مفهوم **"Architecture of Distributed Operating System"**
(هندسة نظام التشغيل الموزع) بطريقة منظمة وبسيطة:

ما هو**Distributed Operating System (DOS)** ؟

نظام التشغيل الموزع هو نظام يجعل مجموعة من الحواسيب المختلفة) أو العقد (Nodes تعمل وكأنها **نظام واحد متكامل**.
أي أن المستخدمين لا يشعرون أنهم يتعاملون مع عدة أجهزة — بل مع جهاز واحد كبير.

هندسة نظام التشغيل الموزع(Architecture)

الهندسة تشرح **كيف يتم تنظيم وربط** الأجزاء المختلفة. هناك عدة أنماط أو أشكال لهندسة أنظمة التشغيل الموزعة، منها:

1. Client-Server Model

- **الفكرة :**
- هناك أجهزة تقوم بدور **العملاء (Clients)** تطلب خدمات معينة، وأجهزة تقوم بدور **الخوادم (Servers)** تقدم هذه الخدمات.
- مثال: جهازك يطلب ملف من سيرفر تخزين مشترك.

مكونات:

- **Clients:** ترسل طلبات (مثل طلب طباعة مستند).
- **Servers:** تعالج الطلبات وترد بالنتيجة.

---

2. Peer-to-Peer Model (P2P)

- **الفكرة :** لا يوجد خادم مركزي، كل جهاز يمكن أن يكون **عميل وخادم** في نفس الوقت.

- مثال: برامج مشاركة الملفات مثل.BitTorrent

مكونات:

- كل عقدة مسؤولة عن إرسال واستقبال البيانات.
- توزيع المهام بالتساوي تقريبًا.

---

3. Multicomputer Model

- **الفكرة :**عدة حواسيب متصلة بشبكة وتعمل كأنها جهاز واحد.
- كل جهاز له معالج خاص به، ولكن يتم إدارة الموارد بشكل مركزي.
- مثال: أنظمة الحوسبة السحابية الضخمة.

مكونات:

- **أجهزة مستقلة** لكنها تدار بواسطة **نظام تشغيل موزع** واحد.

مكونات هندسة نظام التشغيل الموزع بشكل عام

- **Process Management:**إدارة العمليات التي قد تعمل عبر أكثر من جهاز.
- **Resource Management:**إدارة الموارد (معالجات، ذاكرة، طابعات...) وتوزيعها.
- **Communication:**آلية الاتصال بين الأجهزة (شبكة - بروتوكولات).
- **File System:**نظام ملفات مشترك (مثلاً يمكنك فتح نفس الملف من عدة أجهزة).
- **Security:**تأمين الاتصالات وإدارة الحقوق.
- **Fault Tolerance:**التعامل مع فشل الأجهزة بدون التأثير على النظام بالكامل.

✨لماذا نستخدمDistributed Operating Systems ؟

- **المرونة :**يمكن إضافة أو إزالة أجهزة بسهولة.
- **السرعة :**توزيع الأحمال على أكثر من جهاز.
- **الاعتمادية :**إذا تعطل جهاز، يكمل الآخرون العمل.
- **مشاركة الموارد :**الاستفادة من قدرات أكثر من جهاز.

أمثلة على أنظمة تشغيل موزعة

- **Amoeba**
- **Plan 9 from Bell Labs**
- **Inferno**
- **Google's internal systems (بشكل غير مباشر)**

ما هوArchitectural Style ؟

الأسلوب المعماري (Architectural Style)هو **نمط أو طريقة منظمة** لتصميم أنظمة البرمجيات أو الأنظمة التقنية، بحيث يحدد:

- كيف يتم **تنظيم** المكونات.(Components)
- كيف يتم **التواصل** بينها.(Communication)
- وما هي **القواعد** التي تضبط تفاعل هذه المكونات.

بمعنى آخر:

هو قالب تصميمي عام يوجه بناء النظام بطريقة معروفة ومجربة.

أشهر أنواعArchitectural Styles

1. Layered Architecture (الطبقات)

- النظام يقسم إلى طبقات مستقلة.
- كل طبقة تؤدي دورًا معينًا وتتفاعل مع الطبقات القريبة فقط.

مثال:

- Layer 1: واجهة المستخدم.
- Layer 2: منطق الأعمال.
- Layer 3: إدارة البيانات.

مثل: بناء تطبيقات الويب

— Frontend → Backend → Database.

---

2. Client-Server Architecture

- النظام يتكون من **عملاء (Clients)** يطلبون خدمات، و**خوادم (Servers)** تقدم هذه الخدمات.
- العملاء لا يحتاجون معرفة أين أو كيف تتم المعالجة.

مثال:

- متصفحك (Client) يتواصل مع خادم Google (Server).

---

3. Peer-to-Peer Architecture (P2P)

- كل مكون يمكن أن يكون **عميل وخادم** معًا.
- لا يوجد خادم مركزي مسيطر.

مثال:

- تورنت (BitTorrent) ، برامج مشاركة الملفات.

---

4. Event-Driven Architecture

- النظام يتفاعل بناءً على **الأحداث**.**(Events)**
- عندما يحدث حدث معين، النظام يستجيب له بوظيفة محددة.

مثال:

- ضغط زر في تطبيق ← ظهور رسالة.

---

5. Microservices Architecture

- النظام يتكون من **مجموعة خدمات صغيرة مستقلة**.
- كل خدمة تقوم بوظيفة واحدة، وتتواصل عبر الشبكة) عادةً.(API

مثال:

- نظام متجر إلكتروني: خدمة تسجيل دخول مستقلة عن خدمة الدفع، مستقلة عن خدمة الشحن... الخ.

---

6. Service-Oriented Architecture (SOA)

- قريب منMicroservices ، ولكن هنا التركيز يكون على **خدمات أكبر وأثقل** تتكامل مع بعضها.
- غالبًا يستخدم بروتوكولات مثل SOAP و.WSDL

---

7. Pipe-and-Filter Architecture

- النظام مبني كسلسلة من **المرشحات (Filters)** متصلة بواسطة **أنابيب**.**(Pipes)**
- كل فلتر يعالج البيانات ثم يمررها للفلتر التالي.

مثال:

- معالجة ملفات الصوت أو الفيديو: قراءة ← تحويل ← ضغط ← إرسال.

لماذا نحتاج Architectural Styles ؟

- لتسهيل فهم النظام وتصميمه وصيانته.
- لتوزيع الأدوار وتقليل التعقيد.
- للحصول على أنظمة أكثر مرونة وتوسعية (Scalable) وقابلة للتطوير بسهولة.

ملاحظة مهمة

- أحيانًا يستخدم النظام **مزيج من عدة أنماط** معًا لتحقيق أفضل أداء.
- اختيار الـ Style يعتمد على متطلبات المشروع: السرعة؟ القابلية للتوسع؟ التوزيع؟

ما هو Layered Architecture ؟

معمارية الطبقات (Layered Architecture)هي أسلوب تصميم يتم فيه **تقسيم النظام إلى طبقات مستقلة**، بحيث:

- كل طبقة تؤدي مجموعة محددة من الوظائف.
- كل طبقة تتعامل فقط مع الطبقة التي فوقها والتي تحتها.

الفكرة الأساسية:

كل طبقة تعتمد فقط على الخدمات التي توفرها الطبقة التي تحتها، وتخدم الطبقة التي فوقها.

مكوناتLayered Architecture

عادةً يتكون من 3 إلى 5 طبقات، وأشهر تقسيم يكون بهذا الشكل:

1. Presentation Layer (طبقة العرض)

- مسؤولة عن التفاعل مع المستخدم.
- تعرض البيانات وتستقبل الأوامر.

أمثلة:

- واجهة المستخدم الرسومية.(GUI)
- تطبيقات الموبايل أو صفحات الويب.

---

2. Application Layer / Business Logic Layer (منطق الأعمال)

- تعالج المنطق والقواعد الخاصة بالنظام.
- تطبق القوانين والشروط المرتبطة بالتطبيق.

أمثلة:

- حساب الضرائب في برنامج مبيعات.
- التحقق من صحة كلمة المرور.

---

3. Data Access Layer (طبقة الوصول للبيانات)

- تتعامل مع قراءة وكتابة البيانات.
- تفصل منطق الأعمال عن تفاصيل قواعد البيانات.

أمثلة:

- تنفيذ استعلامات.SQL
- التعامل مع API خارجي لاسترجاع بيانات.

4. Database Layer (طبقة قاعدة البيانات)

- تحتوي على البيانات الحقيقية المخزنة.
- تشمل قواعد البيانات أو أنظمة الملفات.

➖أمثلة:

- MySQL، PostgreSQL، MongoDB.

كيف تتواصل الطبقات مع بعضها؟

- المستخدم ⇦ يتعامل مع **Presentation Layer**.
- العرض ⇦ يرسل الأوامر لـ **Business Logic Layer**.
- المنطق ⇦ يستدعي **Data Access Layer** عند الحاجة.
- الوصول ⇦ يتواصل مع **Database Layer** لحفظ أو قراءة البيانات.

كل طبقة تتعامل مباشرةً فقط مع الطبقة القريبة منها! .

✨مميزات Layered Architecture

✅تنظيم الكود وسهولة الفهم.
✅سهولة صيانة النظام وتطويره.
✅إمكانية إعادة استخدام الطبقات في مشاريع أخرى.
✅التبديل بين الطبقات بدون تأثير كبير (مثلاً تغيير قاعدة البيانات بدون تغيير الواجهة).

⚡عيوب Layered Architecture

❌أحيانًا يؤدي إلى أداء أقل (لأن كل طبقة تمرر البيانات للطبقة التالية).
❌ممكن أن تصبح الطبقات متشابكة إذا لم يتم الالتزام بالفصل الجيد.

✍تخيل الشكل البسيط:

```
+--------------------------+
| Presentation Layer       |
+--------------------------+
| Business Logic Layer     |
+--------------------------+
| Data Access Layer        |
+--------------------------+
| Database Layer           |
+--------------------------+
```

📚أمثلة حقيقية

- تطبيقات الويب الحديثة) مثل.(Django, .NET MVC
- تطبيقات الهواتف.
- أنظمة الشركات الكبرى (مثل أنظمة إدارة الطلاب، أنظمة البنوك).

ما هو(Service-Oriented Architecture (SOA ؟

معمارية موجهة نحو الخدمات (SOA)هي أسلوب تصميم يتم فيه **بناء النظام كمجموعة من الخدمات المنفصلة والمستقلة.**

كل **خدمة**:

- تؤدي وظيفة معينة.

- تتواصل مع الخدمات الأخرى عبر **شبكات الاتصال**)غالباً باستخدام بروتوكولات مثلHTTP ، SOAP، .(REST
- يمكن إعادة استخدامها في أنظمة أخرى.

الفكرة الأساسية: كل جزء من النظام هو "خدمة" مستقلة يمكن أن تتكامل مع غيرها.

□خصائصSOA

- **الاستقلالية** :كل خدمة تعمل بشكل مستقل.
- **التواصل عبر الشبكة** :الخدمات تتحدث مع بعضها عن طريق رسائل.(Messages)
- **إعادة الاستخدام** :الخدمة الواحدة يمكن استخدامها في أكثر من نظام.
- **التجريد** :المستخدم أو النظام الآخر لا يحتاج معرفة تفاصيل تنفيذ الخدمة، فقط كيف يتواصل معها.

مكونات SOA الأساسية

1. Service Provider (مزود الخدمة)

- هو الكيان الذي ينشئ وينشر الخدمة.
- يسجل تفاصيل الخدمة في "دليل الخدمات.(Service Registry) "

2. Service Consumer (مستهلك الخدمة)

- هو الكيان الذي يستخدم/يستدعي الخدمة.
- يكتشف الخدمات عبر "دليل الخدمات" ثم يتواصل مع مزود الخدمة.

3. Service Registry (دليل الخدمات)

- قاعدة بيانات أو مكان يتم فيه تسجيل كل الخدمات المتوفرة مع وصف لها.
- يساعد المستهلكين في العثور على الخدمات المناسبة.

🛠️مثال بسيط

تخيل شركة لديها:

- خدمة تسجيل دخول.
- خدمة معالجة مدفوعات.
- خدمة إرسال رسائل بريد إلكتروني.

كل خدمة تعمل بشكل مستقل.
تطبيق الويب أو تطبيق الموبايل يستخدم هذه الخدمات عبر الشبكة.

⚡مقارنة معMicroservices ؟

Microservices	SOA	وجه المقارنة
صغير جدًا	كبير نسبيًا	حجم الخدمة
خفيفة(REST, gRPC)	ثقيلة) مثلSOAP)	طريقة الاتصال
كل خدمة لها قاعدة بيانات مستقلة	عادةً مشتركه	قاعدة البيانات
الأنظمة الحديثة والخفيفة	أنظمة المؤسسات الضخمة	مناسب لـ

)بمعنى آخر Microservices :تعتبر نسخة أخف وأحدث من(.SOA

✨مميزاتSOA

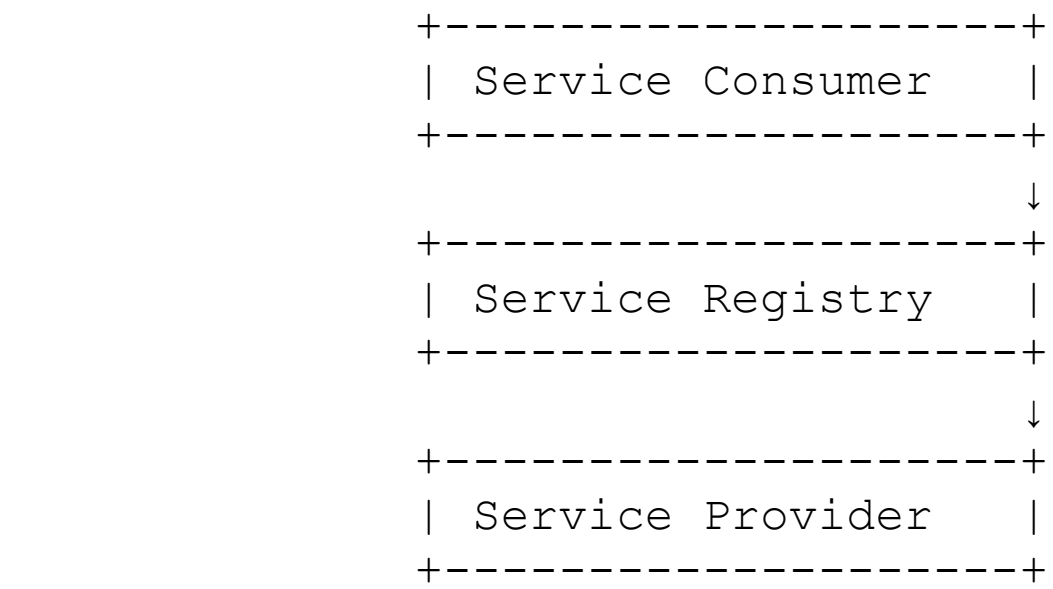
✅مرونة عالية في بناء أنظمة معقدة.
✅إعادة استخدام الخدمات عبر مشاريع مختلفة.

✅تسهيل التكامل مع أنظمة قديمة.(Legacy Systems)
✅تحسين الصيانة والتوسيع.

⚠عيوبSOA

❌ممكن زيادة التعقيد بسبب كثرة الخدمات والاتصالات.
❌الأداء قد يتأثر بسبب طبيعة التواصل عبر الشبكة.
❌الحاجة لإدارة جيدة للأمان والتكامل بين الخدمات.

✍الشكل البسيط لمعماريتها:



🏭أمثلة حقيقية

- أنظمة البنوك الكبيرة.
- شركات الاتصالات.
- أنظمة المستشفيات الضخمة.
- ERP Systemsمثل.SAP

أولاً: ما هو Coordination؟

في أنظمة البرمجيات أو الأنظمة الموزعة،
التنسيق (Coordination) يعني **كيفية تنظيم التواصل والتفاعل بين مكونات النظام** بحيث تعمل معاً بطريقة صحيحة ومنظمة.

مثلاً:

- كيف تتزامن الخدمات مع بعضها؟
- كيف تنتقل الرسائل أو البيانات بينهم؟
- كيف يعرف جزء من النظام متى وأين يتواصل مع جزء آخر؟

بدون تنسيق جيد → يحصل فوضى في النظام!

ثانياً) Temporal Coupling :الترابط الزمني(

تعريف:

- يحدث عندما **يجب** أن يكون **طرفان أو أكثر متاحين في نفس اللحظة** حتى يتم التفاعل بينهم.
- إذا كان أحد الأطراف غير متاح، العملية كلها تفشل.

ببساطة: لازم يكونوا "أونلاين مع بعض" علشان يكملوا الشغل.

مثال بسيط:

- مكالمة هاتفية: الطرفين لازم يكونوا متصلين بنفس الوقت.
- طلب API مباشر: الخدمة A تطلب من الخدمةB ، إذا B غير متاحة → فشل الطلب.

مشاكل:Temporal Coupling

- يسبب هشاشة في النظام (لو جزء بسيط تعطل، تتوقف العملية كلها).
- يقلل من المرونة (صعب التوسع أو التعامل مع الأعطال).

ثالثاً) Referential Coupling :الترابط المرجعي(

تعريف:

- يحدث عندما **يعرف أحد الأجزاء معلومات مباشرة أو تفاصيل محددة عن جزء آخر** لكي يستطيع التعامل معه.

يعني: جزء يعرف اسم أو موقع أو تفاصيل جزء آخر بالتحديد.

مثال بسيط:

- خدمة A تحتاج أن تعرف عنوان URL لخدمة B لكي تتواصل معها.
- أو برنامج يحتاج معرفة اسم المستخدم وكلمة السر للدخول إلى قاعدة بيانات.

مشاكل:Referential Coupling

- يجعل النظام معتمد بشكل قوي على مكونات معينة.
- يصعب التغيير أو الاستبدال (مثلاً لو غيرت اسم الخدمة أو عنوانها، لازم تغير كل شيء يعتمد عليها).
- يقلل القابلية للصيانة والتوسعة.

هدف التصميم الجيد:

- **التقليل من ← Temporal Coupling** مثلا باستخدام طوابير الرسائل (Message Queues) أو الأحداث (Events) بدلاً من الاتصال المباشر.
- **التقليل من ← Referential Coupling** مثلا باستخدام اكتشاف الخدمات (Service Discovery) أو كسر الاعتمادات المباشرة بين المكونات.

الخلاصة البسيطة:

الخطر	المعنى	المصطلح
يقلل المرونة	لازم الطرفين يكونوا موجودين بنفس الوقت	Temporal Coupling
يقلل القابلية للصيانة	أحد الأجزاء يعرف تفاصيل الجزء الآخر	Referential Coupling

---

ما هو التنسيق (Coordination) بشكل عام؟

كما قلنا قبل:
= **Coordination**تنظيم التعاون بين مكونات النظام أو العمليات بطريقة صحيحة ومنظمة.

في الأنظمة الموزعة أو الأنظمة الكبيرة، في أكثر من نوع من التنسيق بحسب الطريقة التي تتفاعل فيها المكونات مع بعضها.

أنواع التنسيق(Types of Coordination)

1. Direct Coordination (التنسيق المباشر)

تعريف:

- مكون واحد يتواصل **مباشرة** مع مكون آخر.
- يتم عبر رسائل مباشرة أو استدعاءات.(Calls)

مثال:

- خدمة A ترسل طلب مباشر لخدمة B عبر.REST API
- مكالمة هاتفية بين شخصين.

المميزات:

- بسيط وسريع في الأنظمة الصغيرة.

العيوب:

- يزيد من الترابط (Coupling) بين المكونات.
- لو تعطل طرف، يتعطل التواصل.

---

2. Indirect Coordination (التنسيق غير المباشر)

تعريف:

- لا يتم التواصل بشكل مباشر بين المكونات.
- يتم عبر وسيط) مثل Message Queue أو.(Event Bus

مثال:

- خدمة A ترسل رسالة إلىQueue ، خدمة B تقرأ الرسالة لاحقًا.
- نشر أحداث (Publish/Subscribe) حيث لا تعرف الجهة المرسلة من سيستقبل الحدث.

المميزات:

- يقلل الترابط.(Coupling)
- يعطي مرونة أكبر وتعامل أفضل مع الأعطال.

3. Temporal Coordination (التنسيق الزمني)

تعريف:

- يعتمد على **التوقيت**، أي الأطراف يجب أن تتزامن (يكونوا متاحين بنفس الوقت).

مثال:

- مكالمة فيديو بين أكثر من شخص.
- Clientيتصل بـ Server ويحتاج جواب فوري.

المميزات:

- تفاعل لحظي وسريع.

العيوب:

- لو أحد الأطراف غير متاح ← فشل العملية.

---

4. Spatial Coordination (التنسيق المكاني)

تعريف:

- يعتمد على أن يعرف الطرفان معلومات معينة عن بعضهم (مثل الموقع، العنوان، معرف الخدمة).

مثال:

- خدمة تحتاج معرفة عنوان IP لخدمة أخرى للتواصل معها.

المميزات:

- بسيط في الأنظمة الصغيرة.

العيوب:

- يضعف القابلية للتوسيع والتغيير.

---

5. Referential Coordination (التنسيق المرجعي)

تعريف:

- أحد المكونات يحتاج معرفة **الهوية أو المرجع** للمكون الآخر لكي يتفاعل معه.

)لاحظ أن Spatial و Referential قريبين، لكن Referential يركز أكثر على المعرفات والهوية(.

مثال:

- برنامج يحتاج اسم قاعدة البيانات أو معرف خاص بخدمة ما لكي يتواصل معها.

🎯مقارنة سريعة:

أهم ميزة	مثال	يعتمد على	نوع التنسيق
سرعة وبساطة	REST API Call	الاتصال المباشر	Direct
تقليل الترابط	Message Queue	وسيط(Broker)	Indirect
تفاعل لحظي	مكالمة فورية	التوقيت المتزامن	Temporal
بساطة في التواصل	معرفة عنوان خدمة	معرفة الموقع	Spatial
ربط محدد وواضح	معرفة معرف خدمة	معرفة الهوية	Referential

🛠تلخيص سريع:

كلما قللنا الترابط (Coupling) بين الأجزاء، كان النظام **أكثر مرونة وأسهل في التطوير والصيانة**.

🔥ملاحظة:

في الأنظمة الحديثة) مثل(Microservices ، نفضل عادة استخدام:

- **Indirect Coordination** (عن طريق Kafka أو RabbitMQ مثلاً).
- **التقليل من Temporal و Referential Coupling.**

🔥:

ما هو System Architecture (هندسة النظام)؟

System Architectureيعني:

➔**التصميم الشامل أو الإطار العام** لكيفية بناء النظام،

➔وكيف تتفاعل المكونات المختلفة مع بعضها البعض.

بمعنى آخر:

- كيف نقسم النظام إلى أجزاء؟
- ما هي المكونات الأساسية؟
- كيف تتواصل المكونات مع بعضها؟
- كيف يتم توزيع الأدوار والمسؤوليات؟

✅الفكرة الأساسية هي **التخطيط الذكي** لبناء نظام قوي، منظم، وسهل التطوير والصيانة.

مكونات System Architecture عادةً:

1. **مكونات النظام:(Components)**
مثل: قاعدة بيانات، خوادم، واجهة مستخدم، خدمات مساندة، إلخ.
2. **التواصل بين المكونات:(Communication)**
مثل: REST APIs, Messaging Systems, Direct Calls.
3. **طرق التوزيع:(Distribution)**
هل النظام مركزي أم موزع(Distributed System) ؟
4. **التنظيم الداخلي:(Internal Organization)**
هل نعتمد على طبقات(Layers) ، أو خدمات مستقلة (Microservices)، أو نماذج أخرى؟
5. **الخصائص غير الوظيفية Non-functional) Requirements):**
مثل الأداء، الأمان، قابلية التوسع، الموثوقية.

أنواع شائعة من:System Architecture

نوع الهندسة	وصف مبسط	مثال
Monolithic Architecture	النظام بالكامل كتلة واحدة	تطبيق قديم فيه كل شيء داخلياً
Layered Architecture	النظام مقسم إلى طبقات مستقلة	تطبيق بثلاث طبقات: عرض، منطق أعمال، بيانات
Microservices Architecture	النظام مكون من خدمات صغيرة مستقلة	أمازون، نتفليكس
Client-Server Architecture	جهاز العميل يطلب من الخادم	مواقع الإنترنت العادية
Peer-to-Peer (P2P) Architecture	جميع الأجهزة تتشارك مع بعض	تورنتBitTorrent -
Event-driven	النظام مبني حول	أنظمة إشعارات الوقت

مثال	وصف مبسط	نوع الهندسة
الحقيقي	الأحداث(Events)	**Architecture**

🎁مثال عملي بسيط:

لو نصمم **تطبيق متجر إلكتروني**:

- **واجهة المستخدم :(Frontend)** موقع أو تطبيق يعرض المنتجات.
- **خدمة الطلبات :(Order Service)** تدير الطلبات والمشتريات.
- **قاعدة البيانات :(Database)** تخزن معلومات العملاء والمنتجات.
- **خدمة الدفع :(Payment Service)** تعالج عمليات الدفع.
- **نظام إشعارات :(Notification Service)** يرسل رسائل للمستخدمين.

☞هذا كله جزء من :**System Architecture**تقسيم النظام إلى مكونات + كيف تتفاعل مع بعض.

🎯لماذا System Architecture مهم؟

- يجعل النظام **منظماً** وسهل الفهم.
- يسهل **الصيانة والتطوير**.
- يحسن من **الأداء وقابلية التوسع**.
- يزيد من **الأمان والموثوقية**.
- يوفر **مرونة** لتحديث أو استبدال أجزاء معينة بدون كسر النظام كله.

✨الخلاصة البسيطة:

System Architecture = الخريطة الأساسية لبناء النظام.
بدونها، المشاريع تصير فوضى وتواجه مشاكل كبيرة مع التوسع أو الصيانة.

📄 ما هو Simple Client-Server Architecture ؟

هو **أبسط وأشهر نمط** للتواصل بين جهازين أو أكثر:

- طرف يسمّى **العميل (Client)** ← يطلب خدمة أو بيانات.
- طرف يسمّى **الخادم (Server)** ← يقدم الخدمة أو البيانات المطلوبة.

✅العميل يعتمد على الخادم لتوفير الوظائف المطلوبة.

⚙كيف يعمل؟

1. **العميل (Client)** يبدأ الطلب ← مثلا يطلب صفحة ويب أو يسجل دخول.
2. **الخادم (Server)** يستقبل الطلب ← يعالجه ويرسل الرد المناسب.
3. **العميل** يستلم الرد ويعرضه للمستخدم.

🔁التفاعل دائماً يبدأ من العميل.

مكونات:Simple Client-Server

الوظيفة	المكون
يرسل الطلبات ويعرض النتائج للمستخدم.	**Client**
يعالج الطلبات ويرسل الردود.	**Server**

المكون	الوظيفة
Network	وسيط الاتصال بين العميل والخادم (مثل الإنترنت أو شبكة محلية).

🔥أمثلة واقعية:

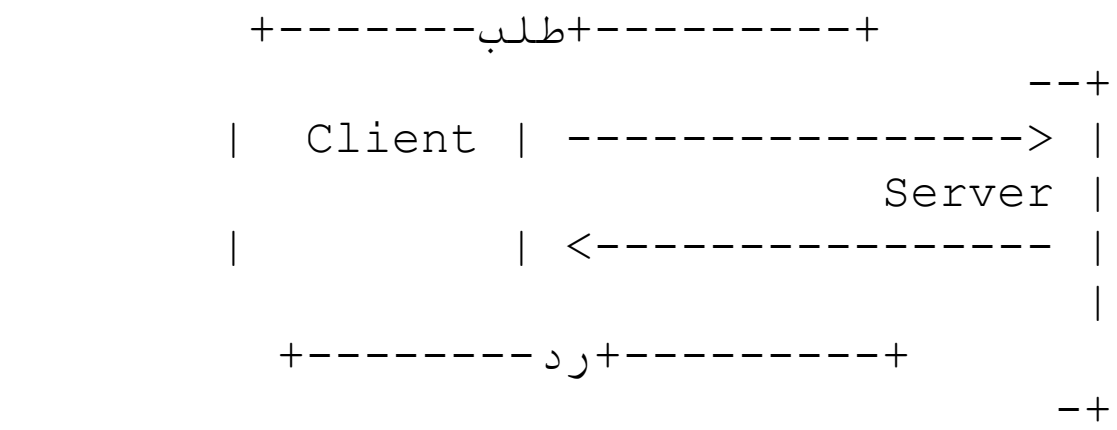
- عندما تفتح متصفح الإنترنت وتدخل رابط موقع ➔ أنت (العميل) تطلب صفحة ➔ الخادم يرد بالصفحة.
- تطبيقات البريد الإلكتروني ➔ جهازك يطلب رسائل جديدة من خادم البريد.
- لعبة أونلاين ➔ اللاعب يرسل أوامر لخادم اللعبة، والخادم يعيد الحالة الجديدة.

🎯خصائص:Simple Client-Server Architecture

✅**بسيط وسهل التنفيذ.**
✅**مناسب للأنظمة الصغيرة والمتوسطة.**
✅**تقسيم واضح بين العملاء والخادم.**

❌**لو تعطل الخادم، يتوقف النظام كله** .(Single Point of Failure)
❌**الخادم قد يصبح عبء إذا عدد العملاء كبير جداً**)مشكلة التحميل العالي.(

رسم تخطيطي بسيط:



ملاحظة مهمة:

مع تطور الأنظمة، ظهر تحسينات على هذا النموذج مثل:

- **Multi-tier Architecture** ← تقسيم المهام إلى أكثر من طبقة (مثلاً واجهة أمامية + منطق أعمال + قاعدة بيانات).
- **Load Balancers** ← لموازنة التحميل على أكثر من خادم.

لكن مبدأ **Client-Server** هو الأساس في كل هذه التطورات.

الخلاصة:

Simple Client-Server = عميل يطلب، خادم يرد.
أساس كل شيء بسيط ولكنه مهم جداً لبناء أنظمة أكبر وأكثر تعقيداً.

ما هي Multitiered Architectures ؟

Multitiered Architecture (أو **N-Tier Architecture** تعني:
← تقسيم النظام إلى **عدة طبقات (Tiers)** ،

➔كل طبقة تكون مسؤولة عن وظيفة محددة،
➔والطبقات تتواصل مع بعضها بطريقة منظمة.

✓الهدف: جعل النظام **أكثر تنظيمًا، سهولة في التطوير، وإدارة أفضل للصيانة.**

□الفكرة الأساسية:

بدلاً من دمج كل شيء في مكان واحد) زي Client-Server العادي(، نقوم بتقسيم النظام إلى طبقات مختلفة، كل طبقة مسؤولة عن جزء محدد.

مثال شائع جدًا :**نظام بثلاث طبقات(Three-Tier Architecture)**

الطبقة(Tier)	الوظيفة	مثال
Presentation Tier	واجهة المستخدم	صفحات الويب أو التطبيقات
Application Tier (Logic)	منطق الأعمال	سيرفر يعالج العمليات
Data Tier	إدارة البيانات	قواعد البيانات

🛠كيف تعمل Multitiered Architecture عمليًا؟

1. **المستخدم** يتفاعل مع الواجهة.(Presentation Layer)
2. الواجهة ترسل الطلب إلى **منطق الأعمال (Application Layer)** لمعالجة الطلب.
3. منطق الأعمال يتواصل مع **قاعدة البيانات (Data Layer)** لاسترجاع أو حفظ المعلومات.
4. يتم إعادة النتيجة إلى الواجهة ثم إلى المستخدم.

لـ

ماذا نستخدمMultitiered Architecture ؟

✓**فصل المهام بوضوح** ←كل طبقة متخصصة ومركزة في وظيفتها.
✓**سهولة التعديل** ←ممكن تعدل طبقة بدون تأثير كبير على الباقي.
✓**تحسين الأمان** ←منع العملاء من الوصول المباشر إلى قواعد البيانات.
✓**قابلية التوسع ← (Scalability)** تستطيع توسيع كل طبقة بشكل مستقل حسب الحاجة.
✓**مرونة أعلى** ←تغيير التكنولوجيا في طبقة معينة بدون تغيير النظام كله.

تخطيط مبسط:

```
lua
نسختحرير
                    +--------------------+
) <--  | Presentation Tier  |تطبيق أو
متصفح(
                    +--------------------+
                              ↓
                    +--------------------+
) <--  | Application Tier   |خادم
يعالج البيانات، منطق الأعمال(
                    +--------------------+
                              ↓
                    +--------------------+
) <--  | Data Tier          |قاعدة
بيانات مثل MySQL أو(Oracle
                    +--------------------+
```

🔥أمثلة حقيقية:

- **موقع تسوق إلكتروني:**
  - الواجهة تعرض المنتجات.
  - الخادم يعالج عمليات الطلب والدفع.
  - قاعدة البيانات تخزن معلومات المنتجات والعملاء.
- **تطبيقات الهواتف الذكية:**
  - التطبيق يرسل طلبات إلى سيرفر.
  - السيرفر يعالج العمليات.
  - البيانات تُخزن أو تُسترجع من قاعدة بيانات سحابية.

✨أنواع أخرى متقدمة:

- **Two-Tier Architecture:** فقط واجهة + قاعدة بيانات مباشرة.
- **Three-Tier Architecture:** واجهة + منطق أعمال + قاعدة بيانات.
- **N-Tier Architecture:** أكثر من 3 طبقات) ممكن إضافة طبقات مثل Cache Layer ، Security Layer، وغيرها.(

⚡الخلاصة:

Multitiered Architecture = تقسيم ذكي للنظام إلى طبقات مستقلة، لكل طبقة شغلها الخاص.
تسهل حياتنا لما نطور، نحدث أو نوسع النظام.

ما هو Three-Tiered Architecture ؟

هو نوع من **Multitiered Architecture**،
وفيه النظام ينقسم إلى **ثلاث طبقات رئيسية**، كل طبقة لها وظيفة محددة.

✅الهدف: جعل النظام **أكثر تنظيمًا، قابلية للصيانة، وأسهل في التوسيع.**

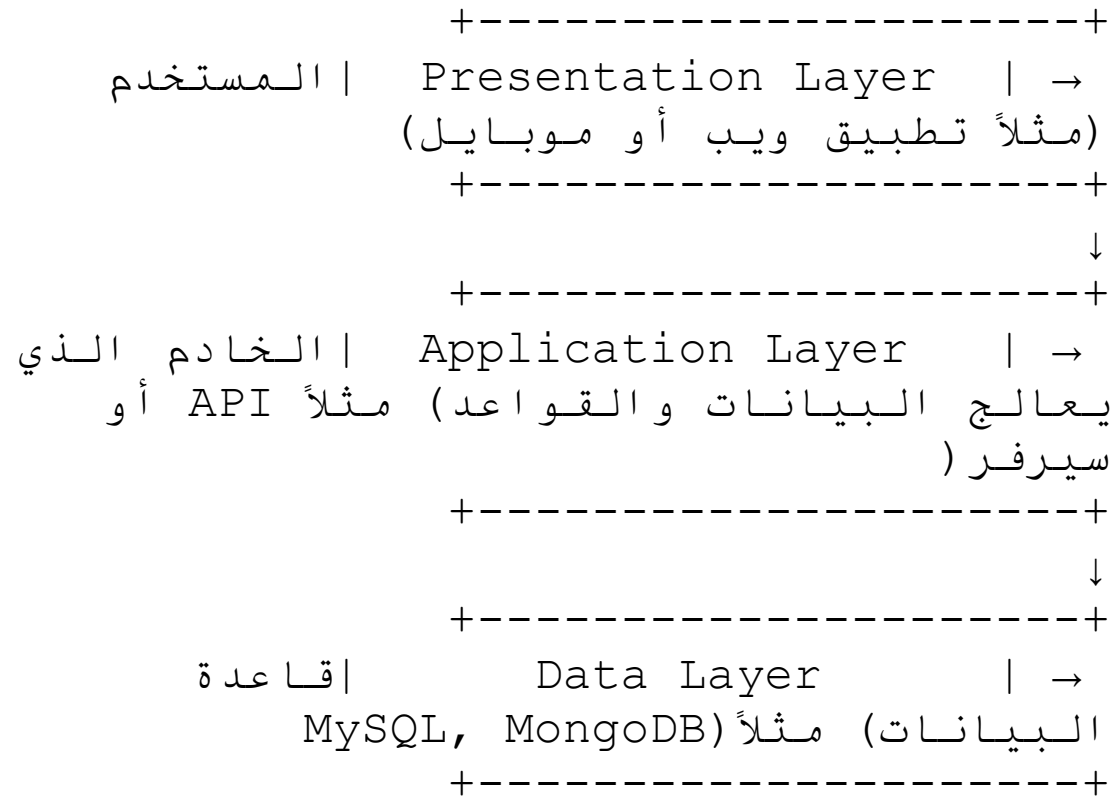
الطبقات الثلاث:(Tiers)

الطبقة	الوظيفة	شرح مبسط
Presentation Tier	واجهة المستخدم	الواجهة التي يتفاعل معها المستخدم مثل موقع ويب أو تطبيق موبايل.
Application Tier (Business Logic)	منطق الأعمال	يعالج العمليات والقواعد التجارية، وينظم الاتصال بين الواجهة والبيانات.
Data Tier	طبقة البيانات	تخزين البيانات (مثل قواعد البيانات) واسترجاعها عند الطلب.

⚙كيف يعمل Three-Tiered Architecture ؟

1. **المستخدم** يتفاعل مع **واجهة المستخدم (Presentation Layer).**
2. هذه الواجهة ترسل الطلب إلى **طبقة منطق الأعمال (Application Layer).**
3. منطق الأعمال يتواصل مع **طبقة البيانات (Data Layer)** لاسترجاع أو تعديل البيانات.
4. يتم إعادة النتيجة إلى التطبيق ومن ثم عرضها للمستخدم.

🖼تخطيط بسيط:



مثال عملي:

موقع تسوق إلكتروني:

- **Presentation Tier:** صفحة المنتجات - تعرض معلومات المنتج للمستخدم.
- **Application Tier:** الخادم يعالج طلبات شراء المنتج، ويتأكد أن الكمية متوفرة، يحسب السعر مع الضرائب.
- **Data Tier:** تخزين معلومات المنتجات، المستخدمين، والطلبات في قاعدة بيانات.

مميزات: Three-Tier Architecture

✓ **تنظيم قوي :** كل طبقة مستقلة عن الأخرى.
✓ **قابلية للصيانة :** تستطيع تعديل أو تطوير طبقة معينة بدون التأثير على الباقي.

✅**تحسين الأمان :**المستخدم لا يتواصل مع قاعدة البيانات مباشرة.
✅**قابلية للتوسيع :**ممكن تكبير كل طبقة بشكل مستقل حسب حاجة المشروع.
✅**سهولة التحديث :**تحديث الواجهة أو منطق الأعمال بدون تغيير قاعدة البيانات.

⚡**مقارنة سريعة مع:Two-Tier Architecture**

Three-Tier	Two-Tier
العميل يتواصل مع خادم، والخادم يتواصل مع قاعدة البيانات.	العميل يتصل مباشرة مع قاعدة البيانات.
أكثر تنظيمًا وقابلية للتوسع	أقل تنظيمًا
مناسب للتطبيقات الكبيرة والمعقدة	مناسب للتطبيقات الصغيرة

✨**الخلاصة:**

Three-Tiered Architecture = واجهة + منطق أعمال + قاعدة بيانات، مفصولين بوضوح النظام يكون قوي وسهل التطوير.

📄 ما هو Symmetrically Distributed System Architecture؟

Symmetrically Distributed Systemيعني أن النظام يتكون من **عدة مكونات موزعة عبر الشبكة**، وكل مكون (أو عقدة) في النظام له **دور متساوي** ولا يوجد مكون رئيسي يتحكم في باقي المكونات. جميع العقد (الأجهزة أو الأنظمة) تعمل بشكل **متساوٍ ومتوازي**.

✓الفكرة الأساسية هنا هي أن **كل عقدة أو مكون لديه نفس القدرة على المعالجة والتواصل**.

الخصائص الرئيسية:

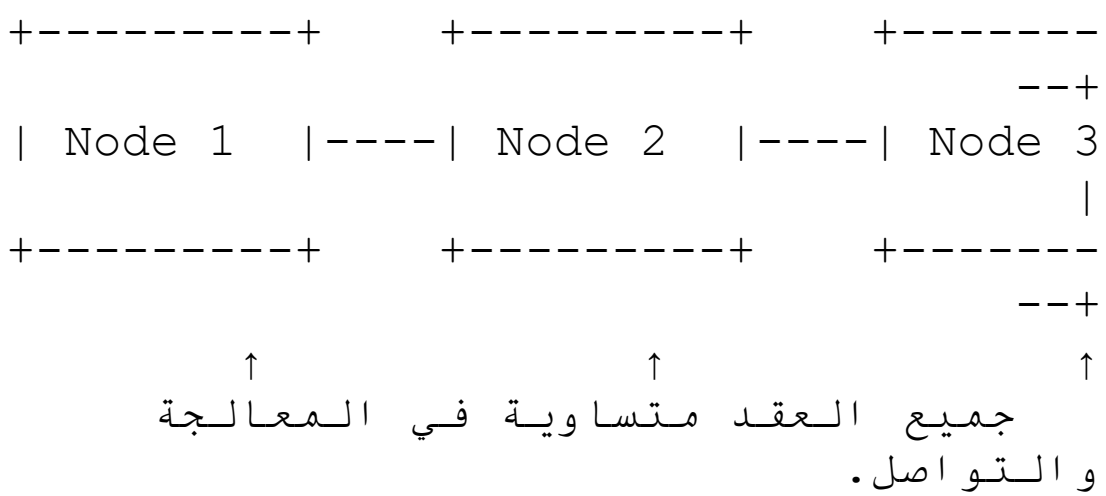
1. **التوزيع المتساوي:**
   كل عقدة أو جهاز في النظام لديه نفس الحقوق والقدرات ولا يعتمد على عقدة واحدة رئيسية.
2. **التفاعل المتوازي:**
   العقد تتواصل مع بعضها وتتعامل مع المهام بالتوازي، مما يحسن الأداء ويساهم في توزيع الحمل بشكل أفضل.
3. **الاستقلالية:**
   كل عقدة قادرة على العمل بشكل مستقل (على الرغم من أنها تتواصل مع باقي العقد).
4. **المرونة:**
   بإضافة عقد جديدة، يتم توسيع النظام بسهولة دون التأثير على بقية العقد.

كيف يعمل Symmetrically Distributed System ؟

- في هذا النوع من الأنظمة، كل عقدة تكون **مستقلة تمامًا**.
- هذه العقد تتواصل مع بعضها عبر الشبكة باستخدام بروتوكولات معينة.
- كل عقدة في النظام تتشارك **المهام الحسابية** وتساهم في **حل المشكلة ككل**.

مثال :في **نظام موزع** مثل **Blockchain**أو **شبكات) P2P نظير إلى نظير(**، كل جهاز (أو عقدة) في الشبكة يعمل بنفس الطريقة ولا يوجد خادم رئيسي أو "رئيسي."

🖼️رسم تخطيطي مبسط:



🔥أمثلة على:Symmetrically Distributed Systems

1. **Blockchain**:
في شبكات مثل **Bitcoin**أو **Ethereum**، لا يوجد خادم رئيسي أو سلطة مركزية. كل **عقدة** في الشبكة تعمل على إضافة وتوثيق المعاملات بشكل متساوٍ.
2. **شبكات:Peer-to-Peer (P2P)**
في شبكات مثل **BitTorrent**، كل جهاز (أو **Peer**في الشبكة يشارك في تحميل ورفع الملفات. لا يوجد جهاز رئيسي، الجميع يعمل بشكل متساوٍ.
3. **أنظمة الحوسبة الموزعة:**
مثل أنظمة الحوسبة السحابية حيث تقوم العقد بتنفيذ أجزاء من العمليات الحسابية بشكل متوازي.

🎯مزايا:Symmetrically Distributed Systems

✅مرونة وموثوقية عالية:

- إذا تعطلت إحدى العقد، يبقى النظام يعمل بفضل العقد الأخرى التي تواصل العمل.

✓التوزيع العادل للحمل:

- نظراً لتوزيع المهام بالتساوي بين العقد، فإن الضغط على النظام يوزع بالتساوي، مما يعزز الأداء.

✓تحمل الأعطال:

- النظام لا يتأثر إذا تعطل جزء منه لأن هناك عقدًا أخرى يمكنها التعامل مع المهام.

⚠عيوب:Symmetrically Distributed Systems

✗التحديات في التنسيق والإدارة:

- بما أن كل عقدة لها نفس الحقوق، يصبح التنسيق بين العقد أمرًا معقدًا خاصة في الأنظمة الكبيرة.

✗تعقيد الشبكة:

- كل عقدة تحتاج إلى التفاعل مع غيرها مما قد يضيف تعقيدًا في إدارة الشبكة.

✨الخلاصة:

Symmetrically Distributed Systemsهي أنظمة تحتوي على **عقد متساوية في المعالجة والتواصل**، مما يساهم في زيادة الأداء، التحمل، والمرونة. هذا النمط مثالي للأنظمة التي تتطلب توزيعًا عادلًا للمهام مثل **الأنظمة الموزعة، شبكاتP2P ، والـBlockchain.**
