Application Design

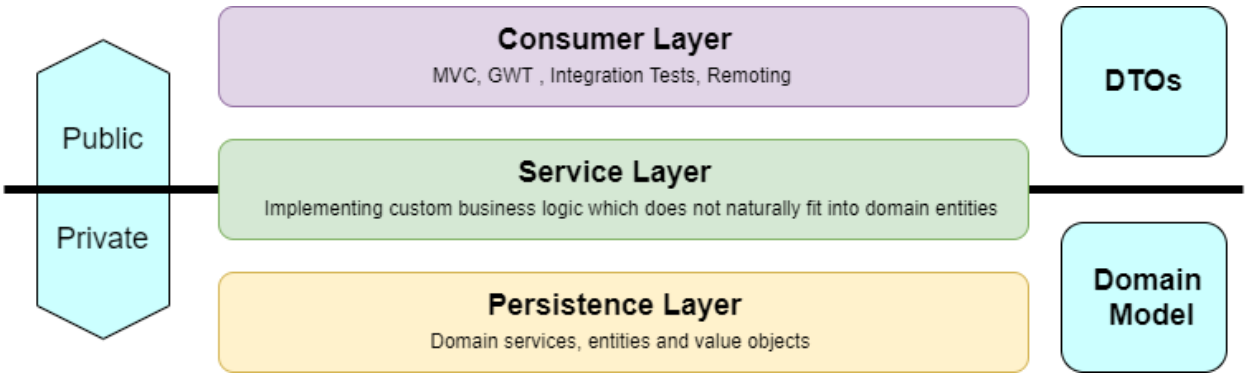
مفهوم

در این یادداشت سعی می‌کنم اصولی که در طراحی نرم افزار باید رعایت کنیم رو بنویسم.

معماری سه‌لایه در نرم‌افزار - Three layer application architecture

مکانیزم لایه بندی روشی است که در آن ما کد های مربوط به قسمت های مختلف را به شکل جداگانه از هم دیگر می‌نویسیم این کار باعث می‌شود که هر لایه مجموعه وظیفه مشخصی داشته باشد و نوشتن تست و توسعه و همچنین نگهداری کد راحت تر باشد.

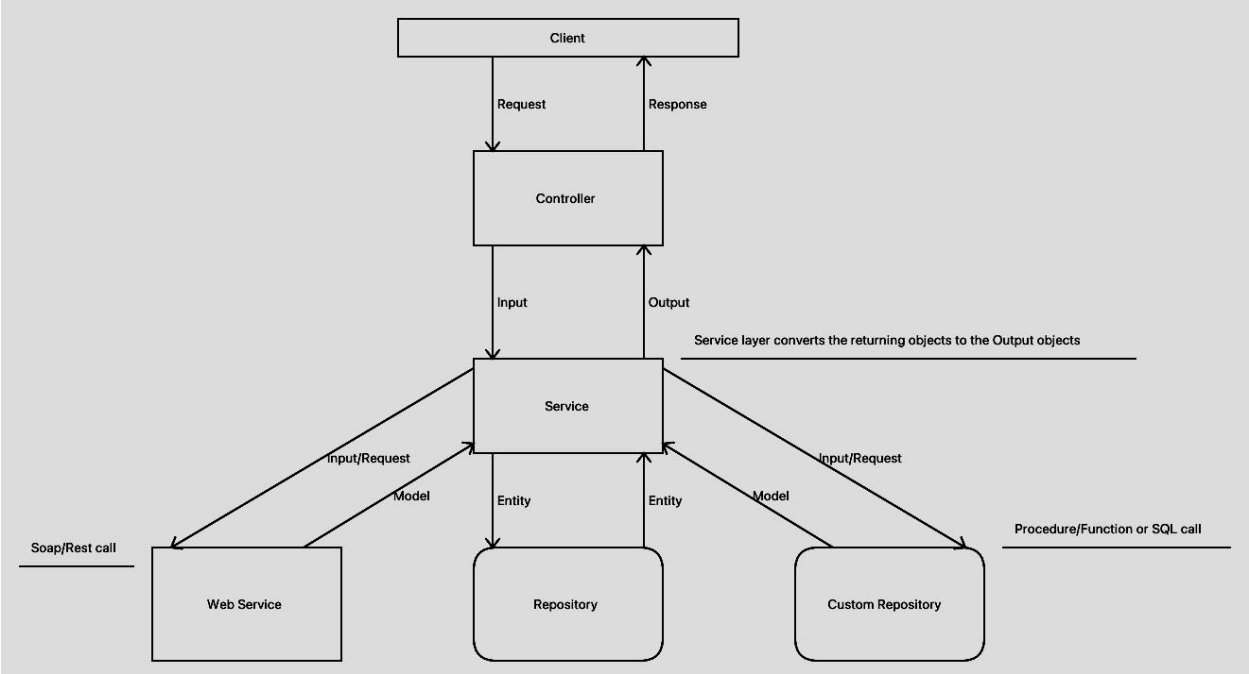
یک معماری معروف در لایه بندی ها معماری سه لایه است که در تصویر زیر آن را مشاهده می‌کنید :



سه لایه اصلی این معماری عبارت اند از :

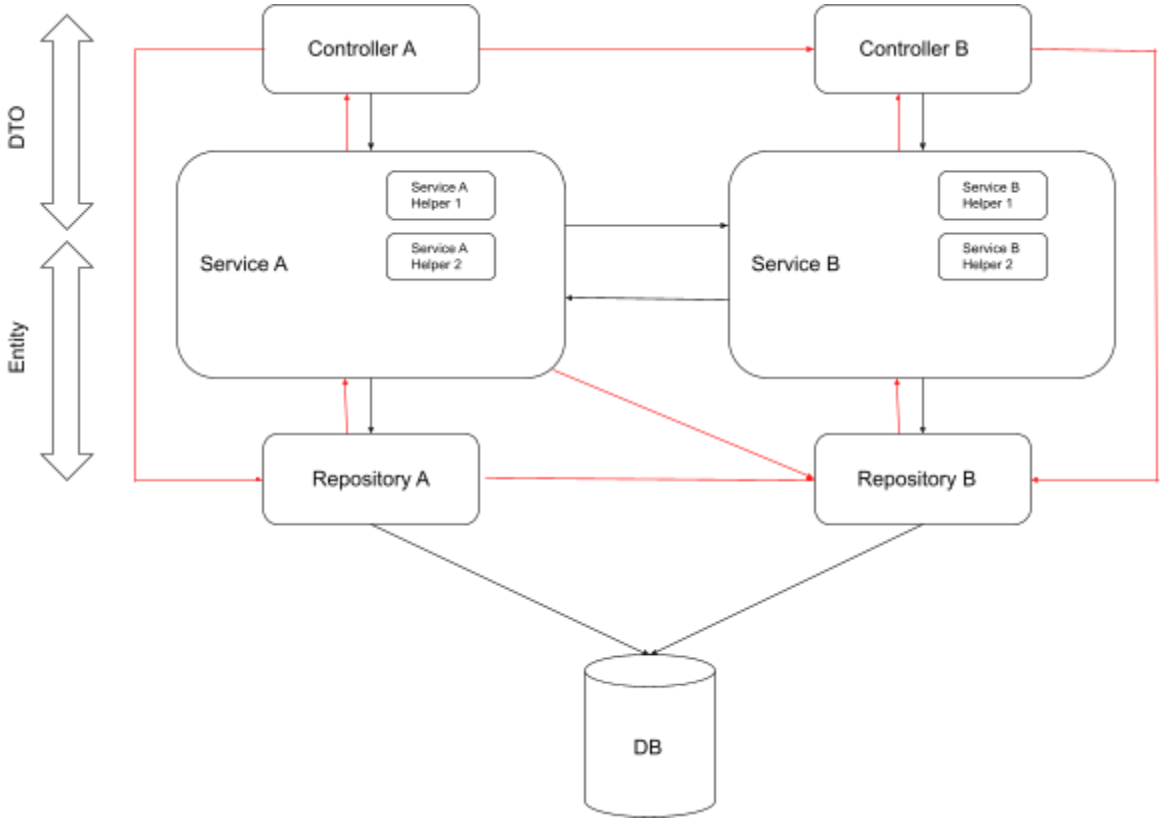
1. Controller : این لایه وظیفه ارتباط با خارج را بر عهده دارد. در واقع هر چیزی که مربوط به api و نحوه ارتباط با خارج باشد در این لایه قرار می‌گیرد.
2. Service : این لایه Business logic را بر عهده دارد
3. Repository : این لایه با Database ارتباط دارد و کار های مربوط به ارتباط با دیتابیس در این لایه اتفاق می‌افتد

برای درک بهتر و دقیق تر کارکرد و ارتباط بین لایه ها به تصویر زیر توجه کنید :



یک نام گذاری دیگری برای لایه ها وجود دارد ، در این نام گذاری لایه ها و کارکردشان تغییر نمی‌کند و فقط اسم آنها عوض می‌شود. این نام گذاری به شرح زیر است :

1. Service
2. Manager
3. DAO



نکته : در تصویر بالا خطوط قرمز روابط و وابستگی های غیر مجاز را نشان می‌دهد

با توجه به تصویر بالا قواعد زیر را در نظر بگیرید :

- در لایه Controller :
  - یک کنترلر به هیچ کنترلر دیگری وابسته نیست.
  - یک کنترلر به هیچ DAO ای وابسته نیست.
  - فقط کدهایی را که نمی توان – به دلیل وابستگی به کتابخانه UI و یا فناوری های REST یا سایر پروتکل‌های سرویس دهی - به لایه Service انتقال داد در این کلاس قرار دارند.
  - Controller یک Feature فقط با Service خود در ارتباط است.
  - یک متد در لایه controller تنها یک متد از لایه service را فراخوانی می کند. یعنی چند مراجعه از یک کنترلر به Service ممنوع است. (چرا؟ چون باعث تزریق منطق به لایه سرویس می شود. منطق باید به کلی در لایه service مدیریت شود.)

- در لایه Service :

- کلاس‌های این لایه عمدتاً در نام خود پسوند Service دارند.
- هیچ وابستگی به کتابخانه‌های REST یا SOAP در این کلاس وجود ندارد.
- service از وجود کلاس controller بی اطلاع است.
- از پکیج تکنولوژی ORM هیچ کلاسی import نشده است.
- همه متدهای public دارای Transactional@ می‌باشند.
- منطق برنامه در این لایه منعکس شده است. یعنی هر چیزی را که بتوان مستقل از کتابخانه واسط کاربر یا سرویس‌های REST و SOAP و امثال آن‌ها و کتابخانه ORM (Hibernate, Jpa) پیاده کرد به این لایه انتقال داده شده است. این کلاس تنها می‌تواند برای پیاده‌سازی منطق برنامه از کلاس‌های موجودیت (*Entity) کمک بگیرد.
- Service یک Entity تنها با Repository خود در ارتباط است.
- کلاس های Service پتانسیل تبدیل به God Class را دارند. در مواقع لازم کد های Cohesive را دور هم جمع کنید و به عنوان Helper یک Service یک کلاس استخراج کنید.

- در لایه Repository :
  - کلاسهای این لایه پسوند Repository دارند.
  - فقط کدهایی که به کتابخانه ORM یعنی JPA یا Hibernate وابسته است در این لایه قرار دارد.
  - Repository به هیچ Repository دیگر وابستگی ندارد.
  - Repository از وجود لایه های بالاتر بی اطلاع است.
  - ورود Business logic به Repository ممنوع است.

دسترسی لایه های مختلف به هم :

- چرا از Service B در Controller A استفاده نکنیم؟

اگر Controller A برای انجام کارهایش هم به Service A نیاز دارد و هم به Service B یعنی یک سری منطق در این لایه نوشته ایم. تمامی این منطق باید به یک لایه پایین تر منتقل شود.

- چرا استفاده از Dao B در Service A ممنوع شده است؟

فرض کنید در لایه ی Service B، بخواهیم به ازای هر حذف یا اضافه شدن به دیتابیس، منطق خاصی صورت بگیرد. حال اگر Service A دسترسی مستقیم به Dao B داشته باشد امکان دور زده شدن این منطق وجود دارد. به عبارت دیگر، استفاده مستقیم از Dao یک feature دیگر می تواند باعث نادیده گرفته شدن بخشی از منطقی شود که در Service ها نوشته شده است.

- چرا تزریق Dao در Dao ممنوع شده است؟

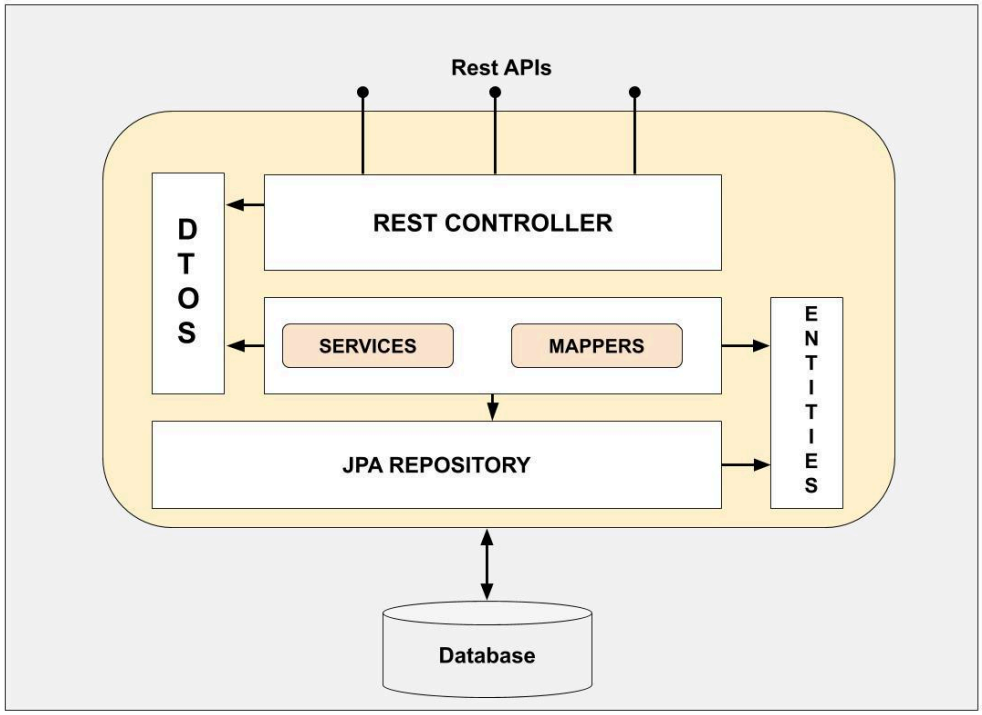
دلیل این نیز مانند بخش قبل است. می خواهیم Service هر فیچر کنترل کاملی بر روال دستکاری داده ها داشت باشد و در جریان هر گونه فراخوانی لایه Dao اش قرار گیرد.
از سوی دیگر نیاز به تزریق Dao در Dao علامتی از نشت Business Logic در لایه Dao است. وگرنه نباید چنین نیازمندی احساس شود.

اعتبار سنجی ورودی کاربر

ما در سمت سرور سه لایه ی service ، controller و repository را داریم ، ورودی کاربر از طریق لایه controller وارد می‌شود. دو شکل از اعتبار سنجی داریم که می‌توانیم در دولایه ی controller و service پیاده سازی کنیم.
در لایه کنترلر درست بودن و فرمت درست داده را می‌توانیم چک کنیم که این کار می‌تواند با فریمورک bean validator انجام شود اما نوعی از اعتبار سنجی نیز وجود دارد که در سطح بیزنس لاجیک است که این شکل از اعتبار سنجی باید در لایه service انجام شود.

الگوی DTO

DTO ها یا همان Data Transfer Object ها Object هایی هستند که داده را بین پروسس های مختلف جابه‌جا می‌کنند.



در تصویر بالا محل هایی که از DTO ها و محل هایی که از Entity ها استفاده می‌کنیم قابل مشاهده هستند.

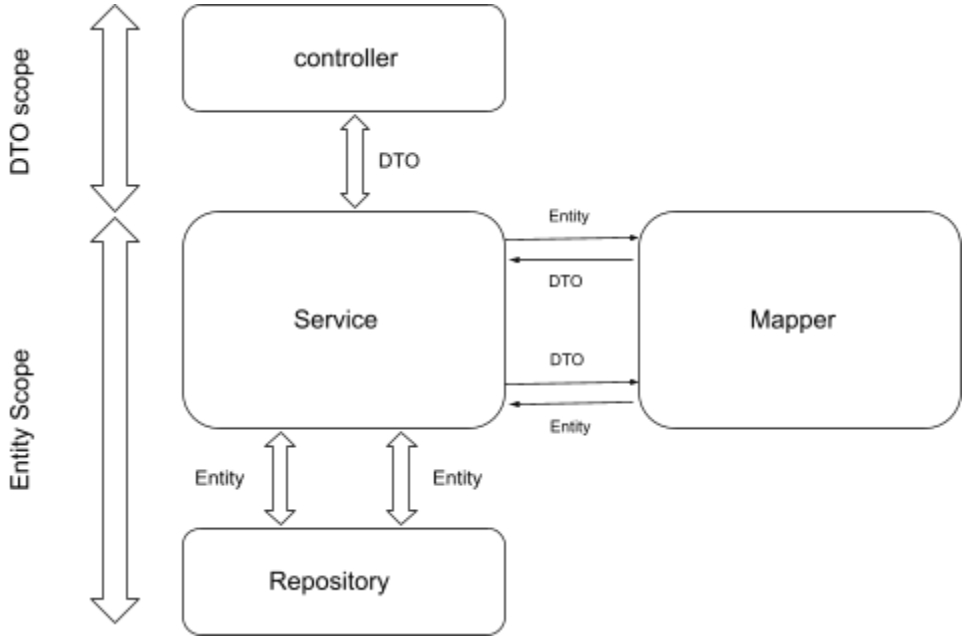
نکته : With DTOs we can build different views from our domain models

دلایل اصلی استفاده از DTO ها :

- فرض کنیم در presentation model ما ترکیب چند domain model را نیاز داریم. در این حالت اگر از DTO استفاده کنیم رفت و برگشت هایی که بین کلاینت و سرور وجود دارد کم می‌شود. یعنی کاهش فراخوانی هایی که بین کلاینت و سرور اتفاق می‌افتد.
- استفاده از DTO باعث می‌شود Domain model از دید کاربر مخفی بماند و این ریسک نمایش داده به کاربر را کم می‌کند. جداسازی Domain model از مدل هایی که در لایه presentation استفاده می‌شوند.

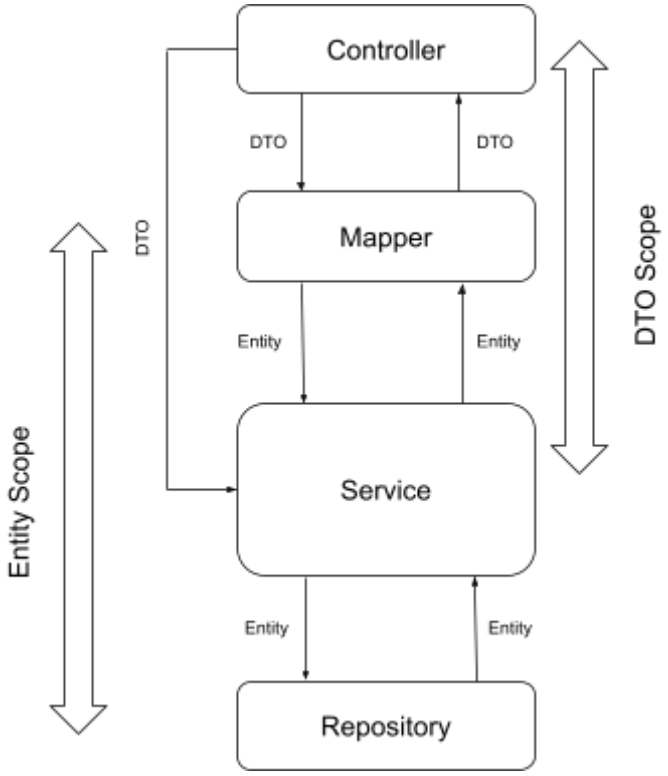
نکته : ما در قسمتی از کد خود نیاز داریم که یک تبدیل بین DTO و Entity داشته باشیم. این تبدیل کجا باید انجام شود ؟ راه حل های مختلفی برای این مورد مطرح شده است :

راه حل یک :



در این تصویر لایه Service از یک util class به نام mapper برای تبدیل بین entity ها و DTO ها استفاده می‌کند.

راه حل دو :



در این راه حل mapper به عنوان یک لایه بین Controller و service قرار می‌گیرد و وظیفه map کردن را اجرا می‌کند.
توجه شود که در این حالت mapper لایه بسته نیست. لایه بسته به لایه ای گفته می‌شود که در آن لایه های بالاتر اجازه دسترسی به

لایه های پایین تر را ندارند به عنوان مثال لایه service لایه بسته است چرا که لایه controller که در بالای آن قرار دارد نمی‌تواند به لایه های پایین تر از service مثل DAO دسترسی داشته باشد. همانطور که در تصویر مشخص است لایه mapper لایه بسته نیست چون لایه controller که در بالای آن قرار دارد می‌تواند به لایه پایین تر از mapper که لایه service است دسترسی داشته باشد.

چند نکته :

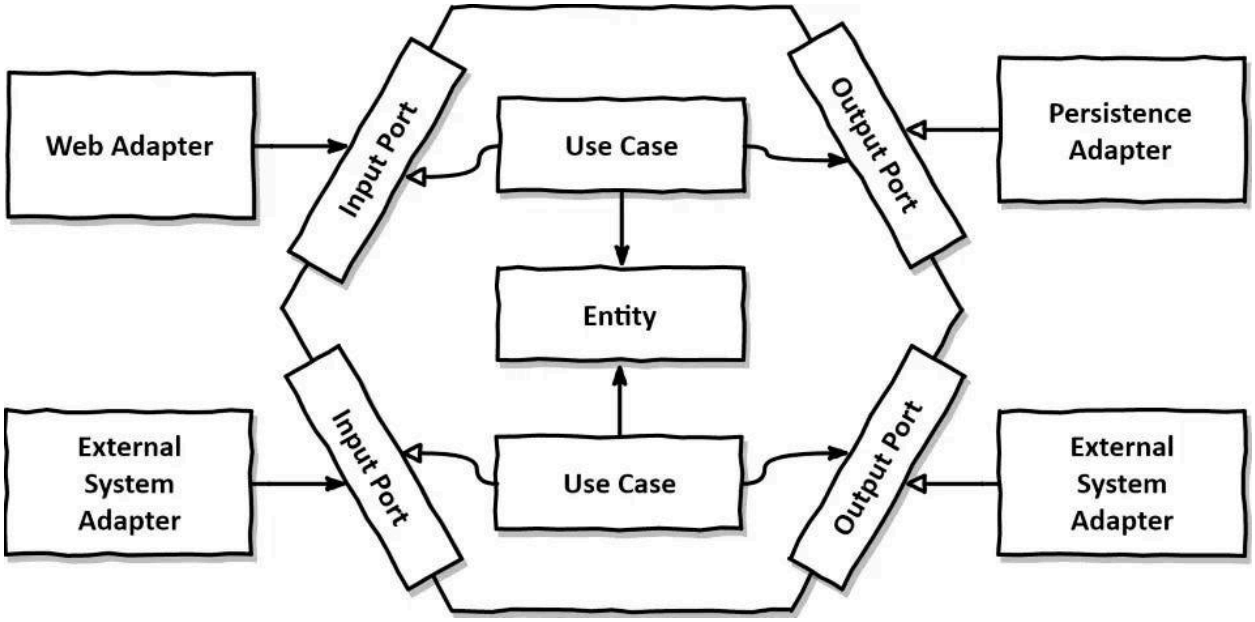
نکته : DTO ها به نوعی POJO هستند یعنی فقط داده را در خود ذخیره می‌کنند پس نباید Business logic را در آنها قرار دهیم.

نکته : یکی از اشتباهات رایج در ساخت DTO ها آن است که به ازای هر مورد یک DTO بسازیم. ساخت DTO های مختلف باعث می‌شود که تعداد آنها زیاد شود در نتیجه مدیریت آن ها سخت می‌شود. پس باید سعی کنیم قانون reuse بودن را رعایت کنیم.

معماری شش‌ضلعی در نرم‌افزار

لینک کتاب اصلی
یک مقاله مرتبط با این معماری همراه با مثال
یک مقاله مرتبط با این معماری با توضیحات خوب
سورس کد نمونه از پیاده سازی این معماری (بسیار مفید)

معماری شش ضلعی با این هدف ایجاد شد که بتواند بیزنس لاجیک را به کلی از فریمورک و دیگر وابستگی ها جدا کند یعنی بتواند کاری کند که domain به جایی وابستگی نداشته باشد.
نام دیگر این معماری پورت‌ها و آداپتورها ports & adaptors است.
در معماری شش ضلعی به جای آنکه domain ما به فریمورک ها و کتابخانه ها و ... خارجی وابسته باشد، وابستگی به شکل معکوس پیاده سازی می‌شود یعنی وابستگی به سمت domain است. پیاده سازی چنین چیزی با استفاده از مفهوم Dependency Inversion ممکن است.
در معماری شش ضلعی (بر خلاف معماری لایه‌ای که بالاتر توضیح دادیم) وابستگی ها به سمت داخل و domain است.



همانطور که در تصویر بالا مشاهده می‌کنید هسته برنامه در یک شش ضلعی قرار گرفته و ارتباط این هسته با محیط بیرون از طریق پورت ها صورت میگیرد. این پورت‌ها Dependency Inversion را برای ما پیاده سازی می‌کنند.
توجه شود که مفهوم شش ضلعی فقط یک نام گذاری است که نشان دهد هسته برنامه از دیگر قسمت ها مجزا است و این یعنی دنبال چیزی به نام شش ضلعی نباید بگردیم. 🙂
در این معماری برنامه از قسمت های مختلف تشکیل شده :

هسته : هر چیزی که در درون شش ضلعی قرار بگیرد جزئی از هسته است. در داخل هسته ما فقط کد های جاوا داریم و این کد ها هیچ وابستگی به کتابخانه های مربوط به دیتابیس یا کتابخانه های مربوط به سرویس REST و ... ندارند.

پورت ها : دو نوع پورت داریم : پورت خروجی و پورت ورودی.
همانطور که نام آنها نشان می‌دهد پورت های ورودی دروازه ورود به domain و هسته برنامه هستند و پورت‌های خروجی دروازه های ورودی از سمت هسته برنامه به سمت خارج برنامه هستند.
توجه شود که چون قصد داریم Dependency Inversion را پیاده سازی کنیم port های ما از جنس interface هستند.

آداپتور های متصل به input port ها : آداپتور هایی که در سمت چپ تصویر قرار دارند و به پورت های ورودی دسترسی دارند به ما این امکان را می‌دهند که بتوانیم از طریق پورت ها به domain دسترسی داشته باشیم.
نکته : در آداپتور های ورودی ما interface مربوط به پورت های ورودی را پیاده سازی نمی‌کنیم بلکه به instance های مربوط به این پورت ها دسترسی داریم و می‌توانیم توابع داخل آن را فراخوانی کنیم.
به عنوان مثال اگر ما بخواهیم یک درخواست rest را پیاده سازی کنیم ، کلاسی که قرار است درخواست REST ما را دریافت کند و انوتیشن RestController را دارد در واقع همان Web Adapter در تصویر بالا است.
کلاس هایی که در این قسمت ساخته می‌شوند در نام خود controller داشته باشند و از انوتیشن RestController استفاده کنند (اجباری نیست). همچنین می‌توانیم یک انوتیشن هایی با نام مناسب و مشابه با انوتیشن های Service, RestController و repository که در معماری سه لایه داشتیم بسازیم و در لایه های این معماری از آنها استفاده کنیم.

پورت های ورودی : این پورت ها در واقع interface هایی هستند که مشخص می‌کند چه توابعی یا بهتر است بگوییم چه امکاناتی در داخل هسته وجود دارد و آداپتور ها (آنهایی که در سمت چپ قرار دارند) می‌توانند آن توابع را فراخوانی کنند.
اینترفیس هایی که برای پورت های ورودی ساخته می‌شوند معمولا در انتهای نام خود UseCase دارند.

قسمت use case : این قسمت business logic سیستم ما را در خودش دارد. کلاس هایی که برای use case میسازیم در واقع input port ها که interface هستند را پیاده سازی می‌کنند در نتیجه وقتی آداپتور های ورودی توابع مربوط به input port را فراخوانی می‌کنند در واقع کد های داخل use case ها اجرا می‌شوند.
کلاس هایی که برای این قسمت ساخته می‌شوند در انتهای نام خود Service دارند و اینترفیس های مربوط به پورت های ورودی (همان اینترفیس هایی که در نامشان UseCase دارند) را پیاده سازی می‌کنند.
این کلاس های در صورت نیاز می‌توانند انوتیشن service یا انوتیشن component داشته باشند و یا اینکه خودمان انوتیشن جدید با نام UseCase بسازیم که این انوتیشن در واقع در خودش انوتیشن component را داشته باشد.

نکته : در کلاس های use case (که پیاده سازی پورت های ورودی را انجام می‌دهند و در نام خود Service دارند) از انوتیشن Trasnactional استفاده می‌کنیم.

پورت های خروجی : پورت های خروجی توسط کد های Use Case که در هسته قرار دارند استفاده می‌شوند و این امکان را به هسته می‌دهند که بتواند با خارج از هسته ارتباط داشته باشند.
برخلاف پورت های ورودی که در نام خود UseCase داشتند، اینترفیس های مربوط به پورت خروجی در انتهای نام خودشان Port دارند.

آداپتور های متصل به output port ها : این آداپتور ها در واقع output port ها (که اینترفیس هستند) را پیاده سازی می‌کنند و قسمت Use Case با فراخوانی توابع موجود در output port ها در واقع کد های داخل adapter ها که در سمت چپ قرار دارند را پیاده سازی می‌کنند.
توجه شود که مثلا اگر من بخواهم با دیتابیس در ارتباط باشم باید در قسمت adaptor (آنهایی که سمت راست هستند) مکانیزم ذخیره و بازیابی را قرار دهم همچنین entity class ها به دیتابیس map میشوند هم در این قسمت قرار می‌گیرند و در هسته قرار نمی‌گیرند چون ما اجازه نداریم از کتابخانه های مرتبط با دیتابیس در هسته اصلی برنامه استفاده کنیم. در واقع ما در سطح هسته کلاس های model خود را داریم و آن ها را به آداپتور های دیتابیس (سمت راست تصویر) پاس می‌دهیم. در آن آداپتور های کلاس های model ما به کلاس های entity نگاشت می‌شوند و در نهایت در دیتابیس ذخیره می‌شوند.
به عنوان مثال اگر بخواهیم ارتباط با یک دیتابیس را در این قسمت از کد پیاده کنیم می‌توانیم مجموعه کلاس های زیر را داشته باشیم :

- کلاس های Entity که مدل هستند و به دیتابیس map می‌شوند. معمولا در انتهای نام خود Entity دارند.

- کلاس های Repository که با دیتابیس در تعامل هستند و کار های ذخیره ، بازیابی و ... را انجام می‌دهند. معمولا در انتهای نام خود Repository دارند و می‌توانند از انوتیشن Repository استفاده کنند.

- کلاس های Mapper که وظیفه تبدیل کردن کلاس های model که در هسته نرم‌افزار ساخته می‌شوند به کلاس های Entity که قرار است در دیتابیس map شوند را دارند. معمولا در انتهای نام خود mapper دارند.

- کلاس های Adapter که در انتهای نام خود Adapter دارند. این کلاس ها اینترفیس مربوط به پورت های خروجی را پیاده سازی می‌کنند و همچنین به کلاس های mapper و کلاس های repository دسترسی دارند

نکته : یک کلاس Adapter می‌تواند به چند کلاس repository دسترسی داشته باشد.

نکته : در این معماری چون لایه ها از هم جدا هستند نیاز به تبدیل کردن بین کلاس ها زیاد دیده می‌شود یعنی در کد های آداپتور مجبور به map کردن کلاس های مدل به یکدیگر هستیم و این سربار وجود دارد.

نکته : از معماری DTO نیز می‌توانیم در معماری شش ضلعی استفاده کنیم. یعنی در آداپتور های سمت چپ کلاس های DTO را ساخته و آن ها را به کلاینت ها تحویل می‌دهیم.

نکته : ساختار کد و پکیج بندی را می‌توانید از داخل این پروژه نمونه ببینید.

Dependency Inversion

این مفهوم را با یک مثال توضیح می‌دهیم. فرض کنید یک کلاس داریم که برای ما داده را می‌خواند. اگر فایل ما به فرمت ایکس.ام.ال. باشد این کلاس به یک شکل خاص داده را می‌خواند. اگر داده به شکل جی.سون. باشد دوباره کلاس به شکل دیگری داده را می‌خواند. پس بسته به نوع داده ای که می‌خواند رفتارش متفاوت است.
حال فرض کنید به جای این که ما هر بار با توجه به نوع فایل رفتار را عوض کنیم (یعنی کلاس ما وابسته به نوع فایل باشد) این رفتار را برعکس کنیم. یعنی یک اینترفیس طراحی کنیم و بگوییم که سیستم ما به این شکل و با این توابع داده ها را می‌خواند. حال هر کسی که می‌خواهد داده اش توسط کلاس ما خوانده شود این اینترفیس را پیاده سازی می‌کند و اینترفیس پیاده سازی شده را به کلاس ما پاس می‌دهد.
با این کار کلاس ما دیگر به نوع داده وابسته نیست.

Dependency Injection

فرض کنید که یک کلاس (به نام کلاس یک) داریم و این کلاس برای اینکه بتواند کار هایش را انجام به یکسری کلاس دیگر نیاز دارد. روش های مختلفی وجود دارد که این کلاس ها را بدست کلاس یک برسانیم. به عنوان مثال یک راه این است که در داخل کلاس یک ، بقیه کلاس ها را new کنیم.
این کار خیلی خوب نیست چون داینامیک نیست چرا که هارد کد کرده ایم و گفته ایم که کلاس یک فقط با این کلاس ها کار می‌کند.
همچنین تست نویسی برای این شکل از کلاس ها سخت است.
یک راه دیگر این است که از طریق setter و یا constructor این کلاس ها را بدست کلاس یک برسانیم . این روش روش خوبی است به این خاطر که داینامیک است چرا که از بیرون این کلاس ها را به کلاس یک می‌دهیم در نتیجه می‌توانیم کلاس با پیاده سازی که مد نظر داریم را پاس بدهیم.

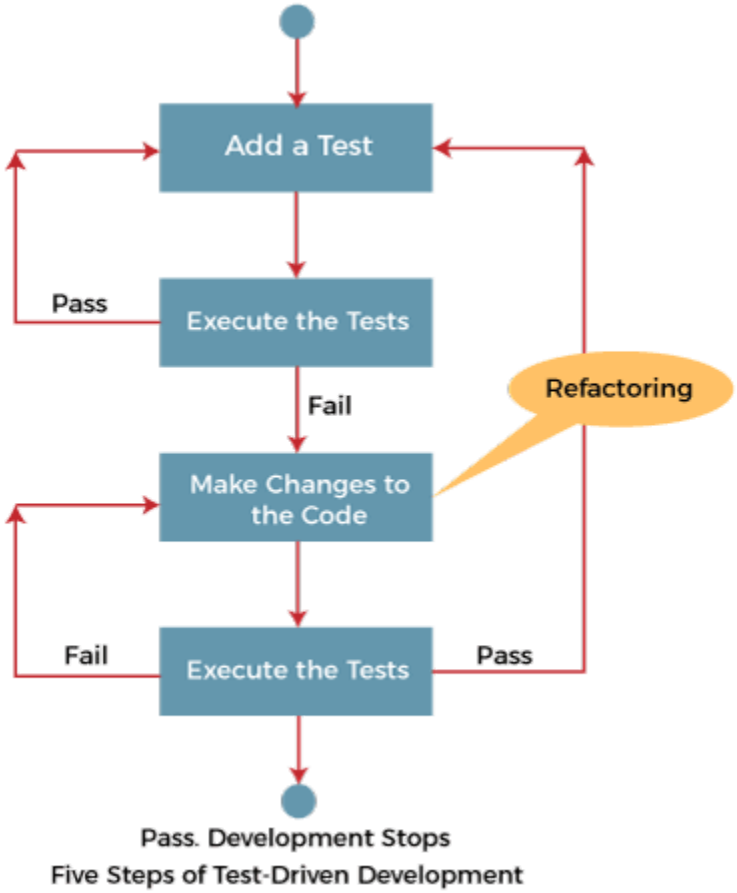
نکته : DI در واقع پیاده سازی از IoC است.

Inversion Of Control

مفهوم IOC container رو داریم که میگه ای برنامه نویس اگر با تزریق کردن مستقیم داخل کد مشکل داری این کار رو نکن یک فایل کانفیگ داریم و تو داخل این فایل بگو که این وابستگی بین کامپوننت ها به چه شکل هست. یعنی di توسط یک IOC container انجام می‌شود.
به عنوان مثال Spring یک ioc container است که وابستگی بین کامپوننت ها رو برایش مشخص می‌کنیم و اون عمل inject رو برای ما انجام می‌دهد.
به عبارت دیگر وظیفه ی تزریق کردن وابستگی ها از گردن برنامه اصلی برداشته می‌شود و این وظیفه به گردن framework و یا IoC Container می‌افتد.

TDD

در روش Test Driven Development ایده آن است که ابتدا تست بنویسیم و سپس کد نوشته شود. بعضی معتقد اند که "تستی که برایش کد نوشته شده، با کدی که برایش تست نوشته شده متفاوت است".
در شکل زیر چرخه‌ی TDD قابل مشاهده است :



Five Steps of Test-Driven Development

نکته ای که باید در نظر داشته باشیم آن است که در این روش یک دفعه کل تست را نمی‌نویسیم بلکه قدم به قدم تست را می‌نویسیم و بعد از چندین بار جابه‌جایی بین تست و کد اصلی، کد ما تکمیل می‌شود.

استفاده از TDD موقع کار علاوه بر اینکه کد باکیفیت و تست شده نوشته میشود چند خوبی دیگر دارد :

1. گاهی پیش میاد که روی یک ایشو کار می‌کنیم و فقط توسعه میدیم و در آخر تست های مربوط به این ایشو رو می‌نویسیم. این کار باعث میشه در آخر فراموش کنیم برای بعضی قسمت ها تست بنویسیم در نتیجه اگر از TDD استفاده کنیم در همون لحظه برای کد هامون هم تست می‌نویسیم.
   یا اگر مثلا ایشو شامل تغییرات زیادی بود خیالمون راحته که برای همه تغییرات تست داریم.

2. گاهی موقع توسعه یک ایشو کاری پیش میاد و ممکنه ایشوی ما نصفه و نیمه رها بشه. وقتی بعد از چند هفته میایم سراغ ایشو اگر از TDD استفاده کرده باشیم مطمئن هستیم که قسمت هایی که قبلا نوشتیم تست دارن و با خیال راحت تغییرات رو میدیم.

SOLID

اصول solid یا solid principles کمک می‌کنند که شما کد های بهتری بنویسید. نام SOLID از پنج اصل مطرح شده برای طراحی Object Oriented گرفته شده است.

- Single Responsibility
- Open Close
- Liskov Segregation
- Interface Segregation
- Dependency Inversion

Single Responsibility

هر کلاس فقط باید یک کار را انجام دهد و نه اینکه تمام کار ها را به دوش یک یا چند کلاس بندازیم.

Open Close

کلاس باید برای extention باز باشد و برای modification بسته باشد.
فرض کنید کلاس های circle و cube را داریم . یک کلاس دیگر هم داریم که برای ما مساحت را حساب می‌کند به نام CalcArea. اگر ما بخواهیم به ازای هر شی جدیدی که میسازیم بریم داخل کلاس CalcArea و بنویسیم if شی مورد نظر ما از جنس فلان بود به شکل مناسب مساحت را حساب کن کار خوبی نکرده ایم چرا که modification کرده ایم.
راه حل این مسئله این است که یک interface بسازیم و بگویم که تابعی به نام getArea داریم. حال هر کلاس جدید ما کافی است این interface را پیاده سازی کند. در نتیجه به ازای ساخت هر کلاس جدید به شکل خودکار توابع مورد نظر را پیاده سازی می‌کنیم.
با این کار ما modification مربوط به کلاس CalcArea را از بین برده ایم (یعنی هر بار لازم نیست برویم و به ازای یک shape جدید آن را تغییر دهیم) و همچنین extension را باز گذاشته ایم چرا که در کلاس CalcArea ما شی هایی که از اینتر فیسی که ساخته ایم را میگیریم و تابع getArea را فراخوانی می‌کنیم یعنی بدون تغییر کلاس CalcArea میتوانیم extension داشته باشیم و شی های جدیدی ایجاد کنیم که با این کلاس به خوبی کار می‌کنند.
به عبارت دیگه نباید کلاسی رو rewrite کنیم به این خاطر که یک شی جدید ساخته شده است.

Liskov Substitution

در این اصل هر subclass باید قابل جایگزینی باشد توسط پدرش به عبارت دیگر :
Every subclass or derived class should be substitutable for their base or parent class.

به شکل ساده یعنی ما باید به شکل درستی از وراثت استفاده کنیم به عنوان مثال فرض کنید که اینترفیس به نام Shape داریم که برای ما یک سری کار انجام می‌دهد. حال اگر بیایم و یک کلاس به نام Car بسازیم و سوپر کلاس آن را shape بگذاریم کار اشتباهی کرده ایم چرا که در کلاس Car ما نمی توانیم بعضی از توابع داخل Shape را پیاده سازی کنیم. پس اگر در جایی از کد بنویسیم :

```
Shape sh = new Car();
```

کامپایلر به ما خطا نمی‌دهد (چون کلاس پدر می‌تواند به فرزندش اشاره کند) ولی اگر در حین کد کسی یکی از توابعی که مربوط به shape است و car آن را پیاده سازی نکرده است را فراخوانی کند به مشکل می‌خورد.

Interface Segregation

اینترفیس ها نباید کلاس ها را مجبور کنند به پیاده سازی کاری که نمی‌توانند. اینترفیس های بزرگ باید به اینترفیس های کوچک شکسته شوند.
فرض کنید که یک اینترفیس داریم به نام Shape که توابع getVolume و getArea دارد. حال اگر من یک کلاس circle داشته باشم که Shape را پیاده سازی می‌کند به مشکل می‌خورم چون نمی‌تواند getVolume را پیاده سازی کند. در اینجا باید اینترفیس خود را کوچک کنیم تا این مشکل برطرف شود. به عنوان مثال دو اینترفیس به نام های Shape و ThreeDimentionalShape داشته باشیم و کلاس Circle فقط کلاس shape ارث بری کند و کلاس های مربوط به شی های سه بعدی از هر دو اینترفیس ارث بری کنند.

Dependency Inversion

بر اصل دو نکته تاکید دارد :

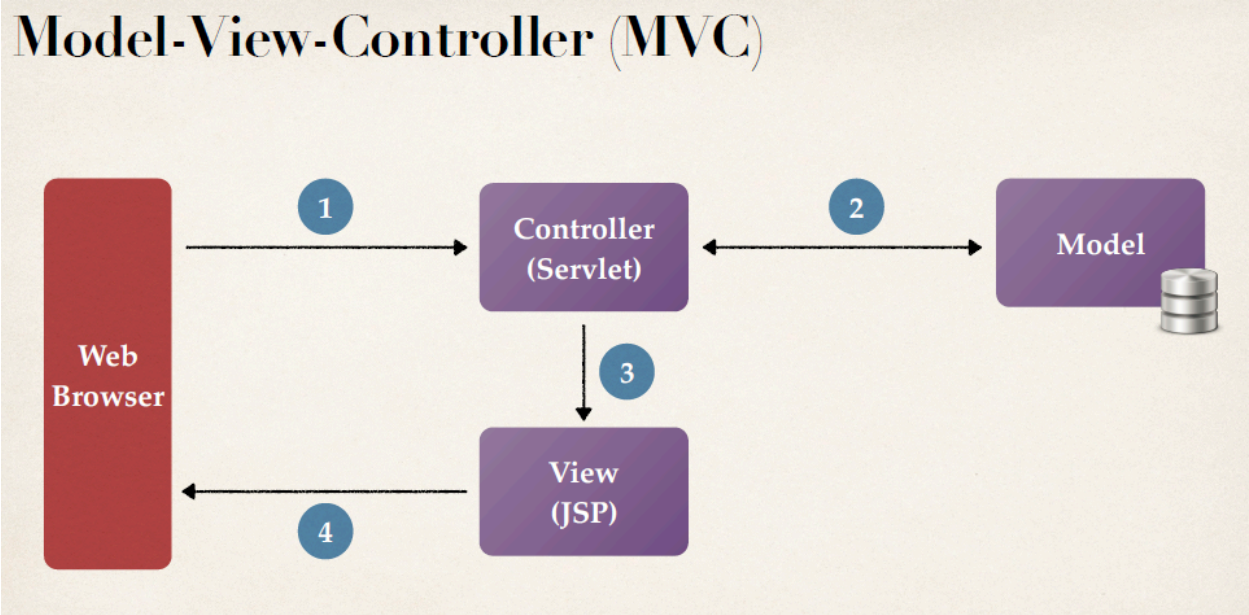
1. یک ماژول سطح بالا نباید به یک ماژول سطح پایین وابسته باشد
   high-level modules should not depend on low-level modules

2. هر دو ماژول باید به شکل انتزاعی وابسته باشند.
   both modules should depend on abstractions

با invert کردن ارتباط وابستگی امکاناتی مثل testability , flexibility و maintainability افزایش میابد.
این مفهوم را با یک مثال توضیح می‌دهیم. فرض کنید یک کلاس داریم که برای ما داده را می‌خواند. اگر فایل ما به فرمت ایکس.ام.ال. باشد این کلاس به یک شکل خاص داده را می‌خواند. اگر داده به شکل جی.سون. باشد دوباره کلاس به شکل دیگری داده را می‌خواند. پس بسته به نوع داده ای که می‌خواند رفتارش متفاوت است.
حال فرض کنید به جای این که ما هر بار با توجه به نوع فایل رفتار را عوض کنیم (یعنی کلاس ما وابسته به نوع فایل باشد) این رفتار را برعکس کنیم. یعنی یک اینترفیس طراحی کنیم و بگوییم که سیستم ما به این شکل و با این توابع داده ها را می‌خواند. حال هر کسی که می‌خواهد داده اش توسط کلاس ما خوانده شود این اینترفیس را پیاده سازی می‌کند و اینترفیس پیاده سازی شده را به کلاس ما پاس می‌دهد.
با این کار کلاس ما دیگر به نوع داده وابسته نیست.

الگوی MVC

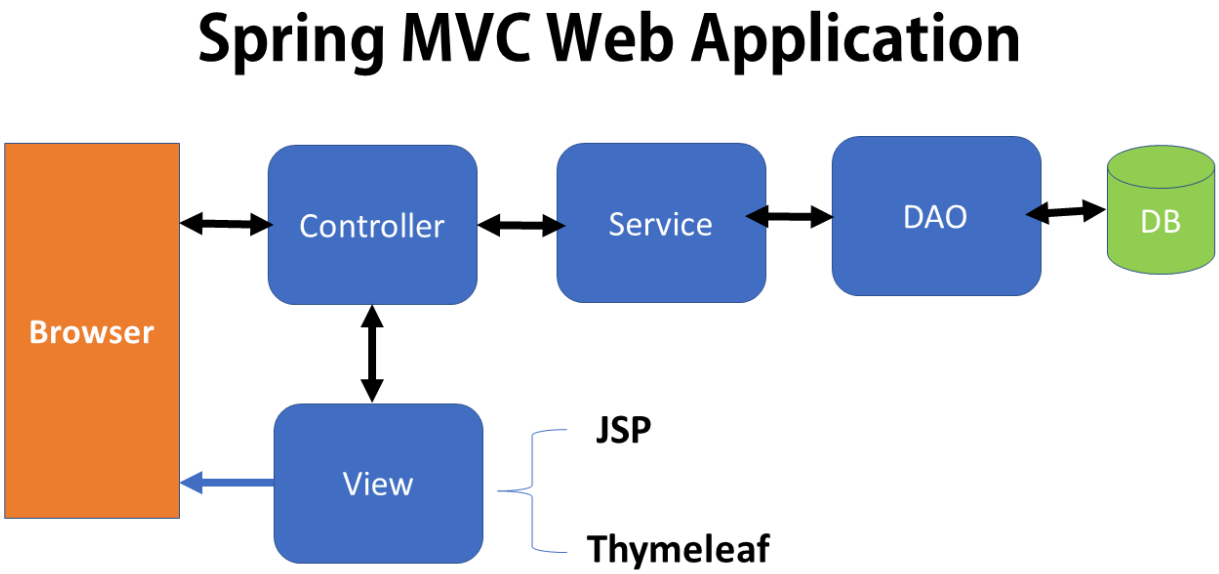
الگوی MVC در زمانی که زبان smalltalk وجود داشته است شکل گرفته است. در این الگو ما سه قسمت اصلی model, view و controller را داریم.



در الگوی mvc باید بدانیم که controller و view قسمت های UI ما را میسازند و هر چیزی که به UI مرتبط نباشد را در Model قرار می‌دهیم. پس یعنی قسمت هایی مثل entity ها repository ها و ... همگی در model قرار میگیرند.

در الگوی MVC درخواست کلاینت به Controller میرسد. اولین کاری که controller می‌کند validation است یعنی پارامتر هایی که از سمت کلاینت ارسال شده اند را بررسی می‌کند.
بعد از آنکه controller کار های validation را انجام داد با model ارتباط برقرار می‌کند تا کارهای مربوط به business logic و کار های غیر ui را انجام دهد و model بعد از انجام کارهایش جواب را به controller برمی‌گرداند. توجه شود که model هیچ کاری با view ندارد.
بعد از آنکه model جواب را به controller داد ، controller باید تصمیم بگیرد که با توجه به جواب model چه صفحه ای باید نشان داده شود مثلا اگر کار مدل موفقیت آمیز بود صفحه یک نشان داده شود و در غیر این صورت صفحه دو نشان داده شود.
همچنین اگر نیاز باشد داده ای در view نشان داده شود controller آن داده را به view ارسال می‌کند و view آن را نشان می‌دهد.

در یک برنامه spring boot الگوی mvc به شکل زیر خواهد بود :

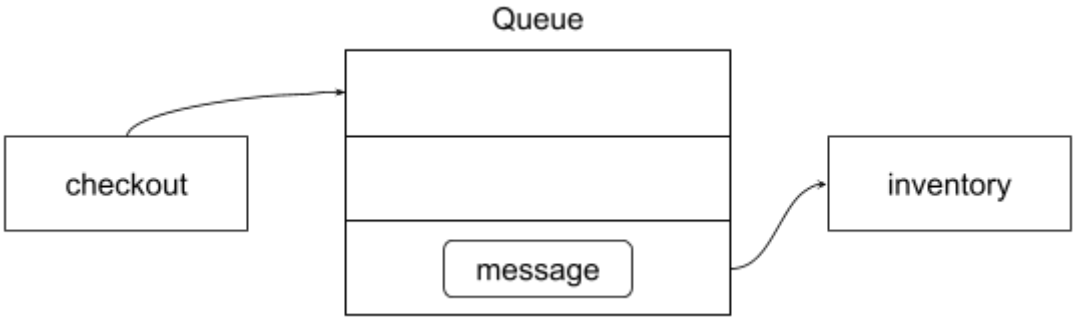


مفهوم Message Broker

[این ویدیو از IBM مفید است.]

مفهوم Message Queue

فرض کنید دو سرویس مجزا به شکل زیر داریم که قصد دارند با هم دیگر ارتباط برقرار کنند. ارتباط به این شکل است که سرویس checkout یک درخواست به شکل tcp را برای inventory میفرستد و منتظر جواب می‌ماند و سپس کارش را ادامه می‌دهد.
این روش چند مشکل دارد اگر تعداد درخواست ها زیاد شود سیستم قفل می‌کند یا مثلا اگر سرویس inventory پایین بیاید دیگر نمی‌توان به کار خود ادامه داد.
برای حل این مشکل یک Queue بین دو سرویس قرار می‌دهند. سرویس checkout به جای آنکه درخواست را مستقیما به inventory بفرستد، درخواستش را داخل queue گذاشته و خود inventory از داخل Queue از آن پیام استفاده می‌کند.
این کار باعث می‌شود در درخواست های زیاد سرویس قفل نشود.



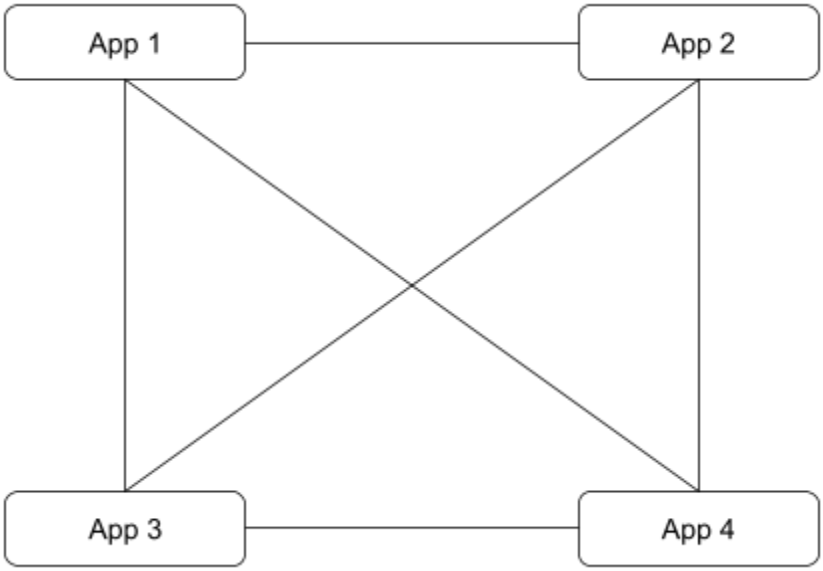
استفاده از message broker چند خوبی دارد :

امکان decouple : سرویس های checkout و inventory از هم مجزا هستن و checkout میگویید پیام را در صف قرار بده و دیگر کاری با بقیه ماجرا و اینکه چه کسی پشت قرار دارد ندارد. همچنین سرویس های checkout و inventory روی سروی ها مجزا می‌توانند اجرا شوند و این یعنی decouple بودن

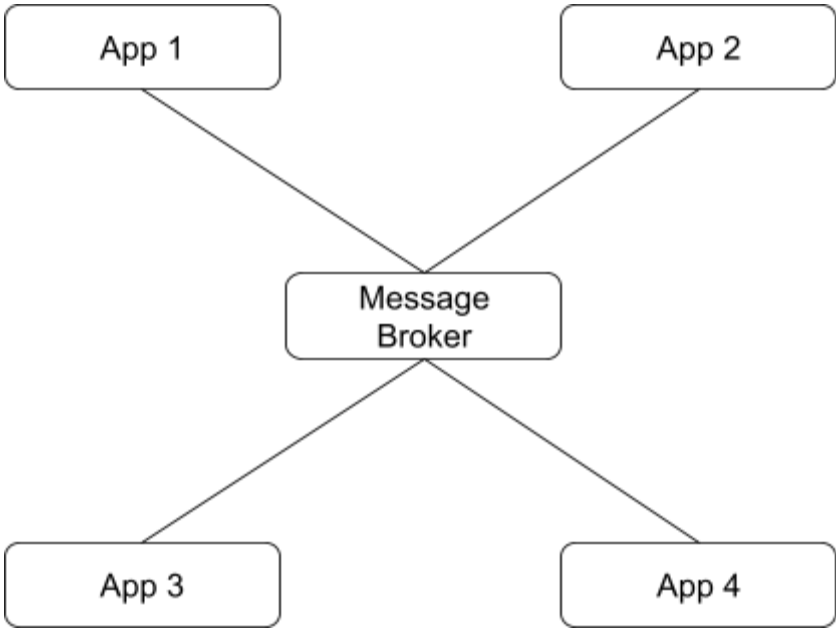
امکان scalable : اگر تعداد درخواست ها از سمت checkout زیاد باشد Queue دائما پر می‌شود و دیگر نمی‌تواند به درخواست ها جواب بدهد به همین خاطر می‌توانیم سرویس های inventory را افزایش دهیم و بیش از یک سرویس inventory داشته باشیم.

توضیح Message Broker

در زمان های قدیم که تازه تلفن‌ها ایجاد شده بودند اگر خانه A میخواست با خانه B صحبت کند باید حتما یک سیم مستقیم از خانه A به خانه B وجود میداشت. این عملکرد باعث میشد که بعد از گذشت زمانی تمام خانه ها با سیم به هم وصل شوند و شلوغ شوند و همچنین کارهایی مثل برطرف کردن مشکلات اتصال و غیره سخت بوده است. برای حل این مشکل مرکز های تلفن ایجاد شدند که خانه ها همگی به مرکز های تلفن متصل می‌شدند و مرکز تلفن اتصال بین خانه ها را برقرار می‌کرد.
ارتباط بین اپلیکیشن ها نیز به همین شکل است. فرض کنید بخواهیم ارتباط را بدون Message Broker ها ایجاد کنیم :



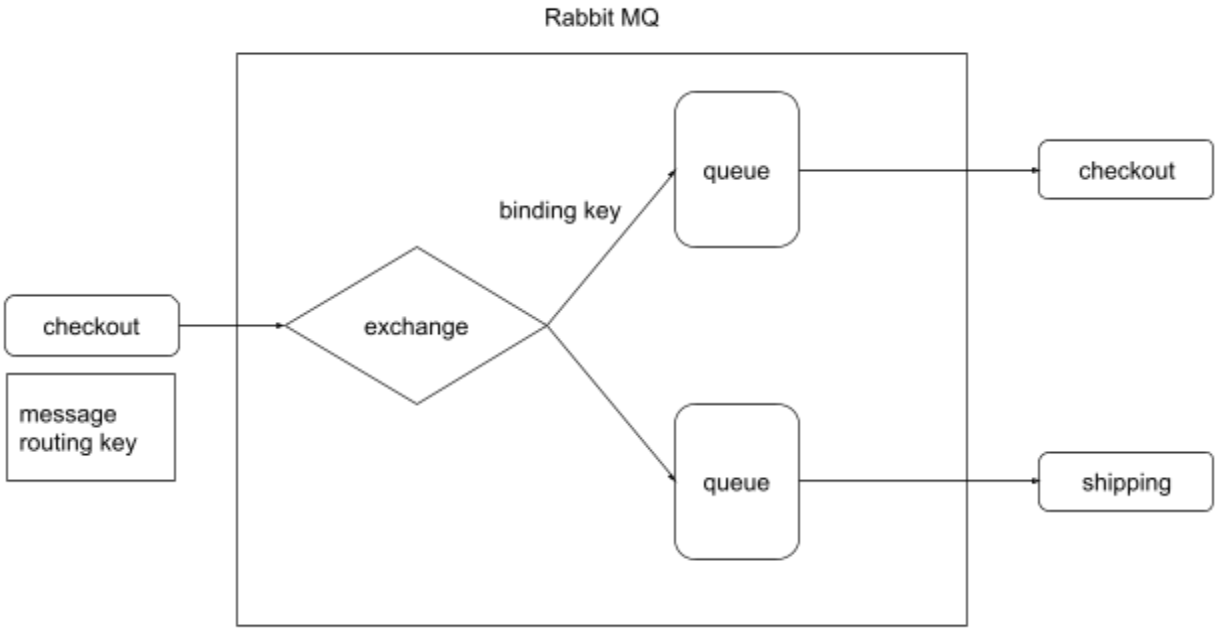
حال اگر از message broker استفاده کنیم به شکل زیر است :



نکته : message broker ها راه حلی برای ارتباط های async هستند یعنی در ارتباط های async تعریف و استفاده می‌شوند.

توضیح Rabbit MQ

یک از شناخته شده ترین message broker ها Rabbit MQ است. Rabbit MQ در واقع پیاده سازی شده ی AMQP که Advanced Message Queue Protocol است میباشد.
ساختار Rabbit MQ به شکل زیر است :



در تصویر بالا مشاهده می‌کنید که در داخل Rabbit MQ دو قسمت داریم، یکی exchange و دیگری queue.
قسمت Exchange مثل یک مرکز پست است که می‌تواند تشخیص دهد هر بسته به کجا باید ارسال شود و قسمت Queue هم که یک صف از پیام ها را در خود دارد.

نکته : queue ها از یک طرف به exchange متصل هستند و از طرف دیگر به consumer ها که در شکل بالا consumer ها checkout و shipping هستند. برای اینکه یک message از queue به consumer ها ارسال شود دو رویکرد وجود دارد :

- روش push api : در این روش consumer ها subscribe می‌کنند و در واقع پیام به سمت consumer ها ارسال می‌شود.

- روش pull api : در این روش consumer پیام را از queue درخواست می‌کند. این روش کارا نیست و بهتر است استفاده نشود.

نکته : می‌توانیم چندین exchange داشته باشیم و یک exchange می‌تواند به چند queue متصل شود.

نکته : از آنجایی که احتمال قطعی در یک شبکه وجود دارد Message Broker ها پیام ها را از queue ها حذف نمی‌کنند. در واقع یک مرحله acknowledgement باید وجود داشته باشه. یعنی بعد از اینکه consumer پیام رو از داخل queue خواند به broker پیام میده که پیام رو خونده و broker پیام رو از داخل queue حذف می‌کنه.
به این امکان message acknowledgement گفته می‌شود.

نکته : exchange ها می‌توانند durable یا transient باشند. در حالت durable با restart کردن message broker بعد از بالا آمدن سیستم exchange وجود دارد و به کار خود ادامه می‌دهد اما اگر به شکل transient باشد باید خودمان exchange را دوباره ایجاد کنیم.

سرویس checkout یک پیام دارد و آن پیام را به exchange می‌دهد و exchange تصمیم می‌گیرد که این پیام را به کجا ارسال کند. هر پیام یک routing key دارد که exchange بر اساس آن تصمیم می‌گیرد که پیام را به کجا ارسال کند.

قسمت exchange وظیفه‌ی ارسال پیام رو بر عهده دارد و نحوه‌ی عملکردش به چهار حالت زیر می‌تواند باشد :

- روش default : در این حالت exchange ها به queue ها از طریق binding key ها bind می‌شوند. در این حالت binding key ما نام همان queue است. حال اگر یک message بیاید که مقدار routing key آن همان نام queue باشد پیام با توجه به این نام به همان queue ارسال می‌شود.
این روش در rabbit mq عمل نمی‌کند.

- روش direct : در این روش هر queue با استفاده از یک binding key به exchange متصل می‌شود. زمانی که یک message به exchange میرسد مقدار routing key مربوط به message با مقدار exchange مربوط به queue بررسی می‌شود و اگر یکسان بود به آن queue ارسال می‌شود.

- روش fanout : در این روش exchange یک پیام را دریافت می‌کند و آن را به همه Queue ها ارسال می‌کند و اصلا routing key را چک نمی‌کند.

- روش topic : فرض کنید مقدار routing key در message ما به شکل ship.shoes باشد و یک queue گفته باشد که مقدار binding key اش (یعنی همان مقداری که exchange بر اساس آن queue را مشخص می‌کند) به شکل ship.* باشد . در این حالت این پیام برای این queue ارسال می‌شود.

- روش header : در قسمت header پیام یک مقدار تنظیم می‌شود که exchange بر اساس آن تصمیم می‌گیرد پیام را به کجا ارسال کند. زمانی که از این روش استفاده می‌کنیم دیگر routing key در نظر گرفته نمی‌شود.

مزیت های Rabbit MQ :

- امکان Cross Language : هر کدام از سرویس های ما می‌تواند به یک زبان برنامه نویسی باشد مثلا سرویس checkout با جاوا باشد و سرویس shipping با استفاده از پایتون باشد.

- امکان security : امکان احراز هویت برای سرویس ها می‌توان ایجاد کرد که ارتباط ایمن باشد.

- امکان acknowledge : یعنی یک پیام از یک queue خارج نمی‌شود مگر اینکه سرویس گیرنده پیام به Rabbit MQ خبر بدهد که من پیام را گرفته‌ام.

AMQP Methods

در AMQP یکسری متد وجود دارد که عملکردی شبیه به HTTP method ها دارند. مثلا , exchange.declare exchange.declare-ok و ... نمونه هایی از این متد ها هستند. در AMQP انواع متد ها وجود دارد و این متد ها در class هایی دسته بندی میشوند. توجه شود که مفهوم method و class در اینجا با مفاهیم آن در برنامه نویسی متفاوت است.

نکته : بعضی از متد ها می‌توانند پارامتر هایی را نیز همراه خود داشته باشند که دیتا های اضافی را حمل می‌کنند مثلا در تصویر زیر name و type و ... را داریم.

Virtual Hosts

ویژگی virtual host این امکان را به ما میدهد که بتوانیم group, user, exchange, queue و ... رو به شکل مجزا از هم دیگر داشته باشیم یعنی بتوانیم محیط های ایزوله داشته باشیم.

استفاده از RabbitMQ در Spring boot

ابتدا rabbitmq را روی سیستم خود نصب می‌کنیم.
برای استفاده از rabbitmq در spring boot ابتدا وابستگی زیر را به پروژه خود اضافه کنید :

```
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

حال در فایل application.properties اطلاعات مربوط به اتصال به rabbitmq را وارد می‌کنیم :

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

در کد زیر سه بین تعریف کرده ایم یکی exchange و دیگری queue است و سومی هم دو bean قبلی را به هم متصل می‌کند.

```
@Bean
Exchange exchange() {
   return ExchangeBuilder
       .directExchange("sina-exchange")
       .durable(true)
       .build();
}

@Bean
Queue queue() {
   return QueueBuilder
       .durable("sina-queue")
       .build();
}

@Bean
Binding bind(Queue queue, Exchange exchange) {
    return BindingBuilder
        .bind(queue)
        .to(exchange)
        .with("sina-routing-key")
        .noargs();
}
```

تا اینجا توانستیم queue و exchange مورد نظر خود را بسازیم. حال باید یک پیام ارسال کنیم.
برای ارسال پیام به شکل زیر عمل می‌کنیم :

```
@Component
public class MessageProducer {

	@Autowired
	private AmqpTemplate amqpTemplate;

	public void sendMessage() {
	Message message = new Message("my message".getBytes());
	amqpTemplate.send("sina-exchange", "sina-routing-key", message);
	}
}
```

همانطور که در کد بالا مشخص است یک Component ایجاد کرده ایم که در آن با استفاده از AmqpTemplate یک شی از جنس Message را ارسال کرده ایم.

حال باید یک receiver هم داشته باشیم که پیام ارسال شده را دریافت کند :

```
@Component
public class MessageConsumer {

	@RabbitListener(queues = "sina-queue")
	void listener(Message message) {
	System.out.println(new String(message.getBody()));
	}
}
```

در این کد هم یک component داریم که با استفاده از RabbitListener مشخص کرده ایم به کدام queue گوش می‌دهد و در این متد یک Message را دریافت کرده ایم که معادل با همان message ای است که ارسال شده است.

نکته : در هنگام اجرای برنامه ، در rabbitmq چیز هایی مثل queue و exchange و ... که ما ساخته ایم در آنجا ساخته نمی‌شود بلکه باید یکبار فرآیند ارسال پیام فراخوانی شود تا queue و exchange و ... ساخته شوند.