

**@Qualifier**

Eğer beanimize değer atamak istiyorsak spring bu tanımlamalardan hangisinin kullanılacağına @Qualifier anotasyonu ile karar verir. Kullanımı --> @Qualifier(“beanName”)

# **Convention over configuration kavramını seçtiğiniz bir örnek üzerinden açıklayınız.**

Conventiona uyduğunuz sürece olaylar hızlı bir şekilde gelişir. Geliştirici esnekliğini azaltır ama iş yükünden kurtarır. Maven dependency management tool'unda da kullanılan bir tekniktir. Doğru dosyaları doğru klasörlere koyarsak her sey kendiliğinden calisir. <http://devnot.com/2016/convention-over-configuration-coc-nedir/> sitesindeki robot örneği üzerinden esinlendiğim bir örnek: Normalde kapalı ortamlardan dışarı çıkarken kapıyı kendimize doğru çekeriz. İçeri girerkense kapıyı ileriye doğru iteriz. Fakat eve geldik diyelim anahtar bizde yoksa bu durumda evdeki insanlar kapıyı açar biz açmayız. Bu özel durumun tanımlanması olayı Conventiıon over Configuration kavramına örnek olarak gösterilebilir.

# SOLID prensiplerini kısaca açıklayınız. Sizce her koşulda bu prensipler çerçevesinde mi kod yazılmalıdır ? Neden ?

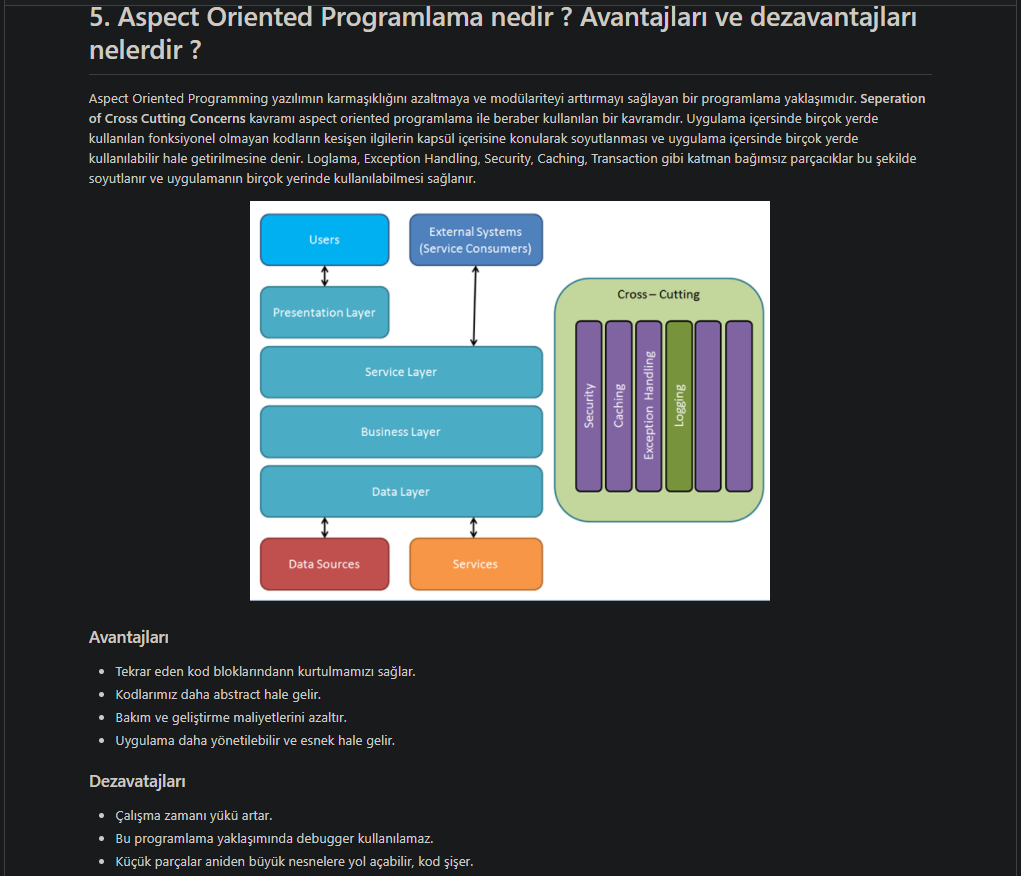
Solid prensipleri uzun süreye yayılacak OOP prensiplerinin uygulandığı büyük projelerde uygulanmalıdır ve gereklidir. Bu şekilde projedeki kodlar daha temiz ve esnek hale gelir, üstüne inşa edilmesi daha kolay olur. Proje büyüdükçe birşeyler eklemek istediğimiz zaman bazı yapıları tamamen değiştirmek zorunda kalmayız. Fakat küçük çaplı projeler, belirli amaca hizmet eden küçük script yapılar için solid presiplerinin birebir uygulanması gerektiğini düşünmüyorum. Proje zaten küçük, üstüne birşeyler inşa etmeyeceksem, bir kere kodlayıp sonsuza kadar kullanıcaksam niye bu prensipleri birebir uygulamak için zaman harcayayım diye düşünürüm. Tabiki de bazı prensipleri uygulamak küçük projeler için gerekli olabilir. Fakat bütün prensipleri de uygulamaya çalışmak da gereksiz olabilir.

### S — Single-responsibility principle

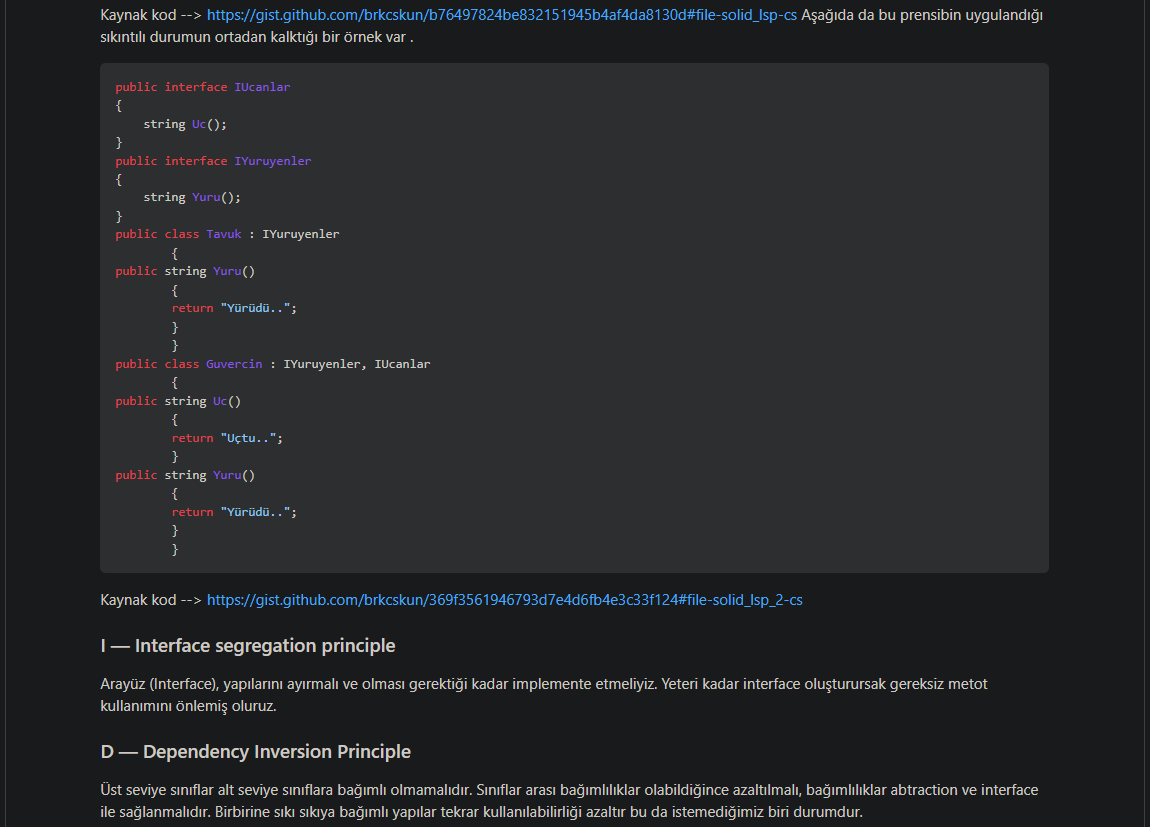
Bu prensibe göre bir sınıfa bir görev yüklenmeli birsürü işi yerine getirmeye çalışmamalıdır. Sınıfın veya metotun değişmek için bir sebebi olmalıdır. Birden fazla durum o sınıfı veya metotu değiştirmeye çalışırsa birden fazla sorumluluk yüklenmiş demektir ki bu da bu prensine aykırıdır.

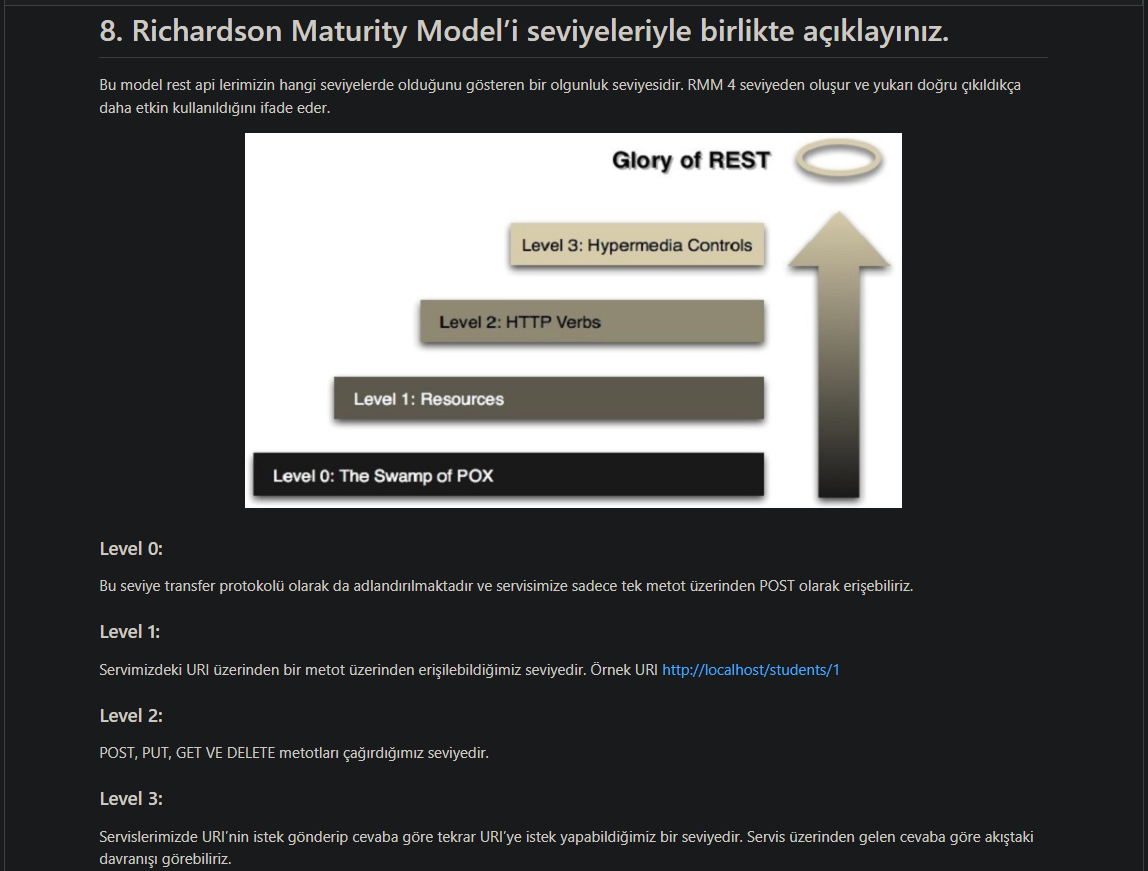
### O — Open-closed principle

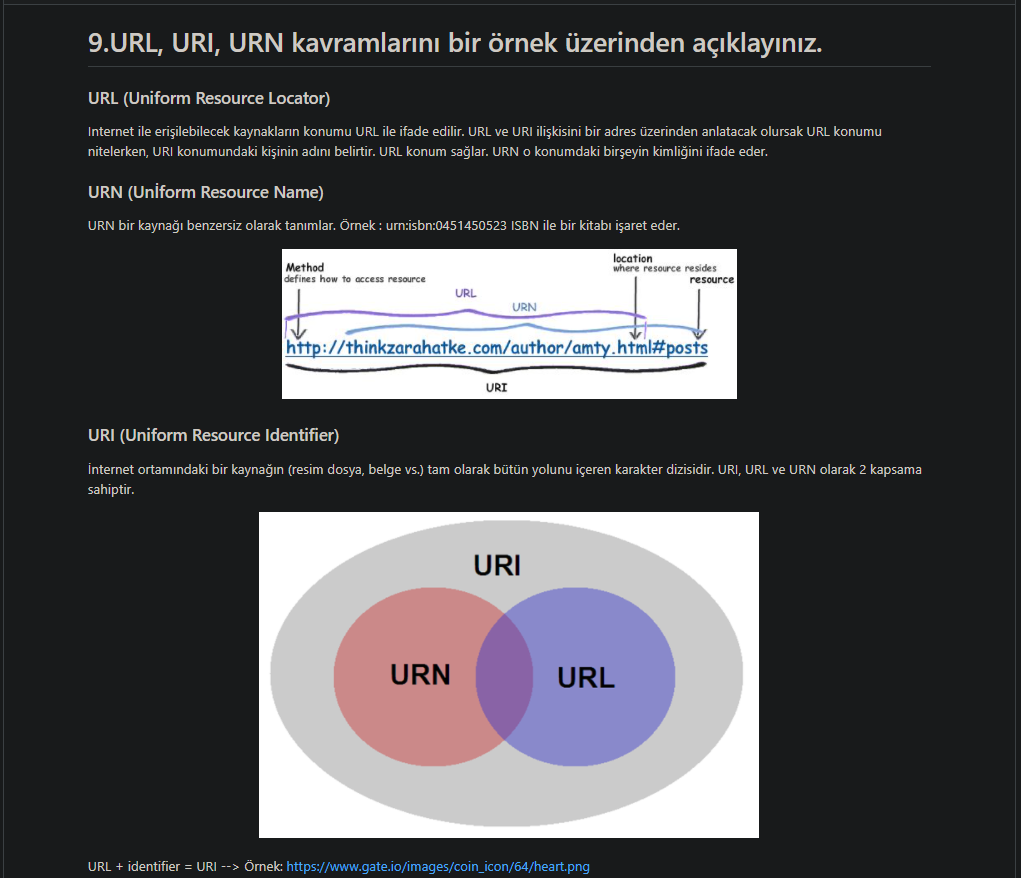
Bu prensibe göre classlar gelişime açık fakat değişime kapalı olmalıdır. Yani metot veya sınıflar var olan özelliklerini korumalı fakat yaptığı işin daha iyi yapılabilir olmasına izin vermelidir. O sınıfı veya metotu evirip çevirip daha iyi hale getirebilirsin ama yaptığın iş değişmesin aynı kalsın demektir.

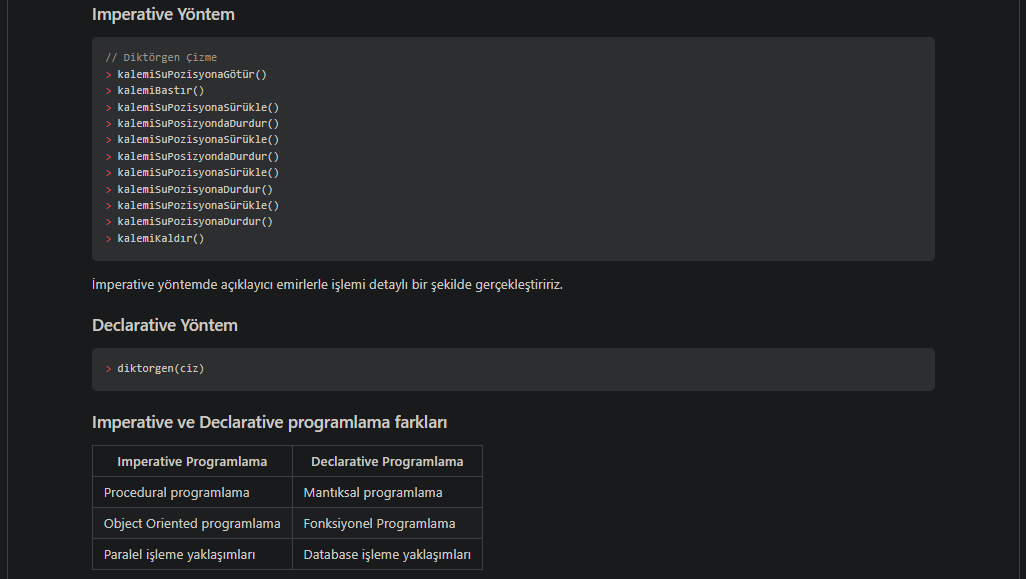












# Scalability nedir ? Horizontal ve Vertical Scalability kavramlarını açıklayınız.

Scalability (Ölçeklenebilirlik) sistemdeki artan kullanıcı sayısıyla beraber bu duruma sistemin nasıl tepki verdiği ile ilgili bir kavramdır.

**Vertical scalability (dikey ölçeklenebilirlik)** Var olan makinemizin donanım özelliklerinin arttırılarak (cpu, ram) sistemin requestleri karşılama yeteneğinin arttırılmasıdır.

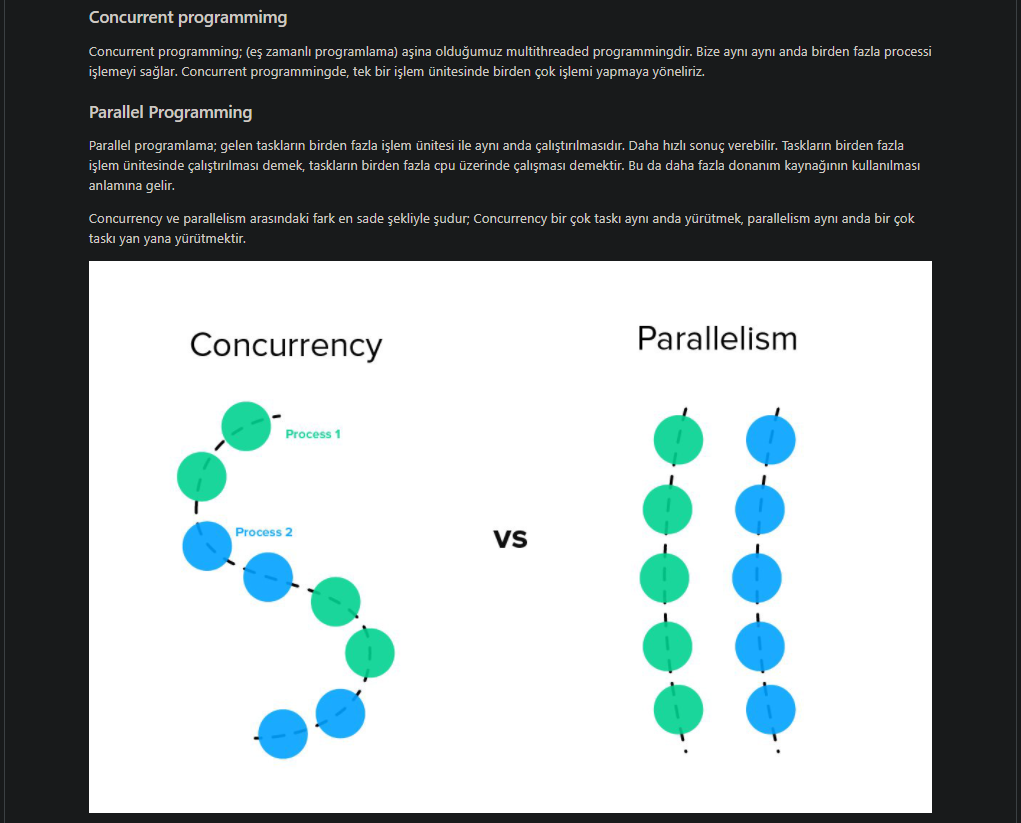
**Horizontal (yatay ölçeklenebilirlik)** Var olan kaynak havuzuna yeni makine ekleyerek sistem yükünü dağıtma ve requestleri karşılama yeteneğinin arttırılmasıdır

# Behavior Driven Development (BDD) nedir, neyi amaçlamaktadır ?

Test driven developmentın karmaşıklığını gidermek için ortaya çıkan bir test yaklaşım biçimidir. Yazılım süreçleri daha test odaklı gider. Test senaryoları Given, when, then olarak 3 başlıkta ilerler.

* Given --> belirlenen test senaryosu bu başlıkta oluşturulur.
* When --> testin gerçekleşeceği zamanı anlatır.
* Then --> test ile alakalı belirlenen senaryonun ve aksiyonun sonucu burada belli olur. Kısaca test sonucu burada belli olur.

Müşteri ihtiyaçlarına yönelik test senaryoları koda dökülerek test senaryoları hazırlanır. Bu yüzden ihtiyaçlar daha iyi karşılanır.



### Semaphore

Semaphoreu anlatmak için otel örneği kullanılabilir. Otelin resepsiyoncusu semaphore'u; odaları müşterilere vermek için kullanır. Aklında otel tamamen boşken otelde kaç odanın müsait olduğunu tutar, ki bu otelin oda sayısıdır. Müşteri geldikçe aklındaki bu oda sayısını düşürür, insanlar çıkış yaptığında bu sayıyı arttırır. Bu rakam sıfıra ulaştığında boşta odasının olmadığını ve isterlerse lobide beklemelerini söyler. Bu bekleyenler sırayla alınırlar.

Mutex'ler Critical Section yönetimi sağlamak üzere tasarlandığı için kendilerini bırakan Thread'in alan Thread olmasını gerekli tutmaktadır. Semaphore'da ise böyle bir kısıtlama yoktur hatta birçok durumda Semaphore'u bırakan Thread alan Thread'den farklıdır.Lock ve mutex'in aksine senkronize erişimi tamamen bekletmek için değil sınırlandırmak için kullanılabilir denilebilir. Sınırlı bir resource a sahipsiniz, aynı anda 3 thread'in bu kaynağa erişmesini istiyorsunuz daha fazlası sizin için bir sorun oluşturuyor.Kodun o kısmını bütün thread'lerin kullanımına açmıyorsunuz, orada sayaca sahip bir lock kullanıyorsunuz. İşte semaphore bu. Semaforların çalışması sırasında bölünmezlik (atomicity) ön plandadır. Yani bir semafor’un içerisinde yapılan birden fazla iş, program tarafından sanki tek bir iş gibi algılanmalı ve araya başka işin girmesine izin verilmemelidir. Semaphore, belirli bir kaynak için tüketici sayısını sınırlamanın bir yoludur.

# High Availability (HA) kavramını kısa açıklayınız.

Yüksek erişilebilirlik, 7×24 çalışan sistemlerin gerek otomatik gerekse manual sebeplerden dolayı yazılım veya donanımdan kaynaklanacak kesintilere uğramaksızın hizmet vermesidir. Internet gibi dağıtık ortamlarda hizmet veren sistemlerin kısa da sürse kesintiye uğraması önemsenecek maddi kayıplara neden olabilir. Büyük yazılım firmaları sistemlerin 7×24×365 hizmet vermesini garanti etmek için yani high availability sağlamak için çeşitli teknikler kullanırlar. Bazıları ;

* Failover Clustering (Yedekli Kümeleme )
* Log Shipping (Günlük Gönderme)
* Data Replication (Veri Yineleme)
* Database Mirroring (Veritabanı Aynalama)

Herhangi bir sistemin yüksek düzeyde erişilebilirlik sunması için, sistemin parçalarının kullanılmadan önce iyi tasarlanmış ve kapsamlı bir şekilde test edilmiş olması gerekir

# Core Domain, Supporting Domain, Generic Domain kavramlarını DDD kapsamında açıklayınız.

Core domain - iş için gerekli olan en önemli alt alan. Onsuz, iş başarısız olur.

Supporting domain - iş için core domain etki alanından daha az değerli olan alt domain. Onsuz iş, bir süre hayatta kalabilir . Ancak yine de oldukça önemlidir (core domaini destekler), ayrıca alana özeldir ve geliştirilmesi gerekir.

Generic Domain - ?

## Conway’in Kanunu’ nu (Conway’s Law) açıklayınız.

“Sistemleri tasarlayan organizasyonlar kendi iletişim yapılarının birer kopyasını üretmekle sınırlıdır”

ÖNEMLİ

## Continuous Integration, Continuous Delivery ve Continuous Deployment kavramlarını açıklayınız.

### Continuous Integration

Yaptığımız geliştirmelerin, refactorlerin, yazdığımız kodların, kontrollerden geçtiği sürece denir. Bu süreçte yapılan geliştirmelerin sorunlara yol açıp açmadığı kontrol edilir, sistemin çalışırlığı test edilir. bu geliştirmeler belli testlere tabi tutulur, kontrol edilir, sistemde bi yerleri kırarsa veya hataya zorlarsa, ilgili developerlara bu bilgilendirmeler yapılarak, sorunun bir an önce giderilmesi sağlanır, bu şekilde her zaman çalışabilen bir sürüm sağlanmış olur.

### Continuous Delivery

Yapılan geliştirmelerin CI sürecinden (bir önceki süreç) geçtikten sonra build alınıp deploya hazır hale gelmesinin sağlandığı aşamadır.

### Continuous Deployment

Burda da paketler ürünler sunucuya kurulur, canlıya alınır.

### Yaygın kullanılan CI/CD araçları

* Jenkins
* Travis CI
* CircleCI
* Jetbrains TeamCity
* Atlassian Bamboo
* GitLab
* Codeship
* Buddy
* Semaphore

Spring dışında dependency injection için kullanabileceğimiz framework’ler / kütüphaneler nelerdir ? (Herhangi bir programlama dili için olabilir.)

Dependency injection nedir ilk önce onu açıklayalım. Dependency injection yazılımı oluşturan yapıların birbirleri ile olan bağımlılıklarını azaltmayı amaçlayan ve o yazılım parçalarını olabildiğince bağımsızlaştıran bir programlama tekniğidir. Kelime anlamı olarak bağımlılıkların dışardan eklenmesi anlamına gelir. Dependency injection sayesinde yazılımı oluşturan parçaların birbirleri ile olan sıkı bağı azaldığı için uygulamaya yeni özellikler eklenip çıkarılabilmesi daha kolay hale gelir. Uygulamaya yeni özellikler ekleneceği zaman değiştirilmesi veya müdahale edilmesi gereken yerlerin sayısı minimuma iner. Daha test edilebilir yapılar ortaya çıkar. Dependency injection yapabilmek için belli teknikler kütüphaneler ve frameworkler vardır.

Dependency Injection teknikleri

Constructor injection : Bağımlılıklar, sınıfa constructor ile eklenir.

Setter injection : Bağımlılık, setter metodu ile parametre geçilerek sağlanır.

Interface inejction : Bağımlılığı, aktaracak olan bir metot sağlar.

Spring mvc nedir

Enhance ne demek

Pojo nedir

DEPENDENCY INJECTION – BAĞIMLILIK ZERKİ

Obscurity nedir

Dependency inejction bir inversion of control yapısıdır. Bu nedemek kontol daima frameworkde olmalı, objelerde değil,

inversion of control –

framework akış kontrolünü ele aldığı zaman inversion of control olur. Uygulamamız frameworkün componenti olur.

Bütün frameworkler bu yöntemle çalışır

Framework application kodunu çağırır.

Dependency injejction ioc ile başarılıdır.

Dependency inejction ioc ile elde ettiğimiz şeylerden biri. Birisi prensip diğeri mekanizma, mekanizma olan ioc sanırım.

Bean springinn ioc containerları tarafından yönetilen objelere bean denir. Spring ioc containerı beanın objelrini oluşturur. Onları biraraya getirip lifecycle ı yönetir.

@Required setter metotları için kullanılıyor ve depreceated oldu. Sadece setter metotları ike kullanışablir

@Component

Annote ettiği sınıfın bir component olduğunu söyler

Sınıdlardan önce kullanılır. O sınıfın instance larını bean yapar

@Autowired, @Qualifier, @Primary, @Value, @Profile, @Conditional,