمل وكيف يمكن التحكم بها لاحقا في الفصول القادمة بإذن الله.

إذا، اعتمدنا ال Monorepo كنموذج لمشروعنا، وعليه يمكننا تخيل ما يلي:

سيكون لدينا AppShell و SignIn و Catalog و Account Details و Payment و MyAccount (المجلد الذي يحتوي الـ Account Details وال Payment Details).

شاهد النتيجة النهائية التي سنصل إليها عند الانتهاء من هذا الفصل:



فائدة

من أسباب الضلال: رد الحق بمعايير نظر خاطئة -والعياذ بالله-، لذلك كان ضبط الأفهام على معيار الوحي وتصحيح النبي -صلى الله عليه وسلم- لمقاييس النظر من أهم ما جاء به الوحي من سبل الهداية.

- مما تعلمته من كتاب القبس الوهاج، أحمد بن يوسف السيد

تصميم وبناء ال APP SHELL:

وسنبدأ في أول جزء لدينا وهو: ال AppShell، ولقد تحدثنا عنه سابقا بما يكفي... والآن الجزء المهم هنا هو أن تطوير ال AppShell سنقوم بتوكيل تطويره إلى فريق يوسف، وهو فريق لديه خبرة عميقة في مبادئ تصميم الأنظمة، لأن هذا الجزء بالتحديد يجب أن يكون بأبسط طريقة ممكنة كما ذكرنا سابقا، ويجب ألا يحتوي على Business Domain محدد له! كما أن العمل هنا لن يتعارض مع عمل المطورين أثناء عملهم بالفرق الأخرى لأنها لا تتطلب جهدا كبيرا أو صيانة معقدة... لكن من أهم ما يجب أن يراعيه المطورين هنا:

١. ال Domain Leak: يجب ضمان ألا يحدث أي تسرب لأي business code داخل ال App Shell، فيجب أن يبقى دوما عاما ومحايدا!

٢. ال Global Routing: يتم إدارة التنقل بين ال micro frontend من خلال ال App Shell، لذلك عليهم أن يحرصوا على ربط ال micro بروابطها الصحيحة.
من الذاكرة: سنقوم في مثالنا الحالي بوضع الروابط مباشرة داخل ال Navigation لي ال App Shell، ما هي الطرق الأخرى التي يمكننا استخدامها بحيث نتمكن من إدارة الروابط الحالية أو تعطيلها دون الحاجة لإعادة نشر ال App Shell؟ ^^... أرأيت مجددا أهمية ما كنا نتكلم عنه سابقا ^^.

٣. التأكد من أن عملية ال mounted وال unmounted تمت بشكل صحيح لل micro frontend.

٤. ضمان أن ال Shared Dependencies تم تجميعها من ال micros المختلفة في ملفات js مشتركة لتحسين الأداء وتقليل التكرار... وهنا يظهر أيضا ميزة من أجمل مزايا ال module federation، هل تتذكرها؟

٥. التحقق من الأداء.

٦. تنظيم اجتماعات مع باقي الفرق لمشاركة الممارسات والسلوكيات النموذجية أثناء العمل.

٧. إذا كان هناك bottlenecks فيقع على عاتقهم تحليل ذلك وتحسين الأداء عند الحاجة...

والآن شاهد الصورة التالية، فمن خلالها استطعنا إنشاء chunks لل shared library فيها ال react ستكون مركزية وذات إصدار واحد مستخدم داخل جميع ال micro frontend، كما يمكننا التحكم بالعديد من الخصائص الأخرى منها تثبيت أرقام الإصدارات المطلوبة.

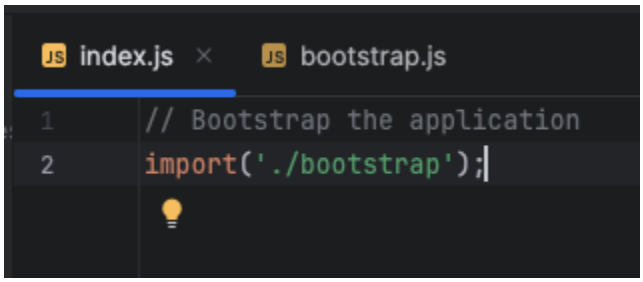
```
shared: {
  "react": {
    singleton: true,
  },
  "react-dom": {
    singleton: true,
  },
  "@mui/material": {
    singleton: true,
  },
  "@mui/icons-material": {
    singleton: true,
  },
  "@emotion/react": {
    singleton: true,
  },
  "@emotion/styled": {
    singleton: true,
  },
  "react-router-dom": {
    singleton: true,
  },
},
```

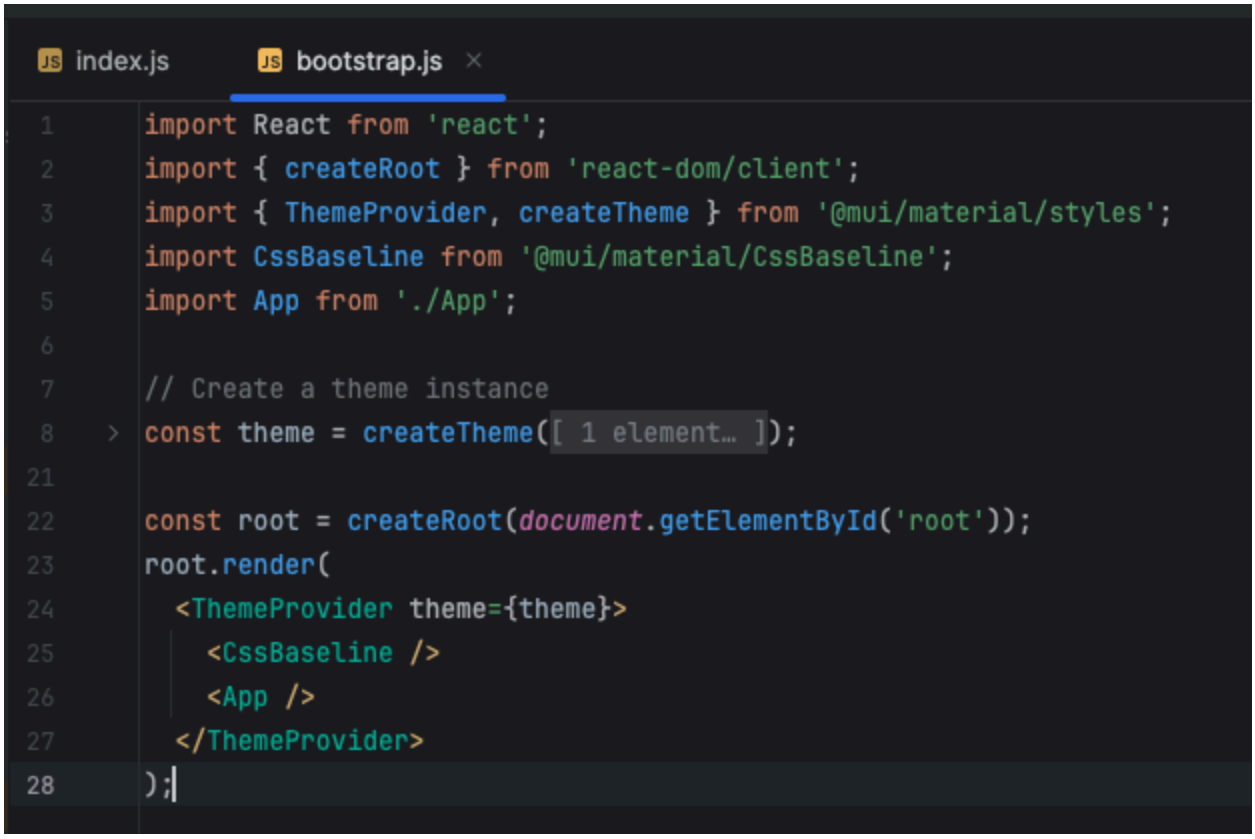
كما أن من النقاط المهمة هي تحديد ال Remotes URL كما ذكرنا سابقا... على سبيل المثال:

```
plugins: [
  new HtmlWebpackPlugin({
    template: "./public/index.html",
  }),
  new ModuleFederationPlugin({
    name: "host",
    remotes: {
      header: "header@http://localhost:8101/remoteEntry.js",
      signin: "signin@http://localhost:8102/remoteEntry.js",
```

والآن سنتحدث عن نقطة مهمة جدا لم نتحدث عنها سابقا... عند استخدام ال Module Federation نقوم بإنشاء ملف bootstrap، هذا الملف يتم عمل import له داخل ال index، وال index فقط فيها سطر ال import لل bootstrap، والسبب في ذلك يعود أننا نحتاج لتأخير عملية تحميل الكود الأساسي حتى تكون جميع المتطلبات جاهزة، فال webpack لا يستطيع معرفة ال shared module حتى يتم تحميلها، وهي من الممارسات الموصى بها... وطبعا هناك طرق أخرى مثل استخدام ال eager والتي تضمن تحميل جميع ال shared library قبل ال application code، لكن إذا قمنا بذلك فإننا سنقوم بتحميل ال dependency على شكل synchronously، في حين أن تحميلها بالشكل الذي تحدثنا عنه سيكون asynchronously، مما يعني أننا لسنا بحاجة لتحميل جميع المتطلبات من البداية، ويتم تقسيم الملفات ل chunks صغيرة... في حين أنها مع ال eager سيكون لدينا bundle كبيرة... وهذا سيظهر أثره على الأداء مثل تأخر زمن الـ TTFB ^^.

شاهد الصورتان التاليتان:





والآن بعد أن قمنا بكتابة الإعدادات الخاصة بال App Shell، سنذهب لإضافة ال Routing، ولأجل تحقيق هذه الغاية في react هناك العديد من الطرق، سأستخدم طريقتي المفضلة وذلك من خلال ال react-router-dom...

من خلال ال react-router-dom يمكننا إعادة توجيه المستخدم لل micro التي يحتاجها، ويمكن أيضا استخدامها في ال micros الأخرى لإعادة التوجيه لل local routing، لهذا ستجد أننا قمنا بوضعها بال shared ^^.

وحتى نقوم بإعادة توجيه المستخدم لل micro المطلوبة فهناك خطوتين بسيطتين وهما: تضمين ال Micro Frontend التي نحتاجها وربطها بالرابط...

شاهد الصورتين التاليتين:

```
// Lazy load remote
const Header = React.lazy(() => import('header/Header'));
const Signin = React.lazy(() => import('signin/Signin'));
```

```
const App = () => {
  return (
    <Router>
      <Box sx={{ display: 'flex', flexDirection: 'column', minHeight: '100vh' }}>
        {/* Header */}
        <Suspense fallback={<CircularProgress />}>
          <Header />
        </Suspense>

        {/* Main content */}
        <Container component="main" sx={{ flex: 1, py: 3 }}>
          <Routes>
            <Route path="/signin" element={
              <Suspense fallback={<CircularProgress />}>
                <Signin />
              </Suspense>
            } />
          </Routes>
        </Container>
      </Box>
    </Router>
  );
};
```

من الأمور الجميلة في React وجود ال Suspense والتي تمثل component يسمح لنا بعرض Fallback UI أثناء وقت تحميل ال Lazy Component مثل ال micro التي لدينا

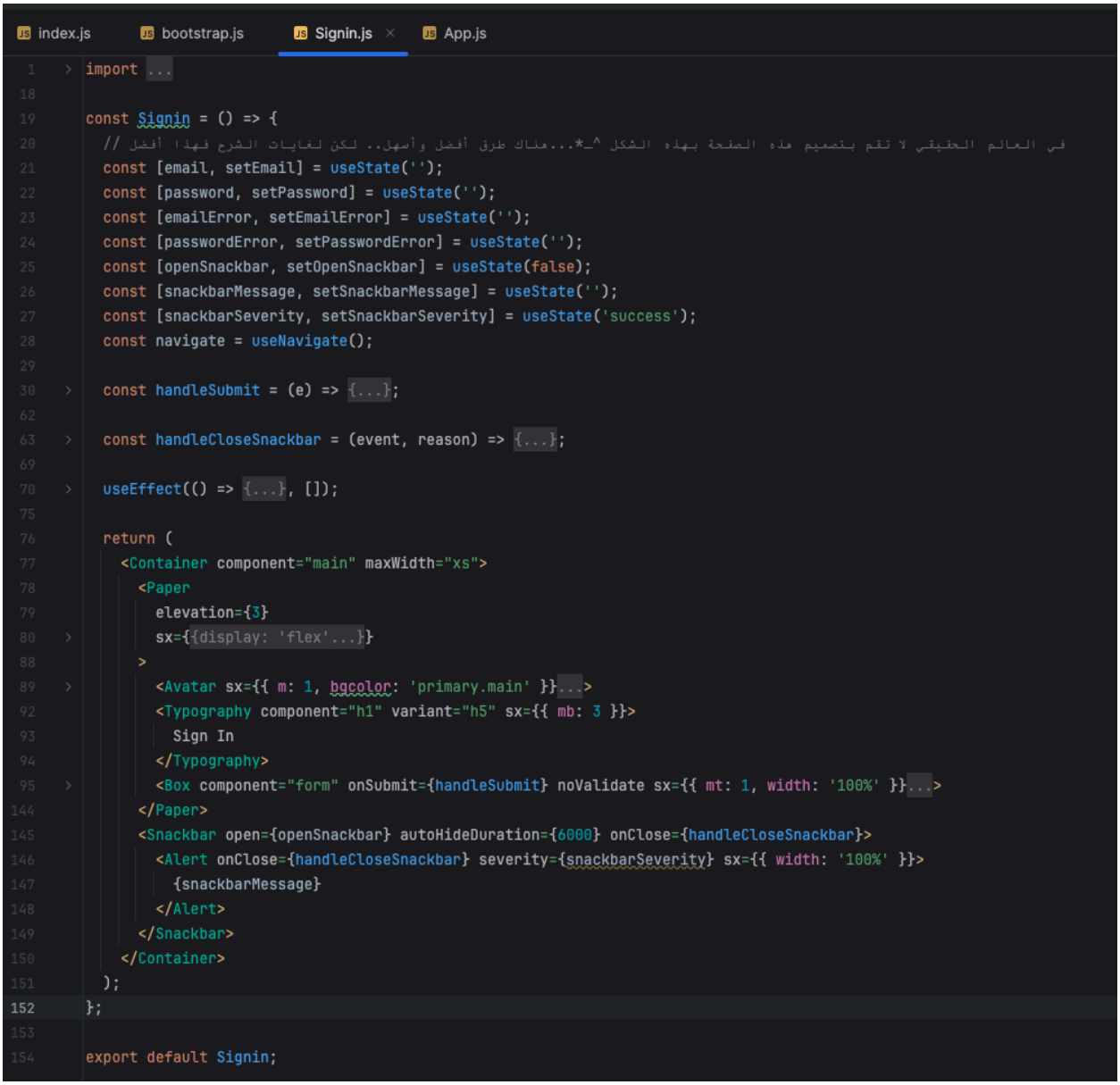
أو أثناء انتظار البيانات... فبدلا من أن تظهر صفحة فارغة سيظهر لدينا حسب الصورة أعلاه Loader ^^.

أما فيما يتعلق بمشاكل ال css المحتملة ونحو ذلك فلا خوف منها لأننا استخدمنا ال MUI style وسنعتمد على تنسيق موحد، كما أن وجود sx والتي تمثل inline style ستحمينا أيضا من المشاكل الخاصة بالتعارض... ومع ذلك، يمكننا لو أردنا تخصيص ال classes ووضع prefix بكل سهولة.

ال Authentication Micro-Frontend:

لا يوجد مسؤولية كبيرة تقع على عاتق ال FE في هذا الجزء من النظام... الشغل الحقيقي هنا عند ال BE، لذلك سيكون من السهولة تصميم وإدارة هذه الصفحة... لذلك ستكون أول مهمة لدينا هي بناء البنية التحتية لل micro وإنشاء صفحة ال Signin.

شاهد الصورة أدناه:



أما فيما يخص ال webpack config فكما تعلمنا سابقا:

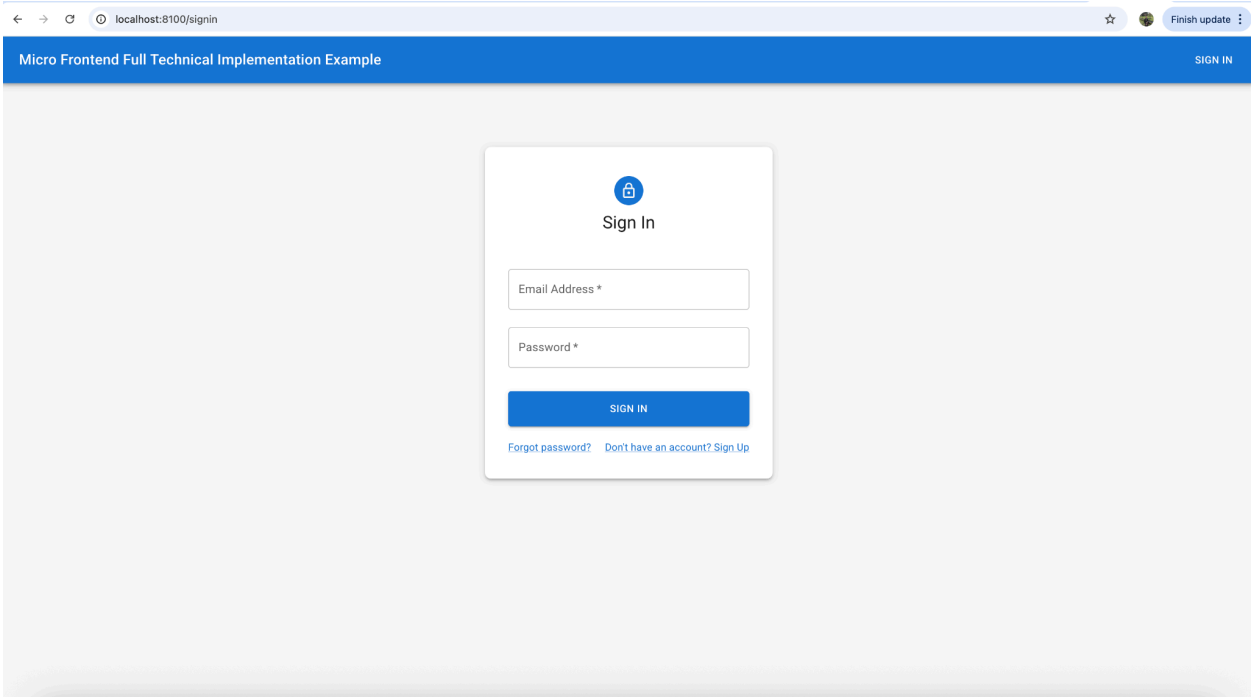
بناء على المثال الافتراضي الذي قمنا بتصميمه سنقوم بتخزين ال token داخل ال localStorage (لغايات تبسيط المثال فقط...، والتوكن هنا فقط true ^^).

طبعا يوجد لدينا هنا ارتباط بين نتائج ما نحصل عليه من بيانات في هذه الجزئية وبين ال micros الأخرى المختلفة... فمثلا لا ينبغي تحميل صفحة ال catalog لمستخدم لم يقم بتسجيل الدخول أساسا! ولو افترضنا وجود نظام صلاحيات فلا يفترض أن يشاهد المستخدم العادي صفحة المدير ونحو ذلك! وهذا يرتبط ارتباطا وثيقا بكيفية إدارتنا لهذه النقاط...

عادة ما يكون التحقق على مستوى كل micro في حالة ال Vertical Split، لأنه يمكن الوصول إليها من رابط خارجي وليس شرطا من ال app shell، في حين عادة ما يكون التحقق من ال app shell في حالة ال horizontal، ومع ذلك يمكننا ضمان التحقق من خلال ال app shell وال micro معا... وكل ذلك حسب طبيعة المشروع والمعمارية التي تم اعتمادها والمعايير التي تم رسمها.

لغايات المثال، سنقوم بالتحقق فقط على مستوى ال App Shell.

شاهد نتيجة تركيب أول صفحة:



ال Header Micro-Frontend:

لقد قلنا أن هناك العديد من الطرق للتعامل مع العناصر التي مثل ال Header، وهذا المثال سنقوم باعتبار أن ال Header هي micro مستقلة بذاتها وليست مجرد عنصر ثابت داخل ال app shell، وحتى نعطي مزيدا من الحماس، هذه ال header ستقوم بنقلنا بين ال

micros المختلفة، كما أنها ستتحدث تلقائيا من خلال ال event عند تسجيل الدخول ودون تحميل الصفحة... لكن هذه ال micro لا يوجد فريق مسؤول عنها، فماذا سنفعل؟

الجواب في البحث عن الفريق المناسب من ضمن الأفرقة التي تم تخصيصها للعمل على المشروع ليعمل عليها إو استحداث فريق لأجلها... وهنا سنقوم بإعطاء هذه المسؤولية لفريق علي باعتبار أن العمل من ناحية ال FE ليس صعبا هناك، مما يتيح لهذا الفريق العمل هنا أيضا بشكل مستقل، كما أن هناك ترابط بين هذان المكونان...

بناء على المعلومات السابقة، سنحتاج إلى ما يلي:

١. بناء أول shared utils وستحتوي بداخلها ال eventBus الذي يجب أن يكون ضمن ال shared الخاصة بالمشروع، ويتم تحميل نسخة واحدة منه فقط!

```
// 2ness.com => هذا مثال توضيحي، لكن هناك العديد من الإعدادات المهمة والأمنية التي يجب أن تراعيها أو يمكنك استخدام مكتبة جاهزة !
class EventBus {
  constructor() {
    this.events = {};
  }

  on(eventName, listener) {...}

  emit(eventName, data) {...}

  off(eventName, listenerToRemove) {...}
}

const eventBus = new EventBus();

export default eventBus;
```

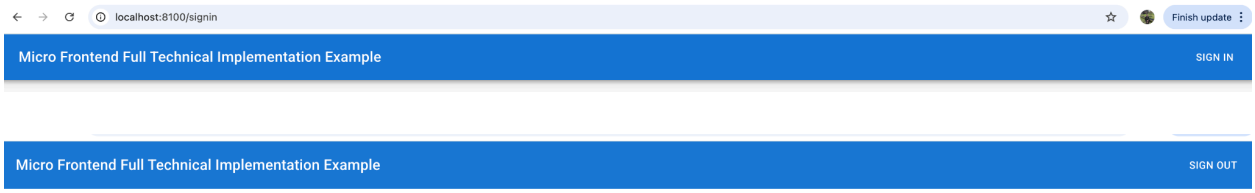
2. إطلاق event عند تسجيل الدخول في صفحة ال signin لإعلام ال micro المختلفة بهذا الخبر.

```
// Emit login event through eventBus
eventBus.emit('user:login', { email });
```

٣. بناء ال header وإضافة ال eventlistener عند حدوث تسجيل دخول أو خروج إلى header micro.

```
useEffect(() => {
  // Listen for login events
  const loginHandler = () => {
    setIsLoggedIn(true);
  };

  // Listen for logout events
  const logoutHandler = () => {
    setIsLoggedIn(false);
  };

  // Subscribe to events
  eventBus.on('user:login', loginHandler);
  eventBus.on('user:logout', logoutHandler);

  return () => {
    eventBus.off('user:login', loginHandler);
    eventBus.off('user:logout', logoutHandler);
  };
}, []);

const handleLogout = () => {
  localStorage.setItem('isLoggedIn', "false");
  eventBus.emit('user:logout');
};
```

الصورة النهائية التي سنخرج بها حتى هذه اللحظة:



ملاحظة: هل تتذكر ما تحدثنا عنه حول الأفكار أو الآليات المناسبة للتعامل مع العناصر التي تتعلق بالإعدادات أو الروابط وآلية إعدادها والأفكار الممكنة لهذا الغرض؟ ما علاقة هذا بال header هنا؟ وكيف يمكننا استخدام ما ذكرناه سابقا لإضافة الروابط دون عمل deploy؟ هل عرفت قيمة ما تعلمناه حتى هذه اللحظة ^^.

ال Catalog Micro-Frontend:

على نفس النمط الذي قمنا بتطبيقه بال micros السابقة سنقوم بذلك هنا، لكن لدينا مميزة إضافية هنا، أن هذه ال micros تمثل SPA ومعماريتها Vertical فيها local route، وهذا يعني حاجتنا لإدارة ال Global Route للوصول إلى هذه ال micro، ثم ستقوم ال micro بإدارة الروابط المتعلقة بها مع ضمان الحفاظ على عملها بشكل مستقل كموقع مستقل، وضمان عملها مع ال project الخاص بنا ك micro.

والعملية سهلة، فنحن اعتمدنا ال client side composition...

هذه ال micro تحتوي بداخلها:

١. ال Main Component والتي تمثل العنصر الأساسي الذي سيدير عملية ال routing داخل ال micro، وسيكون لدينا هنا صفحتين، صفحة قائمة المنتجات وصفحة تفاصيل كل منتج.

```
const Catalog = () => {
  return (
    <Container component="main" sx={{ py: 3 }}>
      <Routes>
        <Route index element={<ProductList />} />
        <Route path="list" element={<ProductList />} />
        <Route path="details/:id" element={<ProductDetails />} />
        <Route path="*" element={<Box>404</Box>} />
      </Routes>
    </Container>
  );
};
```

٢. صفحة قائمة المنتجات: لا يوجد فيها شيء مميز، لكن سنقوم بوضع زر لإعادة التوجيه لصفحة التفاصيل الخاصة بالمنتج.

```
<Button
  size="small"
  component={RouterLink}// import {Link as RouterLink} from 'react-router-dom';
  to={`/details/${product.id}`}
>
  View Details
</Button>
```

٣. صفحة تفاصيل المنتج: أيضا لا يوجد شيء مميز سوى أننا سنقوم بإضافة زر لإعادة التوجيه لصفحة قائمة المنتجات.

```
<Button
  size="small"
  component={RouterLink}// import {Link as RouterLink} from 'react-router-dom';
  to={`/details/${product.id}`}
>
  View Details
</Button>
```

٤. سنقوم بإضافة رابطة هذه ال Micro إلى ال header وسيكون الرابط الخاص بها هو: "catalog/".

الآن بعد انتهينا من هذه، لو قمنا بتطبيق المثال سنجد أن هناك مشكلة خطيرة، وهي أن الروابط تعمل بشكل ممتاز عندما يعمل المشروع بشكل مستقل، لكن لن تعمل داخل ال app shell الخاص بنا... والسبب في ذلك يعود إلى أن ال Global route سيكون مثلا 2nees.com/catalog في حين أن ال micro عندما تعمل بشكل مستقل سيكون ال base path لها هو "/" مباشرة... وللتعامل مع هذه المشكلة هناك العديد من الطرق، وتختلف حسب طبيعة هل هذه ال micro ستكون فقط ك micro داخل مشروع أم أنها عبارة عن مشروع مستقل يعمل وحده ويعمل داخل ال micro... ولتبسيط العمل ولأننا نستخدم ال react router، فسنقوم بعمل wrapper لل Link الخاص بال react router لتصبح النتيجة كما يلي:

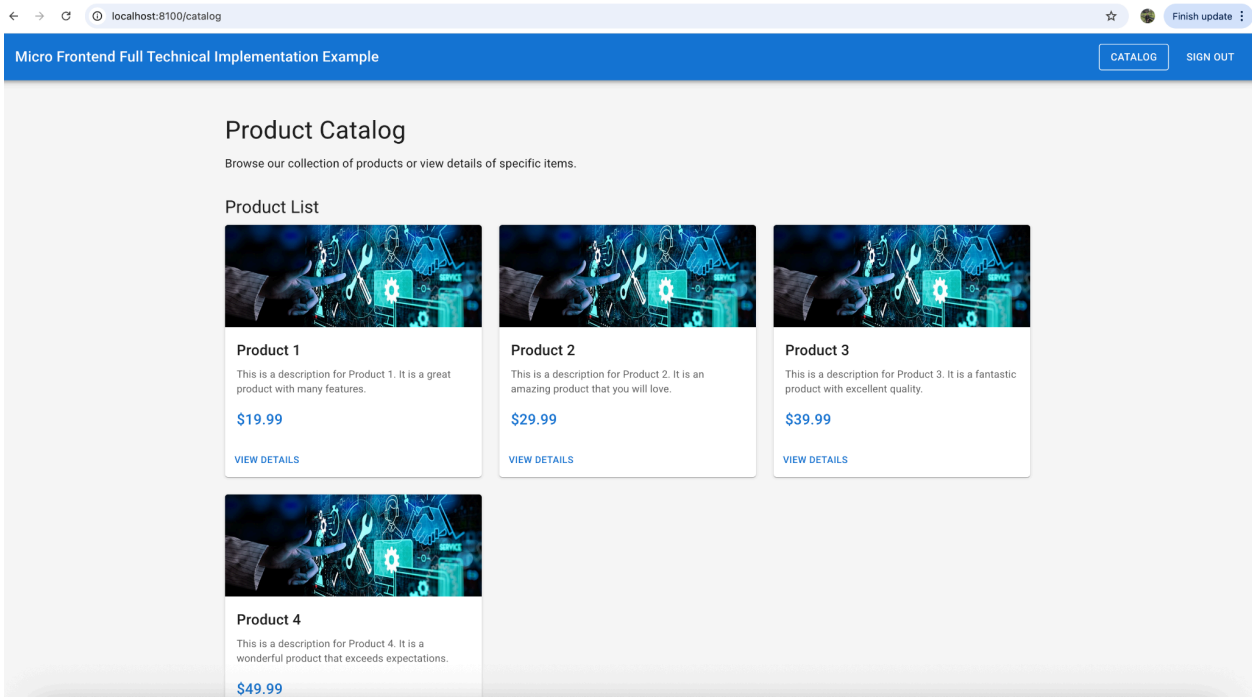
```jsx
import {Link as RouterLink, useLocation} from 'react-router-dom';
import React, {useMemo} from "react";

const ForExampleLink = React.forwardRef((props, ref) => {
  const location = useLocation();

  const toUrl = useMemo(() => {
    if (location.pathname.startsWith('/catalog')) {// static بشكل path لغايات المثال قمنا بوضع ال
      return `/catalog${props.to}`;
    }

    return props.to;
  }, [location.pathname]);

  return <RouterLink ref={ref} {...props} to={toUrl} />;
});

export default ForExampleLink;
```

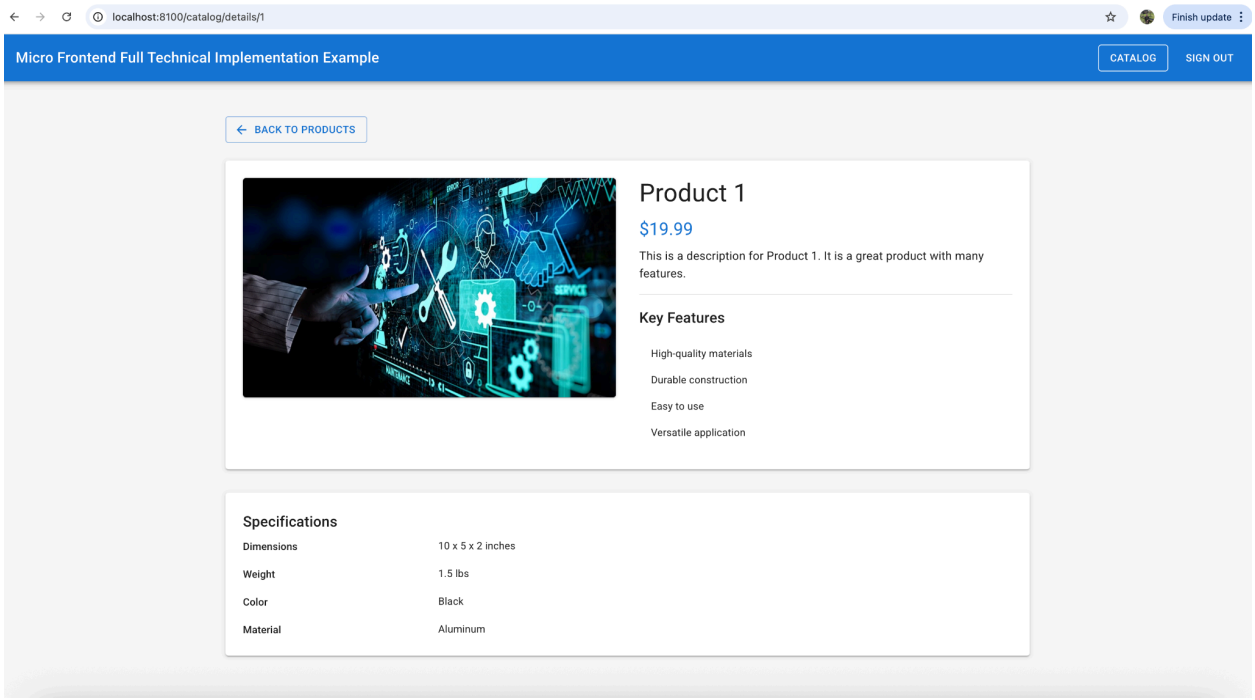
```jsx
<Button
  size="small"
  component={ForExampleLink}
  to={`/details/${product.id}`}
>
  View Details
</Button>
```

```jsx
const BackLink = () => <Button
  variant="outlined"
  component={ForExampleLink}
  to="/list"
  startIcon={<ArrowBackIcon />}
  sx={{ mb: 3 }}
>
  Back to Products
</Button>;
```

ملاحظة: في الإصدارات القديمة من react router كان يتم الاعتماد على useRouteMatch ومن ثم أخذ ال path ووضعه ك prefix... في الإصدارات الجديدة يكفي استخدام ال basename للتعامل مع هذه المشكلة، وهو حل ممتاز إذا كانت ستعمل ك micro دائما...

بناء على ذلك ستكون النتيجة النهائية كما يلي:





كما أنها تعمل بشكل مستقل ^^:

localhost:8103

Product Catalog

Browse our collection of products or view details of specific items.

Product List

Product 1

This is a description for Product 1. It is a great product with many features.

$19.99

VIEW DETAILS

Product 2

This is a description for Product 2. It is an amazing product that you will love.

$29.99

VIEW DETAILS

Product 3

This is a description for Product 3. It is a fantastic product with excellent quality.

$39.99

VIEW DETAILS

Product 4

This is a description for Product 4. It is a wonderful product that exceeds expectations.

$49.99

VIEW DETAILS

localhost:8103/details/1

BACK TO PRODUCTS

Product 1

$19.99

This is a description for Product 1. It is a great product with many features.

Key Features

- High-quality materials
- Durable construction
- Easy to use
- Versatile application

Specifications

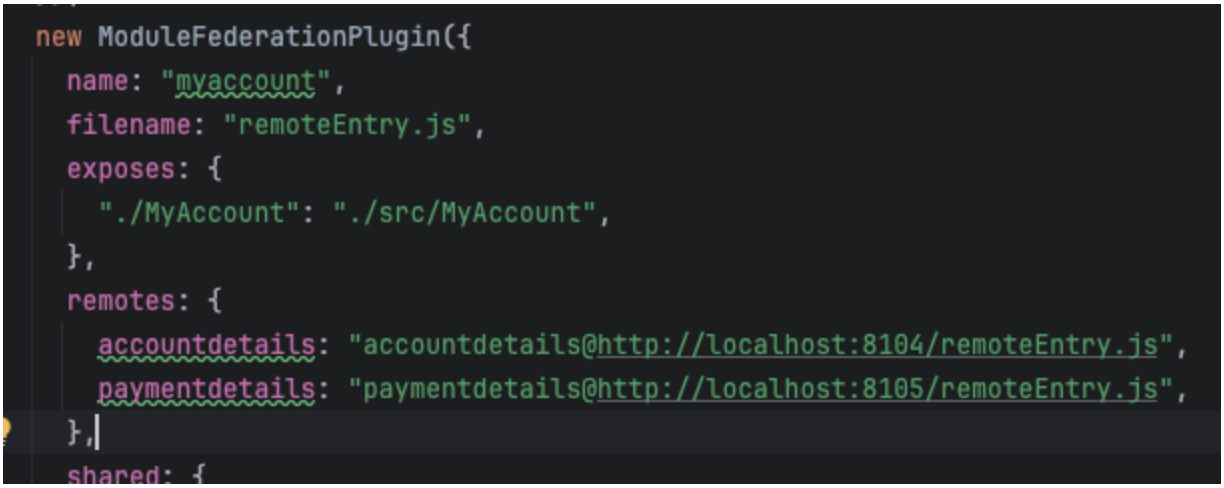
Dimensions	10 x 5 x 2 inches
Weight	1.5 lbs
Color	Black
Material	Aluminum

ال Account Management Micro-Frontend:

وصلنا الآن لآخر micro سنقوم بتصميمها، هذه ال micro تتميز بميزة مهمة، وهي أنها تحتوي أكثر من micro frontend بداخلها، وهذا جعلنا نتجه لاستخدام ال horizontal split كمعمارية للتعامل معها، لكن كيف سيتم ذلك ونحن نتعامل أيضا مع تقسيم Vertical في باقي الصفحات؟!

الجمال يأتي هنا، من خلال ال Webpack federation يمكننا عمل expose لل micro وبنفس الوقت يمكننا جعلها هي نفسها remote ل micros أخرى ^^... هذا الجمال الذي تحدثنا عنه سابقا، والذي بدوره قد يقودنا لنقطة خطر كبيرة وخطيرة تحدثنا عنها سابقا وهي ال bidirectionally، لذلك يجب أن نحافظ قدر الإمكان على اتجاه واضح مسبقا... فمثلا الاتجاه عندنا في هذا المثال واضح من ال app shell إلى ال micro، ولا يوجد اعتمادية ليكون سير العمل من ال micro إلى app ومن ال App إلى ال micro... في بعض الأحيان يمكن تجاوز هذه القاعدة لتحقيق بعض الأهداف التي لا غنى عن تحقيقها سوى من خلال هذه الطريقة، لكن يجب عليك أن تحرص على الحد منها...

إذا بناء على ما ذكرناه، سيكون الشكل الجميل لل Webpack كما يلي:



لاحظ أن ال Myaccount ستكون هي remote لل app shell، وبنفس الوقت هي host لل account details وال payment details ^^.

خطوات العمل:

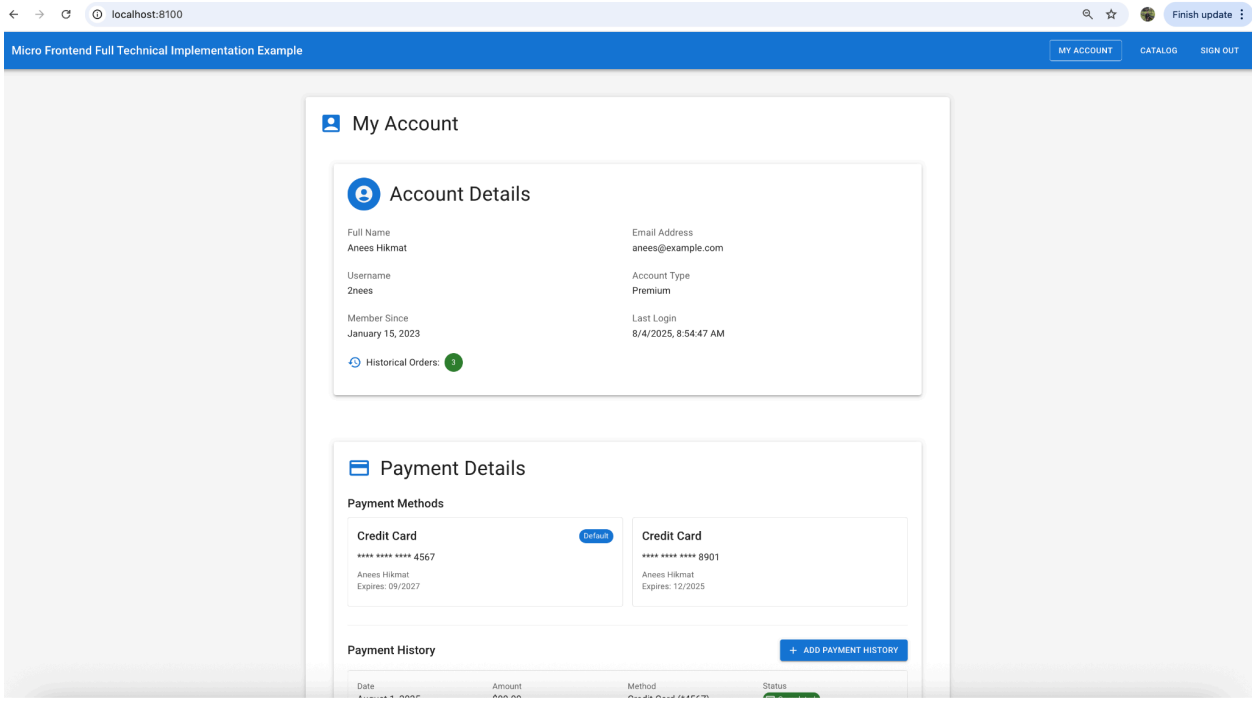
١. سيقوم فريق مهدي ببناء ال micro الخاصة بال payment details.

٢. سيقوم فريق علي ببناء ال account details.

٣. سيقوم فريق مهدي -حسب توزيع المسؤوليات الذي ذكرناه مسبقا- ببناء ال myaccount وربط الإعدادات لتجميع هذه ال micros داخل ال myaccount.

٤. سيتم إضافة الرابط الجديد لل Header micro.

النتيجة النهائية:



بهذا يصبح لدينا المثال كامل وجاهز للعمل بفضل الله. ^^

ويمكنك تصفح المثال وتشغيله مباشرة من خلال هذا الرابط.

* تذكر: من الممارسات المهمة التي ذكرناها سابقا عند تصميمنا ل micros فيها event هي كتابة أو توثيق أسماء ال event التي تعتمد عليها في حالة الإرسال أو الاستقبال.

لكن، هل هذا يكفي؟ بكل تأكيد لا، هناك العديد والكثير من الخصائص التي أشرنا إليها أو مررنا عنها مرور الكرام، وهناك ملاحظات تم كتابتها ومشاركتها في الكود الخاص بالمثال

الذي تم إرساله أو من خلال ال README الخاص بالمثال... جميعها ملاحظات مهمة تتأثر بطبيعة المشروع والتوجه والنظرة... وتذكر أن عملك على مشروع واعتمادك لمعمارية مثل هذه من الأساس قائم على فكرة أن المشروع يجب أن يعيش لفترة طويلة... فإذا كان سيموت سريعا بسبب سوء التطوير فلا داع لكل هذا الجهد *_*.

فائدة

إن خطورة المعايير الثقافية التي ارتسمت لأمة ما أو بيئة ما تكمن في إنزالها على الوحي، لتقييمه أو رده أو رد بعض منه! ومن ذلك قوله تعالى: "قَالُوا أَنَّىٰ يَكُونُ لَهُ الْمُلْكُ عَلَيْنَا وَنَحْنُ أَحَقُّ بِالْمُلْكِ مِنْهُ وَلَمْ يُؤْتَ سَعَةً مِّنَ الْمَالِ"

- مما تعلمته من كتاب القبس الوهاج، أحمد بن يوسف السيد

الفصل الخامس: Build and Deploy Micro-Frontends

إن تبنّي استخدام ال Micro-Frontends يتطلب وجود استثمارا في الاستراتيجيات والخطوات المتبعة في ال CI/CD، وذلك لتقليل التعقيد الناتج عن وجود الكثير من الأجزاء التي سيتم جمعها في نفس الصفحة، ومع استخدام ال Microservices، يصبح من الضروري أتمتة ال build وال deploy لوجود عدد كبير من artifacts، خاصة في معمارية ال horizontal split حيث نُدير العشرات من ال micros، في حين أن ال vertical split سيكون أسهل نسبيًا لأنه يشبه في إعداده مواقع ال SPAs التقليدية.

إن التحسين المستمر لل CI/CD أمر ضروري لتحسين ال Developer Experience (DX)) وتسريع feedback loops، وهذا ما سنحاول تغطيته في هذا الفصل بإذن الله، وسنحاول عرض بعض الأدوات أو أفضل ممارسات ال CI/CD التي يمكنها أن تساعدنا، كما سنتطرق لمفهوم ال Fitness Functions لاختبار خصائص المعمارية المختارة أثناء عملية التطوير.

ملاحظة ١: يقصد بال artifacts في هذا السياق الملفات الناتجة عن عملية ال build، والتي يمكن نشرها أو تشغيلها مثل ملفات ال js bundle أو ال Docker Image أو ملفات ال Static Html, Css... إلخ، باختصار هي النتيجة القابلة لل Deploy بعد ال Build، وهي ما يتم رفعه لل server أو ال cdn ليتم تشغيله.

ال Automation Principles:

العمل مع الـ Micro-Frontends يتطلب تطويرًا مستمرًا لل Automation Pipeline، وإهمال هذا الجانب قد يؤثر سلبًا على سرعة تسليم الأعمال المرتبطة بكل فريق داخل المشروع، وهذا سيتسبب بإضعاف الثقة في عمليات ال deploy على ال production. مما قد يؤدي -في أسوأ الحالات- إلى وجود حالة من الإحباط بين المطورين مع تزايد شعور الخوف من أي عمل جديد، مما يهدد نجاح المشروع ككل، كما أن عملية إدارة هذه الأجزاء يدويا صعبة جدا -وأراها مستحيلة في الواقع العملي-، لذلك، يُعد بناء Automation Pipeline قوي ومتين أمرًا جوهريًا لا غنى عنه، وهذا يشمل ال:

١. ال Continuous Integration: ويقصد بها عملية دمج التغييرات البرمجية من جميع المطورين بشكل تلقائي مع ال main branch -عادة- بشكل متكرر، وفيها تتم عملية ال build وتشغيل ال test، وهي مهمة للتأكد من أن التعديلات الموجودة لن تفسد العمل الحالي مع القدرة على كشف الأخطاء ومعالجتها بشكل أسرع، فمثلا لو كانت هناك مشكلة بال build فسنعلم أن هناك مشكلة يجب أخذها بعين الاعتبار، وإذا كان هناك مشكلة في أحد الاختبارات فذلك يعطينا انطباعا عن وجود خطأ تم تجاوزه ويجب حله ومعرفة سبب تجاوزه... بالنهاية أنت هنا تضمن أن الكود سيعمل حتى لو قام أكثر من مطور بنفس اليوم بعمل merge... من الأمثلة على هذه العملية ما نشاهده يوميا أثناء عملنا، ففي كل مرة نرسل بها "Pull Request"، يتم تشغيل اختبارات تلقائية للتأكد من أن كل شيء يعمل بشكل صحيح.

٢. ال Continuous Delivery: وهي الخطوة التالية بعد ال CI وهي تمثل المكان الذي سيقوم بتجهيز نسخة قابلة للنشر عبر إنشاء ال artifact المناسبة، لكن هنا لا يتم عمل deploy على ال production بشكل تلقائي، بل يكون جاهزا لذلك بنقرة زر من خلال وجود العامل البشري... (أي أن القرار بيد البشر في تحديد لحظة الجاهزية للنشر).

٣. ال Continuous Deployment: وهي نفس ال Continuous Delivery لكن مع فرق مهم وهو عدم وجود العنصر البشري، فعملية النشر هنا تتم بشكل تلقائي بناء على قواعد معينة، وهذه تحتاج لوجود آلية اختبار قوية وثقة عالية بجودة الشيفرة البرمجية ومخطط العمل، وهدفها يكون تسريع العمل لأقصى درجة.

بعد هذه المقدمة، يمكننا القول أن مبادئ الأتمتة هي:

١. اجعل دورة التغذية الراجعة (Feedback Loop) سريعة قدر الإمكان.

٢. قم بالتكرار والتحسين المستمر (Iterate Often) لاستراتيجياتك في الأتمتة.

٣. مكّن الفرق من اتخاذ القرارات المناسبة بشأن micros التي تقع ضمن مسؤولياتهم.

٤. قم بوضع الحدود بشكل واضح (Guardrails) لتعمل الفرق ضمنها وتتخذ قراراتها مع الحفاظ على الجودة والاستقرار.

٥. تعريف استراتيجية اختبار قوية (Solid Testing Strategy) لجميع ال micros على جميع المستويات.

والآن، لنذهب للغوص داخل كل مفهوم... ^^

١. اجعل دورة التغذية الراجعة (Feedback Loop) سريعة قدر الإمكان

تعد هذه النقطة واحدة من أساسيات أي نظام تمت أتمتته، فعندما يقوم أي مطوّر بكتابة شيفرة برمجية جديدة أو تعديل شيفرة برمجية قديمة، فعليه أن يعرف بسرعة ما إذا كان كل شيء ما زال يعمل كما هو متوقع أم لا... وتنبع أهمية ذلك من أن الانتظار الطويل لمعرفة نتائج التغييرات سيبطئ من سرعة الفريق على إنجاز الأعمال، كما أن ذلك سيقلل من الحماس الذي يكون عند المطورين لحظة رفع أي عمل جديد، ومن الأمثلة الجميلة التي نشاهدها بشكل مستمر عند رفع أي شيفرة برمجية إلى GitHub Repo تمت أتمتها، أن الاختبارات الخاصة بال CI تبدأ مباشرة وفي أقل من دقيقة -عادة في ثواني قليلة-، وتظهر النتيجة في دقائق كحد أقصى!

بناء على ما سبق، نلاحظ أن تحليل الخطوات الموجودة مهم جدا لتحديد الخطوات التي يمكن أن تنفذ بشكل متوازي (Parallelized) والخطوات التي يجب أن تنفذ بشكل تسلسلي، والحلول التي تدعم كلا الأمرين هي الحلول الممتازة... مثلا يمكن تطبيق ال unit test ليعمل بشكل متوازي بدلا من الانتظار لوقت طويل حتى تنفيذ الاختبار على المئات أو الآلاف منها! في حين أن ملفات المتعلقة بال migration الخاصة بقواعد البيانات يجب ضمان تنفيذها بشكل متسلسل لضمان عدم حصول أي خطأ...

إن التعامل مع ال Micro frontend يبسط أو يحسن من الاستراتيجيات المتبعة في الأتمتة، فهي تمثل أجزاء صغيرة تجعل من ال CI أسرع... قارن بين سرعة ال CI في مشروع يحتوي ألف unit test وبين micro فيها خمسين unit test... ومع ذلك، هناك تعقيد يجب أن نأخذه بعين الاعتبار، وهو الحاجة لبناء وصيانة جميع ال pipeline التي سيتم بناؤها لهذه ال micros، لأن القيام بذلك بشكل يدوي يعني كارثة دولية تنبئ بدمار العالم! والحل ببساطة من خلال تطبيق مبدأ ال Infrastructure as Code أو اختصارا IaC.

ال Infrastructure as Code:

يعتبر هذا المفهوم أسلوبا لإدارة ال Infrastructure الخاصة بأي شيفرة برمجية مثل الشبكات وال virtual machines وال load balancers ونحو ذلك، وهذا يتم من خلال ملف كتبت فيه مجموعة من الخطوات التي تصف ما نحتاجه ويتم ترجمتها لما نحتاج فعلا، وهذا ما يطلق عليه ال Descriptive model، لذلك تكون النتيجة من تنفيذ هذا الملف هي environment متشابهة لجميع ال micro التي تستخدمها... وبهذا لو كان لدي ١٠ micro فلن نخاف لو احتجنا إضافة خطوة أو تعديلها، لأن تعديل الملف سيقوم بتعديل جميع ال pipeline لجميع ال micro، ومن الأمثلة العملية على ذلك هو كتابة ملف YAML في خطوات إنشاء server معين على AWS... شاهد مثال عملي على Github actions:

ملاحظة: هناك العديد من الطرق المفيدة حتى لتحسين المثال الذي بالأسفل، كما يمكن استخدام بعض الأدوات المفيدة والدمج بينها للحصول على تجربة فريدة... مثل النظر إلى ال Affected micro وبناء ال pipeline ليسحب فقط هذه ال micros... أو استخدام

أدوات خارجية للتحسين من ال pipeline... وكل ذلك مرتبط باحتياجاتك وحجم المشروع ومتطلبات العمل.

```yaml
name: Micro Frontend CI/CD

# تحديد متى يتم تشغيل الـ pipeline
on:
  push:
    branches:
      - main        # عند كل push على الـ main
  pull_request:     # وعند كل pull request

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest

    strategy:
      matrix:
        frontend:
          - homepage
          - dashboard
          - notifications
          - analytics    # قائمة الـ Micro Frontends التي سيتم بناءها واختبارها

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: 22

      - name: Install dependencies for ${{ matrix.frontend }}
        working-directory: ./aneesmfe/${{ matrix.frontend }}
        run: npm install
        # الانتقال إلى مجلد الـ Micro Frontend الحالي ثم تثبيت كل الحزم المطلوبة (dependencies)

      - name: Build ${{ matrix.frontend }}
        working-directory: ./aneesmfe/${{ matrix.frontend }}
        run: npm run build
```

٢. قم بالتكرار والتحسين المستمر (Iterate Often) لاستراتيجياتك في الأتمتة

إن عملية الأتمتة ليست شيئا يبنى لمرة واحدة أول المشروع ثم يترك، بل هي عملية مستمرة من التطوير والمراقبة والاختبار، وكل ذلك للحفاظ على سرعة العمل، لذلك:

١. يجب مراقبة أداء ال pipelines وعرض التفاصيل أو التقارير بشكل واضح (مثل وجود dashboard أو شاشة تعرض وقت ال build المستغرق ونحو ذلك).

٢. إذا كان الوقت يستغرق من ٨ ل ١٠ دقائق، فهذا يعني أننا بحاجة للتحسين وبشكل إجباري!

٣. عمل مراجعة شهرية إذا كانت ال pipelines بطيئة، وكل دورة مثل ٣ أو ٤ شهور في حال كانت جيدة.

هذه العملية مهمة لتسريع ال feedback loop التي تحدثنا عنها في أول مبدأ من هذه المبادئ.

٣. مكّن الفرق من اتخاذ القرارات المناسبة بشأن micros التي تقع ضمن مسؤولياتهم

إن مشاركة المطورين في بناء ال pipelines الخاصة بال micros التي يعملون عليها يساعد على تحسين عملية الأتمتة وتسريعها والوصول إلى أفضل الخطوات التي يمكن تطبيقها واستخدامها،

وذلك ينبع من أن لكل فريق تحدياته وأولوياته المختلفة، فمثلا قد يهتم فريق ال payments بموضوع الاختبارت وإضافة security checks أكثر من فريق المنتجات الذي قد يكون تركيزه منصبا على الأداء وتحسينه...

هناك نقطة تقنية مهمة هنا، وهو أن ال micros قد تختلف بحاجاتها وهذا سيجعل من ال pipeline تستخدم وصفا مختلفا لتحقيق الأتمتة، وهذا قد يكون نابعا من وجود أكثر من stack أو تقنية مستخدمة ضمن نفس المشروع... لذلك، تعمل المؤسسات على إنشاء guardrails (حدود إرشادية) للفرق، وهو ما سنتحدث عنه في النقطة التالية [٨٨].

٤. قم بوضع الحدود بشكل واضح (Guardrails) لتعمل الفرق ضمنها وتتخذ قراراتها مع الحفاظ على الجودة والاستقرار

ال Guardrails هي قواعد أو خطوط إرشادية تُحدد للفرق التقنية حتى تساعدهم على العمل بشكل صحيح ومنظم لضمان السير في الاتجاه الصحيح وبما يتوافق مع المؤسسة ككل، وهي بذلك تتداخل مع النقطة السابقة.

فمثلا تكون مراقبة الأداء الخاص بال CI/CD ووضع الصلاحيات والقواعد الخاصة بالنشر من مسؤولية الشركة، لكن إضافة ال scripts وخطوات إنشاء ال artifact من مسؤولية الفريق، وهذا لا يعني الفصل بين الشركة والفرق، بل تمكين الفرق من اتخاذ قرارات تقنية مسؤولة تسهم في تحسين النتائج النهائية مع مراقبة ذلك ونتائجه ضمن حدود وتوجيهات واضحة.

مثال عملي، سنفترض أن شركتي تسمى: قبيلة ^^، وعليه تم وضع حدود إرشادية وكانت كما يلي:

1. مسؤولية الشركة:

- اختيار أداة موحدة لل CI/CD وستكون Jenkins
- إعداد ال Security الخاصة بال CI/CD ومنع الوصول الغير مصرح به وإعطاء أقل صلاحيات ممكنة اعتمادا على مبدأ ال Least Privilege، وتخزين ال API tokens وكلمات المرور ونحوها بطريقة مشفرة وآمنة ومنع ظهورها بالملفات أو بال logs...
- إعداد ال SSO للمطورين بحيث يسمح للموظفين باستخدام حساب الشركة للدخول إلى jenkins مثلا دون إنشاء حسابات جديدة، وهذا لضمان التحكم المركزي والمنع التلقائي لأي موظف غادر الشركة من تسجيل الدخول...
- إعداد ال Logging وتتبع كل من قام بتشغيل ال pipeline والأحداث والأوامر التي حصلت ومكان الفشل وإرسال إشعارات بذلك...
- تحديد صلاحيات النشر في ال environments المختلفة
- وضع قواعد تتعلق بال code review قبل عمل merge لأي PR
- وضع قواعد أو خطوات للتعامل مع أي فشل يحصل بال pipeline.
- بناء Shared CI/CD template فيها المراحل الأساسية الجاهزة والمشتركة بين الفرق مثل ال test, build, security check...

2. مسؤولية الفرق:

- كل فريق يكتب السكربتات الخاصة به مثل ال npm run build أو غيرها بما يتوافق مع طبيعة الكود الخاص بهم.
- وضع إعدادات التحسين مثل ضغط الملفات أو ترتيب الخطوات.
- تحديد الآلية المناسبة حتى نحصل على artifact قابلة للنشر، هل ستكون static file أم Docker...
- تحديد ال custom checks الخاصة بهم، مثلا كتابة مجموعة من الأوامر التي تتحقق من أن ملفات الترجمة محدثة أو اختبار توافقية micro مع micro أخرى...

٥. تعريف استراتيجية اختبار قوية (Solid Testing Strategy) لجميع ال micros على جميع المستويات.

هناك حاجة لتخصيص وقت -وهو استثمار مهم- لوضع استراتيجية اختبار قوية ومتينة، خصوصًا إضافة اختبار ال end-to-end للمشروع الخاص بنا، فإذا كان لدينا عدة micro-frontends وفي كل view هناك عدة micros مستخدمة من عدة فرق فإننا نريد التأكد من أن تطبيقنا يعمل من البداية وحتى النهاية بشكل صحيح، وهذا يشمل التأكد من أن عملية الانتقال بين الصفحات مغطاة وتعمل بشكل سليم، وهذا قبل نشر ال artifacts الخاصة بأي نسخة جديدة على ال production.

ويضاف إلى ذلك أهمية وجود ال unit testing وال integration testing، لكن لا يوجد تحديات خاصة أو كبيرة تواجهنا هنا مع micro-frontends في هذا الجانب.

ال Developer Experience:

إن واحدة من الاعتبارات الأساسية عند العمل باستخدام micro-frontends هي تجربة المطور (DX). وعلى الرغم من أن ليس كل الشركات يمكنها دعم فريق مخصص لتحسين تجربة المطورين، إلا أنه يمكننا إنشاء فريق افتراضي داخل المؤسسة لهذا الغرض. هذا الفريق سيكون مسؤولًا عن إنشاء الأدوات التي يحتاجها المطورين وتحسين تجربة عملهم مع ال micro-frontends لتجنب أي عوائق أثناء تطوير هذه ال micros... وهذا يشمل عملية التطوير أثناء البرمجة نفسها وآلية الاختبار والفحص ومالجة المشاكل في الجزء الذي يقع تحت مسؤوليتي، وعلى الجميع أن يعرف مسؤولياته وأنه مسؤول عن جزء من النظام وليس النظام كاملا، وفي نفس الوقت لا يتنافى هذا مع حس المسؤولية المرتبط بالأجزاء الأخرى، وإنما ضبط حدود الفرق وأجزاء عملهم... كما أن هذا الفريق سيكون مسؤولا أو حريصا على جعل ال DX قابلة للتوسع وغير منغلقة أمام التقنيات الجديدة، فمثلا لو صدر إصدار جديد أفضل أو مكتبة أخرى أفضل من التي بين يدينا الآن، وهناك فائدة من استخدامها لتحسين ال DX لا مانع من ذلك... وكما أن عملية مراقبة الأداء وال CI/CD عملية مستمرة، فهذه عملية مستمرة أيضا ^^

هناك أسئلة يجب أن يجيب عنها هذا الفريق أو المسؤول أو المؤسسة عند اعتماد استخدام ال micro frontend، وإجابة هذه الأسئلة ستؤثر على ال DX، منها:

- كيف يمكن اختبار الجزء الذي أعمل عليه (يشمل ال test وال performance... وغيرها)؟
- كيف يمكن مشاهدة النتائج الخاصة بال micro التي أعمل عليها ضمن المشروع؟ وكيف سأتحقق من توافقها مع الأجزاء الأخرى؟
- هل الأدوات الموجودة تساعدني حقيقة على إتمام الأعمال أم أنها ستشكل عائق أمامي؟
- كيف سنضمن تناسق ال UX؟
- كيف سنشارك البيانات بين ال micros؟

وغيرها مما ذكرناه سابقا أيضا ضمن الفقرات...

إجابات هذه الأسئلة ستؤثر بشكل كبير على تجربة المستخدم وشغفه في العمل... تخيل الفرق بين مؤسسة اعتمدت ال Client side composition، وشركة اعتمدت على ال Edge composition، أيهما سيمتلك DX أفضل؟ وهذا نفسه ينطبق لو غصنا في التفاصيل الخاصة لهذه الإجابات وما يترتب عليها.

وهنا يظهر دور هذا الفريق (فريق ال DX) في اتخاذ القرارات المناسبة وتقديم الأدوات واقتراحها أو بناءها ونحو ذلك.

ال Horizontal Versus Vertical Split:

إن القرار الذي تم اتخاذه لاعتماد ال Horizontal Split أو ال Vertical Split في مشروعنا سيؤثر بشكل كبير على ال DX، وقد قلنا مرار أهمية هذا القرار وأثره على مختلف الجوانب...

في حالة اعتماد ال Vertical Split في مشروعنا، فإن كل micro-frontend سيبنى كصفحة HTML أو تطبيق SPA مستقل وتحت إدارة فريق واحد، وهذا يجعل تجربة التطوير مشابهة لتطوير ال SPA التقليدي، وهذا يعني بالضرورة تجربة مطور ممتازة.

كما أن ال test بأنواعه المختلفة يتم بسهولة نسبية هنا، ويستطيع كل فريق اختبار الجزء الخاص به، بما في ذلك الانتقالات بين micro-frontends، وسيتمكن من رؤية ذلك بشكل مباشر وسهل، لكن لضمان التكامل بين ال micros يجب التأكد من أن جميع هذه ال micro تم تحميلها بشكل صحيح داخل الـ Application Shell، ويُفضل هنا أن يتولى الفريق المسؤول عن تطوير الـ Shell تنفيذ اختبارات ال End-to-End الخاصة بالتوجيه بين الأجزاء وما يرتبط بها.

في حين أن اعتماد ال Horizontal Split سيقودنا لتحديات أكبر تقلل من جودة تجربة المطور، فهنا تقع عليه مسؤولية إدارة أكثر من micro غالبا أو micro في أكثر من view، أو كلاهما معا! وفي هذه الحالات فإننا سنحتاج إلى آلية وأدوات تضمن عملية الاختبار أثناء ال run time والتحقق من أن الأجزاء المختلفة تعمل معا، وألا تعارض بين ال dependencies، كما يقع على عاتقه فهم كيفية التواصل بين ال micros المختلفة التي طورتها فرق مختلفة... لذلك لا يوجد أدوات قياسية هنا، وغالبا ما تلجأ الشركات لبناء حلولها الخاصة

أو استخدام الحلول التي تتوافق مع احتياجاتها، لذلك تعتبر تجربة المطور هنا أكثر تحديا، وهذا يدعو فريق ال DX يهتم بتطوير الأدوات لتسهيل حياة المطورين، فكما يقوم المصمم بالبحث عن أفضل UX لزوار الموقع يقوم فريق ال DX بالبحث عن أفضل DX لهم. ^^

ال Frictionless Micro-Frontends Blueprints:

إن ال DX لا تتأثر فقط بالأدوات، بل أيضا كيف يمكننا إنشاء أو إضافة micro جديدة، فكلما زاد لدينا عدد ال micro أصبحت الحاجة لأتمتة هذه العملية أمرا ضروريا... لذلك يعد استخدام ال Command Line لإنشاء ال micro frontend واحدة من أشهر الخيارات المستخدمة فعليا في عالم ال micro، بحيث نقوم ببناء template فيه جميع المتطلبات وال dependency التي نحتاجها والمتبعة داخل الشركة والمشتركة مع كل micro -ال guardrails-... هذه العملية تعتبر ممتازة وجميلة ومهمة من حيث تسريع آلية العمل وزرع الثقة والراحة في نفس المطور، كما يساعد ذلك أي مطور جديد على البدء مباشرة بالعمل ضمن القواعد والإرشادات التي اعتمدتها المؤسسة مع المطورين.

ال Environments Strategies:

في الشركات المتوسطة والكبيرة عادة ما تكون ال environments الموجودة هي ال testing وال staging وال production، بحيث تمثل ال testing أول مرحلة بعد سحب

الشيفرة البرمجية من جهاز المطور إلى المكان الذي يشترك فيه جميع المطورين، وهو المكان الذي يستخدم لعمل test سريع على العمل الذي تم رفعه، لذلك تعتبر هذه ال env غير مستقرة... ثم يتم نقل العمل من هذه ال env إلى ال staging والتي تمثل env شبيهة بال production بشكل كبير، وتستخدم لاختبار وفحص النسخ شبه النهائية قبل نشرها، ثم يتم نقل العمل من هذه ال env إلى ال prod والذي يتواجد عليه المستخدمون الحقيقيون، لذلك تكون هذه ال env محمية وعليها قيود كثيرة لمنع حدوث أي خطأ مقصود أو غير مقصود، كما يتم حمايتها من الوصول اليدوي *-*... وهنا يظهر دور فريق ال DX أيضا، فكما تلاحظ مجموعة القيود والضوابط التي تم وضعها كانت من قبل هذا الفريق، كما أن توفير أدوات ال deploy المناسبة والترقية بشكل آمن وسريع من وظائفه، ومن هنا يظهر دورهم أيضا في ابتكار آلية لتحسين بيئة العمل من خلال ال On-demand Environment...

ال On-demand Environment:

بدلاً من الاعتماد فقط على ال env الثلاث التقليدية، يمكن إنشاء envs مؤقتة عند الطلب تُستخدم لاختبار أجزاء معينة من النظام، ثم يتم حذفها مباشرة بعد انتهاء الاختبار، وهذا له فوائد كثيرة منها إمكانية فحص ال micros بشكل معزول، وإجراء ال end-to-end على جزء صغير من النظام، وتمكين أصحاب العمل أو ال Products manages من معاينة المزايا أو جزء منها قبل إطلاقها أو حتى قبل عمل merge لها مع النظام الأساسي... كما يمكننا من خلال هذا الأسلوب إنشاء أكثر من env في وقت واحد لغايات التجربة والفحص ثم يتم

حذفها تلقائيا، وكل هذا سيوفر من التكاليف بشكل كبير لأن هذه ال envs ستستخدم لفترة قصيرة...

ال Version Control:

يعد ال Git واحد من أشهر وأكثر الأنظمة استخداما حول العالم، فمن خلاله يمكنك إدارة الإصدارات الخاصة بالشيفرة البرمجية، مما يتيح تتبع التغييرات على الشيفرة البرمجية على مر الوقت، ومن هنا تظهر أهمية تحديد ال Branching Strategy، وال Repository Strategy.

ال Branching Strategy:

ويقصد بها آلية أو كيفية تنظيم الأعمال داخل ال repo الواحدة بهدف تنظيم سير العمل بين المطورين، وإدارة الميزات (features) والإصلاحات (fixes) والإصدارات (releases) بكفاءة، فمثلا هل سنقوم بكل التعديلات على branch واحدة؟ أم سنقوم بإنشاء branch جديدة لكل ميزة أو إصلاح؟

إن تحديد الآلية المناسبة مهم لتسهيل التعاون بين المطورين وتحسين آلية ال merge بين الشيفرات البرمجية دون أخطاء وبما يتناسب مع مخطط سير العمل الذي تم اعتماده في المؤسسة، ومن أشهر الأمثلة على الاستراتيجيات المستخدمة:

١. ال Git flow: وهنا يتم تقسيم العمل إلى عدة أنواع من ال branches، مثل:

*. ال main: النسخة المستقرة.

*. ال develop: برانش تدمج فيه الميزات الجديدة أثناء التطوير.

. ال feature/: برانش مؤقتة لإنشاء ميزات جديدة.

. ال release/: برانش لإعداد إصدار أو نسخة جديدة من المشروع.

. ال hotfix/: برانش لإصلاح مشاكل على production مباشرة.

وهذا الأسلوب من الأساليب المناسبة في الفرق الكبيرة والمشاريع المعقدة.

٢. ال Trunk-Based Development:

يتم العمل هنا على ال main branch أو على branches عمرها قصير جدا بحيث لا يتجاوز اليوم أو يومين، وهذه الآلية مناسبة للمشاريع أو الفرق الصغيرة وال CI/CD ذات النشر التلقائي المستمر. -شخصيا لا أحب هذا الأسلوب نهائيا، لأنه يتطلب انضباطا عاليا من جميع المطورين والالتزام بعمل merge للتغيرات الصغيرة أولا بأول بسبب وجود الكثير من العمليات التي سيتم عمل merge لها، وهذا أراه من خبرتي صعب التطبيق في شركاتنا الصغيرة أو الناشئة... أما الشركات الكبيرة ذات الانضباط العالي والقواعد الصارمة، فيختلف هذا الأمر هناك، بل يعدون هذا الأسلوب ممتازا لهم لأنه ينقذهم من ال Merge Hell، كما أن له تحديات كبيرة مع ال monorepo يجب معالجتها في حال استخدامه هناك-.

٣. ال Feature Branching:

يتم هنا إنشاء branch مستقلة لكل ميزة والعمل عليها بشكل مستقل، وهي ممتازة عند الحاجة للتحكم الدقيق بكل مزية ومراجعة الشيفرة البرمجية قبل عمل ال merge بسهولة.

ال Repository Strategy:

ويقصد بها الطرق التي يتم بها تنظيم الشيفرة البرمجية والمكونات المختلفة لمشروع ما داخل ال Repositories، أي هل سنضع كل المشروع في repo واحدة؟ أم نقسمه إلى عدة repo مستقلة؟

وبذلك لدينا نوعين رئيسين هما:

١. ال Monorepo: هنا يتم وضع جميع أجزاء التطبيق داخل repo واحدة فقط، مثل الواجهات، والملفات، والخدمات وغيرها... فمثلا يمكن أن يكون لدينا مجلد frontend ومجلد backend ومجلد auth داخل repo واحدة اسمها aneesCourse... شاهد الصورة أدناه:

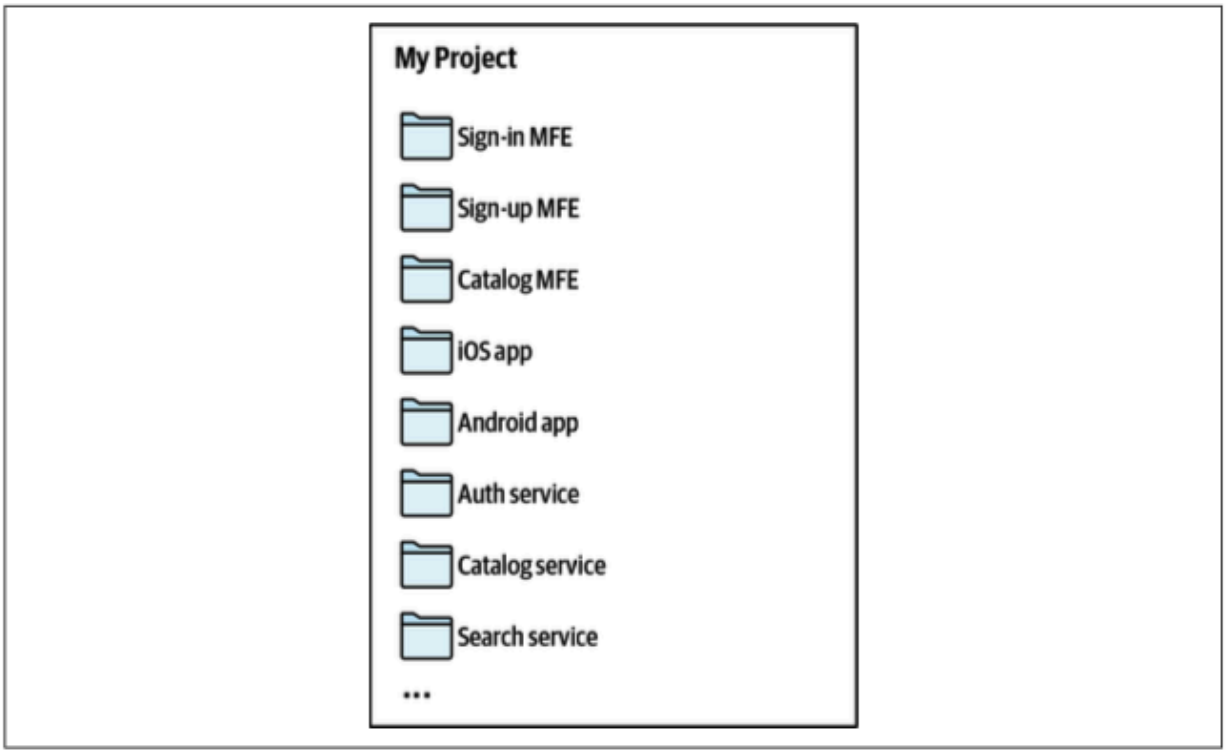


Figure 6-1. Monorepo example where all the projects live inside the same repository

هذا الأسلوب يقدم مزايا متعددة مثل:

*. إمكانية استخدام المكتبات المشتركة بسهولة أو نقل جزء من الكود من أحد الخدمات وجعله متاحا للجميع لاستخدامه بدلا من إعادة كتابته أكثر من مرة في أكثر من مكان...

*. سهولة التعاون والتواصل بين الفرق، وفهم المشروع ككل.

*. يشجع الفرق على متابعة التحديثات الخاصة بال dependencies وال apis versions ونحو ذلك، ومواكبة التغيرات...

*. سهولة دخول المطورين الجدد إلى المشاريع المبنية بهذا الأسلوب.

*. يمكن للموظفين اكتساب خبرة ممتازة من خلال النظر للشيفرات البرمجية المتعددة الموجودة من الفرق والمطورين الآخرين واستلهام الأفكار من أعمالهم...

كما أن لهذا الأسلوب مميزات فله أيضا عيوب أو تحديات مهمة مثل:

*. يتطلب استثمارا دائما في أدوات الأتمتة، وغالبا ما تصبح الأدوات المتوفرة مفتوحة المصدر غير مناسبة مع نمو حجم المشروع بشكل كبير...

*. مع زيادة حجم المشاريع داخل ال monorepo فإننا سنكون بحاجة أيضا إلى توسعة ال pipeline، فشركات مثل جوجل وفيسبوك وغيرها أوضحت أن هذا يتطلب منها فرقا كاملة ومستقلة لإدارة هذه الجزئية والحفاظ على أداء هذه الأدوات ضمن ال pipeline، لذلك قامت هذه الشركات بتصميم build tools خاصة بها وأتاحتها للاستخدام للمطورين الذين يعانون من هذه المشكلة... وكمثال عملي لو صارت ال monorepo لدينا فيها ١٠ مشاريع ولدينا أكثر من ١٠٠ مبرمج، فإن هذا قد يجعل زمن بناء المشروع كامل مع ال test هو نصف ساعة، تخيل لو حدثت ١٠٠عملية merge يوميا فقط من قبل المئة مبرمج -وهو رقم بسيط- كم سيكون الوقت المستغرق؟! وكيف سيؤثر هذا على ال feedback loop؟! وكيف ستكون حالة ال Queue لل pipeline؟! لذلك كان الحل الذي وضعته هذه الشركات مرتكزا على أفكار جميلة منها عمل build فقط للأجزاء التي تغيرت فقط، كما تم توزيع عملية ال build على مئات ال servers بدلا من أن تكون العملية على server واحد... ومن أمثلة هذه الأدوات: Bazel.

*. ترابط المشاريع بشكل مفرط بحيث يمكن أن نصل لمرحلة لا يمكن عمل deploy لهذه المشاريع إلا معا، مما قد يمنع مشاركة ال micro التي تم تصميمها مع مشاريع أخرى لارتباطها الوثيق بما تم تصميمه هنا.

*. في ال monorepo ال Trunk-Based يجلب معه تحديات كبيرة، فهناك ضغط كبير على ال main branch بسبب وجود الكثير من عمليات ال mega التي تتم يوميا على نفس ال branch، وأي خلل صغير هنا سيؤثر على الجميع، كما أن أي شيفرة برمجية تم تعديلها تؤثر على المشاريع الأخرى ستجعل الجميع متأثرا بذلك وعليهم معالجة هذا التأثير مباشرة، ولا يوجد عزل كاف للتغيرات الكبيرة ولا يمكن بسهولة إيقاف مزية بعد عمل ال merge... ومع ذلك يعد هذا الخيار من الخيارات التقنية المناسبة لهذا الأسلوب! بشرط توفر الالتزام العالي والانضباط الشديد والاستثمار العالي والفعال في الأدوات المستخدمة، لذلك ذكرت هذه النقطة من ضمن عيوب وتحديات ال monorepo.

*. يمكن أن يصبح ال Git History فوضويا بشكل لا يصدق في هذه الاستراتيجية إذا لم يكن هناك انضباطا من المطورين... تخيل ١٠ مطورين قاموا بعمل commits على أجزاء مختلفة مثل ال fe, be, auth وكانت ال messages هي: fix, update, test final, test final 1 وهكذا، هذا كله سيجعل من عملية تتبع المشكلة وإصلاحها أو العودة لمكان معين أمرا صعبا...

في الختام وقبل الانتقال للاستراتيجية الثانية المستخدمة، فإن ال monorepo واحدة من الخيارات الممكنة والمتاحة للتعامل مع ال micro frontend، وهناك أدوات جميلة تساعد على إتمام هذه العملية وجعلها عملية لطيفة مثل ال NX...، مع ذلك فهذا يتطلب التزاما

كبيرا ووعيا بأهمية الحفاظ على جودة الشيفرة البرمجية والحذر من ال coupling بين ال micros...

٢. ال Polyrepo: هنا يتم تخزين كل جزء من المشروع في repo منفصل، لذلك يطلق عليه أيضا multirepo، وهنا حسب مثالنا السابق فسيكون ال fe داخل repo اسمها aneesFeCourse وال be داخل aneesBeCourse وال auth داخل aneesAuthCourse وهكذا... شاهد الصورة أدناه:

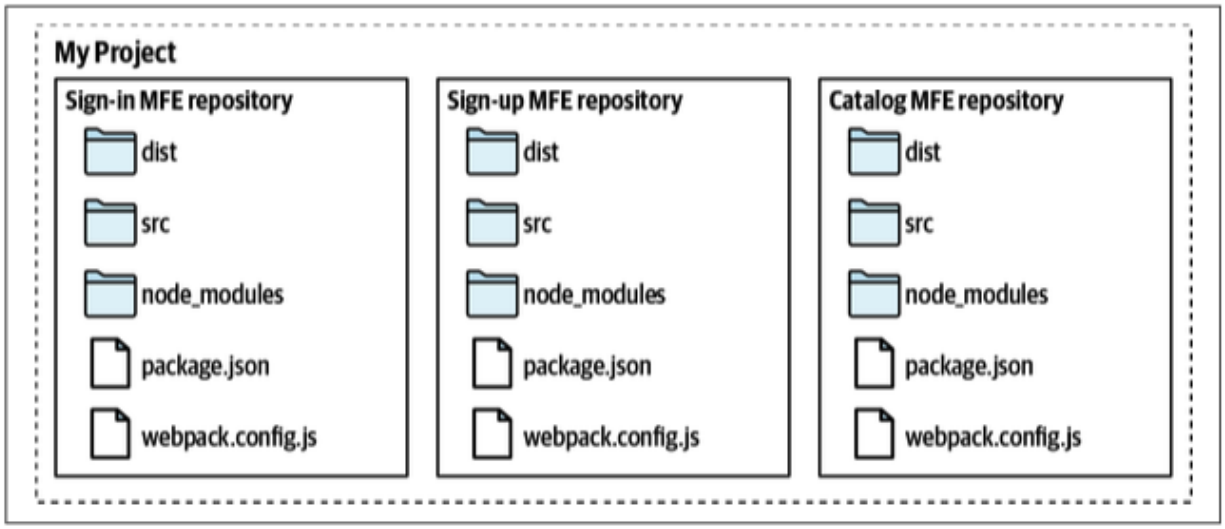


Figure 6-2. Polyrepo example where we split the projects among multiple repositories

هذا الأسلوب يقدم مزايا متعددة مثل:

*. يمكننا استخدام ال branch strategy المناسبة لكل repo على حدا بما يتناسب مع طبيعتها واحتياجاتها...

*. لا يوجد خطر على باقي الفرق، فكل فريق يعمل باستقلالية دون التأثير على الآخرين.

*. أسلوب ممتاز في ضبط وتحديد آلية التواصل بين المشاريع من خلال APIs، وتحديد العلاقات بين producers وال consumers ووضع الآليات التي تضبط أي تغييرات يمكن أن تتسبب بأي breaking changes، فمثلا لو كان لدينا فريق A يعمل على ال Login Service وفريق B يعمل على ال UI لتطوير واجهة تستخدم هذه ال Service، فإن الفريق A ملزم بتحديد البيانات المقبولة لهذه ال service وما هو ال output المحتمل منها وآلية التعامل مع الأخطاء فيها... وهذا ما يطلق عليه بال Contract.

*. تحكم دقيق في الصلاحات، فلو كانت لديك أجزاء مهمة أو خطيرة من الشيفرة البرمجية تخشى من الوصول غير المصرح به إليها، فإن ال polyrepo مفيد جدا، لأنك يمكن أن تعطي صلاحيات دقيقة لكل فريق أو قسم على هذه ال repo.

*. الاستثمار بأوات الأتمتة هنا أقل وأوفر من ال monorepo، فعملية ال build ستكون أسرع وال CI/CD tools أبسيط وغير مكلفة، ويمكن استخدام IaC بين الفرق.

كما أن لهذا الأسلوب مميزات فله أيضا عيوب أو تحديات مهمة مثل:

*. صعوبة في اكتشاف المشاريع، لذلك يجب وضع آلية واضحة للتسمية بحيث يمكن البحث عن أي مشروع بسهولة.

*. تكرار الشيفرة البرمجية، فدون حوكمة واضحة قد يتكرر الجهد وتزيد المشاكل، وقد لا يتم مشاركة الكود بين الفرق بالشكل المناسب.

*. قد تختلف جودة الشيفرة البرمجية من repo إلى أخرى بسبب عدم الالتزام بالتحديث الدوري لهذه ال repo.

فائدة

قد يصبر الصالحون عل الشدائد والمصاعب لكّنّهم قد لا يصبرون عل النِّعَم والرَّخاء، وهذا يذكرني بأكثر من ورد ذكرهم من كبار المجرمين في القرآن الكريم، فتراهم أسياد أقوامهم، وأشرافهم، وأغناهم... ومع ذلك كان رخاؤهم عاملا مشتركا من عوامل كفرهم واستكبارهم، وهذا يزيد من أهمية استحضار باب الهداية وسؤال الله الهداية والثبات ونحو ذلك؛ في الرخاء والشدة!

- مما تعلمته من كتاب القبس الوهاج، أحمد بن يوسف السيد

ال A Possible Future for a Version Control System:

إن أي استراتيجية من الاستراتيجيات المختلفة التي تحدثنا عنها والتي يمكننا استخدامها مع ال Version Control لن تكون حلاً مثاليًا لكل الحالات، بل ستكون الحل الأنسب حسب السياق الذي نعمل عليه، وتذكر أن الأمر دائما عبارة عن موازنة بين المزايا والعيوب لكل أسلوب حسب سياق المشروع ^^.

هناك أسلوب آخر لم نتحدث عنه في ال Repo Strategy، وهو استخدام نظام هجين يجمع بين الأسلوبين بحيث يمكننا الاستفادة من مزايا كل استراتيجية وتقليل عيوب كل واحدة منها.. شاهد الصورة أدناه:

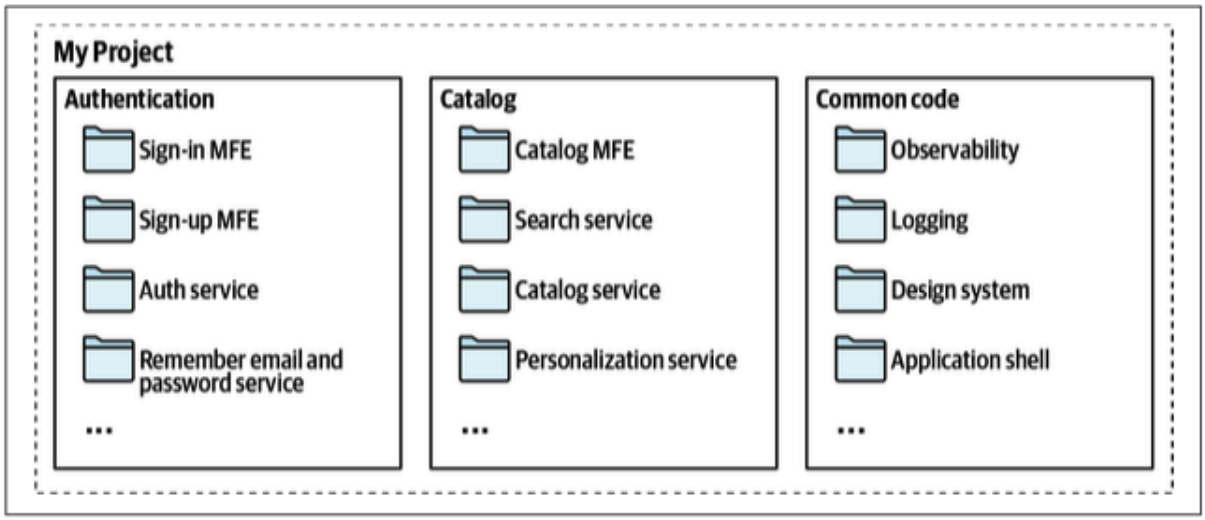


Figure 6-3. A hybrid repositories approach, where we can combine monorepo and poly-repo strengths in a unique solution

كما تلاحظ في الصورة فإن فكرة النظام الهجين هذا قائمة على مبدأ استخدام ال monorepo ضمن subdomain محدد، و Bounded Context واضح، ويتم استخدام ال polyrepo بين ال subdomain المختلفة حيث يفضل الاستقلالية... فمثلا ال Auth كما في الصورة هي بذاتها polyrepo بالنسبة للمشروع، وهي نفسها monorepo لجميع الأجزاء المرتبطة بال auth من FE... وهذا الأسلوب له تحدياته أيضا من أهمها أنه يتطلب تخطيطا دقيقا لكيفية تقسيم السياقات الخاصة بها لأنها مرتبطة بال repo، ويحتاج لتوثيق فعال وقوي ونظام حوكمة واضح...

ال Continuous Integration Strategies:

قبل أن نبدء في هذه النقطة أحب أن أشير إلى نقطة مهمة حول مفهوم ال DevOps، فال DevOps يمثل مزيج من الفلسفات الثقافية والممارسات والأدوات التي تزيد من قدرة المؤسسة على تقديم التطبيقات والخدمات بسرعة عالية، وبذلك لم تعد فرق ال Development وال Operations معزولة عن بعضها البعض، بل وصل هذا الأمر في المؤسسات لدمج هذان الفريقان في فريق واحد -في بعض الأحيان-، حيث يعمل الجميع على دورة حياة التطبيق من البداية وحتى النهاية، أي من التطوير والاختبار إلى النشر والتشغيل، مع تطوير مجموعة من المهارات التي لا تقتصر على وظيفة واحدة فقط.

ملاحظة: يقصد بالفلسفات الثقافية -cultural philosophies- لل DevOps بالقيم والمعتقدات والسلوكيات التنظيمية التي تشكل طريقة التفكير لدى الفرق وكيفية تفاعلها معا، وذلك يشمل تشجيع التعاون وتحمل المسؤولية بشكل مشترك طوال فترة حياة المشروع والشفافية فيما بينهم، والتعلم من الأخطاء والتركيز على CI بدلا من الاكتفاء بالوضع الراهن... لذلك، إذا سمعت DevOps يقول: "جميعنا نتحمل المسؤولية من التطوير إلى التشغيل لأننا فريق واحد"، أو "الخطأ الذي حصل فرصة للتعلم، فلنجعل موثوقية سير العمل أولوية لدينا"، فاعلم أنه يتحدث من مرتكز فلسفي نابع عن ثقافة ال DevOps التي تبحث عن تحسين طريقة التفكير وآلية العمل وسلوك الفرق في المؤسسة...

والآن، لماذا بدأنا في هذه المقدمة في هذا الباب؟

الجواب ينطلق من أن الرسالة الأساسية التي نريد إيصالها: أن فرق التطوير نفسها يجب أن تتحمل مسؤولية الـ CI وأتمتة الـ Pipelines، وذلك بدلاً من الاعتماد على فرق خارجية، لأن ذلك يسرع عملية التطوير ويحسن جودة المنتج، ومع انتشار ثقافة الـ DevOps، أصبح من المعتاد أن تُمنح الفرق مرونة وأدوات لتنفيذ الـ CI وفق حدود ومعايير الشركة، وفي الـ Micro-frontends يزداد هذا الأمر أهمية لأن هناك أجزاء مستقلة من التطبيق يجب بناؤها ونشرها بشكل متكرر وموثوق، ويمكن أن تستخدم الفرق أدوات مختلفة مثل Rollup أو Webpack بحسب طبيعة ما يقومون به من أعمال، وهذا التنوع يسمح بتجربة ومقارنة الأدوات في بيئات حقيقية، كما أن طريقة تنفيذ الـ CI تعتمد على طريقة تقسيم الـ

Micro-frontends -عمودي أو أفقي-، لأن كل أسلوب منهما يفرض طريقا مختلفا في الاختبارات والنشر.

كما أننا يجب أن ندرك أنه لا يوجد تطبيق CI واحد يناسب جميع الـ Micro-frontends؛ فالكثير منها يعتمد على المشروع ومعايير الشركة والنهج المعماري، فعلى سبيل المثال عند تنفيذ الـ Micro-frontends مع الـ Vertical Split، فإن جميع مراحل الـ Pipeline لل CI ستكون مشابهة لمراحل الـ SPA التي اعتدنا عليها، كما يمكننا إجراء اختبارات الـ End-to-end قبل النشر ونحو ذلك...

في حين أن الـ Horizontal Split سيتطلب مزيدا من التفكير حول الوقت المناسب لتنفيذ مهمة معينة، فعند إجراء اختبار الـ End-to-end سيتعين علينا تنفيذ هذه المرحلة في بيئة الـ Staging أو الـ Production؛ وإلا فسيتعين على كل Pipeline فردي أن يكون على دراية بجميع مكونات التطبيق، مع جلب أحدث إصدار من جميع الـ Micro-frontends ودفعها إلى environment مؤقتة! وهذا حل يصعب صيانته وتطويره ^^

وهذا كله يعيدنا إلى نقاط مهم تحدثنا عنها سابقا أو أشرنا إليها من خلال الشروحات، وهي تتعلق بوجود حدود واضحة ومرسومة توضح صلاحيات ومسؤوليات كل فريق، ومتى يمكن اعتبار تغيير معين هو تجاوز يجب منعه...

على سبيل المثال:

من أدوار ال DevOps تصميم وصيانة ال infrastructure لل CI/CD من servers و build agents ونحو ذلك، وتحديد المعايير والسياسات المتعلقة بالأمان وال deploy وال

performance، وتوفير ال environments المختلفة وضمان استقرارها ومراقبة ال pipelines وضمان جاهزية الأنظمة ونحو ذلك.

لكنهم لن يتدخلوا في منطق الشيفرة البرمجية نفسها أو طريقة تنظيمه إلا إذا كان لذلك تأثير مباشر على ال deploy أو الأمان... ولا يفرضون أدوات أو إعدادات تفصيلية للمشروع إلا إذا كان لذلك سبب تقني وفني قوي مثل مشاكل أداء أو توافقة أو أمان..

في حين أن من أدوار المطورين إعداد وإدارة ال pipelines الخاصة بمشاريعهم داخل الإطار الذي رسمه فريق ال DevOps، والتأكد من أن شيفرتهم البرمجية تمر بمراحل التحقق التي قامو ببناءها مثل ال unit test وال integration test قبل عملية ال merge... كما أن اختيار الأدوات المناسبة للحصول على الـ Artifacts القابلة للنشر من وظيفتهم، أكان ذلك من خلال webpack أو rollup أو غيرها، وتقليل الوقت الخاص بال Feedback loop سيكون من مسؤولياتهم، وإصلاح المشاكل التي تظهر بال pipelines بسبب التغييرات التي يقومون بها...

لكنهم لن يتدخلوا في ال infrastructure لل pipeline التي وضعها ال devops أو يقومون بتغييرها دون موافقة ال DevOps أو مناقشة السبب التقني معهم، كما يجب منعهم من إضافة أو إزالة أي موارد تخص أي environment غير المخصصة للتطوير، إلا ضمن إجراءات محددة! فمن غير المعقول أن يقوم المطور بتغيير إعداد يتعلق بال production دون المرور بإجراءات نضمن فيها أن التغيير صحيح ومخطط سير العمل صحيح وبموافقة ال DevOps!

ولا يظن أحد أن هذا يعني انفصال أو عزل بين الفرق، بل هو أسلوب تلاق يضمن حدود كل فريق منهم، ويدعم التواصل المباشر للحصول على المساعدة لتحسين الـ CI من قبل كل فريق أمام المشاكل التي تواجهه... وكثير من الشركات تعمل بهذا الشكل أو بشكل قريب منه دون أن تدري ذلك!

ال Testing Micro-Frontends:

جميعنا يدرك أهمية اختبار الشيفرة البرمجية الخاصة بنا، وأهمية اكتشاف الأخطاء أو العيوب في أقرب وقت، وهذا الأمر يزداد أهمية إذا اعتمدنا الـ Micro-frontends كمعمارية أو أسلوب تطوير في مشروعنا...

إن العمل مع الـ Micro-frontends لا يعني أننا سنقوم بتغيير طريقة تعاملنا مع الممارسات التي تعلمناها سابقا بال testing لل Frontend، لكن العمل مع ال Micro سيضيف تعقيدا إضافيا إلى ال Pipeline عند إجراء الـ End-to-End Testing، وبما أن الـ Unit Testing والـ Integration Testing لا يختلفان عن أي بنية أخرى لـ Frontend، فسوف نركز هنا على الـ End-to-End Testing لأنه التحدي الأكبر والأهم في عالم الـ Micro-frontends ^^

ال End-to-end testing:

إن الغاية من الـ End-to-End Testing اختبار ما إذا كان التطبيق يعمل كما هو متوقع من البداية وحتى النهاية ^^، وهذا كله لضمان الحفاظ على تكامل البيانات بين الأجزاء المختلفة في مشروعنا.

والسؤال الآن، متى يمكننا إجراء الـ End to End testing؟

١. قبل ال deploy على environment مؤقتة يتم إنشاؤها عند الطلب لهذا الغرض.

٢. قبل ال deploy على environment موجودة مسبقا.

٣. قبل ال deploy على الـ production environment... لكن من خلال استغلال ال Feature Flags لتشغيل أو تعطيل بعض المزايا بشكل حسب الحاجة، مع وجود آلية تضمن تحديد المستخدمين الذين سيتمكنون من مشاهدة هذه التغييرات بعد ال deploy... ثم إذا كانت الأمور ممتازة يتم توسعة هذه العملية...

كما تلاحظ هناك فروقات دقيقة بين كل إجراء، وكل إجراء قد يتقاطع مع غيره من ناحية ما، أو أن يتم دمج سلوك معين مع إجراء آخر... وكل هذا مرتبط ارتباطا وثيقا بمستوى التعقيد وخطورة التعديلات ورؤية المؤسسة ونحو ذلك... فمثلا العديد من المؤسسات لا تفضل استخدام ال End to End testing على الـ Production، وتفضل أن يتم ذلك قبل أن تصل النسخة النهائية لل production، لكن هذا سيأخذنا لمعالجة التعقيدات المترتبة على التعامل مع العديد من ال micros الناضج منها وغير الناضج ^^... طبعا لن نتعمق بهذه التفاصيل فهي مرتبطة بموضوع ال Testing نفسه، وهذا الموضوع خارج سياق هذا الكتاب،

لكن تأكد أن ما قلناه هو مخلص مخل، لأن كل إجراء أو سلوك يرتبط ارتباطا وثيقا بعدة عوامل مثل وجود Third party library على ال production ونحو ذلك!

ال Vertical-split end-to-end testing challenges:

شاهد الصورة التالية أولا ثم اتبعني ^^

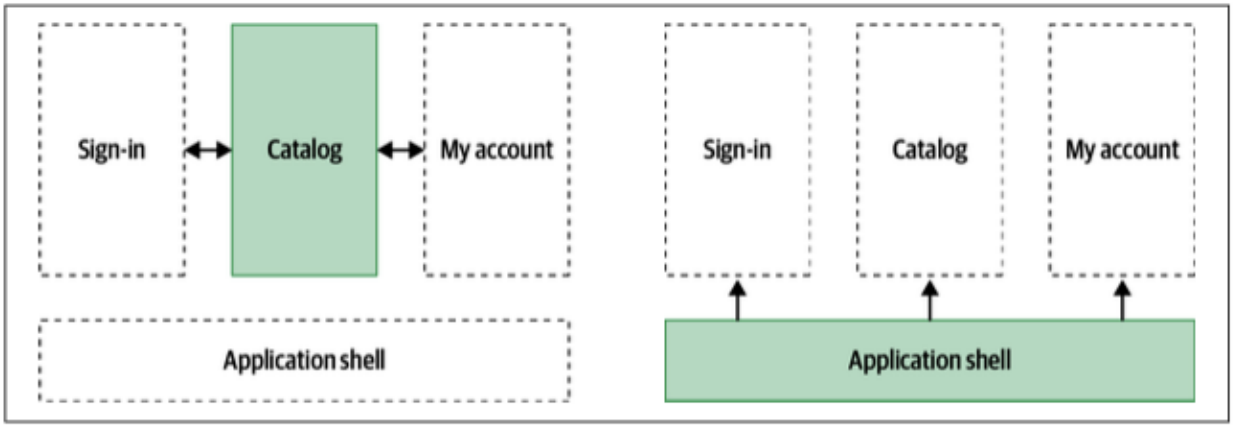


Figure 6-4. An end-to-end testing example with a vertical split architecture

عند اعتمادنا على معمارية ال Vertical Split، فإن العمل يتم تقسيمه بحيث يكون لكل فريق جزء معين من التطبيق يديره كاملا، وهذا يسهل على كل فريق التركيز على اختبار وظائف ذلك الجزء بشكل دقيق كما لو كان تطبيقا مستقلا، والآن افترض كما في الصورة أننا فريق ال Catalog وقام المستخدم بتسجيل الخروج، فما الذي سيحدث؟ وإذا كان المستخدم في ال Catalog وضغط على تغيير معلومات الحساب فما الذي سيحدث؟ إن كل تفاعل من هذه التفاعلات يمثل تحديا للفريق، لذلك يجب على الفريق الذي يعمل على جزء محدد أن يكتب اختبارات تتجاوز حدوده ليضمن أن التطبيق يتصرف بشكل صحيح عند التنقل بين

الأجزاء المختلفة، بالمقابل، الفرق الأخرى التي تدير الأجزاء الأخرى يجب أن تفعل الشيء نفسه من جهتها! وهذا يعني أن عملية تسجيل الخروج ستقودني إلى ال Sign in Micro Frontend، وهنا يلزمني كفريق Catalog من التحقق من عملية تسجيل الخروج تمت وتم تحويلي فعلا إلى آل sign-in.

بالإضافة لذلك لدينا الـ Application Shell ^^، وهو المسؤول عن التحكم في ال global routing الخاص بكل ال micros، وهنا يظهر دور فريق ال App Shell الذي يتوجب عليه أن يختبر أن ال routing تعمل بشكل صحيح، وأن التطبيق يقوم بتحميل الـ Micro-frontend المناسب عند طلب الرابط الخاص به.

ال Horizontal-split end-to-end testing challenges:

شاهد الصورة أولا ثم اتبعني ^-*

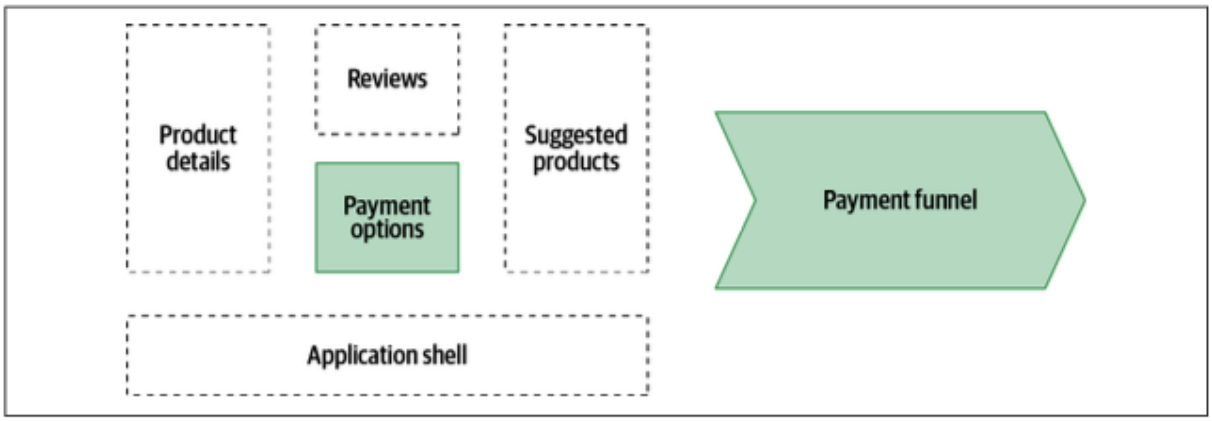


Figure 6-5. An end-to-end testing example with a horizontal-split architecture

إن الـ Horizontal Split يعني أن الـ Micro-frontends قد تظهر في أكثر من مكان أو أكثر من View داخل التطبيق، وهذا يطرح سؤالا مهما: من هو المسؤول عن إجراء اختبارات الـ End-to-End؟؟

من الناحية التقنية فإن ما ناقشناه في تحديات الـ Vertical Split لا يزال صحيحا وموجودا هنا، لكن يضاف لدينا مستوى جديد من التعقيد الذي يتوجب علينا إدارته ^^، فمثلا كما تشاهد في الصورة: فريق الـ Payments قد يكون مسؤولًا عن جزء الدفع الذي يظهر في صفحات مختلفة، وهذا يطرح سؤالا مهما: هل الفريق نفسه سيكون المسؤول عن اختبار كل هذه السيناريوهات التي يظهر فيها جزء الدفع؟ الإجابة نعم للأسف ^^، الفريق سيكون مسؤولا عن التأكد من أن وظائف الـ Micro-frontend التي يديرها تعمل بشكل صحيح في كل الحالات، **لكنه لن يستطيع فعل ذلك بمفرده!** فهذا الأمر يتطلب تنسيقا مستمرا مع الفرق الأخرى ومسؤولية واضحة في توزيع اختبارات الـ End-to-End، حتى لا تتكرر الجهود ولا تتعارض الاختبارات مع بعضها البعض، وحتى تتوزع الأحمال بين الفرق المختلفة! كما أن من التحديات المهمة التي يجب أن نراعيها أن وجود عدة فرق تتشارك في نفس ال View = أن أي تعديل من فريق معين قد يؤثر على صحة ال testing للفرق الأخرى، وهذا قد يجعل الصيانة والتحديثات أكثر تعقيدا، لكن وبالرغم من هذه التحديات؛ يمكن للفرق إجراء اختبارات ال End-to-End بشكل ناجح في معمارية الـ Horizontal Split، لكن يجب أن تكون هناك حوكمة واضحة وتنظيم محكم لتجنب المشاكل التي قد تظهر لاحقا.

ال Testing technical recommendations:

بناء على كل ما ذكرناه سابقا، يمكننا أن نقول هناك ثلاثة طرق رئيسية يمكنك اتباعها لتنفيذ ال End to End testing، وهي:

١. تشغيل ال End to End على environment تحتوي جميع ال micros، وهذا الخيار يضمن أن ال environment مستقرة وفيها كل ال micros, إلا أنها ستبطئ من ال feedback loop.

٢. استخدام environment عند الطلب -on demand- بحيث يتم جمع جميع الموارد المطلوبة لكل test، لكن هذه العملية معقدة ومكلفة خاصة في التطبيقات الكبيرة.

٣. استخدام ال Proxy Server لاختبار ال Micro التي يعمل عليها الفريق، مع تحميل بقية الأجزاء من ال Staging أو ال Production، وجمال هذا الأسلوب يكمن أننا سنكون قادرين على تجربة جميع الحالات وجلب ما نحتاجه بناء على خطوات سير الاختبار!، ويقل التعقيد وال depnedancy الخارجية، وتقل خطورة وجود نسخ غير مستقرة أو الحاجة لإعداد environment على الطلب ^^، ويمكن إعداد ذلك بسهولة من خلال ال WebPack.

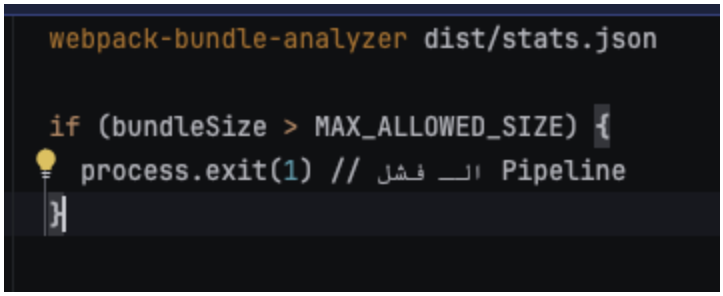
شاهد هذا المثال الذي يوضح كيفية إعداد الـ Web Pack Proxy.

ال Fitness Functions:

من النقاط المهمة التي يجب علينا الاعتناء بها كمطورين هي الذهاب للنادي للحفاظ على لياقتنا البدنية *ـ*... لحظة، يبدو أن العنوان قد أثر علي وذهب باتجاه آخر ^^...

إن عملية الحفاظ على لياقتنا البدنية تشبه ما نريده هنا من الحفاظ على جودة ال architecture الخاصة بنا، والتأكد من وجود آلية موضوعية يمكننا استخدامها لتقييم سلامة ال architecture، وهذه نقطة مهمة... فكيف يمكننا التأكد من أن القرارات التي تم اتخاذها تم اتباعها من قبل جميع الفرق؟ وكيف سنتأكد أن قراراتنا أصلا صحيحة دون أن نتمكن من قياسها؟!

إن الوسيلة التي توفر تقييما موضوعيا لسلامة وجودة بعض الخصائص المعمارية هي ما يطلق عليه ال Fitness Functions ^^، وهناك العديد من الخطوات الموجودة ضمن ال Automation Pipeline التي يمكننا استخدامها لتقييم خصائص المعمارية مثل ال Static Analysis لفحص ال Cyclomatic Complexity أو ال Bundle Size... مثلا، إذا كنا نتحدث عن micro frontend وتم وضع معيار لحجم البيانات التي سيتم تحميلها من قبل المستخدمين والتي لا ينبغي لنا تجاوزها فإن وجود Fitness Function هنا سيكون أمرا ممتازا، وعليه سنقوم بقياس حجم ال Bundle الصادر ومقارنتها مع الحجم المسموح به، شاهد هذا المثال:



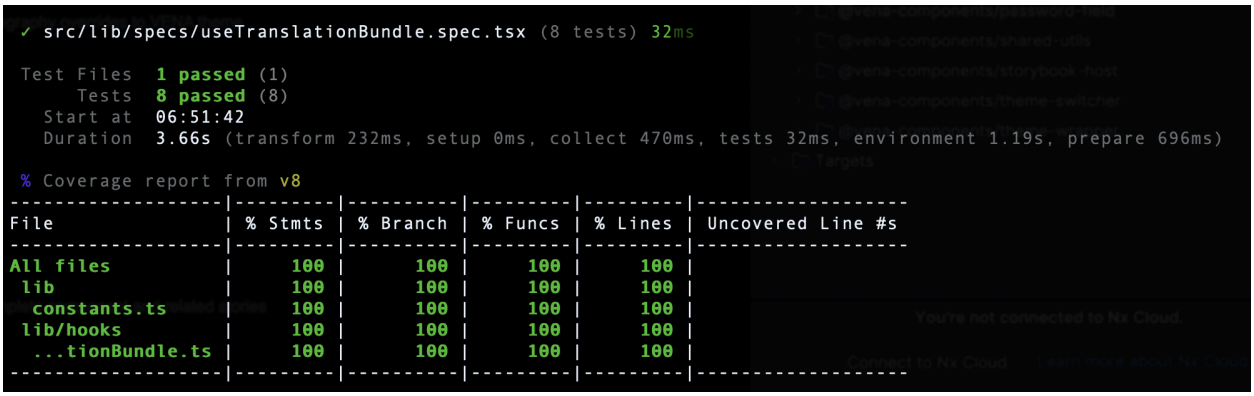
والآن، دعونا نستعرض بعض الخصائص المعمارية المهمة والتي يجب أن ننتبه إليها عند تصميم ال Automation Pipeline لمشروع Micro Frontend:

١. ال Bundle Size: يجب علينا تخصيص حجم محدد لكل Micro-Frontend وتحليل النتائج إذا ما تم تجاوز الحد مع معرفة السبب، وفي حالة المكتبات المشتركة فيجب علينا مراجعة حجم كل المكتبات المشتركة.

٢. ال Performance Metrics: يمكننا استخدام أدوات مثل ال Lighthouse للتحقق مما إذا كان الإصدار الجديد من التطبيق يحافظ على نفس مستوى الأداء أو أفضل مقارنة بالإصدار الحالي.

٣. ال Static Analysis: توجد العديد من أدوات الـ Static Analysis في بيئة JavaScript مثل ال Qodana والتي يمكنك من خلالها حساب ال Cyclomatic Complexity مثلا، وبناء على ذلك يمكننا فرض معايير صارمة لمنع إكمال Pipeline إذا تم تجاوزت الشيفرة البرمجية مستوى التعقيد القياسي... وللمزيد من المعلومات حول ال Cyclomatic Complexity يمكنك تحميل هذا الكتاب أو الذهاب مباشرة إلى الصفحة رقم ٥٣.

٤. ال Code Coverage: يتم هنا تحديد نسبة مئوية للاختبارات التي تم تنفيذها على المشروع، بحيث نتأكد قدر الإمكان أن أغلب الوظائف الموجودة تمت تغطيتها ب test... لا يوجد ضمان هنا لجودة ال test، لكنه يقدم لمحة عامة عن مستوى التغطية، وعادة ما تكون نسبة التغطية هي ٨٠٪ وفوق، وعادة ما يكون أكثر من ٩٥٪ لا داعي له إلا بظروف معينة، شاهد الصورة أدناه:



٥. ال Security: وهنا يتم التأكد من أن الشيفرة البرمجية لا تنتهك أي قوانين أو قواعد وضعها فريق الـ Security أو الـ Architecture.

ال Deployment Strategies:

في عالم ال Micro frontend تعد عملية ال deploy واحدة من أهم المراحل والمحطات التي نمر بها، لذلك كانت الاستقلالية في النشر -والتي تحدثنا عنها سابقا- واحدة من أهم العناصر أو المرتكزات الأساسية لذهابنا لعالم ال micro frontend وتقسيم المشروع بناء على ذلك! وهنا نؤكد مجددا على أن ال Micro frontend يجب أن تكون مستقلة تماما عن

بعضها البعض، بحيث يمكننا نشر أو تحديث micro دون الحاجة لتحديث الـ micros الأخرى... وتأكد أنك في اللحظة التي تحتاج فيها إلى "تنسيق" عملية النشر بين عدة micro frontend فإنها اللحظة المناسبة والنقطة الحرجة لإعادة النظر في القرارات التي اتخذناها في معماريتنا والعودة لرسم الحدود مجددا... فاحذر من ال coupling لأن تأثيرها خطير جدا...

بعد هذه المقدمة البسيطة والمراجعة السريعة لما ذكرناه سابقا، فإننا يجب أن نهتم أيضا بالاستراتيجيات التي يمكننا استخدامها عند عملية النشر لأي micro جديدة، وهذا مهم ومفيد جدا لتجنب أي مشكلة ممكن حدوثها عن المستخدمين... وهناك أساليب متنوعة يمكننا استخدامها للحصول على أفضل تجربة بأقل مخاطر ممكنة، وهذا ما سنتعرف عليه في هذا الموضوع ^^

ال Blue-Green Deployment Versus Canary Releases:

في هذا الباب سنتعرف على استراتيجيتين جميلتين: ال Blue-Green وال Canary Releases ...^^

١. ال Blue-Green: هذه الاستراتيجية قائمة على مبدأ تقليل وقت التعطل (ال downtime) وعدد المخاطر المحتملة، لذلك يتم تشغيل اثنتان من ال environment، يطلق على الأولى: البيئة الزرقاء وعلى الثانية البيئة الخضراء ^^...

البيئة الزرقاء تمثل الإصدار الحالي أو النسخة التي يستخدمها الناس في الوقت الحالي، في حين تشير البيئة الزرقاء إلى الإصدار الجديد الذي نرغب بتحويل الناس إليه، وسير العمل في هذا الأسلوب يتم من خلال رفع النسخة الخضراء على ال Production وعمل test كامل لها وقبل توجيه الناس إليها... ثم بعد التأكد من عملها بشكل صحيح نقوم تغيير اتجاه المستخدمين من النسخة الزرقاء إلى النسخة الخضراء دفعة واحدة ^^، وذلك يتم من خلال ال Router.

شاهد الصورة F6-6:

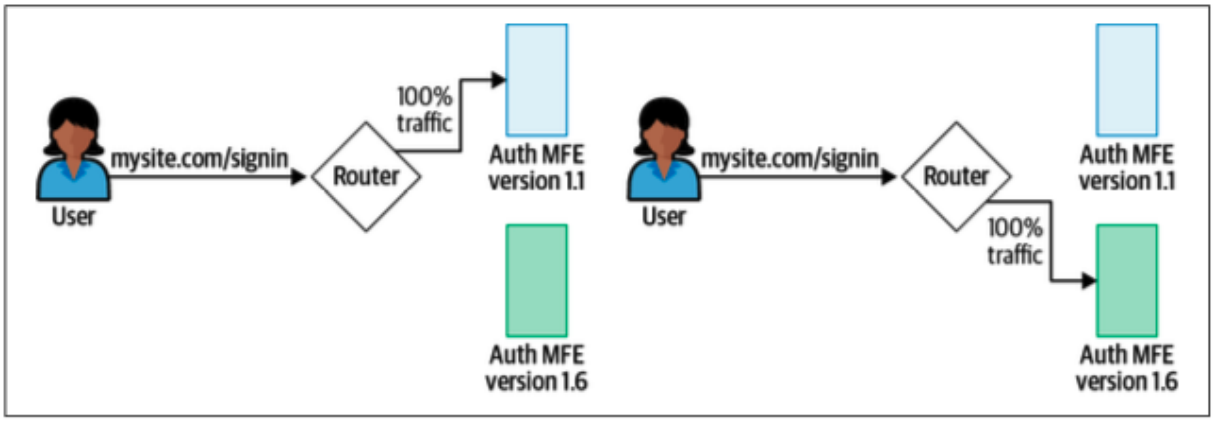


Figure 6-6. Blue-green deployment

إن من أهم القوائد التي يقدمها هذا النهج هو سرعة ال Rollback العالية، فكل ما يلزم للقيام بذلك عند حدوث مشكلة معينة هي إعادة توجيه المستخدمين إلى البيئة الزرقاء... كما أنها تقلل من المخاطر المحتملة لأن النسخة الجديدة تم اختبارها فعلا على ال Production...

٢. ال Canary Releases: هذه الاستراتيجية قائمة على مبدأ لا تنام بين القبور ولا تحلم أحلام مزعجة *ـ*... أي أنها قائمة على مبدأ نشر التحديثات الجديدة بشكل تدريجي ومراقب، فبدلا من تحويل كل المستخدمين إلى النسخة الجديدة -الكناري-؛ يتم تحويل نسبة محددة أو أشخاص محددين فقط، ثم يتم زيادة هذه النسبة وعدد الأشخاص مع مرور الوقت والتأكد من كل شيء، لهذا يعد عنصر الرقابة هنا مهم جدا ومركزي، وأكثر ما يهتم به هنا هو ملاحظة أي مؤشرات قد تظهر مثل زيادة عدد الأخطاء الظاهرة بال log أو بطء الاستجابة أو انخفاض عدد المستخدمين... إلخ، كما أن التعامل هنا مع المشاكل سهل، فإذا لوحظت أي مشكلة يتم تحويل المستخدمين مجددا إلى النسخة القديمة...

شاهد الصورة F6-7:

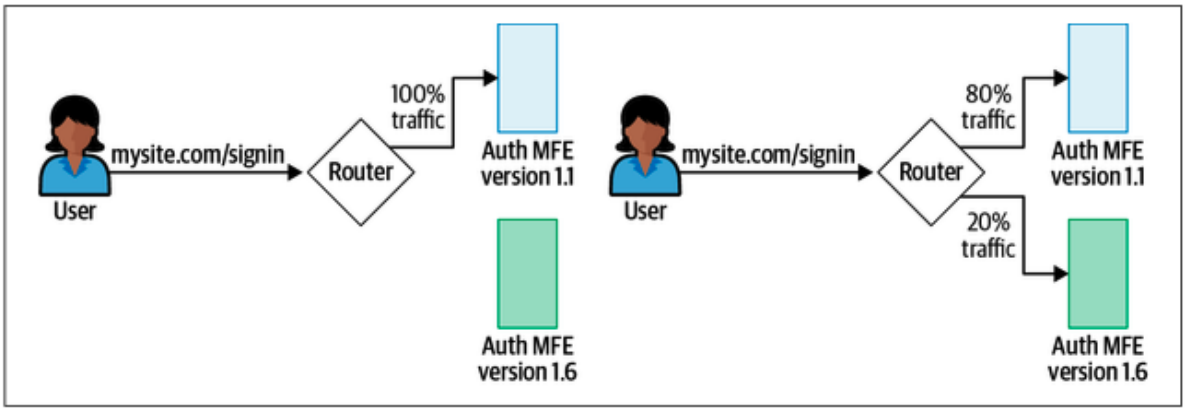


Figure 6-7. Canary release

ملاحظة مهمة: لاحظ أن ال Router موجود في كلا الأسلوبين، وال Router هنا يشير إلى مفهوم عام يستخدم للتعبير عن العديد والكثير من الوسائل التي يمكننا استخدامها لإعادة توجيه الناس، وهذا يرتبط ارتباطا وثيقا بالآلية التي سنعتمدها لإعادة التوجيه ومكان وضع هذه الآلية... مثلا: إذا كان الأسلوب الذي اعتمدناه لدينا هو ال client side؛ فيمكننا وضع عملية التحويل في ال app shell... أي أن عملية التوجيه هنا ليست بالضرورة أن تكون Hardware، بل يمكن أن يكون logic برمجي تم بناءه لتحقيق هذا الغرض.

شاهد الصورة T6-1 والتي تمثل مجموعة من الخيارات الممكن استخدامها لتحقيق هذه الغاية...

Table 6-1. Router options available for canary releases and blue-green deployments

	Blue-green deployment or canary release mechanism
Client-side routing	Application shell Configuration passed via static JSON or backend APIs
Edge-side routing	Logic running at the edge (e.g., AWS Lambda@Edge)
Server-side routing	Application server logic API gateway Load balancer

ال Strangler Pattern:

إن هذه الاستراتيجية مهمة جدا للشركات التي لديها بالفعل مشروع قائم وعليه الكثير من المستخدمين... إن ال Blue-Green وال Canary Release مناسبة في حال كانت ال micro frontend تم نشرها في ال production وتعمل حقيقة، لكن ماذا لو كنا نحتاج إلى استبدال الموقع القديم بموقع جديد يعتمد على هذه المعمارية -mfe-؟!

وهنا يأتي دور ال Strangler Pattern، وهو أحد الحلول الجميلة التي تفضلها الشركات، والذي يستخدم بال micro-service وقمنا بسرقته هنا أيضا :P...، وال Strangler Pattern هو استراتيجية تستخدم لتحديث أو إعادة بناء نظام برمجي كبير وقديم - Legacy Application- عن طريق استبدال أجزاء منه تدريجيا بأجزاء جديدة مبنية على بنية حديثة مثل الـ micro-frontends.

وفكرته الأساسية قائمة على مبدأ إنشاء تطبيق جديد بجانب التطبيق القديم، ومن ثم البدأ بإعادة كتابة وظائف معينة من التطبيق القديم في micro frontend في التطبيق الجديد، أي عملية نقل تدريجية للوظائف من الموقع القديم إلى الجديد بدلا من الانتظار لشهور أو سنوات حتى يتم الانتهاء من النسخة الجديدة... عند اكتمال نقل وظيفة معينة يتم خنقها في المشروع القديم وتوجيه المستخدمين للنسخة الجديدة منها، حتى يتم حنق جميع الوظائف القديمة ^^، وهو ما يشبه ما تفعله الأشجار في الغابات الاستوائية، ومنها أخذ الإسم... انظر للصورة التالية:

إذا كيف يعمل هذا الأسلوب؟

١. لدينا تطبيق كامل يعمل يمثل النسخة القديمة ويجب أن نحافظ عليه طوال فترة العمل.

٢. يتم البدأ بعملية التطوير التدريجية، بحيث يتم اختيار جزء صغير من المشروع له قيمة عالية في المشروع مثل صفحة تسجيل الدخول، ويتم إعادة بنائها من خلال ال Micro frontend.

٣. يتم نشر الجزء الجديد على ال production في منصة خاصة له.

٤. يتم إعادة توجيه المستخدمين من الجزء القديم إلى الجزء الجديد، وفي مثالنا من صفحة تسجيل الدخول القديمة إلى تسجيل الدخول التي بناؤها على شكل micro.

٥. تستمر هذه العملية حتى يتم استبدال جميع الخصائص واحدة تلو الأخرى حتى نرى أن جميع المستخدمين صار يتم تحويلهم إلى النسخة الجديدة.

هل عرفت الآن لماذا تحب هذا الأسلوب العديد من الشركات؟!

إن هذا الأسلوب يقدم قيمة مستمرة للمستخدمين، فعملية التغيير والتطوير تتم بشكل مستمر بدلا من الانتظار الطويل والجمود حتى تكتمل النسخة الجديدة، كما أنه يقلل من ال Big-bang failue لأنه يسمح لك باختبار وتجربة الأجزاء الجديدة في ال production أولا بأول ودون التأثير على النظام بأكمله، ويمكن إرجاع المستخدمين للنسخة القديمة في حالة وجود مشاكل، وهذا كله يسمع للمطورين بالتعلم من الأخطاء أولا بأول وتحسين أساليبهم في التعامل مع التحديات المختلفة...

لكن كما هي العادة، الحياة ليست وردية، والكمال لله -سبحانه وتعالى- وحده، فلدينا تحديات خطيرة، أهمها التعقيد، فهذه الاستراتيجية ستزيد من التعقيد الخاص بالمشروع، لأن ذلك غالبا ما سيتطلب تعديلا على النسخة القديمة حتى تتمكن من التعامل مع النسخة الجديدة، كما أننا نحتاج لبناء Router للتحكم بعملية التوجيه بين النسختين [٨]، وهذا سيقودنا لتعقيد جديد وهو حاجتنا للاحتفاظ بأكثر من نسخة وغالبا ما قد تكون النسخة القديمة والنسخة الجديدة المكتملة ونسخة hybird والتي فيها النسخة القديمة مع ال micros التي كان يتم إضافتها، وذلك كله لضمان القدرة على التراجع السريع عند وجود أي مشكلة أو في حالات الطوارئ...

شاهد الصورة F6-8:

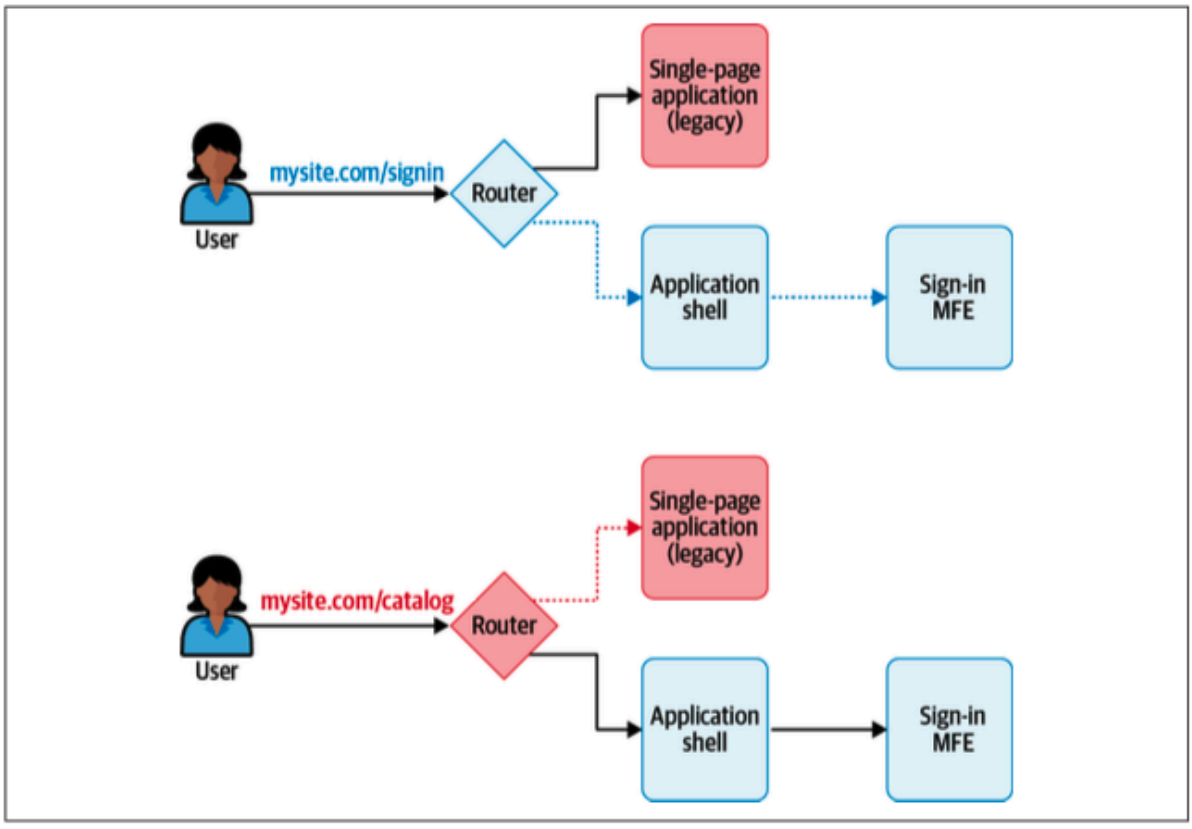


Figure 6-8. A strangler pattern where the micro-frontends live alongside the legacy application so that we can create immediate value for the users and the company instead of waiting for the entire application to be developed

ال Observability:

إن آخر نقطة مهمة لدينا في هذا الباب، هي القدرة على مراقبة النظام والتحقق من صحة النظام والقدرة على تتبع الأخطاء وتحليل الأداء ونحو ذلك.

تكمن أهمية هذا الجزء في باب النشر لأنه:

١. هي المرحلة الأخيرة في ال Feedback loop، والتي من خلالها تغلق هذه الحلقة، فبعد نشر أي جزء جديد على ال Production يجب علينا التحقق من أنها تعمل بشكل صحيح... من دون وجود آلية للمراقبة = فلن تعرف إذا حدثت هناك مشكلة أم لا.

٢. تصحيح الأخطاء بسرعة، فبدلا من محاولة البحث عن الأخطاء وإعادة صناعتها على ال local device -وقد يكون الأمر مستحيلا-، فإن وجود أدوات تحفظ سير عمل المستخدمين وتحفظ ال Stack Trace سيكون له أثر إيجابي على قدرتنا في معالجة الأخطاء... وهذا سيقودنا لقدرتنا على تحديد المشكلة بدقة ومنها تحديد مكان المشكلة ونسبتها لأي frontend أو end أكثر...

هذه النقطة ليست رفاهية! بل هي ضرورة قصوى خصوصا في المشاريع الكبيرة... لذلك حاول ألا تتجاهلها *-*.

فائدة

المصلح ليس مطالباً بأن يعرف كل الخطوات والمراحل الإصلاحية التي سيسلكها، وإنما عليه أن يبذل ما عليه، ويفعل ما يمكن، فإذا أُغلقت الأبواب في وجهه كما أَغلَق البحر الطريق على موسى -عليه السلام-؛ فإنه مطالب بالاستهداء والتوكل، إذ إن موسى لم يكن يعلم المخرج من هذا الإغلاق حتى أوحى الله إليه بأن يضرب بعصاه البحر فانفلق فكان كل فرق كالطود العظيم.

- مما تعلمته من كتاب أنوار الأنبياء، أحمد بن يوسف السيد

الفصل السادس: Automation Pipeline for Micro-Frontends: A Case Study

الآن بعد أن ناقشنا في الفصل السابق بشكل نظري حالات تطبيق الأوتوميشن والخيارات الممكنة والخصائص المختلفة والفروق التي بينها، دعونا ننتقل لمثال تطبيقي تخيلي لتطبيق تلك المفاهيم النظرية على مشروع يعتمد على ال micro-frontend، وتذكر أن ما سنقوم بافتراضه هنا هو اختيار لغايات العرض، لكن في المشاريع الحقيقية يجب اختيار التقنيات والأساليب المناسبة للمشروع... باختصار لا تأخذ ما سنكتبه هنا من أساليب كأنه المعيار الصحيح الذي يجب عليك اتباعه في مشروعك!

لنفترض أن لدينا مؤسسة اسمها قبيلتكم *_*، هذه المؤسسة تقدم العديد من الخدمات المهمة، وهي في مرحلة تمكن مطوريها وتثق بهم لمعرفة وتحديد الأدوات الأفضل التي يجب استخدامها بال automation pipeline، والمؤسسة قامت بوضع ال guardrails بشكل واضح، والمطورين يدركون هذه الحدود ويعملون بناء عليها...

هيكلة المشروع وفلسفة العمل لديهم:

١. تم اعتماد ال Vertical Split للمشروع، حيث كل فريق مسؤول عن بناء وتطوير صفحة أو جزء كامل من التطبيق، وكل micro ستتكون من HTML, CSS, JS خاصة به.

٢. اختيار الأدوات التقنية المناسبة سيكون منوطا للفرق ضمن القواعد العامة للمؤسسة.

٣. تم اعتماد ال monorepo كاستراتيجية لل repo الخاصة بالمشروع.

٤. تم اعتماد ال Trunk-based Development ك branch strategy.

٥. تم اعتماد أسلوب ال fix-forward لمعالجة الأخطاء، حيث يتم معالجة الأخطاء مباشرة ونشرها مباشرة.

٦. تم تحديد ال enviroment المطلوبة للمشروع لتكون كما يلي:

*. ال DEV: وهي بيئة التطوير وتعمل بنمط ال continuous deployment.

*. ال STAGE: وهي بيئة ما قبل ال Production، وتستخدم للاختبار والتأكد من أن الكود يلبي المتطلبات الخاصة بال Business، وسيقوم فريق ال UAT من التأكد من هذه المتطلبات.

* ال PROD: وهي المكان النهائي لعرض المشروع للمستخدمين ^^.

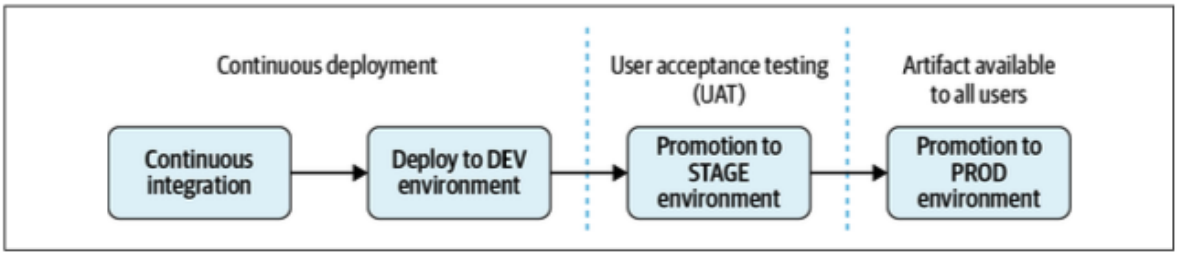


Figure 7-1. An example of an environments strategy

٧. ال automation pipeline مصمم لضمان جودة الشيفرة البرمجية، مع وجود لوحة تحكم داخلية تسمح للمطورين وفرق الجودة بالتحكم في عملية ترقية ال artifact من بيئة إلى أخرى.

٨. يجب على الفرق أن تقوم بإجراء جميع أنواع الاختبارات، وهي ال Unit testing وال Integration testing وال End to End testing.

٩. يجب إضافة ال Fitness Function، ولقد قام فريق ال architecture بوضع القواعد المناسبة ليتم مراقبتها والتحقق منها، وأن كل شيفرة برمجية جديدة تلتزم بما تم الاتفاق عليه.

١٠. في هذه المرحلة سيتم تجنب استخدام الـ Feature flag لتجنب التعقيد الإضافي، وسيتم تجنب ال Release Branch لأننا توجهنا لاستخدام ال Trunk-Based.

١١. عملية النشر سيتم التحكم بها من خلال dashboard، مما سيسمح لنا بمراجعة وتتبع كل خطوة...

إذا، بناء على هذه المعطيات، سيكون ال High level automation strategy كما الصورة:

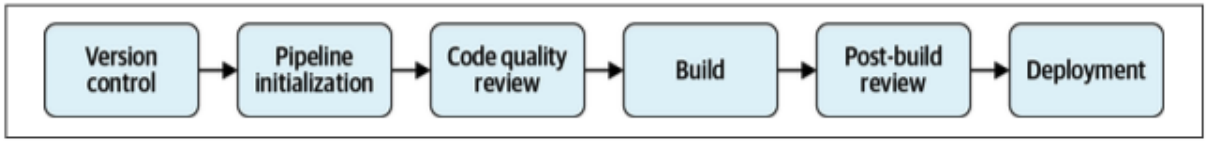


Figure 7-2. High-level automation strategy design

والآن سنذهب للحديث عن هذه الخطوات:

1. ال Version control

2. ال Pipeline initialization

3. ال Code-quality review

4. ال Build

5. ال Post-build review

6. ال Deployment

معلومة: Defect Costs Rise over Time -ارتفاع تكاليف العيوب بمرور الوقت-: يشير هذا المفهوم إلى أن تكلفة إصلاح الأخطاء البرمجية تزداد بشكل كبير وأحيانًا بشكل أُسّي كلما تم اكتشافها في مرحلة متأخرة من دورة حياة التطوير، وهو مبدأ أساسي من مبادئ هندسة البرمجيات... باختصار إن هذا المبدأ يؤكد على أهمية اكتشاف الأخطاء ومعالجتها في المراحل المبكرة من التطوير، مثل مرحلتي التصميم والتطوير، بل إن تكلفة الإصلاح قد تتضاعف إلى ٢٥ ضعف إذا تم اكتشاف المشكلة على ال Production مقارنة بمرحلة التصميم! وتخيل أن هناك معاهد مثل IBM أوصلت الرقم ل ١٠٠ ضعف أو أكثر!

١. ال Version Control:

قلنا أن شركة قبيلتكم اعتمدت ال Monorepo كاستراتيجية لل Repo الخاصة بها، وهناك العديد الأدوات التي يمكننا استخدامها لمساعدتنا في إدارة الـ monorepoe مثل ال Lerna وال NX وال Turborepo... وسنقوم باعتماد ال NX كأداة لمشروعنا...

إن هذه الأدوات تقدم مزايا ممتازة لإدارة ال monorepo، من أهمها ال Housing، والتي يقصد بها عملية إدارة ال dependency، فمثلا بدلا من تنزيل نفس المكتبة لكل micro frontend، سيتم تنزيلها مرة واحدة على مستوى ال root directory، واستخدامها من قبل جميع المشاريع، وهذا يوفر مساحة التخزين ويسرع من عملية التحميل... وهذا أيضا سيقودنا لإمكانية تخزين ال vendors التي نعمل عليها، مثلا يمكننا تخزين ال react على cdn لفترة طويلة لأن احتمالية تغييرها قليل أو وقت تحديثها قد يكون طويل لاستقرار هذه المكتبات... وهذا سيقلل الضغط ويزيد من سرعة التحميل...

كما أننا سنقوم باستخدام ال github كنظام للتحكم في الريبو والأتمتة ونحو ذلك، وهو يقدم ميزات كبيرة منها وجود ال Github Marketplace والذي يتيح لنا استخدام سكربت جاهزة لتنفيذ مهام محددة في ال pipeline مثل إضافة سكربت لل linting... أو يمكننا كتابة script خاصة بنا لتنفيذ هذه المهام...

سيتم تطبيق قواعد ال linting عند كل commit جديدة، وسيتم تشغيل ال test check عند فتح أي PR أو تحديثه...

٢. ال Pipeline initialization:

مرحلة تهيئة ال Pipeline هي الخطوة الأولى التي يتم تنفيذها تلقائيا لكل ال micro-frontend قبل بدء أي عمل آخر، وهدفها الأساسي هو تجهيز بيئة العمل للخطوات التالية.

هذه المرحلة تشمل عدة إجراءات شائعة يتم تنفيذها لكل ال micro-frontend، بما في ذلك: ال Cloning لل micro-frontend repository داخل ال container، وتثبيت جميع ال dependencies اللازمة لها...

انظر إلى الصورة F7-3:

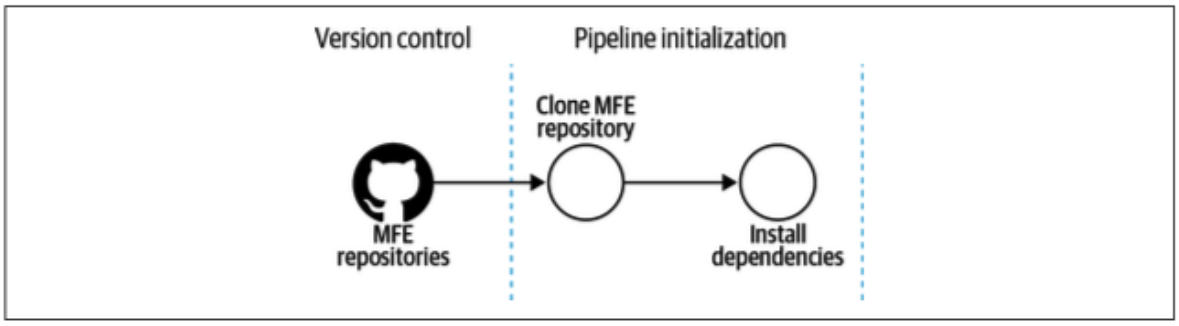


Figure 7-3. Pipeline initialization stage, showing two actions: cloning the repository and installing the dependencies

في هذه الصورة يمكننا أن نرى الجزء الأول من ال automation pipeline حيث نقوم بتنفيذ إجراءين رئيسيين: cloning لل micro-frontend repository وتثبيت ال dependencies عبر أوامر ال yarn أو ال npm...

أهم ما يجب علينا مراعاته هنا هو جعل عملية ال repository cloning عملية سريعة قدر الإمكان، فنحن لا نحتاج إلى كامل تاريخ ال repository عند تشغيل عملية ال CI process، لذلك من الممارسات الجيدة استخدام خيار command depth لجلب آخر commit فقط، خصوصا عند استخدام ال monorepo، نظرا لأن هذه ال repository قد يزداد حجمها بسرعة كبيرة ^^... شاهد المثال:

```
- git clone --depth 1 https://github.com/qabelatomAccount/qabelatqomRepo
```

٣. ال Code-quality review:

في هذه المرحلة من ال pipeline سنقوم بالتحقق من جودة الشيفرة البرمجية ومراجعتها والتأكد من كونها تلتزم بمعايير الشركة، شاهد الصورة أدناه:

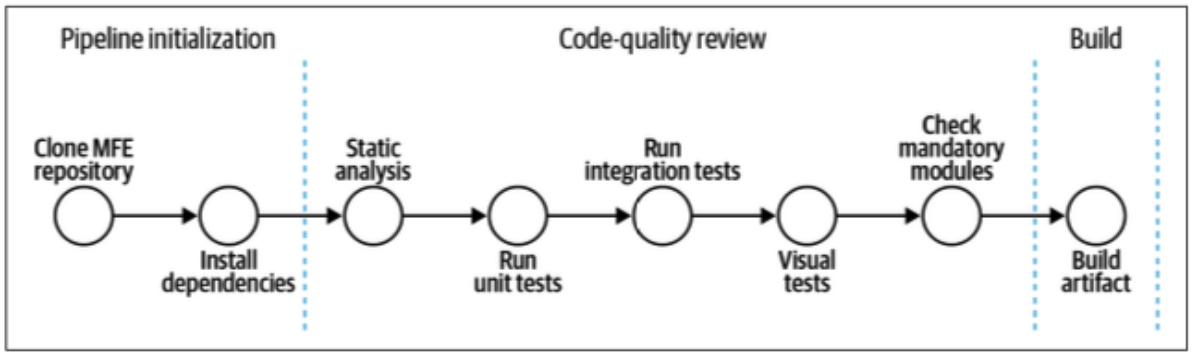


Figure 7-4. Code-quality checks like unit testing, static analysis, and visual regression tests

كما تلاحظ في الصورة، فبعد ال Pipeline initialization لدينا العديد من الخطوات التي يجب علينا القيام بها قبل ال build، وهي:

*. ال Static Analysis، وهنا ستتم عملية فحص الشيفرة البرمجية بهدف اكتشاف الأخطاء المحتملة أو المتعلقة بالجودة -كما تحدثنا سابقا-، ومنها سنراقب ال Cyclomatic Complexity، فهو سيساعدنا على فحص جودة الشيفرة البرمجية من خلال وجود آلية قياس

ملموسة -ولقد تحدثنا عن ذلك من قبل-... إذا وجدنا أن التعقيد كبير، فسيتم إيقاف ال pipeline لحل المشكلة...

*. ال Run unit test وال Integration Test: سيتم تطبيق هذه الآليات من خلال استخدام ال jest، وهي من المهام ذات الأولوية، والتي لا تعطيها اهتماما كثير من الشركات للأسف! كما أن عملية الفحص هنا يجب أن تكون سريعة أو أسرع من التطبيقات التي بنيت monolithic لأن الأجزاء المراد فحصها هنا صغيرة، وبنائها كذلك أسهل مما لا يترك عذرا لتجاهلها...

*. ال Visual Tests: هي واحدة من أنواع الاختبارات التي يمكن استخدامها عن طريق أخذ screenshot لأجزاء من واجهة المستخدم أو الواجهة كاملة ومقارنتها مع النسخة السابقة، وهذا لضمان أن أي إصلاح لم ينتج لنا مشكلة جديدة... وهي مفيدة في ال micro تحديدا لضمان أن جميع ال micro لم تتأثر بتعديل أحد ال micro الموجودة...

*. ال Check Mandatory Modules: هنا سنقوم بالتحقق من جميع ال dependency الموجودة، مثل التأكد من الإصدارات المستخدمة وأنها موحدة مع جميع ال micro، وأن

هناك مكتبات إلزامية تم تحميلها في كل micro مثل مكتبات ال log وال observability... والآلية لذلك بسيطة من خلال التحقق من ال package.json...

ملاحظة مهم: لاحظ أن كثير من الوظائف الموجودة والتي تحدثنا عنها تعبر عن تطبيق عملي لل Fitness Function ^^ مثل ال Cyclomatic Complexity وال Check Mandatory Modules...

٤. ال Build:

في هذه المرحلة سنقوم بإنشاء ال artifacts التي سيتم إرسالها لل production، وسنقوم باستخدام ال Webpack لأجل هذه الغاية.

ستشمل هذه المحطة عملية تحويل ال code من source code إلى build code وعمل minifying له بحيث نتخلص من المسافات الزائدة وال comments والرموز غير الضرورية... إلخ.

ملاحظة: في هذه الخطوة يمكن للفرق أن تجرب أو تقترح أدوات أخرى غير ال Webpack، ويمكن مقارنة النتائج... لكن كل ذلك يجب أن يكون ضمن الحدود التي رسمتها الشركة... ومع ذلك، يفضل أن تتفق جميع الفرق على استخدام أداة موحدة لل build، مع وجود مرنة للتغيير إن وجد خيارات أفضل.

٥. ال Post-Build Review:

تعد هذه المرحلة من المراحل الحاسمة قبل إعطاء الموافقة النهائية والذهاب إلى ال Production، ويتم فيها التأكد من أن الكود فعلا جاهز للنشر ولا خوف من تصديره...

لذلك، سنجد في هذه المرحلة عمليات أساسية وهي:

*. ال Artifacts Storage: هذه الخطوة من الخطوات المهمة، ففيها سنقوم بحفظ نسخة من ال artifacts التي تم إنتاجها داخل repo مخصصة لهذا الغرض، ويمكن استخدام أدوات مثل Nexus لتحقيق هذه الغاية... كما يمكن استخدام حلول أبسط مثل S3... المهم هو وجود Single source of truth لتخزين جميع النسخ الناتجة ليسهل متابعتها وإدارتها...

*. ال End to end testing: لأننا اعتمدنا أن المشروع هو Vertical split، فعملية ال end to end testing ستكون في هذا المكان، وبهذا نضمن أن جميع الأجزاء ما زالت تعمل كما هو متوقع... وهنا اعتمدت قبيلتكم إنشاء on-demand environment لهذا الغرض... -تذكر أن هناك ميزات وعيوب وبدائل لكل أسلوب، يمكنك العودة لتلك المفاهيم-... أما في حالة ال Horizontal split، فإن هذه العملية قد يتم تأجيلها لل Production، وتذكر أننا في هذه الحالة سنحتاج إعداد ال feature flag وال mock data عند الحاجة... -راجع ما قلناه سابقا-.

*. ال performance checks: عملية التحقق من الأداء عملية مهمة جدا لضمان أن النسخ النهائية الجديدة ما زالت تلبي متطلبات النظام، وضمن الحدود التي رسمتها الشركة، وإضافة هذه العملية لم تعد عائقا أو عملية صعبة، بل هي عملية سهلة، ومن الأدوات المشهورة لتحقيق هذه الغاية ال lighthouse التي تم تصميمها من جوجل... وهذا سيتيح لنا التحقق من الأداء وال accessibility ونحو ذلك.

شاهد الصورة F7-5

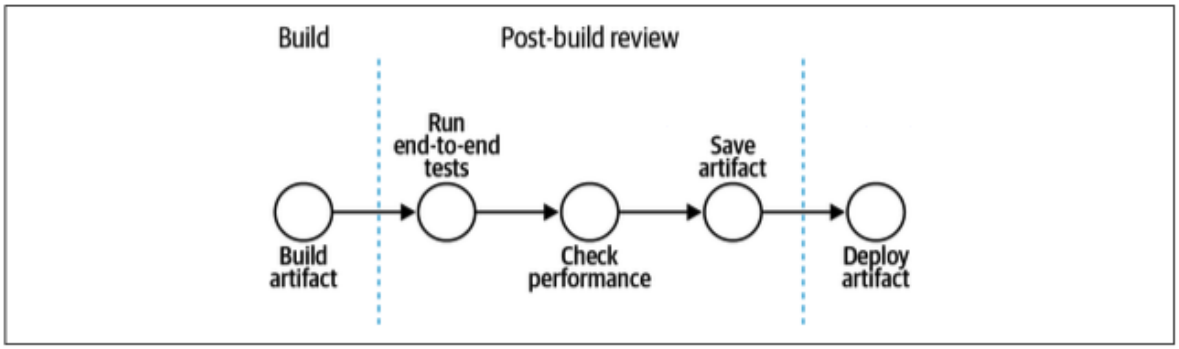


Figure 7-5. In the post-build review, we perform additional checks before deploying an artifact to an environment

٦. ال Deployment: وهي المرحلة الأخيرة في محطتنا ^^... وفي هذه المرحلة سيتم تسليم النسخة الجديدة للمستخدمين من خلال نشرها على ال Production، وهنا يجب علينا أن نراعي تسلسل خطوات النشر وآلية وعمل النشر والتحكم في الإصدارات... في هذه المرحلة

وخصوصا في البدايات، تجنب التعقيد الكبير والاستثمار الكبير قبل فهم الاحتياجات الفعلية للمشروع... ابدأ بأبسط الطرق وأرخصها ^^

الفصل السابع: Backend Patterns for Micro-Frontends

هناك اعتقاد شائع عند كثير من المطورين بأن الـ micro-frontends لن تنجح إلا إذا كانت مقترنة بالـ microservices، وذلك لضمان وجود استقلالية كاملة لكل جزء من التطبيق، وقد يعتقد البعض أن التطبيقات الضخمة المصممة monolith لا يمكن أن تتوافق أبدا مع الـ micro-frontends... لكن هذا كله غير صحيح، فالواقع مختلف عن هذه التخيلات والتحليلات ^^! فالـ micro-frontends تتمتع بمرونة كبيرة، ويمكنها العمل مع أنواع مختلفة من الأساليب الموجودة بال BE.

لذلك، سنتعرف في هذا الفصل كيف يمكننا أن نبني نظام يحقق التكامل بين ال BE وال FE، وسنتعمق في كيفية عمل الـ micro-frontends مع ال monoliths ومع الـ microservices وال BFF...

ال API Integration and Micro-Frontends:

هناك الكثير من الأساليب والأنماط المختلفة التي يمكن استخدامها للتعامل مع ال API من خلال ال micro frontend، وسنأخذ نحن منها ثلاثة أنماط مختلفة وهي:

١. ال Service dictionary:

وفكرة هذا النمط ببساطة هي جلب روابط ال API من خلال ملف معين موجود على السيرفر أو API مخصصة لهذا الغرض، وآلية العمل هنا هي أن ال micro frontend تقوم

بجلب الروابط من خلال تحميل ال Service dictionary... هذا الملف الذي تم تحميله فيه كل الروابط التي نحتاجها كاملة، ثم يقوم ال frontend باستدعاء الرابط المناسب حسب الخدمة التي يحتاجها...

ملاحظة مهمة: هذا الأسلوب ليس مخصصا لل micro-frontend، بل هو أسلوب يستخدم في حالة ال monolithic بالأساس! ويمكن استخدامه هنا... ما أقصده أن هذه الأفكار ليست بالضرورة أن تكون خاصة بال micro frontend، بل هي أفكار يمكن أن تستخدم في أكثر من معمارية أو مكان أو أسلوب بحسب حاجة النظام ومتطلباته... فلا تغلق عقلك ^^.

2. ال API Gateway:

تمثل ال API Gateway واحدة من الطرق المشهورة للتعامل مع ال request القادمة من ال clients إلى ال server، وهي تمثل Single Entry point، أي نقطة وصول واحدة لأي micro service سنقوم باستخدامها، فمثلا إذا كان لدينا هذه ال microservice أو أكثر: micro1.2nees.com/shop/33 و micro2.2nees.com/product/1 فإننا من خلال ال API gateway سنقوم ببناء نقطة وصول تسمى api.2nees.com وهي من ستقوم بالتواصل مع ال micro service، لتصبح النتيجة api.2nees.com/product/1 و api.2nees.com/shop/33 ^^

ومن الفوائد المهمة لل API Gateway التحكم بال Routing من خلال مكان واحد بدلا من أن يقوم بذلك ال client، والتحقق من ال Auth وضبط ال rate limiter ومراقبة المشاكل ونحو ذلك.

٣. ال BFF: ال BFF هو نمط تم تطويره على غرار ال API Gateway، لكنه بدلا من أن يخدم جميع واجهات ال Client من خلال API Gateway واحدة يقوم ببناء APi Gateway مخصصة لكل نوع... مثلا واحدة ل Web وواحدة لل Mobile وواحدة لل Iot...

إن أهم ما يقدمه هذا الأسلوب هو القدرة على تخصيص البيانات حسب حاجة ال Client، فمثلا قد تحتاج ال Iot معلموات قليلة فقط ومحددة، في حين أن ال mobile قد يحتاج معلومات أكثر تفصيلا إلا أنها قد تكون مبنية بهيكلة مختلفة للبيانات وبشكل أقل تفصيلا من الويب... وهذا بالضرورة سيقلل من عدد ال request القادمة من ال client لأن البيانات يتم جلبها بناء على حاجته هو...

والآن شاهد الصورة F8-1:

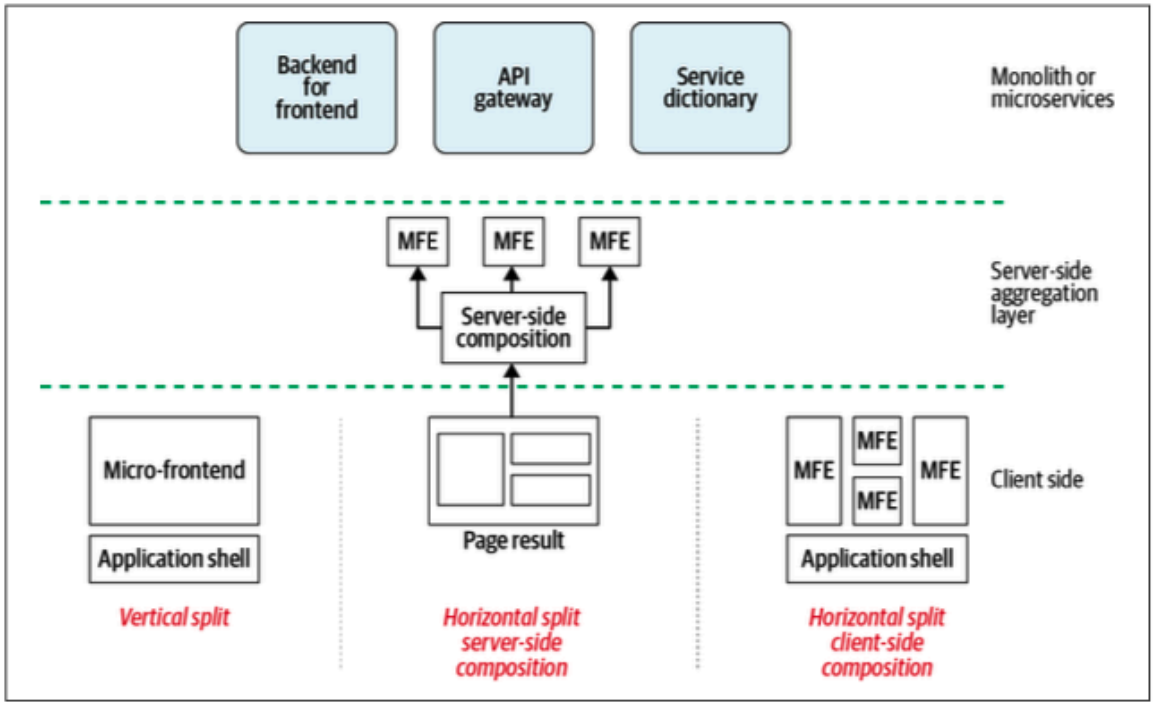


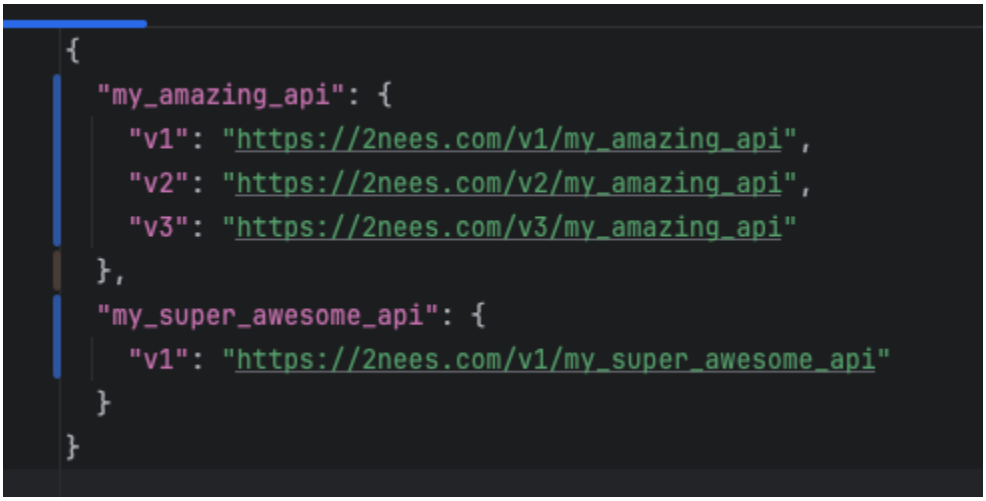
Figure 8-1. Micro-frontends and API layers

قبل أن ننتقل للمواضيع بشكل أكثر تفصيلا، تذكر أننا يمكن أن نستخدم أو أن ندمج أكثر من أسلوب... كما أن هناك أساليب للتعامل مع ال API تكون من خلال ال frontend، ومع ذلك فإن القيام بهذه العملية وإدارتها من خلال الفرونت اند في عالم ال micro قد تعتبر خطوة خطيرة، لذلك لا ينصح بها...

العمل مع ال Service Dictionary

كما تحدثنا في المقدمة، فإن الـ Service Dictionary ليست أكثر من قائمة تحتوي على الـ endpoints المتاحة للاستخدام والمقدمة إلى الـ micro-frontend، وهذا يسمح لنا باستخدام الـ API دون الحاجة إلى كتابة الـ endpoints بشكل يدوي داخل كود ال client، وعادة ما يتم توفير الـ Service Dictionary من خلال ملف JSON أو من خلال API يجب استخدامها كأول طلب صادر من الـ micro-frontend، كما يمكن تضمين ال Service Dictionary مع ملف ال config الخاص بال micros، لأن هذا الملف سنقوم بتحميله أيضا في البداية وقبل أي شيء، وهذا يعتبر أيضا أحد الإعدادات...

شاهد المثال:



```
{
  "my_amazing_api": {
    "v1": "https://2nees.com/v1/my_amazing_api",
    "v2": "https://2nees.com/v2/my_amazing_api",
    "v3": "https://2nees.com/v3/my_amazing_api"
  },
  "my_super_awesome_api": {
    "v1": "https://2nees.com/v1/my_super_awesome_api"
  }
}
```

هذا المثال يوضح فكرة ال Service Dictionary، ومع بساطته فهنا نرى قيمة رائعة لهذا الأسلوب وهي إمكانية إدراج جميع ال APIs المدعومة حاليا، وبهذا كل client يمكنه الاستمرار باستخدام الإصدار الخاص طالما لم يكن هناك break changes... لكن علينا

الحذر من الحالات التي تتطلب حديث فوري مثل إصدار نسخة للموبايل فيها break changes أو مشاكل أمنية، ومن الحلول الجميلة للتعامل مع هذه المشكلة هي إعداد آلية للتحديث الإجباري... فكم مرة رأيت تطبيقا لم يعد بإمكانك استخدامه دون تحديث بعد صدور إصدار جديد؟

بالنسبة لل micro frontend، فإن هذا الأسلوب يقدم ميزة لطيفة للفرق التي تعمل على هذا الأسلوب، وهي معرفة ما يجري أو ما تقوم به الفرق الأخرى، فإذا قام أحد الفرق بتحديث إصدار ما أو إضافة API ما، فإن الجميع سيعرف ذلك، وهذا مفيد من ناحيتين، في حالة كانت الشركة تعتمد ال Component Teams فهذا يعني أن كل فريق سيكون مسؤولا بشكل كامل على مجال عمله، ففريق لل FE وفريق لل BE... في هذه الحالة مجرد إضافة أو تعديل ال service سيكون لدى ال FE team علم بذلك، وفي حالة ال cross-functional teams فإن هذه المعرفة ستكون مطلوبة ومهمة أكثر من الحالات الاعتيادية... كما أن الفريق نفسه يمكنه معرفة الكثير من الأمور المتنوعة ضمن نطاق المشروع لأن الجميع ما تجمعهم غالبا روابط مشتركة، وهذا سيكون ممتازا إذا كان الفريق Two-Pizza Team.

ملاحظة: ال Two-Pizza Team هو مفهوم نشأ في شركة أمازون، وفكرته تمحور حول حجم الفريق المناسب للعمل، والقاعدة تقول إن قطعتين من البيتزا إذا لم تكن تكفي لإطعام كامل الفريق، فهذا الفريق كبير جدا... وبلغة أخرى الفريق يجب ألا يزيد عن ١٠ أشخاص!

والفكرة وراء هذا المفهوم ليس توفير المال، بل تقليل عدد الروابط بين الأشخاص في المجموعة قدر الإمكان، فعدد الروابط بين أعضاء الفريق يمكن وصفها بالمعادلة الرياضية التالية:

$n(n - 1) / 2$

بحيث يمثل n عدد الأشخاص، فإذا كان عدد الفريق هو ٦ فهناك ١٥ رابطا بينهم، وإذا كان العدد ١٢ عضوا فسيكون لدينا ٦٦ رابطا... وهذا يشير إلى أن عدد الروابط بين أعضاء الفريق يزداد بشكل أسي وليس بشكل خطي [٨٨].

مثال على الروابط لفريق مكون من ٤ أعضاء:

A مع B

A مع C

A مع D

B مع C

B مع D

C مع D

إن الهدف من هذه القاعدة هو إبراز أن زيادة حجم الفريق لا تؤدي دوما إلى زيادة الإنتاجية! بل على العكس، يمكن أن يؤدي هذا السلوك إلى **زيادة التعقيد في التواصل والتعاون، مما يُقلل من الكفاءة.**

والآن قد تتعجب لماذا تطرقت لهذه النقطة ضمن السياق، والسبب ببساطة يعود لفكرة شائعة أخرى وهي أن تحسين مستوى التواصل لوحده قد يكون كافيا لضمان التواصل الجيد

وتشارك المعلومات، لكن الحقيقة ليست كذلك، تخيل معي هذا الفريق المكون من ١٢ شخصا وبينهم ٦٦ رابط... هل يمكن ضمان هذا التحديث بين كل هذه الروابط؟! بكل تأكيد لا، ولعل أشهر المشاكل التي قد تحدث هي نسيان تحديث إصدار أو رابط إحدى الخدمات بعد إطلاق نسخة جديدة منها... وهذا ما يمكن تلافيه إذا كان لدينا تواصل جيد وأسلوب مثل هذا الأسلوب!

وأخيرا، إن استخدام الـ service dictionary مع الـ micro-frontends المجمعة على ال client side سيكون ذا قيمة كبيرة لنا، فالـ BFFs والـ API gateways أكثر ملاءمة للتجميع على ال server side نظرا لارتباط الـ micro-frontend مع طبقة البيانات الخاصة به.

العمل مع ال API Gateway:

كما تحدثنا في المقدمة أيضا، فإن الـ API Gateway تمثل نقطة دخول موحدة ومركزية للوصول إلى الـ microservices، وبهذا يستواصل ال Client مع نقطة واحدة فقط [٨٨].

شاهد الصورة F8-7:

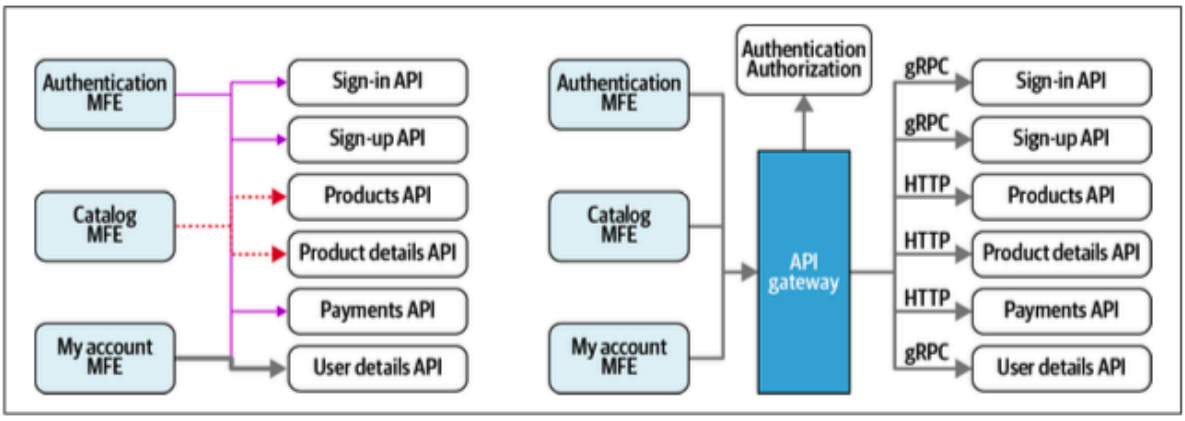


Figure 8-7. An API gateway pattern simplifies the communication between clients and server and centralizes functionalities like authentication and authorization via edge functions

لاحظ جمال الصورة وكمية ما أخفته ال API Gateway من تفاصيل عن ال client، ولاحظ أن عملية الاتصال التي تمت وتحديد البروتوكول المناسب لها تم تحديده وتطويره من قبلنا، دون أن نزعج ال client بذلك.

لكن كما لهذا الأسلوب من مميزات جميلة فلديه عيب خطير يجب الاعتناء به وعدم تجاهله، وهو ال Single Point of Failure، لذلك يتم إنشاء cluster لل Api Gateway للتقليل من خطورة الفشل... كما أن المؤسسة إذا كان يعمل فيها المئات من المطورين، فسنحتاج إلى حوكمة قوية لضبط أي إضافة أو تغيير أو حذف على الـ APIs من ال Gateway!

لتجاوز هذه العيوب هناك العديد من الحلول، مثل:

١. استخدام API gateway لكل business domain، وفيه يتم إنشاء API gateway واحدة للمنتجات، وأخرى للمستخدمين، وثالثة للمدفوعات... وهذا يقدم مزايا رائعة منها التخلص من ال Single Point of Failure... كما يمكننا من استخدام الأدوات أو الأنماط

التي نحتاجها مع كل domain، مثلا يمكننا استخدام ال BFF لخدمة تطبيقات الموبايل وجلب معلومات من أكثر من domain، مثلا order.api.com/mobile-view ستقوم بجلب معلومات الطلب ومكان السائق الذي ينقل الطلب ونحو ذلك، في حين يمكننا بالاكتفاء ب APi gatway مباشرة على بوابة المستخدمين للتحقق من الصلاحيات ونحو ذلك.

لاحظ أن هذا الأسلوب يعطينا استقلالية أكبر وتحكم أدق، لكنه يتطلب جهدا أكبر في البدايات، لكنه يعطينا مرونة واستقلالية أكبر على المدى الطويل ^^.

٢. يمكننا الدمج واستخدام ال API gateway مع ال Service dictionary في حالة ال client-side composition، وفي هذه الفكرة يمكننا استغلال وجود ال API gateway لتمثل نقطة دخول واحدة لجميع ال micro service، وسيقوم ال client بطلب قاموس ال APIs ومن ثم عمل التجميع على ال client بناء على ال Routes الخاص بهذه الصفحة...

٣. يمكنك استخدام الـ API gateway مع ال Server Side Composition أيضا، وفكرة عملها أن ال micro service ستتحدث مع API gateway داخلية، وستقوم ال micro-frontend بالحديث مع هاي ال API gateway... وسيكون لدينا API gateway أخرى خارجية يتعامل معها ال ،client بحيث تقوم هذه ال API بالتواصل مع ال UI Composition Layer، وستقوم هذه بدورها بالتواصل مع ال Micro fontend (تحدثنا

عن ذلك سابقا، وتذكر أننا في حالة ال server side composition هنا فإننا نتحدث عن horizontal split).

العمل مع ال BFF Pattern:

كما تحدثنا في المقدمة أيضا، يقوم الـ BFF بتجميع البيانات من جميع الـ APIs المطلوبة في request واحد على الخادم، ثم يُعيد response واحد جاهز إلى العميل، وهذا يُقلل من عدد ال request بين العميل والخادم ويُحسن من الأداء، وتذكر أن دور هذا الأسلوب يأتي إذا كانت كمية البيانات كبيرة لديك وقادمة من عدة APIs، أو لديك Cross-Platform.

إذا، هذا الأسلوب يقدم حل مميز لمشكلتين رئيستين وهما:

١. لديك واجهة مستخدم مثل لوحة تحكم مالية، وتحتاج إلى بيانات من عدة APIs مختلفة، بالطريقة التقليدية سيقوم المتصفح بطلب كل هذه الـ APIs بشكل منفصل ثم سنقوم بتجميعها وتجهيزها للاستخدام في ال page view، وهذا سيؤدي إلى زيادة عدد الطلبات من العميل إلى الخادم...

٢. لديك تطبيق ويب وتطبيق موبايل، وطريقة عرض البيانات تختلف اختلافا كبيرا بينهم، فماذا ستفعل؟ -عادة ما يتم الحرص على عرض بيانات مفصلة ودقيقة ومختصرة ومنظمة على تطبيقات الهاتف المحمول، وذلك لتقديم أفضل تجربة للمستخدم وأحسن أداء-...

كما تلاحظ، فإن ال BFF هنا يكون دوره، وهذا ملعبه ^^، وهذه هي الحالات المناسبة لاستخدامه، لا كل حالة موجودة في عالم الويب ^^.

ملاحظة: هناك بعض الحالات الجميلة التي قد تدعوك لاستخدام ال BFF، مثلا في مرحلة الانتقال في ال BE من monolithic إلى microservice.

شاهد الصورة F8-12:

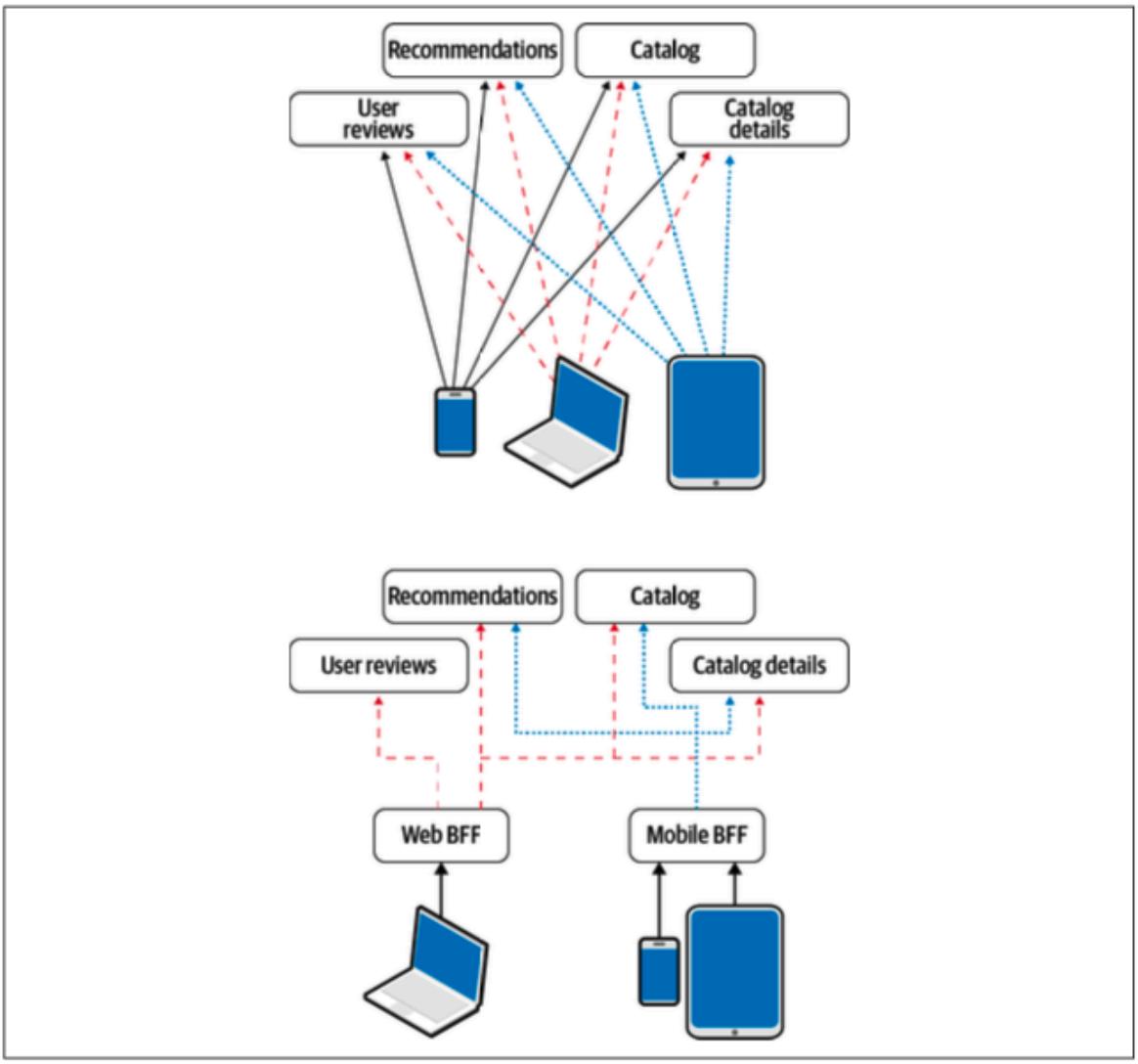


Figure 8-12. On the top, a microservices architecture consumed by different clients; on the bottom, a BBF layer exposing only the APIs needed for a given group of devices—in this case, mobile and web BFF

والآن، السؤال المهم في هذا الموضوع، كيف سيعمل هذا الأسلوب أو كيف سيؤثر علينا ونحن نعمل ك micro-frotnend؟؟

والجواب هنا حسب نوع المعمارية التي قمنا باختيارها لهذا التطبيق، في حالة ال Client side composition لدينا حالتين:

١. ال Vertical split مع ال BFF: وهنا ستكون حياتنا سعيدة، فبكل بساطة سنقوم بطلب ال BFF API ثم عرض هذه البيانات في الصفحة، ويعتبر في حالتنا هذه نهجا فعالا ^^.

٢. ال Horizontal Split مع ال BFF: وهنا ستكون حياتنا لولبية، وتحدياتها كثيرة، لأن الصفحة الواحدة ستتكون من أكثر من micro frontend، وهنا سنحتاج إلى مكان لجلب BFF واحدة لجميع هذه المكونات، والذي سيقدم هذا الخيار هو ال APP SHELL، وهذا سينتهك واحد من أكثر المبادئ أهمية عندنا وهو مبدأ الاستقلالية، وتحدثنا عن مشاكل ذلك سابقا... لذلك لا ينصح في استخدام ال BFF مع هذه المعمارية لأنها قد تلغي مزية ال micro frontend، ومع ذلك، إذا استخدمت إحدى الأفكار للتخلص أو الحد من هذه المشكلة، فعليك دراسة وتحليل الموضوع بشكل دقيق قبل الإقدام على هذه الخطوة...

أما في ال Server side composition:

لن تكون لدينا مشكلة ال Horizontal split التي تحدثنا عنها، لأننا سنقوم باسترجاع البيانات من خلال BFF بحيث تكون جاهزة، وستقوم ال UI Composition Layer بتركيب هذه ال micros وإعطائها البيانات... ^^

بطريقة أبسط، سنقوم بطلب ال BFF API => ثم سيقوم ال BFF بطلب جميع البيانات التي يحتاجها => ثم سيتم إرجاع ال template الخاص بكل micro frontend => ثم سيتم تجميع هذه ال template في صفحة كاملة ليتم إعادتها جاهزة إلى المتصفح. (نفس المخطط الذي شرحناه سابقا، لكن الجديد هو وجود BFF مسؤول عن جلب البيانات).

ال Best Practices للتعامل مع ال API في ال Micro Frontend

سنتطرق لبعض الملاحظات المهمة والتي ينبغي عليك مراعاتها عند تعاملك مع ال API، بحيث نحصل على أفضل توافق ممكن بين ال micro-frontend وال backend.

١. ال Multiple micro-frontends consuming the same API:

إذا كانت لديك أكثر من micro frontend في نفس الصفحة (horizontal split)، وهناك API يتم جلبها من two micros مثلا، فعليك التوقف قليلا وأن تسأل نفسك، هل فعلا قرار التقسيم لدي صحيح؟ أم هل من الأفضل تجميع هذه ال micros معا؟ لا تتجاهل هذا المؤشر، فهو مؤشر مهم وكثيرا ما يحسن من البنية التركيبة للمشروع.

٢. ال APIs come first, then the implementation:

الـ API-First هو نهج يعطي الأولوية لتصميم وتعريف ال APIs قبل البدء في أي عملية بناء للشيفرة البرمجية، فبدلا من أن تقوم الفرق ببناء حلولها بشكل منفصل ثم تحاول ربطها لاحقا، يتم الاتفاق على API Contract أولا، وهذا سيقدم لنا فوائد مهمة منها:

*. يمكن للفرق العمل بشكل متواز ومستقل، فال BE يعرف بالضبط ما سينتجه، وال FE يمكنه البدء ببناء الواجهات بناء على ما تم الاتفاق عليه بال API Contract، وبذلك يمكنه بناء Mock Data دون الحاجة لانتظار ال BE.

*. تقليل المخاطر لأن ذلك يجعل جميع الفرق المعنية تحلل ال API، ومع عملهم يمكن كشف الأخطاء أو الاحتياجات التي لم يتم تغطيتها مبكرا.

*. يحسن التواصل والتعاون بين الفرق خصوصا باختلاف المناطق الجغرافية، وهذا سيدعم استخدام ال RFC -التي تحدثنا عنها سابقا- من الفرق لمناقشة التغييرات الجوهرية والمساعدة على اتخاذ القرار.

٣. ال API consistency:

هناك أهمية بالغة لتوحيد ال API، فوجود APIs متسقة وموحدة سيساعد على فهم أي API موجودة في المشروع وتوقع نتائجها، كما أن إي API جديدة سيتم إضافتها ستكون مفهومة بالنسبة لنا، وهذا سيسهل على الجميع العمل، ويقلل من الأخطاء، ويحسن من منحنى التعلم للمطورين الحالين والجدد ^^، ولعل أشهر الأمثلة على ذلك بتوحيد ال Error Handling - معالجة الأخطاء-.

وهنا لفتة جميلة، وهي أن هذا المبدأ أيضا يجب أن يتم تطبيقه على ال Events Schema عند التواصل بين ال micros المختلفة، مثلا يتم الاتفاق على أن هناك Event سيتم إطلاقه إذا

نجح المستخدم بتسجيل الدخول اسمه user-Sign-In... وبهذا ستقوم الفرق المختلفة المهتمة بهذا ال event بالاستماع إليه، وسيقوم الفريق المعني بتطويره وإرساله... كما يجب أن يتم توثيق هذه المعلومات بالإضافة لما سيتم إرساله من بيانات مع هذا ال event.

٤. ال WebSocket and micro-frontends:

عملية إدارة ال Websocket مع ال micro-frontend تتطلب بعض الانتباه، وذلك حتى لا نقع ضحية لوجود أكثر من Websocket فعال وشغال في نفس الوقت! أي أن الأصل وجود Websocket connection واحدة وليس أكثر من واحدة... تخيل أن لدي في صفحة ١٠ micro frontend تحتاج لاستخدام ال websocket، ثم قمنا بإنشاء ١٠ connection لهذا الغرض!

لذلك، علينا الحرص على إنشاء Unique Connection لكامل التطبيق وإتاحة ال Websocket instance لجميع ال micro-frontend... لذلك، إذا كان التعامل من خلال ال Horizontal split فينصح أن تتم هذه العملية من خلال ال APP Shell، أما إذا كنا نتعامل مع Vertical Split فالحياة سعيدة ويمكن وضع ال Websocket داخل ال micro... أما إذا كانت ال websocket غير مستخدمة إلا في micro محددة، فإنها توضع هناك وبغض النظر عن أسلوب التقسيم المتبع...

كما ينبغي عليك الحرص على معالجة التحديات الممكنة، مثلا تم تحميل ال socket في ال app shell ووصلت بعض الرسائل لكن ال micro لم يتم تحميلها بعد.. وهنا يمكن استخدام ال Message Buffer ومحاولة إعادة إرسال الرسائل ونحو ذلك.

وحتى تتضح الفكرة، انظر لهذا المثال: -تطبيق في حالة ال Horizontal Split وذلك لأهمية معرفة طرق إدارة ال socket به-:

```
// application-shell/socket.js
import { EventEmitter } from 'events';

export const socket = new WebSocket('wss://chat.2nees.com/socket');
export const socketEvents = new EventEmitter();

socket.onmessage = (event) => {
  const data = JSON.parse(event.data);
  // إرسال الرسائل إلى أي Micro-Frontend مسجّل
  socketEvents.emit(data.type, data.payload);
};

socket.onerror = (error) => {
  socketEvents.emit('error', error);
};
```

```
// chat-window/index.js
import { socketEvents } from 'appShell/socket';

// تحديث واجهة الدردشة عند وصول رسالة جديدة
socketEvents.on('newMessage', (message) => {
  renderMessage(message);
});

// التعامل مع الأخطاء
socketEvents.on('error', (err) => {
  showError(err);
});
```

٥. ال The right approach for the right subdomain:

ذكرنا سابقا أن المرونة في تصميم ال micro service وال micro frontend أمر مهم، وما تم الابتداء به لا يعني بالضروره أنه سيبقى صحيحا طوال فترة التنفيذ أو التطوير، والقرارات الصحيحة عند البداية قد تصبح خاطئة عند التقدم بالعمل... كما أن الخيار الأنسب قد يختلف من subdomain إلى آخر، لذلك على الفرق التحلي بالمرونة أثناء العمل وتقبل وجود هذه الاختلافات بدلا من الالتزام بنهج موحد داخل كل subdomain، وكل ذلك بناء على ما تعلمناه سابقا وضمن الضوابط التي تنص عليها قواعد اتخاذ القرار.

٠٦. ال Designing APIs for cross-platform applications:

يجب أن نحرص على أن يكون التحكم في سلوك التطبيق نابعا من configuration يتم جلبها من خلال API وليست مجرد Hard-coded، والسبب في ذلك أن تغيير هذه الإعدادات لن يدفعنا لإصدار نسخة جديدة من المشروع، كما أن ذلك يمكن أن يقلل من بعض المخاطر المحتملة مثل ال Bursty Traffic...

وكمثال عملي لنأخذ ال Polling Strategy، تخيل أن لدينا تطبيقا لمعرفة حالة الطقس، وسيقوم تطبيق الهاتف المحمول بتحديث بياناته كل خمسة دقائق بشكل تلقائي ^^... إذا تم وضع هذا الوقت hard-coded داخل ال client side فإن هذا قد يتسبب في مشكلة إذا كان هناك ضغط على ال servers في هذا الوقت -مثلا ثلجة أو إعصار أو حدث تنتظره الناس-، بينما إذا تم وضعها كإعداد، فإن سلوك التطبيق يمكن تغييره وزيادة وقت التحديث الخاص بالبيانات دون الحاجة لتعطيل التطبيق ^^

فائدة

إن الله -سبحانه وتعالى- وإن كتب النصر لعباده؛ فإنه لم يكتبه لهم دون بذل الأسباب في تحقيق النصر، ومثال ذلك: "ادْخُلُوا الْأَرْضَ الْمُقَدَّسَةَ الَّتِي كَتَبَ اللَّهُ لَكُمْ"، فهي كتبت لهم لكن عليهم أن يدخلوها وأن يبذلوا الأسباب، فلما كان جوابهم لجبنهم وخوفهم وقلة همتهم: "فَإِن يَخْرُجُوا مِنْهَا فَإِنَّا دَاخِلُونَ"... لم يدخلوها وحرمت عليهم ما شاء الله لهم!

- مما تعلمته من كتاب أنوار الأنبياء، أحمد بن يوسف السيد

الفصل الثامن: الجانب البشري وعلاقته بال micro-frontend

لا يقتصر نجاح مشروع الـ Micro-frontends على كتابة الشيفرة البرمجية أو اتباع أفضل الممارسات **التقنية** فحسب؛ بل إن الجانب الأكثر أهمية والذي يجب أن نفكر فيه بعمق: هو الجانب البشري والتنظيمي...

إن إهمال الجانب البشري غالبا ما يضيع قرارات صحيحة أو منطقية لأنها كانت تغطي الجانب التقني فقط، دون الاهتمام بالجانب البشري -العاطفي أو العقلاني-! لذلك تجد كثيرا من المطورين المبدعين بقوا في أماكنهم لأنهم لم يستطيعوا التعبير عن أنفسهم بشكل صحيح! وهناك كثير ممن أعرفهم شخصيا، كان لتسويقهم لأنفسهم الأثر الأكبر في وصولهم لمراتب أعلى من غيرهم -بعد فضل الله سبحانه وتعالى وتوفيقه لهم-؛ رغم أن هناك من هو أكثر منهم مهارة وعلما.

لذلك، إذا قررت تبني ال micro-frontend، فعليك ألا تتجاهل الجانب البشري، وكيف ستبرز الميزات والتحديات التي ستواجهها المؤسسة عند اعتماد هذا النهج، بطريقة تقنية وبلغة بسيطة يفهمها المسؤولين وغير التقنين؛ ومن ذلك:

كيف سيتم تنظيم التواصل بين الفرق المختلفة لضمان التعاون الفعال؟ وكيف يمكن تجنب إنشاء "مجموعات معزولة" (Siloed teams) لا تتواصل مع بعضها البعض؟ وكيف يمكن تمكين المطورين ليتمكنوا من اتخاذ القرارات الصحيحة بشكل مستقل داخل نطاقات عملهم؟... إلى آخره.

إن الـ Micro-frontend يمكن أن يساعد في حل بعض هذه المشكلات، لكن في المقابل قد يزيد من التعقيد إذا لم يتم التعامل مع المعمارية بشكل صحيح، لذلك، من الضروري أن تقوم بتحليل لشكل المؤسسة التي نعمل بها، وكيف يمكن أن تتكيف الفرق ومخطط سير العمل في الشركة مع هذه البنية الجديدة، إن الاستثمار في هذا الجانب من البداية يضمن أن يكون المشروع ناجحا على المدى الطويل، ويقلل من السخط الناجم من الفرق أو المسؤولين النابع من التغيير...

إن أول الأسئلة التي ستطرح عليك عند اقتراحك لل micro هي، لماذا يجب أن نستخدم أو أن ننتقل إلى ال micro-frontend؟

وهذا سؤال وجيه ومهم، والإجابة تنطلق فعلا من وجود حاجة حقيقة لهذا الانتقال، فإذا كانت الحاجة حقيقة لهذه البنية، فيمكن الإجابة من خلال ذكر الفوائد التقنية والتنظيمية التالية -ولاحظ أن هذه القائمة فيها جانب تقني وجانب بشري موجه لل stakeholders-:

- استقلالية الفرق، فالفرق الكبيرة يتم تقسيمها لفرق أصغر، والعمل الكبير يتم تقديمه لأعمال أصغر، وهذا يقلل الاعتمادية ويعطي قدرة أكبر على اتخاذ القرارات التقنية.
- النشر (Deployment) بدون مخاطر، وهذا يعني أن الفريق يمكنه إصدار تحديث لجزء من التطبيق دون التأثير على الأجزاء الأخرى، كما يمكن التراجع عنه بسهولة إذا حدث هناك خطأ ما.

- تقليل الحمل الإدراكي على المطور -Reduced Cognitive Load-، والحمل الإدراكي هو مقدار المعلومات التي يجب على المطور معالجتها لفهم جزء معين من النظام، ففي التطبيقات الضخمة قد يضطر المطور لفهم قاعدة بيانات كبيرة، وشيفرة برمجية معقدة، وتفاعلات متعددة، بينما في الـ Micro-Frontends: يركز المطور على جزء صغير فقط من التطبيق، مما يسهّل فهمه وإتقانه، ويزيد من إنتاجيته.
- ال Faster Iterations، لأن الفرق تعمل بشكل مستقل على أجزاء صغيرة من التطبيق، فيمكنها إصدار التغييرات والمميزات الجديدة بشكل أسرع، فلا توجد حاجة للانتظار حتى يتم إكمال العمل على الأجزاء الأخرى، وهذا يسمح بنشر إصدارات متعددة لخصائص متعددة بشكل أسرع.
- تعزيز الابتكار، فهذا يتيح للفريق التقني باختيار التقنية المناسبة للجزء الذي يعمل عليه... لكن ضمن الشروط والضوابط التي تحدثنا عنها سابقا.
- أشر إلى أن ال Micro-Frontends تسمح بتكرار أسرع لل Features وتقلل من خطر إدخال Bugs إلى التطبيق بأكمله.
- اشرح لماذا قد يساعدك ال Micro-Frontends في تحقيق أهداف العمل.
- عدّد المشكلات التي تحاول حلها باستخدام هذا الأسلوب الجديد: مثل مشكلة الحجم والتعقيد، ومشكلة الاستقلالية والتقييد التقني ونحو ذلك.
- أشر إلى أفضل طريقة وجدتها لتنفيذ ال Micro-Frontends ضمن سياق المؤسسة أو المشروع الذي تعمل عليه.
- حلّل التأثير الذي قد يتركه هذا ال Architecture على تواصل الفرق، وكيف يمكن تفادي المشكلات التي تحدثنا عنها سابقا.

- تحديد وتوضيح الأسلوب الذي توصي به لبناء ال Governance بشكل جيد مما يمكننا من إدارة مثل هذا ال Architecture، ومن ذلك وجود قواعد تضبط الفرق مثل مكان مشترك لمكونات التنسيق أو ال API contracts ونحو ذلك.

أما من التحديات الأساسية التي قد تواجهها والتي ينبغي لك الحديث عنها:

- خطر إنشاء Silos داخل المنظمة، وال Silos هي الصومعة، وفي ذلك إشارة إلى أن الفرق والأقسام قد تعمل بشكل منعزل مع تركيز كل فريق على أهدافه الخاصة فقط، وهذا قد يعيق التواصل الفعال، ووجود ازدواجية في العمل بحيث يعمل فريقان على حل نفس المشكلة، وضعف في الرؤية الشاملة للمشروع أو أهداف المؤسسة... كما أن ذلك سيقلل من سرعة اتخاذ القرار لعدم وجود رؤية شمولية.
- استثمار أكبر في ال Automation Pipelines، فالتحديات هنا مختلفة، ونحن بحاجة لثقافة مختلفة هنا في التعامل مع ال micro...
- خطر التباينات في ال User Interface، وهذا يعني يعني أن تصميم وتجربة المستخدم قد تختلف بين أجزاء التطبيق المختلفة، مما يؤدي إلى عدم الاتساق.
- خطورة التداخل بين ال Component وال Micro frontend
- خطورة بناء علاقة صلبة بين الأجزاء تضيع مفهوم ال micro مثل وجود ترابط بين ال app shell وال micro والاعتماد على ال state لنقل البيانات بين ال micros... إلخ.

هذه بعض النقاط التي يمكننا الحديث عنها وإبرازها... وهناك المزيد بناء على ما ذكرناه سابقا من مميزات وتحديات...

ملاحظة: قبل الوصول لهذه المرحلة، حاول إشراك الأشخاص المعنيين أو سؤالهم واستشارتهم حول ما يتعلق في البيزنس والأفكار الخاصة بالمشروع والتقسيمات التي تراها والحدود التي وضعتها ونحو ذلك... إن هذه العملية ممتازة وتقرب الأشخاص والأفكار، وتحسن من ال PoC الذي ستخرج به، ويعالج بعض الأخطاء المحتملة...

العلاقة بين المؤسسات وال Software Architecture

عندما نختار Software Architecture للمشروع الخاص بنا، فمن السهل أن ننجذب لتقليد الشركات الكبرى التي حققت نجاحا كبيرا، لكن الواقع هو أن هذه البنى أو المعماريات ليست مثالية بذاتها، بل هي مجرد مجموعة من التنازلات (trade-offs).

إن الهدف الرئيسي ليس في العثور على البنية المثالية، وإنما في العثور على البنية التي تتناسب بشكل أفضل مع سياق مشروعنا وطبيعة مؤسستنا. وهذا يشمل:

- احتياجات منظمتك: ما هي أهداف العمل؟ هل الأولوية للسرعة، أم الأمان، أم قابلية التوسع؟
- الموارد المتاحة: ما هو حجم فريقك؟ ما هي مهارات المطورين؟ ما هي الميزانية؟

- نوع المشروع: هل هو تطبيق بسيط أم نظام معقد وواسع النطاق؟

كل بنية برمجية (مثل Monolith أو Microservices) تأتي مع مجموعة من الإيجابيات والسلبيات، فعلى سبيل المثال: قد توفر الـ Microservices مرونة وقابلية للتوسع، لكنها قد تزيد التعقيد بشكل كبير! لذاك، بدلا من البحث عن الكمال -الكمال لله وحده جل في علاه-، يجب على الفرق أن تقيّم خياراتها بعناية وأن تختار البنية التي تتوافق مع تحدياتها...

وهذا الكلام يقودنا للإجابة عن السؤال الذي يسأل دائما، وهو: "ما هو أفضل Software Architecture؟"، والجواب دائما: "يعتمد ذلك على" ^^... أي أن إجابة هذا السؤال تعتمد على كل ما ذكرناه سابقا... فحتى المؤسسة نفسها، قد تعتمد معمارية س في مشروع، ومعمارية أخرى ص في مشروع آخر! بل إن المشروع نفسه قد يتطور مع الزمن ويختلف بسبب اختلافات البيزنس أو تطورات التقنية، فيصبح ما كان جيدا اليوم أو الأمس، سيئا في الغد!

ال Features Versus Components Teams:

لدينا نوعان أساسيان من الفرق يمكننا الاعتماد عليهما في معماريتنا، لكل نوع منهم مزاياه وتحدياته الخاصة، والتي تستوجب منك الانتباه عند اختيار الأسلوب المناسب لك وللمؤسسة، اعتمادا على طبيعة المشروع وخبرة المطورين وأماكن تواجدهم وتوزيعهم...

1. ال Feature Teams:

هي فرق متعددة الوظائف (Cross-Functional) تمتلك جميع المهارات اللازمة لتسليم ميزة كاملة، لذلك عادة ما يكون المطورون في هذه الفرق Full-Stack، وهو يتناسب مع ال Horizontal Split، حيث يمكن أن يكون كل فريق مسؤولًا عن واحدة micro-frontend أو أكثر.

المميزات:

- يمتلك الفريق كامل السيطرة على ميزته التي يعمل عليها.
- التركيز على المستخدم بحيث يعمل الفريق على تحسين تجربة المستخدم بشكل مباشر (هذه نقطة مهمة ومفصلية).
- تقليل الحمل الإدراكي المطلوب من المطورين والفرق.

التحديات:

- قد يتطلب هذا النهج فريقًا آخر (أو أحد الفرق الموجودة) ليكون مسؤولًا عن تجميع الصفحات (Page Composition)، مما قد يقلل من المرونة.

2. ال Component Teams:

هي فرق متخصصة في تطوير component معينة أو جزء محدد من النظام، ويتخصص كل فريق في layer معينة من التطبيق ويكون مسؤولا عنها، على سبيل المثال: فريق لل BE

للتعامل مع ال API، وفريق FE للتعامل مع الواجهات... وتيناسب هذا الأسلوب مع ال Vertical Split، لأن المسؤوليات يتم تقسيمها وتوزيعها بناء على ال layers التقنية.

المميزات:

- تخصص عميق: الفرق تمتلك خبرة عميقة في مجالها.
- دعم تطبيقات ال Cross-Platform: لأن هذا النهج يمكن فريق الـ Backend من تطوير APIs تخدم جميع المنصات أكانت mobile أم web أم غيرها...

التحديات:

- قد يؤدي إلى إنشاء Siloed Teams إذا كان التواصل ضعيفا بين الفرق.

وهنا سنأخذ مثالا غريبا نوعا ما حتى تقترب الصورة لما نشير إليه ^^

لو افترضنا أن لدينا Landing Page نرغب ببنائها، فأي نوع من الفرق هو الأنسب لهذه المهمة؟

إذا كانت إجابتك هي الأول أو الثاني هكذا، فهي إجابة خاطئة *-*... لأن الإجابة يجب أن ترتبط بسياق محدد، والسياق هنا ليس موجودا... هذه أول نقطة، لكن لو كانت إجابتك كالآتي:

هذه الصفحة يمكن أن يتم اعتبارها ك Feature تنسب لفريق ال Feature، ويمكن اعتبارها مجموعة من ال Component التي ستنسب إلى فريق ال Component ومن ثم يقوم ال Feature team بتجميعها... فحينها هذه الإجابة صحيحة ^^.

إن ال Landing Page هي بالتأكيد Feature لأنها ستقدّم Business Value محددة للمستخدم مثل الترويج لمنتج جديد، وفي هذا السياق يمكن أن يكون ال Feature Team مسؤولا عن ال Landing Page بالكامل.

كما يمكن أيضا تكوين ال Landing Page من Components صغيرة قابلة لإعادة الاستخدام، على سبيل المثال قد تحتوي ال Landing Page على ال header و ال product list ونحو ذلك، ففي هذا السياق يمكن أن يكون ال Component Team مسؤولا عن بناء Component Library قياسية يمكن لجميع الفرق الأخرى من استخدامها، ثم سيقوم فريق آخر مثل Feature Team من استخدام هذه ال Components لتجميع ال Landing Page.

هل اتضحت الفكرة الآن؟ الموضوع عبارة عن تنازلات ومقايضات ونظرة للأسلوب الأفضل حسب الوضع الراهن، وهذا يتغير بتغير المعطيات... وغالبا ما تحتوي المؤسسات على كلا الفريقين Hybrid Model Teams كما في سؤالنا السابق.

ال ADR

ال ADR هي اختصار ل: Architectural Decision Records، وهي وثيقة تُستخدم 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؟!

بناء على ما سبق، يمكننا أن نقول أن ما نرجوه من تطبيق ال ADR هو:

- توثيق السياق: بحيث نجيب على سؤال "لماذا؟"، فنحن لن نقول "اخترنا Vertical Split فقط"، بل سنشرح لماذا كان هذا الاختيار هو الأفضل في سياق الشركة في ذلك الوقت، مع الأخذ في الاعتبار المزايا والعيوب (trade-offs) لكل خيار.

- توفير الشفافية: جعل عملية اتخاذ القرار واضحة وشفافة لجميع أعضاء الفريق، حتى لو لم يكونوا حاضرين في المناقشات، وهذا يمنع الالتباس من الغاية في استخدام أسلوب معين أو تقنية معينة، ويقلل من الحاجة إلى إعادة مناقشة نفس المواضيع لاحقا.
- تحسين القرارات المستقبلية: من خلال توثيق الأسباب، والخيارات التقنية التي دعتنا لنهج أسلوب معين، يمكننا نحن والفرق المستقبلية من التعلم من القرارات السابقة وتجنب الأخطاء التي وقعنا بها...

هيكل الـ ADR يحتوى على عدة أقسام أساسية، وهي بالترتيب:

- الحالة (Status): توضيح حالة الوثيقة (مسودة، متفق عليها، إلخ).
- ال Stakeholders: تحديد المسؤولين عن اتخاذ القرار.
- ال Due Date: تاريخ اتخاذ القرار.
- ال Outcome: ويمثل القرار النهائي المتخذ
- ال Introduction: فقرة تصف سياق الشركة والمشكلة التي يحاول ال ADR حلها.
- ال Forces: السبب أو الأسباب التي تدفعنا لتغيير المعمارية.
- الخيارات (Options): تمثل قائمة بالحلول الممكنة، مع تحليل المزايا والعيوب لكل خيار.
- ال Final Decision and Rationale: ملخص القرار النهائي وتوضيح الأسباب المنطقية التي أدت إلى الذهاب مع هذا الخيار دون غيره.
- يمكن إضافة ال Owners وال Appendix أيضا إذا لزم الأمر...

هذا الملف من الملفات الجميلة والمفيدة جدا على المدى الطويل، قد لا تكون هذه الملفات إلزامية للعمل، لكنها تعطي جانبا مهما وعمليا على المدى البعيد... لذلك ننصح ببناء هذا الملف، خصوصا عند أكبر وأول قرار للتغيير.

وسائل مقترحة لتحسين التواصل بين الفرق ومشاركة المعرفة

لقد تحدثنا سابقا عن آليات التواصل وأهميتها على شكل قبسات سريعة، لكن أحب أن أضيف هنا أسلوبين يمكن استخدامهما لتحسين التواصل ومشاركة المعرفة، خصوصا إذا كانت الفرق موزعة على أماكن جغرافية متعددة أو لدينا فرق بتخصصات متعددة ومختلفة.

الأسلوب الأول: هو ال Community of Practice، وهو يطرح فكرة إنشاء مجتمع داخل الشركة من خلال بناء تجمع دوري كل شهر أو نصف شهر للأشخاص الذين يعملون في نفس المجال، مثلا اجتماع شهري لجميع مطوري ال FE، وهذا يساعد على مشاركة المعرفة والخبرات وأفضل الممارسات، بالإضافة إلى التحديات التي حصلت معهم خلال هذه الفترة وكيف قاموا بحلها... وهذا كله سيقدم فائدة عظيمة للجميع، كما أن هذا الفريق يمكن أن يعمل معا (Mob Programming) لحل مشكلة معينة.

ملاحظة: ال Mob Programming هو مصطلح يشير إلى طريقة عمل تعاونية يشترك فيها المطورين على حل مشكلة ما أو تطوير مزية ما بشكل جماعي، على جهاز واحد، وفي نفس الوقت! وهو يشبه ال Pair Programming إلا أن ال Pair يكون بين شخصين، بينما ال Mob يكون بين جميع أعضاء الفريق...

في هذا الأسلوب يتم تقسيم الأعضاء إلى Driver و Navigator و Explorers و Recorder، ويتم تبادل الأدوار فيما بينهم كل ٥ أو ١٠ دقائق... يكون ال Driver هو المسؤول عن كتابة الشيفرة البرمجية، وال Navigator هو قائد الجلسة والمسؤول عن توجيه

ال Driver لما عليه فعله، ويقدم له الأفكار ويحلل المشكلة ويناقش الخيارات الممكنة مع الفريق، وال Explorers يركزون على حل المشكلة ويبحثون عن المعلومات ويقترحون الأفكار، و يتأكدون بأن الجميع متفق على الخطوة التالية، أما ال Recorder فهو الذي يدون القرارات الهامة التي يتخذها الفريق ويسجل الملاحظات ويتابع ما يتم مناقشته... وقد يكون التقسيم هو Driver و Navigator و The Mob، ولن يختلف الأمر كثيرا.

الأسلوب الثاني: هي ال Town Halls، وهي اجتماعات أو أحداث يتم تنظيمها عبر قسم ال IT لتقديم معرفة عامة حول ما يحدث عبر الفرق، مثل إنجازات فريق معين أو ممارسات جديدة تم إدخالها إلى داخل المؤسسة ^^، ويتم عقد هذه الاجتماعات على مستوى القسم التقني كاملا (FE, BE, Devops... إلخ).

ويهدف هذا الأسلوب إلى تقديم نظرة عامة عما يحدث في المؤسسة ومشاركة الإنجازات وتقديم المبادرات وتعزيز الوعي المشترك بين الفرق، وحتى يمكن استخدامها للترحيب بالموظفين الجدد وإدخالهم لجو الشركة مع كسر حواجز الاتصال.

طبعا إذا كان عدد الموظفين كبيرا، فهذا الأسلوب قد لا يعمل بشكل جيد أو لن يكون الخيار الأمثل في مثل هذه الحالة، لذلك يمكن استبدال هذا الأسلوب ب Newsletter دورية يتم إرسالها أو نشرها داخل المؤسسة، ويلتزم الجميع بقرائتها والتفاعل معها...

ال Micro متنوعة المستوى في التعقيد

قد تتعجب من هذا العنوان، لكن يجب عليك أن تعلم أن ال micro التي يتم تصميمها اعتمادا على ال subdomain التي تم تحديدها، تختلف في تعقيدها... وهذا أمر مهم جدا لأنه يرتبط ارتباطا وثيقا بالعامل البشري، فإذا كانت إدارتك لهذه ال micro بطريقة خاطئة فهذا سيتسبب بإرهاق على المطورين وكثرة الاحتكاك والمشاكل النابعة من الإرهاق وتداخل الأعمال أو كثرتها عند فريق وقلتها عن فريق آخر... والعديد من النقاط المؤثرة والجوهرية، ومن هنا يمكننا تقسيم مستوى التعقيد إلى ثلاثة مستويات، وهي:

١. ال Micro ذات التعقيد المرتفع: عادة هذا النوع من ال Micro يبدأ بتعقيد طبيعي -أو هكذا نظن-، ويتم إدراك مستوى التعقيد لاحقا في مراحل أخرى، وعادة ما يكون سبب ذلك هو وجود الكثير من التفريعات المرتبطة بال Business Logic، أو أن نفس ال Business Logic معقد وفيه تفاصيل كثيرة لم يتم تقدير تعقيدها بشكل صحيح، وهذا يؤثر بشكل واضح على الفريق ويزيد من الحمل المعرفي عليهم... كما أن هذا النوع من ال Micro أصعب في الصيانة وأصعب في الدعم، وأكثر احتمالية في الوقوع بال Bugs.

لذلك، لا يتم ترك هذا النوع من ال Micro هكذا، بل يتم النظر إليه مجددا ومراجعة حدوده، مع محاولة لتقسيمه من جديد، فإذا استطعنا تغيير مستوى التعقيد من العالي إلى الطبيعي -النقطة رقم ٣-، فهذا أمر في غاية الروعة.

٢. ال Micro ذات الجهد المرتفع: هذا النوع من ال Micro يتطلب جهدا كبيرا عند الإنشاء، لكنه لا يتطور كثيرا أثناء العمل أو خلال دورة حياة التطبيق، ومعرفة هذه ال Micro مفيدة لأنها تسمح لنا إسناد العديد من ال Micros من هذا النوع إلى فريق معين، مع وجود موازنة في مستوى تعقيد هذه ال Micro وقدرة الفريق على إدارة هذه ال micros، حتى نتجنب إغراق الشباب بكمية وظائف مخيفة :P.

٣. ال Micro ذات التعقيد العادي أو الطبيعي: هذا النوع هو ما نصبو إليه حقيقة وما نرغب في تحقيقه... إنها ال Micro الطبيعية التي نأمل في الحصول عليها، ففيها نحقق الاستفادة العظمى من هذه المعمارية، ويكون الحمل المعرفي أو الإدراكي فيها طبيعي مما يعني فهما كاملا أو فعلا للعمل من قبل المطورين... كما أن هذا النوع يعني أن فريق واحد قادر على إدارة وبناء هذه ال micro والتحكم فيها من الألف إلى الياء، مع استقلالية عالية وسرعة في اتخاذ القرار.

والحمد لله رب العالمين.

الخاتمة

وصلنا الآن إلى نهاية رحلتنا في هذا العالم الجميل، عالم ال Micro Frontend، ولقد تطرقنا إلى العديد من المواضيع الرائعة وشاهدنا الكثير من الأمثلة، وصارت لدينا فكرة عملية حول كيفية العمل على هذه المعمارية، وأهمية المقايضات للحصول على أفضل النتائج الممكنة... وتعلمنا أن أفضل الخيارات تتناسب مع سياق المشروع، وهي عملية مقايضة للحصول على أفضل السيء... وقلنا أن هذه المعمارية يجب أن تنفذ عن الحاجة إليها وليس لأنها Trend أو لأنها شيء جميل! وإياك أن تنسى: "لا تستخدم المدفع الرشاش لتقتل نملة"!

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