## Git Flow: Sürüm Kontrolüne Kapsamlı Bir Yaklaşım

Bilinen bir iş akışı yöntemine Vincent Driessen tarafından tarif edildiği gibi [Git Flow](http://nvie.com/posts/a-successful-git-branching-model/) denir . Git Flow, özellik dalları, sürüm dalları, ana hat veya geliştirme dalları ve düzeltmelerin nasıl ilişkili olduğunu açıklar. Bu yaklaşım, kullanıcılar tarafından indirilen, kütüphaneler ve masaüstü uygulamaları gibi paketlenmiş yazılımlar için çok uygundur.Ancak birçok web sitesi için Git Flow aşırıya kaçmıştır. Bazen, ana hat geliştirme ve serbest bırakma dalları arasında, ayrımı değerli kılmak için yeterince büyük bir fark yoktur. Alternatif olarak, düzeltmeler ve özellik dalları için iş akışınız aynı olabilir.

## GitHub Flow: Basitleştirilmiş Bir Alternatif

GitHub, [GitHub Flow](https://guides.github.com/introduction/flow/" \t "_blank) adlı alternatif bir iş akışı önerir . GitHub Flow, özellik dalları gibi Git Flow ile aynı öğelerin bazılarına sahiptir. Ancak Git Flow'dan farklı olarak GitHub Flow, ana hattı ve sürüm dallarını bir “ana” olarak birleştirir ve düzeltmeleri tıpkı özellik dalları gibi ele alır.Bu basitleştirilmiş model, değişikliklerin hızlı bir şekilde yapılabileceği ve bazen günde birkaç kez kolayca dağıtılabildiği sürekli dağıtım modellerine daha uygundur.

## Git Flow Ne Zaman Ek Karmaşıklığa Değer?

Git Flow'un doğasında bulunan ek karmaşıklığın aşağıdaki gibi belirli durumlarda gerekli olduğunu bulduk:

* Ayrılmış olarak adlandırılmış veya numaralandırılmış sürümleriniz olduğunda
* Bir sonraki sürüm için özellikleri geliştirmeye ve entegre etmeye devam ederken, bir sürüm adayı üzerinde geliştirmeyi dondurmanız gerektiğinde. Bu, örneğin bir sürüm adayının gözden geçirilmesi, hata düzeltmeleri ile kabul edilmesi veya yayınlanmadan önce onaylanması gerektiğinde olabilir. Geliştirme, bir sonraki sürümde aynı anda devam eder
* Yazılımın birden fazla sürümünün bağımsız olarak desteklenmesi ve bakımının yapılması gerektiğinde Bu senaryolardan herhangi biri geçerliyse Git Flow'dan yararlanabilirsiniz. Yoksa, GitHub Flow tarafından daha iyi hizmet edilebilir.

## Pull Request vs. Merge Request

Bir çekme isteğinde, bir ana dalda yaptığınız değişikliklerin bir ana ana dalda birleştirilmesini öneriyorsunuz. Varsayılan olarak, kafa dalı taban dalıyla çakışmadığı sürece herhangi bir çekme isteği herhangi bir zamanda birleştirilebilir. Ancak, çekme isteğini belirli bir dalda ne zaman birleştirebileceğiniz konusunda kısıtlamalar olabilir. Örneğin, bir çekme isteğini masterancak gerekli durum denetimleri geçiyorsa birleştirebilirsiniz .

## Razor Pages

Razor Pages, sayfa tabanlı web uygulamalarını çok daha kolay ve üretken bir şekilde kodlamanızı sağlayan yeni bir yapı sunar. ASP.Net’te olduğu gibi ASP.Net Core çatısında da bulunur. Bu bölüm, Razor Pages tanıtımı ve Visual Studio Code ile kullanılmasını kapsar. Razor @ işareti ile C# kodlarına geçmemizi sağlıyor. HTML ile iç içe kullanım imkanı sunuyor. Benzer şekilde PHP ve ASP ise sadece kendi etiketleri sırasıyla <?php ?> ve <% %> etiketleri arasına kod yazmaya olanak sağlar. Razor biraz daha sade bir yapıya sahiptir. Razor kodları örnekte olduğu gibi tek satırlık kod ise sadece @ işaretinden sonra eğer birden fazla satıra sahipse @{…} arasına yazılır. Değişkenler var anahtar kelimesi ile deklare edilir. C# dosyaları .cshtml uzantılısına sahiptir.

## Console Application nedir?

Console ise grafiksel olmayan bir kullanıcı arayüzüne sahiptir[.](http://www.btdersleri.com/ders.aspx?konu=224) Kullanıcıdan bilgi alma ve gösterme işlemleri siyah bir ekranda gerçekleşir. Visual C# veya Visual VB dillerinde kullanabildiğimiz hazır nesneler Console Uygulamalarında yoktur. Visual Studio, çok gelişmiş özelliklere ve yardımcı araçlara sahip bir dosya editörüdür. NET platformu üzerinde geliştirilen proje dosyaları  
dışında metin dosyaları, sql, rtf uzantılı dosyalar da düzenlenebilir.

## c# json serialize / deserialize

**DataContractJsonSerializer'ı kullanma**

DataContractJsonSerializer sınıfı, JSON'un serileştirilmesine ve serileştirilmesine yardımcı olur. System.Runtime.Serialization.dll derlemesinde bulunan System.Runtime.Serialization.Json ad alanında bulunur. Sınıfı kullanarak bir nesneyi JSON verilerine serileştirebilir ve JSON verilerini bir nesneye serileştirebiliriz.

1. DataContract]
2. **sınıf** BlogSite
3. {
4. [DataMember]
5. **genel dize** Adı {  **get**; **ayarlayın**; }
7. [DataMember]
8. **public string** Açıklama {  **get**; **ayarlayın**; }
9. }

**Serialization**

Serileştirme'de, özel bir .Net nesnesini JSON dizesine dönüştürür. Aşağıdaki kodda, bir BlogSiteclass örneği oluşturur ve özelliklerine değerler atar. Daha sonra, BlogSite sınıfını değiştirerek DataContractJsonSerializer sınıfının bir örneğini oluştururuz ve nesne (BlogSite) yazmak için bir MemoryStream sınıfı örneği oluştururuz. Son olarak, MemorySteam nesnesinden JSON verilerini okumak için bir StreamReader sınıfı örneği oluşturur

1. BlogSite bsObj =  **yeni** BlogSite ()
2. {
3. Adı =  "C-sharpcorner" ,
4. Description =  "Bilgi Paylaş"
5. };
7. DataContractJsonSerializer js =  **new** DataContractJsonSerializer ( **typeof**(BlogSite));
8. MemoryStream msObj =  **yeni** MemoryStream ();
9. js.WriteObject (msObj, bsObj);
10. msObj.Pozisyon = 0;
11. StreamReader sr =  **yeni** StreamReader (msObj);
13. // "{\" Açıklama \ ": \" Bilgiyi Paylaş \ ", \" Ad \ ": \" C-sharpcorner \ "}"
14. **string** json = sr.ReadToEnd ();
16. ) (Sr.Close;
17. msObj.Close ();

**deserialization**

Diziselleştirme işleminde, Diziselleştirmenin tersini yapar, yani JSON dizesini özel bir .Net nesnesine dönüştürür. Aşağıdaki kodda, BlogSite sınıfının bir örneğini oluşturur ve özelliklerine değerler atar. Ardından, BlogSite sınıfını değiştirerek ve bir nesne (BlogSite) yazmak için MemoryStream sınıfının bir örneğini oluşturarak DataContractJsonSerializer sınıfının bir örneğini oluştururuz. Son olarak, MemorySteam nesnesinden JSON verilerini okumak için bir StreamReader sınıfı örneği oluşturur.

1. **string** json =  "{\" Açıklama \ ": \" Bilgi Paylaşım \ ", \" Ad \ ": \" C-sharpcorner \ "}" ;
3. **kullanarak** (var ms =  **yeni** MemoryStream (Encoding.Unicode.GetBytes (json)))
4. {
5. // JSON'dan seri kaldırma
6. DataContractJsonSerializer deserializer =  **yeni** DataContractJsonSerializer ( **typeof**(BlogSite));
7. BlogSite bsObj2 = (BlogSite) deserializer.ReadObject (ms);
8. Response.Write ( "Ad:"  + bsObj2.Name); // İsim:  C-sharpcorner
9. Response.Write ( "Açıklama:"  + bsObj2.Description); // Açıklama: Bilgi Paylaşımı
10. }

## MVC vs MVVM

## MVC (Model-View-Controller):

Controller kullanıcıdan gelen inputları karşılar, ayrıca UI ile ilgili bütün akışı yönetir ve kararları verir. Controller View hakkında hiç birşey bilmez ama View Controller’ı bilir. Görüldüğü üzere Controller ile View arasında 1-n bir ilişki var yani bir Controller birden fazla View tarafından kullanılabilir. Controller kullanıcıdan gelen inputlar doğrultusunda Model üzerinde değişikleri yapar, Model değiştiğini View’e notify eder yani View ile Model arasında Observer ilişkisi var. View, Model’e register olur, görüldüğü üzere bir model’e birden fazla View register olabilir. Aralarında ki observer ilişkisi sayesinde, Model’deki herangi bir değişiklik ona register olmuş bütün View’lere yansır.

## MVVM (Model-View-ViewModel):

Aslında PM View’in state ve davranışlarıyla ilgili bilgiyi kendi üzerine alıyor ve Business Layer ile arasındaki kordinasyonu sağlıyor ve View’e karar vermeyle ilgili çok az şey bırakıyor. View yine stateleri tutuyor aslında. Fakat MVP’nin aksine Presentation Model View ile ilgili hiç bir bilgiye ihtiyaç duymuyor, bu yüzden ki View ile PM arasında 1-n bir ilişki var, bir PM birden fazla View’de kullanılabiliyor, bu kısmıyla MVC’ye benziyor, fakat MVC’nin aksine View üzerinde ki manipulasyonlar PM üzerinden gerçekleşiyor. Aslında şöylede bakabiliriz, PM ile View arasında yine bir observer ilişkisi var, .Net’e kullanım şekillerinden biride **INotifyPropertyChanged** interface’inden türeyip, .Net’in binding alt yapısını kullanması. Zaten özünde yaptığı iş DataBinding. Kendi propertylerini View’in propertyleriyle senkronize ediyor, aynı zamanda state’lerede karar veriyor, mesela şu şu TextBox dolduğunda şu Buton enabled olsun gibi. Tabi herzaman enabled olucak bir kontrol’un state’ini PM’de tutmak anlamsız.  
 PM’in en büyük avantajı, hiç bir View’e ihtiyaç duymadan bir View’in davranışlarını ve Data’sını barındarabilmesi, bu yapısıyla TDD’ye (Test Driven Development) çok uygun. Şu haliyle Prensentation Patternleri arasında en ideal pattern’lerden biri

## Model-View-Update (MVU) :

* A Program, görünümü de içerebilmesi dışında MVVM'de bir tür Görüntüleme Modelidir. Bu yüzden en iyi yerine getirilebilen mümkün olan en küçük özerk birim olarak görülebilir. Yani bir program!
* MVVM gibi mimarilerden farklı olarak, a Modeloldukça keyfi bir işlevsellik setini (hizmetler, veriler, ...) değil, programınızın *durumunu* tanımlar . Böylece pek çok insan terimden rahatsız olmaz ve onun yerine devlet demektedir. Karışıklığı önlemek için buraya modelle bağlı kalıyorum.
* A Commandolmasını beklediğiniz şeydir. Bir eylemi gerçekleştirebilen (eşzamansız olarak) bir şey, ör., Bir API'den veri almak, verileri bir veritabanına geri yazmak veya bir şeyleri hesaplamak. Kendisi yeni bir mesaj göndermeyi tetikleyebilir.
* A Message, tüm parçaları bir arada tutan tutkaldır. Özellikle aşağıda açıklanan güncelleme işlevinin ne yapması gerektiğini tanımlar.
* Init()İşlevi, bir Programın başlangıç noktasıdır. Modelinizin ilk sürümünü oluşturur ve örneğin veri almak için ilk komutu başlatabilir.
* Update()Fonksiyon doğrudan modelini işleyen tek yerdir. Ancak diğer yaklaşımlardan farklı olarak, onu asla değiştirmez, bunun yerine güncellenmiş bir kopya oluşturur. Modelinizin bu yeni sürümüyle birlikte, başka işlemler başlatmak için bir komut da döndürebilir.
* View()İşlev UI tanımlanması için en MVU uygulamalarında sorumludur.