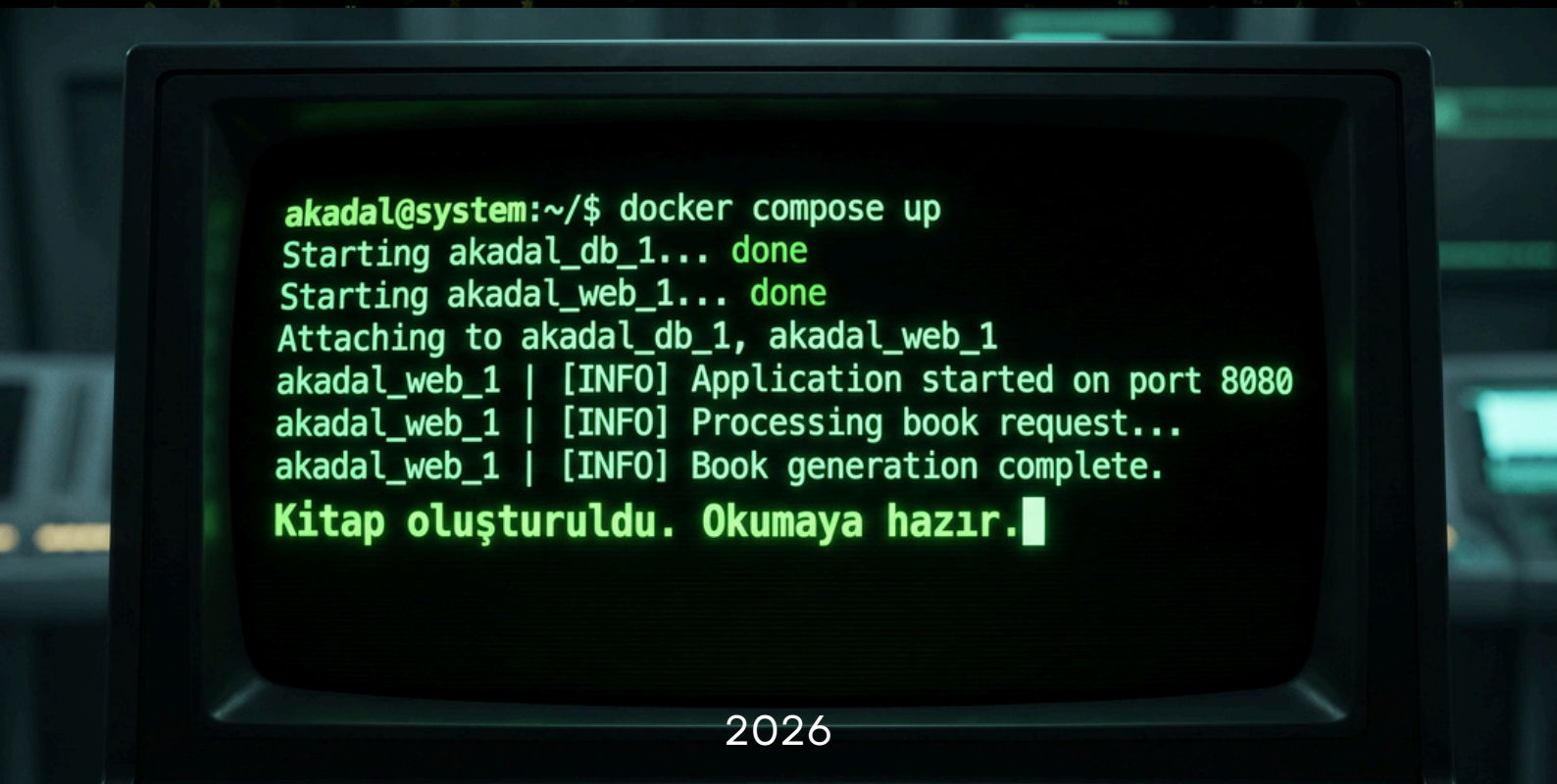
İNETD İnternet Teknolojileri Derneği

Yönetim Bilişim Sistemleri

için

Modern Geliştirme Ortamları

Emre Akadal



2026

Yönetim Bilişim Sistemleri için Modern Geliştirme Ortamları

Emre Akadal

2026

Kitabın Adı

Yönetim Bilişim Sistemleri için Modern Geliştirme Ortamları

Yazar

Emre Akadal

Editör

Mehmet Hakan Satman

Yayın Yönetmeni

Serra Çelik

Yayıncılık Sertifika Numarası

76485

ISBN

978-625-96238-5-6

Baskı

(E-Kitap), 2026

Kapak Görseli

Emre Akadal

Yayınlayan

İnternet Teknolojileri Derneği
Rumeli Cad. No:36 Kat:5 D:502 34363 Şişli İstanbul
inetd.org.tr info@inetd.org.tr yayinevi@inetd.org.tr

Önsöz

Kod yazmak yazılım geliştirmek değildir. Bir sistem ortaya çıkartmak için kodlanmasının yeterli olduğunu düşünen kişi sayısı az değildir. Ancak bir yazılımı ürün haline getirmeye çalışan her kişi ve ekip eninde sonunda taşlı yollarla karşılaşmıştır. Kodlama ve neticesinde elde edilen dijital varlık, bir ürünü ortaya çıkartmanın yalnızca küçük bir adımıdır. Bir dijital ürünün son kullanıcıya ulaştırılabilmesi ve sürdürülebilir olması için planlanması ve uygulanması gereken birçok adım bulunmaktadır. Bu kitap yazılım geliştirme sürecinin kodlama dışındaki kısmını açıklamayı hedeflemektedir. Bir yazılım dilini merkeze almayacağız. Hatta hiç kodlama bilmeseniz bile bu kitap sizin için hala oldukça fayda sağlayabilir. Bir yönetici de bu konulara hakim olarak karar süreçlerini daha verimli hale getirebilir.

Yazılım geliştirmek için yerel bir değişken tanımlamadan tutun da sistemin hizmet vereceği sunucudaki tüm ayarlamalara kadar öğrenilmesi gereken çok sayıda hassas ayrıntı bulunmaktadır. Özellikle Yönetim Bilişim Sistemleri gibi birçok alanı kesiştiren ve köprü görevi gören alanlar ve beceri geliştirmek isteyenler için bu kitap ve içerdiği bilgiler, piyasada ihtiyaç duyulan ve avantaj sağlayıcı unsurları da beraberinde getirmektedir. Bilişim alanındaki birçok konuda olduğu gibi bu konuda da deneyimlemek, öğrenmeyi pekiştirebilecek en önemli eylemlerden birisi olacaktır.

Bilişim alanında üretim yapmak da eğitim vermek de geleceğe dair bir öngörüde bulunmak da çok zor ve tehlikeli. Bir günde birçok şeyin değiştiğine şahitlik ettiğimiz yıllardayız. Yakın zamana kadar yapay zeka[1] (YZ) bizim için şimdiye göre oldukça basit algoritmalar ve bir çalışma alanıyken bugün büyük dil modelleri sayesinde günlük iş süreçlerimizin değişilmez bir parçası haline geldi. Birçok kişi hala hayranlık duymakla birlikte YZ araçlarını çoktan kendi iş ve hobi süreçlerine entegre ettiler ve sayısız çıktı üretmeyi kolayca mümkün kıldılar. Bununla birlikte YZ konusunun riskli tarafına dikkat çeken kişi sayısı da hiç az sayılmaz. Bu konuda karikatürist Emrah Ablak'a ait bir konuşmanın ilgili kısmını aynen aktarmak istiyorum:

> "... Siz ne düşünüyorsunuz yapay zeka konusunda? Yapay zekayı bir araç olarak mı görüyorsunuz? Yapay zeka bir araç değil ama. Çünkü bir araçla ne yapacağını bilirsin. Yapay zeka öyle bir şey değil. Bir sürü şeye yarıyor ve **yaptığı şeyi çok kestiremiyorsun.** Kendi mesleğimden söyleyeyim; görsel üzerine çalışıyorum. **Yapay zekayla görsel üzerinde çalıştığım**

[1] Kitap boyunca yapay zeka kavramı büyük dil modellerini kastetmek için kullanılmıştır. Yapay zeka, dil modellerinden çok daha geniş bir çalışma alanıdır.

> **zaman yapay zekanın bana nasıl sonuç vereceğini asla kestiremiyorum ve her defasında farklı bir sonuç veriyor bana.** Yapay zeka aslında bir araç değil. Yapay zeka kabaca evrim sürecinin devamı gibi bir şey. Biz üst insanı yarattık. Yani evrim biyolojik olmak zorunda değil. Biz bunu dijital olarak başardık ve ileride bir gün yıldızlar arası, gezegenler arası yolculuk yapılacaksa bunu insanlar yapmayacak. Çünkü bizim zaten ne ömrümüz ne fiziğimiz bunun için uygun değil ama dijital bir şeyi her yere yollayabiliriz. Dijital bir şeyle her yeri inceleyebiliriz. Hani herkes 'yapay zeka dünyayı ele geçirecek mi?' diyor ya, geçirecek abi..."

Akademik tarafta da benzer kaygıları görebiliyoruz. Boşuna da değil bu kaygılar. YZ yüzünden birçok kabul edilemez hata ile karşılaştık: uydurma kaynaklar, gerçek olmayan bulgular, makale metninde unutulmuş şablon YZ yanıtları… Profesyoneller bir yandan teknolojiyi kullanmak isterken diğer yandan etik bir ihlal yapmadıklarından emin olmak istiyorlar. Hatta öyle ki; bazı makalelerde YZ araçlarının yazar olarak makalede beyan edildiğini gördük. Garipsedik, güldük ve eleştirdik. Ancak bu aslında etik ihlal yapmadan teknolojiden faydalanmak isteyenlerin çabasının bir sonucuydu. Akademisyenler naif insanlardır. Maaşlarına zam dahi talep etmez, kendilerine verilen görevleri genellikle geri çevirmez ve adlarına sürülecek bir lekeden ölesiye çekinirler. Bu da onları olabilecek en temkinli şekilde yaklaşmaya sevkeder.

Başka hangi teknolojik yenilik insanları böyle derinden etkilemiştir, bilemiyorum. Daktilo ile makale yazmaktan bilgisayara geçişte akademisyenler bilgisayarı da ikinci yazar olarak eklemişler midir acaba? Ya da "acknowledgement" başlığı altında "ben bu makalenin sayısal hesaplamalarında bilgisayar kullandım" demeye mecbur edilmişler midir?

YZ'yi çok yüksek beygirli bir otomobile benzetiyorum ben. İlerlemesi için yapacağınız en ufak bir harekette beklentinizden çok daha fazlasını verir size. Ancak direksiyondaki en ufak bir hakimiyet kaybı feci bir kaza ile sonlanabilir. Bu sebeple YZ ile bir iş yaparken; özellikle bir yazılım geliştirirken ya da bir akademik çalışma sürecinde faydalanırken direksiyona sıkı sıkı sarılmak gerekir. Bu kitaba YZ'nin geliştirme süreçlerinde kullanımıyla ilgili de bir bölüm ekledim. Bölümü okuduğunuzda fark edeceksiniz ki YZ'yi kullanmak istem (prompt) yazmaktan ibaret değil. Ayrıca istem mühendisliği diye bir mesleğin doğacağına da inanmıyorum. Olsa olsa "YZ terbiyeciliği" olabilir.

Levent Alpöge'nin 2026 yılındaki çalışması da YZ'nin akademik çalışmalardaki göz ardı edilemez faydalarına verilebilecek somut örneklerinden birisidir. Stephen Smale'ın 21. yüzyıl problemleri listesinde yer alan Jacobian varsayımı, 1939'dan beri çözülmeyi bekleyen klasik bir soruydu. Alpöge, Anthropic'in Claude Fable 5 modelinden faydalanarak bu varsayımın üç ve daha yüksek boyutlarda doğru olmadığını gösteren kısa ve açık bir karşı örnek ortaya koydu. Sonuçlar diğer bilim insanlarınca incelendi ve teyit edildi. Böylece onlarca yıldır kanıt aranan bir iddia, doğrulanabilir tek bir örnekle çürütüldü. Bu örnekte YZ, bilinmeyen bir şeyi insan yerine ispat etmek gibi bir görev üstlenmedi. Bunun yerine varsayımı geçersiz kılacak bir örnek keşfetti. Formül kolay hesaplanabilir olduğundan dileyen herkes bu örneği kontrol etmek için aynı hesabı tekrarlayabildi; tartışma da spekülasyondan çıkıp "örnek doğru mu, ne anlama geliyor, iki boyut neden ayrı duruyor" sorularına döndü. Yani bilim bir adım daha ilerledi. Akademide YZ'nin değeri tam da bu noktada belirginleşmektedir. Keşfi hızlandırır,

sonuçları herkesin denetleyebileceği hale getirir ve araştırmacının enerjisini tekrarlı ve katma değer faydası olmayan işlerden strateji geliştirme ve süreci yönetme tarafına kaydırır.

Bu kitap içeriğinin sağlayacağı tüm faydalarla birlikte, YZ ile yapılabileceklerin bir örneği olarak da karşınızda durmaktadır. Ders içeriğimi kitaplaştırmak olarak çıktığım bu yolda tüm kitap içeriğini oluşturmakta çok yoğun şekilde YZ araçlarından (Codex, Claude, Grok ve Kimi) faydalandım. Hatta metinlerin neredeyse tamamının oluşturulmasında, düzenlenmesinde, yeniden biçimlendirilmesinde, dil düzeltmelerinde, görsel ve tablo üretimlerinde, hataların tespit edilmesi ve giderilmesinde, güncel kaynakların taranmasında YZ araçlarını yoğun şekilde kullandım. Elbette kitabın tamamı tek bir istem (prompt) sonucu ortaya çıkmadı. Kitabın yazım süreci öncesinde detaylı bir bağlam mimarisi kurmam gerekti. Sonrasında fazlar şeklinde yazım gerçekleşti. İçerik hem mimari hem metin olarak defalarca güncellendi. Sonrasında ise hem doğrudan kendim hem de YZ araçları ile tekrarlı denetimler gerçekleştirdim. Birkaç turluk okuma ve düzeltme aşamalarından sonra kitap elinizdeki görünüme ulaştı. YZ desteğiyle yazılan bir kitabın emeksiz ortaya çıktığı gibi bir yanılgının oluşmamasını diliyorum. Zira doğrudan kendi yazdığım metinleri bu kadar fazla kez okuyup düzeltmemişimdir. Dedim ya, direksiyonu sıkı sıkı tutmak gerekiyor. Aksi halde farklı yollara sapmanın ne kadar kolay olduğunu da birçok projemde deneyimledim. Kitaptaki tüm içeriğin, orijinalliğin ve sorumluluğun tarafıma ait olduğunu da beyan etmek isterim.

Kitabın tamamını titizlikle okuyup dönüşleriyle iyileştiren ve zenginleştiren kitabın değerli editörü Prof. Dr. Mehmet Hakan Satman'a, kitabın yayımlanmasını ve ücretsiz dağıtılmasını mümkün kılan İnternet Teknolojileri Derneği (İNETD) Yönetim Kurulu'na, süreç boyunca destekleri için aileme ve yorumlarıyla bu kitabı zenginleştiren değerli öğrencilerime çok teşekkür ederim.

Doç. Dr. Emre Akadal
2026

Öğrencilerime...

Kısaltmalar ve terimler

Birçok alanda olduğu gibi bilişim alanında da teknik ve mesleki kelimeler var ve bu kelimeler genellikle İngilizce dilindedir. Bu kelimeleri günlük dilde çok fazla kullanıyoruz ancak konu akademik bir kitap olduğunda dengeyi bulmak oldukça zorlaşıyor. Bu kitapta bilişimin doğası gereği çok sayıda İngilizce kelime kullanılmaktadır. Mümkün mertebe çevirilerini de sunuyor olmakla birlikte bazı durumlarda İngilizce karşılıklarını kullanmak konusunda çaresiz bir mecburiyet yaşadım. Bu durumu bir nebze aşabilmek ve hangi kelimenin ne amaçla kullanıldığını sunabilmek için kısaltma ve terimlere dair bir içerik ile başlamanın doğru olacağını düşündüm. Bir göz gezdirmeniz yeterli olacaktır. Kitabı okurken ihtiyaç duymanız halinde bu sayfaya dönerek kavramı inceleyebilirsiniz.

Kısaltmalar

Tablo 1: Kitapta geçen başlıca kısaltmalar

Kısaltma	İngilizce	Türkçe
ADR	Architecture Decision Record	Mimari karar kaydı
AI / YZ	Artificial Intelligence	Yapay zekâ
API	Application Programming Interface	Uygulama programlama arayüzü
CD	Continuous Delivery / Deployment	Sürekli teslim / sürekli dağıtım
CI	Continuous Integration	Sürekli entegrasyon
CLI	Command-Line Interface	Komut satırı arayüzü
DevOps	Development + Operations	Geliştirme ve operasyon bütünleşmesi
DNS	Domain Name System	Alan adı sistemi
E2E	End-to-End	Uçtan uca (test)
GUI	Graphical User Interface	Grafik arayüz
HTTP	Hypertext Transfer Protocol	Hiper metin aktarım protokolü
HTTPS	Hypertext Transfer Protocol Secure	Güvenli hiper metin aktarım protokolü
JSON	JavaScript Object Notation	JavaScript nesne gösterimi
MCP	Model Context Protocol	Model bağlam protokolü
PR	Pull Request	Değişiklik isteği
PRD	Product Requirements Document	Ürün gereksinim belgesi

Kısaltma	İngilizce	Türkçe
SHA	Secure Hash Algorithm	Güvenli özet algoritması
SSH	Secure Shell	Güvenli kabuk
SSL	Secure Sockets Layer	Güvenli yuva katmanı
VCS	Version Control System	Sürüm kontrol sistemi
VM	Virtual Machine	Sanal makine
VPS	Virtual Private Server	Sanal özel sunucu
WSL	Windows Subsystem for Linux	Linux için Windows alt sistemi
YAML	YAML Ain't Markup Language	Yapılandırma dosyası biçimi

Terim karşılıkları

Tablo 2: Kitapta tutulan terim karşılıkları

Terim	Türkçe karşılığı
audit	denetim
artifact	çıktı paketi
branch	dal
branch protection	dal koruması
build (imaj)	imaj oluşturmak
build (yazılım)	derleme
cache	önbellek
changelog	değişiklik günlüğü
chat	sohbet
commit	değişiklik kaydı
container	konteyner
copilot	kodlama asistanı
deployment	dağıtım
development	geliştirme ortamı
domain	alan adı; (bağlama göre) iş alanı
endpoint	uç nokta
environment variable	ortam değişkeni
flag	seçenek bayrağı
health check	sağlık denetimi
host	ana makine
hot reload	anında yenileme
issue	konu kaydı
job	iş (iş hattı adım grubu)
lint	statik denetim
lock file	kilit dosyası
localhost	yerel ana makine
log	günlük
merge	birleştirme
migration	göç, veritabanı taşıma
pipeline	iş hattı

Terim	Türkçe karşılığı
port	bağlantı noktası
production	üretim ortamı
prompt	istem
publish (port)	dışarı/erişime açmak
pull request	değişiklik isteği
registry	kayıt deposu
release	sürüm paketi
repository	depo
review	inceleme
rollback	sürüm geri alma
runner	çalıştırıcı
runtime	çalışma zamanı
secret	gizli bilgi
shell	kabuk
staging (Git)	hazırlama alanı
staging (ortam)	hazırlama ortamı
step	adım
tag	etiket
token	token, jeton
trigger	tetikleyici
volume	kalıcı veri alanı
workflow	iş akışı

İçindekiler

Şekil Listesi

Tablo Listesi

Bölüm 1

Geliştirme Ortamlarına Giriş

Bir ders projesiyle başlayalım. Dört kişilik bir proje ekibi, küçük bir stok takip uygulaması geliştirmek istiyor. Ekipte biri arayüzle, biri veritabanıyla, biri raporlama ekranlarıyla, biri de proje dokümantasyonuyla ilgileniyor. İlk günlerde her şey oldukça basit görünür: Herkes kendi bilgisayarında kodu açar, gerekli programları kurar ve çalışmaya başlar. Ancak proje ilerledikçe küçük farklar büyümeye başlar. Bir ekip üyesinin bilgisayarında çalışan uygulama diğerinde hata verir. Birinde veritabanı sürümü farklıdır, diğerinde kullanılan paket güncel değildir. Bir kişi dosyayı e-posta ile gönderir, başka biri aynı dosyanın eski halini düzenler. Bu noktada problem artık yalnızca kod yazmak değildir; ekip olarak aynı çalışma düzenini sürdürebilmektir.

İşte geliştirme ortamı dediğimiz konu burada anlam kazanır. Geliştirme ortamı, yazılım üretmek için kullanılan araçların toplamından daha fazlasıdır. Bilgisayar, işletim sistemi, terminal, editör, programlama dili, paket yöneticisi, sürüm kontrol sistemi, dokümantasyon, test düzeni, konteyner (İngilizce *container*) düzeni ve yayın süreci bir araya gelerek ekibin çalışma zeminini oluşturur. Bu zeminin amacı herkesi aynı düşünmeye zorlamak değildir; herkesin aynı işi, anlaşılabilir ve tekrarlanabilir biçimde yapabilmesini sağlamaktır.

Bu kitapta geliştirme ortamına şu gözle bakacağız:

Şekil 1.1: Geliştirme ortamının temel katmanları

Şekil 1.1 kesin bir sıra listesi değildir; daha çok geliştirme pratiğinin hangi katmanlardan oluştuğunu gösteren sade bir haritadır. Örneğin bir ders ödevinde CI/CD hattı kurmak gerekmeyebilir; ancak bir şirket projesinde aynı adım kalite kontrolün ve teslim güvenilirliğinin temel parçası haline gelebilir.

Geliştirme ortamlarını türlerine göre düşündüğümüzde de tek bir doğru yoktur. Bir geliştirici, uygulamayı doğrudan kendi bilgisayarında çalıştırarak işe başlayabilir. Aynı proje büyüdüğünde ekip üyeleri arasındaki kurulum farklarını azaltmak için sanal makinelerden veya konteyner düzenlerinden faydalanmak isteyebilir. Uzaktan çalışma, merkezi kaynak yönetimi ya da güçlü donanıma erişim gerektiğinde ise bulut tabanlı ortamlar anlam kazanabilir.

Ancak bu seçenekler birbirinin basit birer alternatifi değildir. Yerel ortam doğrudan ve hızlı bir çalışma düzeni sağlarken kurulum farklarını geliştiricinin sorumluluğuna bırakır. Sanal makineler ve konteyner düzenleri ortamı daha denetlenebilir hale getirebilir; buna karşılık öğrenme, kaynak kullanımı ve bakım yükü getirir. Bulut tabanlı ortamlar erişimi kolaylaştırabilir; fakat maliyet, bağlantı ve veri güvenliği gibi yeni sorular ortaya çıkarır. Bu sebeple geliştirme ortamı seçerken aracın adından önce projenin ihtiyacına bakmak gerekir.

Gördüğünüz gibi geliştirme ortamı seçimi yalnızca teknik bir tercih değildir. Aynı zamanda ekip büyüklüğü, teslim tarihi, bakım sorumluluğu, güvenlik beklentisi ve bütçe ile ilgilidir. Yönetim Bilişim Sistemleri açısından bu bölümün temel amacı da budur: Araçların adını bilmekten önce, bu araçların organizasyonel sonuçlarını okuyabilmek.

1.1 Bu Kitap Hakkında

İlk bakışta bu kitabın ele aldığı konu basit görünebilir: Okuyucu geliştirme araçlarını öğrenmelidir. Bir editör kuracak, terminal kullanacak, Git ile kod paylaşacak, Docker ile uygulama çalıştıracak ve gerektiğinde sunucuya yayın yapacaktır. Bu cevap başlangıç için doğrudur; çünkü geliştirme ortamları gerçekten de somut araçlarla karşılaşarak öğrenilir.

Ancak yalnızca araç isimlerini bilmek çoğu zaman yeterli değildir. Bir ekipte sorun çıktığında problem "Git bilmiyoruz" kadar basit olmayabilir. Belki dal (branch) kullanımı ekip anlaşmasına dönüşmemiştir. Belki kurulum adımları belgelenmemiştir. Belki bir paketin sürümü herkesin bilgisayarında farklıdır. Belki test yapılmadan üretim ortamına geçildiği için müşteri karşısında güven kaybı oluşmuştur.

Bunu küçük bir proje üzerinden düşünelim. Bir proje ekibi, dönem sonunda teslim edilecek bir web uygulaması geliştiriyor olsun. Teslime iki gün kala uygulama bir bilgisayarda çalışıyor, diğerinde çalışmıyor. Bu durumda teknik hata önemlidir; fakat Yönetim Bilişim Sistemleri açısından asıl soru biraz daha geniştir: Bu hata neden son ana kadar fark edilmedi? Kurulum süreci neden tekrarlanabilir değildi? Ekip içinde kim hangi değişikliği yaptı ve bu değişiklik nasıl izlenebilirdi?

Bu kitap, tam olarak bu soruların etrafında konumlanır. Her aracın bütün teknik ayrıntılarını tüketmek yerine geliştirme ortamını iş süreci, ekip iletişimi, maliyet ve risk okuryazarlığı açısından okuyabilecek bir temel kurar. Böylece okuyucu yalnızca komut yazan biri değil, teknik çalışma düzeninin iş sonuçlarını anlayabilen biri haline gelir.

1.2 Geliştirme Ortamı Nedir?

Bir yazılım projesine yeni dahil olduğunuzu düşünün. Size bir bağlantı gönderiliyor, bir depo adresi paylaşılıyor ve "projeyi bilgisayarında çalıştır" deniyor. Bu cümle kısa görünür; ancak arkasında birçok küçük gereksinim bulunur. Hangi işletim sistemi kullanılacak? Hangi programlama dili sürümü gerekli? Veritabanı nerede çalışacak? Gizli ayarlar nasıl verilecek? Uygulamanın başarılı çalıştığını nasıl anlayacağız?

Peki bu gereksinimleri tek bir çatı altında nasıl düşünebiliriz? Bu noktada geliştirme ortamı (development environment) kavramı devreye girer. Geliştirme ortamı, bir

yazılımın yazılması, çalıştırılması, test edilmesi, paylaşılması ve gerektiğinde yayın sürecine hazırlanması için kullanılan teknik ve düzenleyici öğelerin bütünüdür. Bu tanımın içinde araçlar vardır; fakat yalnızca araçlar yoktur. Kurallar, alışkanlıklar, dokümantasyon ve ekip anlaşmaları da bu ortamın parçasıdır.

Örneğin bir Node.js projesinde editör olarak Visual Studio Code, çalışma zamanı olarak Node.js, bağımlılık yönetimi için `npm`, sürüm kontrolü için Git, yerel veritabanı için PostgreSQL ve çalıştırma bilgileri için bir `README.md` dosyası kullanılabilir. Bunların her biri ayrı bir araçtır. Ancak geliştirme ortamı, bu araçların birlikte nasıl çalıştığını ve ekibin bu düzeni nasıl tekrar edebildiğini ifade eder.

Pratik sınırı iyi çizmek gerekir. Geliştirme ortamını anlamak, her işletim sisteminin iç yapısını veya her programlama dilinin derleyici ayrıntılarını öğrenmek anlamına gelmez. Bir aracın projede hangi görevi üstlendiğini, ekip içinde hangi riski azalttığını ve yanlış kullanıldığında hangi bakım maliyetini üretebileceğini görmek bu kitap için yeterli derinliği sağlar.

1.3 Kod Yazmak ve Yazılım Geliştirmek

Kod yazmak, yazılım geliştirmenin en görünür parçasıdır. Bir fonksiyon oluşturmak, bir form ekranı tasarlamak, veritabanından veri çekmek veya bir hata mesajını düzeltmek doğrudan kodla ilişkilidir. Bu sebeple okuyucunun ilk ilgisi çoğu zaman kodun kendisine yönelir. Bu oldukça anlaşılır bir durumdur.

Ancak yazılım geliştirme yalnızca kod üretmek değildir. Yazılım geliştirme; ihtiyacı anlamayı, çözümü tasarlamayı, kodu düzenli biçimde saklamayı, değişiklikleri izlemeyi, ekip içinde paylaşmayı, test etmeyi, yayınlamayı ve sonrasında bakımını üstlenmeyi içerir. Bir başka ifadeyle kod yazmak üretimin bir adımıdır; yazılım geliştirmek ise bu üretimin sürdürülebilir bir sürece dönüşmesidir.

Bu ayrımın kaba bir haritasını Tablo 1.1 sunar. Gerçek projelerde sınırlar elbette daha geçirgendir; yine de karşılaştırma, iki kavramın nerede ayrıştığını görmeyi kolaylaştırır.

Tablo 1.1: Kod yazmak ve yazılım geliştirmek

Bakış açısı	Kod yazmak	Yazılım geliştirmek
Temel odak	Belirli bir işlevi çalıştırmak	Çalışan işlevi sürdürülebilir sisteme dönüştürmek
Zaman ufku	Anlık görev veya ödev	Proje yaşam döngüsü
Başarı ölçütü	Kodun o anda çalışması	Kodun ekip tarafından anlaşılması, test edilmesi ve bakımının yapılabilmesi
Risk	Hata çoğu zaman yerel kalır	Hata teslim tarihi, kullanıcı deneyimi ve maliyet üzerinde etki üretir
İletişim ihtiyacı	Daha sınırlı olabilir	Analist, geliştirici, yönetici ve kullanıcı arasında ortak dil gerekir

Burada kod yazma işi önemsizleştirilmeye çalışılmamaktadır. Tam tersine, iyi yazılım geliştirme iyi kod yazma becerisini de içerir. Kritik nokta, kodun organizasyon içinde nasıl bir iş değeri, maliyet ve risk zincirine bağlandığını görebilmektir.

Bu ayrım pratikte şunu değiştirir: Bir hatayla karşılaştığınızda yalnızca "hangi satır bozuk?" diye sormazsınız. Aynı zamanda "bu hata neden daha önce yakalanmadı, ekip bunu nasıl paylaşmalı, dokümantasyonda ne eksik, yayın sürecinde hangi kontrol yoktu?" diye de sorarsınız. Yazılım geliştirme düşüncesi burada başlar.

1.4 Geliştirme Ortamı Okuryazarlığı

Bir toplantıda geliştirici "üretim (production) ortamdaki paket sürümü yerel (local) ortamdan farklı" dediğinde, bu cümle ilk anda yalnızca teknik bir ayrıntı gibi duyulabilir. Fakat bu ayrıntı müşteri şikayetine, yayın gecikmesine, fazladan mesaiye veya güven kaybına dönüşebilir. Geliştirme ortamı okuryazarlığı tam bu noktada önem kazanır.

Geliştirme ortamı okuryazarlığı, her aracı uzman düzeyinde kullanmak anlamına gelmez. Daha doğru ifade şudur: Bir geliştirme ortamının hangi bileşenlerden oluştuğunu, bu bileşenlerin iş sürecini nasıl etkilediğini ve teknik kararların ekip, maliyet, zaman ve risk üzerinde nasıl sonuç doğurduğunu anlayabilmektir. Bu okuryazarlık, teknik ekip ile iş tarafı arasında sağlıklı bir tercüme zemini oluşturur.

Örneğin bir proje yöneticisi "Docker kullanalım mı?" sorusuna yalnızca "modern bir araç olduğu için kullanalım" diye yaklaşırsa eksik düşünmüş olur. Daha yerinde soru şudur: Ekipte kurulum farkları sık yaşanıyor mu? Uygulama birden fazla servisle mi çalışıyor? Bu aracın öğrenme maliyeti teslim takvimini etkiler mi? Üretim ortamıyla geliştirme ortamı arasındaki farkları azaltacak mı? İşte okuryazarlık, aracı isim olarak bilmekten çok bu soruları sorabilmektir.

Bu bakış açısı ekip içindeki iletişimi de doğrudan etkiler. Teknik bir aracı iş süreci içinde okumak, aracın yalnızca ne yaptığını değil, hangi iş problemini azalttığını da sormaktır. Örneğin Git yalnızca kod saklama aracı değildir; ekipte kimin neyi ne zaman değiştirdiğini görünür kılar. Ancak Git kullanmak tek başına iyi takım çalışması anlamına gelmez. Dal kuralları, commit mesajları, kod inceleme alışkanlığı ve geri dönüş planı yoksa araç vardır, fakat süreç hâlâ zayıftır.

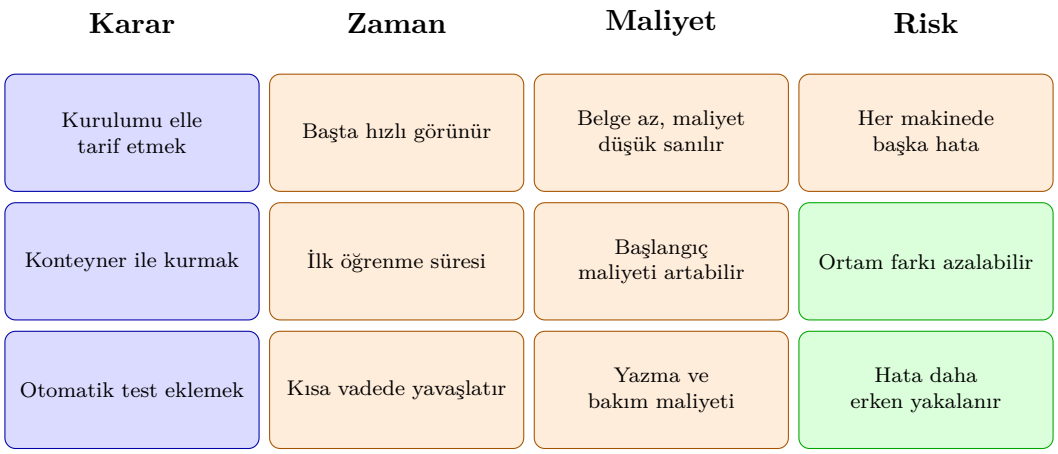
Benzer şekilde, bir projede yazılımcı "dependency çakışması var" dediğinde analist bunu teslim tarihine etkisi açısından, yönetici ise maliyet ve müşteri taahhüdü açısından duymak ister. Aynı cümle farklı kişiler için farklı anlamlar taşır. Bu sebeple geliştirme ortamı okuryazarlığı, teknik terimi iş sonucuna bağlayan ortak bir dil kurmaya yardımcı olur.

Küçük bir örnek düşünelim. Teslime yakın bir zamanda uygulama yeni bilgisayarda çalışmıyorsa yazılımcı paket sürümlerini inceler, analist test senaryolarının hangi ortamda denendiğini sorar, yönetici ise gecikmenin kapsamını görmek ister. Ortak dil kurulmadığında herkes kendi parçasını doğru söylese bile ekip aynı problemi konuşamayabilir. Geliştirme ortamı okuryazarlığının önemi tam olarak burada ortaya çıkar.

1.4.1 Zaman, maliyet ve risk etkisi

Teknik kararlar çoğu zaman teknik gerekçelerle alınır; fakat sonuçları yalnızca teknik alanda kalmaz. Diyelim ki ekip, projeyi herkesin kendi bilgisayarında elle kurması yerine bir konteyner düzeniyle çalıştırmaya karar verdi. Bu karar başlangıçta öğrenme süresi gerektirir. Ancak kurulum hatalarını azaltabilir, yeni ekip üyesinin projeye katılmasını hızlandırabilir ve üretim ortamına geçişte daha öngörülebilir bir zemin sağlayabilir.

Şekil 1.2 bu dengeyi üç kararda özetler. Yeşil, istenen etkiyi; turuncu, ödenen bedeli gösterir.



Şekil 1.2: Teknik kararın zaman, maliyet ve risk etkisi

Görselin söylediği şey sadedir: teknik tercih, iş tarafında zaman planı, bütçe ve güvenilirlik kararıdır. Ayrıntıyı ezberlemek değil, bedeli ve faydayı birlikte okumak yeterlidir.

1.5 Geliştirme Ortamının Temel Bileşenleri

Bir proje bilgisayarda çalışmadığında çoğu zaman tek bir sebep ararız. “Kod bozuk” deriz, “veritabanı çalışmıyor” deriz veya “paket eksik” deriz. Oysa geliştirme ortamı birden fazla bileşenin birlikte çalışmasıyla oluşur. Bu bileşenlerden biri eksik, uyumsuz veya yanlış yapılandırılmışsa uygulamanın tamamı sorunlu görünebilir.

Peki bu bileşenleri nasıl düşünmeliyiz? Geliştirme ortamının temel bileşenleri (İngilizcede development environment components), yazılım üretim sürecini mümkün kılan teknik ve düzenleyici parçalardır. Biz burada bu parçaları ayrı ayrı ezberlemek yerine, her birinin projede hangi görevi üstlendiğine bakacağız.

Basit bir web uygulaması için tablo şu şekilde okunabilir:

Tablo 1.2: Geliştirme ortamı bileşenlerinin görevi

Bileşen	Temel görevi	Beklenen soru
İşletim sistemi ve terminal	Uygulamanın çalışacağı yerel zemini sağlar	Ekip aynı komutları benzer biçimde çalıştırabiliyor mu?

Bileşen	Temel görevi	Beklenen soru
Editör, dil ve çalışma zamanı (runtime)	Kodun yazılmasını ve çalışmasını sağlar	Kullanılan sürüm ve araçlar belgelenmiş mi?
Paket yöneticisi	Dış bağımlılıkları yönetir	Bağımlılıklar tekrar kurulabilir ve izlenebilir mi?
Sürüm kontrolü	Değişiklik geçmişini tutar	Kim, neyi, neden değiştirdi sorusu cevaplanabiliyor mu?
Konteyner ve yayın ortamı	Uygulamayı daha kontrollü koşullarda çalıştırır	Geliştirme, test ve üretim ortamı farkları yönetiliyor mu?

Tablo 1.2 bir kontrol listesi gibi okunabilir; fakat tek başına yeterli değildir. Her proje aynı bileşenleri aynı derinlikte kullanmaz. Küçük bir ders projesinde basit bir README dosyası yeterli olabilirken, kurumsal bir uygulamada otomatik test, konteyner, loglama ve güvenlik kontrolleri zorunlu hale gelebilir. Bu sebeple geliştirme ortamı bileşenlerini anlamak, projenin ölçeğine göre doğru soruları sorabilmektir.

1.5.1 İşletim sistemi, dosya sistemi ve terminal

Bir ofiste herkesin aynı evrak dolabını farklı isimlerle düzenlediğini düşünün. Bir kişi sözleşmeleri "Belgeler" klasörüne, diğeri "Projeler" klasörüne, üçüncüsü masaüstüne koyuyorsa aranan dosyayı bulmak zaman alır. İşletim sistemi, dosya sistemi ve terminal de geliştirme ortamında buna benzer bir zemin oluşturur. İşletim sistemi bilgisayarın temel çalışma düzenini sağlar; dosya sistemi proje dosyalarının nerede durduğunu belirler; terminal ise bu düzene komutlarla erişmemizi sağlar.

İşletim sistemi; işlemci, bellek, depolama ve ağ gibi kaynakların uygulamalar tarafından kullanılmasını düzenler. Geliştirici açısından bunun karşılığı çalışan süreçler, kullanıcı hesapları, dosya izinleri, bağlantı noktaları ve kurulu sistem araçlarıdır. Aynı proje Windows, macOS ve Linux üzerinde benzer bir amaca hizmet etse de dosya yolları, komutlar, izinler ve paket kurma biçimleri farklılaşabilir. Bu farklar hesaba katılmadığında sorun kodda değil, kodun çalıştığı zeminde ortaya çıkar.

Dosya sistemi bu zeminin düzenini görünür hale getirir. Proje kaynakları, yapılandırma dosyaları, bağımlılıklar ve üretilen çıktılar belirli dizinlerde tutulur. Terminalde çalışan kabuk (shell) ise bu dosyalar ve süreçler üzerinde tekrarlanabilir işlemler kurmamızı sağlar. Bir komutu dokümana yazabilir, ekip arkadaşımızla paylaşabilir veya daha sonra bir betiğin parçası haline getirebiliriz. Grafik arayüzle yapılan bir işlemin adımları gözden kaçabilirken komut satırındaki işlem açık bir ifadeye dönüşür.

İşletim sistemlerinin iç mimarisine inmeden önce bir projenin hangi dizinde durduğunu, hangi komutla çalıştığını, hangi dosyanın yapılandırma taşıdığını ve bu bilgilerin ekip içinde nasıl paylaşılacağını görmek yeterlidir. Çünkü yanlış dizinde çalıştırılan basit bir komut bile saatlerce süren hata aramaya dönüşebilir. Kısım 2 içinde işletim sistemi ailelerini, çekirdek kavramını, dosya sistemi düzenini, izinleri, süreçleri, ortam değişkenlerini ve uzaktan erişimi daha ayrıntılı biçimde ele alacağız.

1.5.2 Editör, programlama dili, çalışma zamanı (runtime) ve paket yöneticisi

Kod yazmak için kullandığımız yazılımlar genellikle editör olarak tanımlanır; fakat kodun çalışması için editör tek başına yetmez. Programlama dili kodun ifade biçimini, çalışma zamanı (runtime) bu kodun hangi ortamda çalışacağını, paket yöneticisi ise projeye eklenen dış kütüphanelerin nasıl kurulacağını belirler. Örneğin JavaScript yazarken VS Code editör olabilir, Node.js çalışma zamanı görevini üstlenebilir, `npm` ise bağımlılıkları yönetebilir.

Ancak burada kritik nokta araç adlarından daha fazlasıdır. Ekipte bir kişi Node.js 18, diğeri Node.js 22 kullanıyorsa aynı kod farklı davranabilir. Bu yüzden geliştirme ortamı okuryazarlığı, "hangi dili kullanıyoruz?" sorusunun yanına "hangi sürümle, hangi paketlerle, hangi kurulum adımlarıyla çalışıyoruz?" sorusunu da ekler. Bu da ekip büyüdükçe olası kombinasyon sayısını hızla artırır; iki geliştiricinin ortamının birebir aynı kalması bir noktadan sonra neredeyse imkânsızlaşır.

1.5.3 Sürüm kontrolü, dokümantasyon ve takım akışı

Bir proje birkaç ay sonra yeniden açıldığında genellikle iki soru ortaya çıkar: Bu değişiklik neden yapılmıştı ve yeni katılan biri işe nereden başlayacak? Kodun kendisi bu soruları yanıtlamaz; onu açıklayan bir geçmiş ve ortak bir başlangıç noktası gerekir. Sürüm kontrolü, dokümantasyon ve takım akışı, geliştirme ortamının tam bu boşluğu dolduran ve en sık başvurulan parçaları arasındadır; çünkü bir proje yalnızca dosyaların son halinden oluşmaz. Hangi değişikliğin ne zaman ve neden yapıldığı, gerektiğinde eski bir duruma nasıl dönüleceği ve projeyi açan bir kişinin işe nereden başlayacağı da çalışma düzeninin parçasıdır.

Bu ihtiyacı karşılayan araçlara sürüm kontrol sistemi (version control system, VCS) adı verilir. Bir VCS, dosyalardaki değişiklikleri zaman içinde kaydeder ve belirli durumları yeniden incelemeyi mümkün kılar. Git, bu amaçla kullanılan dağıtık bir sürüm kontrol sistemidir. Her geliştirici projenin geçmişini içeren yerel bir depoyla çalışabilir; değişikliklerini commit (değişiklik kaydı) olarak kaydedebilir, dallarla birbirinden ayırabilir ve gerektiğinde birleştirebilir.

GitHub ise Git'in kendisi değildir. Git depolarını uzakta barındıran ve onların çevresinde iş birliği özellikleri sunan bir platformdur. Değişiklik istekleri (pull request), konu kayıtları (issue), kod inceleme, proje panoları ve otomasyonlar Git ile tutulan geçmişi ekip sürecine bağlar. Başka platformlar da benzer hizmetler sunabilir; dolayısıyla temel beceri github.com gibi belirli bir web sitesini ezberlemeyi hedeflemektense Git'in çalışma şeklini ve yeteneklerini anlamaktır.

Bireysel çalışan biri için Git, güvenli denemeler yapmayı, değişiklikleri anlamlı adımlara ayırmayı ve hata durumunda geçmişi incelemeyi sağlar. Ekipte ise buna eş zamanlı çalışma, sorumluluğun görünür olması ve değişikliklerin birleştirilmeden önce gözden geçirilmesi eklenir. Ancak araç tek başına düzen kurmaz. Commit'lerin taşıdığı değişikliklerin ve eklenen mesajların anlamlı ve tutarlı olması, dal kullanımının ortak kurallara dayanması ve inceleme sorumluluğunun paylaşılması gerekir.

Dokümantasyon bu geçmişin neden ve nasıl tarafını tamamlar. Özellikle Markdown,

sade metin düzeni sayesinde README dosyalarından konu kaydı (issue) ve değişiklik isteği (pull request) açıklamalarına kadar geliştiricinin günlük işinde sürekli karşısına çıkar. Markdown dosyaları Git ile birlikte tutulabildiği için dokümandaki değişiklik de kod değişikliği gibi incelenebilir. Bu sebeple Git ve Markdown, Kısım 4 ve Kısım 5 içinde daha geniş ele alınacaktır.

Basit bir ekip akışı şöyle kurulabilir:

1. Geliştirici kendi dalında (branch) değişiklik yapar; böylece ana çalışma çizgisi korunur.
2. Commit mesajıyla yaptığı değişikliği kısaca açıklar; böylece geriye dönük izleme mümkün olur.
3. Değişiklik isteği açar; böylece ekip değişikliği görür ve tartışır.
4. README veya ilgili dokümanı günceller; böylece bilgi yalnızca yapan kişinin zihninde kalmaz.
5. Değişiklik kontrol edildikten sonra ana dala (main branch) alınır; böylece kalite kontrol sürecin parçası haline gelir.

Bu akışın amacı bürokrasi üretmek değil, ekip büyüdükçe kaybolabilecek bilgiyi kayıt altına almak ve geri dönülebilir bir çalışma düzeni kurmaktır.

1.5.4 Konteyner, CI/CD, sunucu ve dağıtım ortamı

Konteyner, CI/CD, sunucu ve dağıtım ortamı, geliştirme sürecini "benim bilgisayarımda çalışıyor" savunmasından "kullanıcıya güvenilir biçimde ulaşıyor" güvencesine taşır. Konteyner uygulamayı daha kontrollü bir çalışma paketi içinde düşünmemize yardım eder. CI/CD, yani sürekli entegrasyon ve sürekli teslim ve dağıtım (continuous integration / continuous delivery-deployment), kod değişikliklerinin otomatik kontrollerden geçmesini sağlar. Sunucu ve dağıtım ortamı ise uygulamanın gerçek kullanıcıya ulaştığı zemindir.

Bu süreç küçük adımlarla şöyle okunabilir:

1. Uygulama yerel ortamda çalışır; geliştirici temel işlevi doğrular.
2. Konteyner tanımı hazırlanır; uygulamanın hangi koşullarda çalışacağı daha açık hale gelir.
3. Otomatik test veya kontrol hattı çalışır; hata mümkünse erken yakalanır.
4. Uygulama test veya hazırlama ortamına alınır; üretime benzer koşullarda gözlenir.
5. Üretim ortamına yayın yapılır; artık teknik hata kullanıcı deneyimi ve iş sürekliliği meselesine dönüşür.

Bu adımların her biri teknoloji içerir; fakat her biri aynı zamanda kalite ve risk yönetimi adımıdır.

1.6 Neden Bu Teknolojilere İhtiyaç Duyarız?

Standart geliştirme ortamına ihtiyaç duymamızın ilk sebebi oldukça basittir: Aynı projeyi farklı kişilerin benzer biçimde çalıştırabilmesi gerekir. Bir ekip üyesinin bilgisayarında uygulama açılıyor, diğerinde açılmıyorsa ekip zamanı kod geliştirmek yerine

kurulum farklarını anlamaya harcar. Bu da özellikle teslim tarihi yaklaşırken ciddi bir baskı oluşturur.

Ancak standartlaşma, herkesin aynı bilgisayarı kullanması veya her ayrıntının tek tip hale getirilmesi anlamına gelmez. Standart geliştirme ortamı, projenin çalışması için gerekli asgari koşulların açıkça tanımlanmasıdır. Hangi dil sürümü kullanılacak? Hangi paketler kurulacak? Veritabanı nasıl başlatılacak? Ortam değişkenleri nereden alınacak? Testler hangi komutla çalışacak? Bu soruların cevabı kişisel hafızaya bırakılmamalıdır.

Bir ekip senaryosu düşünelim. Beş kişilik bir grup müşteri destek paneli geliştiriyor. Yeni katılan ekip üyesi projeyi çalıştırmak için iki gün harcıyorsa, burada yalnızca teknik bir eksik yoktur; oryantasyon (onboarding) süreci de zayıftır. Aynı şekilde üretime alınacak bir uygulamada hangi ayarın nereden geldiği bilinmiyorsa, risk yalnızca geliştirici bilgisayarında kalmaz, müşteri deneyimine taşınır.

Bu sebeple standart geliştirme ortamı bir kontrol ve iletişim aracıdır. Teknik ekip için tekrar edilebilirlik sağlar; proje yöneticisi için zaman tahminini iyileştirir; iş tarafı için teslim güvenilirliğini artırır. Elbette her standart bir bakım maliyeti doğurur. Fakat standart yoksa maliyet çoğu zaman görünmez biçimde, hata ayıklama ve bekleme süresi olarak geri döner.

Bu fayda dört başlıkta somutlaşır: tekrarlanabilir kurulum, takım içi oryantasyon, ortam farklarından doğan hatalar ve bunların ortak görünümü olan “benim bilgisayarımda çalışıyor” problemi.

1.6.1 Tekrarlanabilir kurulum

Yeni bir ekip üyesi projeyi ilk kez çalıştırmaya çalıştığında genellikle kurulum adımlarını başka birine sorar; o kişi o an müsait değilse ilerleme durur. Tekrarlanabilir kurulum, projenin yalnızca bir kişinin bilgisayarında değil, aynı adımları izleyen herkesin bilgisayarında çalışabilmesidir. Bu yüzden kurulum bilgisi sözlü anlatıma veya WhatsApp mesajlarında kaybolan notlara bırakılmamalıdır. En azından `README.md` dosyasında hangi komutların çalıştırılacağı ve beklenen sonucun ne olduğu yazmalıdır.

Örneğin basit bir Node.js projesinde kurulum kontrolü şöyle gösterilebilir:

```
$ node --version
v20.11.1

$ npm install
added 128 packages in 6s

$ npm run dev
Local: http://localhost:5173/
```

Komutların ürettiği üç sinyal daha değerlidir: Node.js kurulu ve beklenen sürümdedir, proje bağımlılıkları kurulmuştur, uygulama yerel adreste çalışmaya başlamıştır. Ekip bu sinyalleri okuyabiliyorsa hata çıktısını da daha hızlı yorumlayabilir.

1.6.2 Takım içi oryantasyon

Takım içi oryantasyon, yeni bir kişinin projeye katıldığında işi anlayıp katkı verebilme sürecidir. Standart ortam bu süreci kişilere bağımlı olmaktan çıkarır.

Sağlıklı bir oryantasyon akışı genellikle şu adımları içerir:

1. Proje deposu paylaşılır; böylece kodun resmi kaynağı netleşir.
2. Kurulum belgesi okunur; böylece bilgi kişisel anlatıma sıkışmaz.
3. Örnek ortam dosyası hazırlanır; böylece gizli bilgiler paylaşılmadan çalışma düzeni kurulur.
4. İlk test veya geliştirme komutu çalıştırılır; böylece kurulumun başarılı olup olmadığı görülür.
5. Küçük bir değişiklik, değişiklik isteği (pull request) ile gönderilir; böylece takımın katkı kuralı öğrenilir.

Bu adımlar basit görünür. Ancak ekip büyüdüğünde basit adımların yazılı olması, zaman kaybını ve yanlış anlaşılmayı belirgin biçimde azaltır.

1.6.3 Ortam farklarından doğan hatalar

Ortam farklarından doğan hatalar genellikle küçük ayrıntılar gibi başlar. Bir bilgisayarda veritabanı portu farklıdır, diğerinde paket sürümü günceldir, bir başkasında gizli ayar dosyası eksiktir. Fakat bu farklar birleştiğinde ekip aynı hatayı yeniden üretmekte zorlanır. Hata yeniden üretilemiyorsa çözülmesi de gecikir.

Tablo 1.3 sık görülen durumları ve doğurdukları riski gösterir.

Tablo 1.3: Ortam farklarından doğan hatalara örnekler

Durum	Risk
Bir geliştiricide Node.js sürümü farklı	Kod bazı makinelerde çalışır, bazılarında kırılır
`.env` dosyası eksik	Uygulama veritabanına veya servise bağlanamaz
Veritabanı şeması güncel değil	Testte geçmeyen veri hataları oluşur
Satır sonu CRLF / LF karışır	Betikler ve fark çıktıları bozulur
Büyük/küçük harf duyarlı yol	Windows'ta açılan dosya Linux'ta bulunamaz
WSL'de Windows yolundan çalışmak	İzin ve performans sürprizleri çıkar
Farklı kilit dosyası kullanılmış	Kurulan paket ağacı kişiye göre değişir
Docker kapalıyken Compose beklenir	"Bende çalışıyor" yalnızca yerel servise bağlı kalır

Bu durumların her biri iş tarafında gecikme, kalite kaybı veya güven azalması olarak karşılık bulabilir. Ortam farkı küçük bir geliştirici sorunu değil, izlenmesi gereken bir projedir.

1.6.4 "Benim bilgisayarımda çalışıyor" problemi

"Benim bilgisayarımda çalışıyor" cümlesi çoğu geliştiricinin bir noktada duyduğu veya söylediği bir cümledir. Bu cümle ilk anda savunma gibi görünse de aslında bize önemli bir şey söyler: Çalışan şey yalnızca kod değildir; kodun çevresindeki ortam da sonucun parçasıdır.[1]

Dönem projesi teslimine bir gün kala uygulama ekip liderinin bilgisayarında çalışıyor, sunum yapılacak bilgisayarda çalışmıyorsa problem artık yerel olmaktan çıkar. Ekip zaman kaybeder, sunum riske girer, güven duygusu zedelenir. Bu yüzden iyi geliştirme ortamı, başarılı çalışmayı kişisel bilgisayara bağımlı olmaktan çıkarıp ekipçe tekrar edilebilir hale getirmeye çalışır.

1.7 Geliştirme Ortamı Türleri

Geliştirme ortamı denildiğinde çoğu kişinin aklına kendi bilgisayarı gelir. Bu başlangıç için doğaldır; çünkü okuyucular çoğu uygulamayı önce kendi cihazlarında yazar ve çalıştırır. Ancak proje büyüdükçe veya ekip farklı cihazlardan çalışmaya başladıkça yerel ortam tek seçenek olmaktan çıkar.

Bu noktada üç temel ortam türünden söz edebiliriz: yerel geliştirme ortamı, sanal makine veya konteyner tabanlı ortamlar, uzaktan ya da bulut tabanlı ortamlar. Her birinin güçlü tarafı ve bedeli vardır. Bu yüzden seçim yaparken "hangisi daha profesyonel?" sorusu yerine "bizim projemizin ölçeği, ekibin bilgisi, güvenlik beklentisi ve bütçesi açısından hangisi anlamlı?" sorusu daha sağlıklıdır. Aşağıdaki tablo bu üç ortam türünü güçlü yönleri ve bedelleri açısından yan yana koyar.

Tablo 1.4: Yerel, sanal makine ve bulut tabanlı ortamlar

Ortam türü	Güçlü tarafı	Sınırı veya bedeli
Yerel ortam	Hızlı başlamak ve öğrenmek kolaydır	Bilgisayar farkları hata üretebilir
Sanal makine	İşletim sistemi düzeyi daha kontrollüdür	Kurulum ve kaynak kullanımı ağır olabilir
Konteyner tabanlı ortam	Uygulama koşulları daha tekrar edilebilir hale gelir	Konteyner bilgisi ve bakım disiplini gerektirir
Bulut tabanlı ortam	Uzaktan erişim ve merkezi yönetim sağlar	Maliyet, erişim ve veri güvenliği dikkat ister

Tablo 1.4 bunu doğrular: Her satırda kazanılan bir şeyin karşılığında ödenen bir bedel vardır ve bu bedeli göze almadan yapılan seçim sonradan sürpriz doğurabilir. Örneğin küçük bir sınıf içi ödevde yerel ortam en makul seçenek olabilir. Fakat mikroservislerden oluşan bir kurumsal uygulamada konteyner kullanmamak ciddi bakım zorluğu doğurabilir. Dolayısıyla geliştirme ortamı türü seçimi, teknik olduğu kadar organizasyonel bir karardır.

[1] Beaulieu-Jones, B. K., & Greene, C. S. (2017). Reproducibility of computational workflows is automated using continuous analysis. *Nature Biotechnology*, *35*(4), 342. https://doi.org/10.1038/nbt.3780

1.7.1 Yerel geliştirme ortamı

Yerel geliştirme ortamı, projenin geliştiricinin kendi bilgisayarında çalıştırılmasıdır. Öğrenme, hızlı deneme ve küçük projeler için oldukça uygundur. Ancak ekip büyüdüğünde her bilgisayarın farklı işletim sistemi, farklı paket sürümü ve farklı ayar taşıması mümkündür. Bu yüzden yerel ortam kullanılıyorsa kurulum adımları açık yazılmalı, beklenen sürümler belirtilmeli ve "bende çalışıyor" durumunun ekip standardı sayılmaması gerektiği unutulmamalıdır.

1.7.2 Sanal makine ve konteyner tabanlı ortamlar

Sanal makine ve konteyner tabanlı ortamları, proje için daha kontrollü bir çalışma odası hazırlamak gibi düşünebiliriz. Sanal makine, işletim sistemi dahil daha geniş bir ortamı taklit eder. Konteyner ise çoğu durumda uygulamanın çalışması için gereken bağımlılıkları daha hafif ve taşınabilir bir paket içinde düzenlemeye yarar.

Bu ayrımın teknik ayrıntıları Kısım 7 içinde ele alınacak. Şimdilik pratik sonuç önemlidir: Bu tür ortamlar ekipteki farklı bilgisayarların etkisini azaltabilir. Ancak tamamen bedelsiz değildir. Kurulum dosyalarının güncel tutulması, ekip üyelerinin temel kavramları öğrenmesi ve konteyner ile gerçek üretim ortamı arasındaki farkların doğru yönetilmesi gerekir.

1.7.3 Uzaktan ve bulut tabanlı geliştirme ortamları

Uzaktan ve bulut tabanlı geliştirme ortamları, geliştirme kaynaklarının geliştiricinin kişisel bilgisayarı yerine internet üzerinden erişilen bir ortamda bulunmasıdır. Bu yaklaşım zayıf bilgisayarla çalışma, uzaktan ekip yönetimi veya hızlı deneme gibi durumlarda avantaj sağlayabilir. Ancak erişim yetkileri, veri gizliliği, maliyet takibi ve hizmet kesintisi gibi konular dikkatle değerlendirilmelidir. Bulut kolaylık sağlar; fakat kontrol ve sorumluluk sorularını ortadan kaldırmaz.

1.8 Kitabın Kapsamı ve Sınırı

Bir alana yeni girerken en zor şeylerden biri neyi ne kadar öğrenmek gerektiğini kestirmektir. Geliştirme ortamları da böyledir. Terminali öğrenmeye başladığınızda işletim sistemi ayrıntıları karşınıza çıkar; Git'e geçtiğinizde dallanma modelleri, Docker'a geçtiğinizde ağ ve dosya sistemi konuları görünür olur. Her kapı başka bir kapı açar.

Ancak bu kitabın amacı bütün kapıların arkasına sonuna kadar girmek değildir. Burada hedef, Yönetim Bilişim Sistemleri gibi ürün geliştirme odaklı alanlarda, bu süreçte kazanılması gereken geliştirme pratiğini anlayacak, teknik ekiplerle konuşacak, proje risklerini görecek ve temel uygulamaları deneyebilecek düzeyde bir zemin kazandırmaktır. Bilgisayar bilimi derinliği, gerektiğinde saygı duyulacak bir alandır; fakat bu kitabın merkezinde değildir.

Bu sebeple kapsamımızı üç ilkeyle sınırlayacağız: Gerektiği kadar teknik ayrıntı, her teknik ayrıntının yanında iş süreci karşılığı ve her aracın avantajıyla birlikte sınırı. Böylece metin ne yüzeysel bir araç kataloğuna dönüşecek ne de okuyucuyu alanın bütün ayrıntıları altında bırakacaktır.

Bu kitapta çekirdek (kernel), süreç (process), bağlantı noktası (port), dosya izni veya konteyner gibi kavramlarla karşılaşacağız. Fakat bunları teorik bilgisayar bilimi konusu olarak değil, geliştirme pratiğinde karşımıza çıkan karar ve hata noktaları olarak ele alacağız. Örneğin port kavramını ağ programlamanın bütün ayrıntılarıyla değil, yerel uygulamanın neden `localhost:3000` adresinde çalıştığını ve port çakışmasının projeyi nasıl durdurduğunu anlayacak kadar inceleyeceğiz.

Araç ezberi kısa vadede güven verici görünebilir. Hangi komutun ne işe yaradığını bilmek elbette önemlidir. Ancak çalışma düzeni kurulmadığında komut bilgisi dağınık kalır. Bu kitapta bir komutu yalnızca yazıp geçmeyeceğiz; o komutun hangi süreçte kullanıldığını, çıktısının nasıl okunacağını ve ekip içinde hangi kararı desteklediğini de konuşacağız.

Bir sürücünün, otomobil tamircisi kadar motor bilmesi gerekmez; fakat aracın hangi uyarı ışığında durması gerektiğini, bakımın neden geciktirilmemesi gerektiğini ve küçük bir arızanın ne zaman güvenlik riskine dönüşeceğini anlaması gerekir. Geliştirme ortamları için de benzer bir durum vardır. Geliştirici, her aracın iç mekanizmasını sonuna kadar bilmek zorunda değildir; fakat doğru soruları sorabilecek kadar temel bilgiye sahip olmalıdır.

Bu temel bilgi, “hangi sürüm kullanılıyor?”, “bu kurulum başkası tarafından tekrar edilebilir mi?”, “bu hata üretim ortamında neye yol açar?”, “bu karar ekibe ne kadar öğrenme maliyeti getirir?” gibi soruları mümkün kılar. Kitabın sınırı bu soruları sorabilecek zemini kurmaktır.

1.9 Kitabın Akış Planı

Kitap, bir geliştirme ortamını sıfırdan kurup olgunlaştıran bir iş akışı gibi ilerleyecek. Önce bilgisayarımızdaki temel çalışma zeminiyle başlayacağız: işletim sistemi, terminal, dosya düzeni ve komut çıktısını okuma alışkanlığı. Çünkü geliştirme ortamını anlamak için önce nerede durduğumuzu ve hangi komutun neyi değiştirdiğini görebilmemiz gerekir.

Ardından editör, programlama dili, runtime ve paket yönetimi tarafına geçeceğiz. Araç kurulumunun yanında proje bağımlılıklarının nasıl izlendiğini, sürüm farklarının nasıl risk ürettiğini ve bir uygulamanın yerel ortamda nasıl güvenilir biçimde çalıştırıldığını inceleyeceğiz. Daha sonra Git ve takım akışıyla kodun ekip içinde nasıl paylaşıldığını ele alacağız.

Sonraki aşamada konteyner, Docker Compose, servisler, ortam değişkenleri, CI/CD ve dağıtım konularına ilerleyeceğiz. Bu konular uygulamanın yalnızca geliştirici bilgisayarında değil, test ve üretim ortamlarında da öngörülebilir biçimde çalışmasını hedefler. En son güvenlik, gizli bilgiler, bakım, loglama ve sorun giderme başlıklarıyla ortamın sürdürülebilir tarafına bakacağız.

Bölüm 2

İşletim Sistemi ve Bileşenleri

Birçok okuyucu için terminal ekranı ilk bakışta biraz yabancı görünür. Siyah bir pencere, yanıp sönen bir imleç ve birkaç kısa komut… Grafik arayüzlere alışmış biri için bu ortam, bilgisayarın daha teknik ve daha az davetkâr bir tarafı gibi hissedilebilir. Bu duygu oldukça anlaşılırdır.

Ancak geliştirme ve sunucu dünyasında terminal yalnızca "zor görünen" bir ekran değildir. Terminal, farklı işletim sistemlerinde, uzak sunucularda, konteyner içinde çalışan servislerde ve otomasyon süreçlerinde ortak bir çalışma dili sağlar. Geliştiriciler ve sistem yöneticileri çeşitli sebeplerden dolayı bir uygulamanın hangi klasörde çalıştığını görmek, bir servisin açık olup olmadığını anlamak, bir dosyanın izinlerini kontrol etmek veya hata çıktısını okumak için çoğu zaman terminal kullanırlar.

Bu bölümde terminali komut ezberi olarak ele almayacağız. Her komutu bir iş ihtiyacına bağlayacağız: Nerede olduğumuzu görmek, dosyaları listelemek, çalışan süreci anlamak, bir portun kullanılıp kullanılmadığını kontrol etmek, ortam değişkenlerini okumak veya uzak bir sunucuya bağlanmak. Böylece komutlar tek tek ezberlenen parçalar olmaktan çıkıp geliştirme ortamını okuyabilmenin araçlarına dönüşecek.

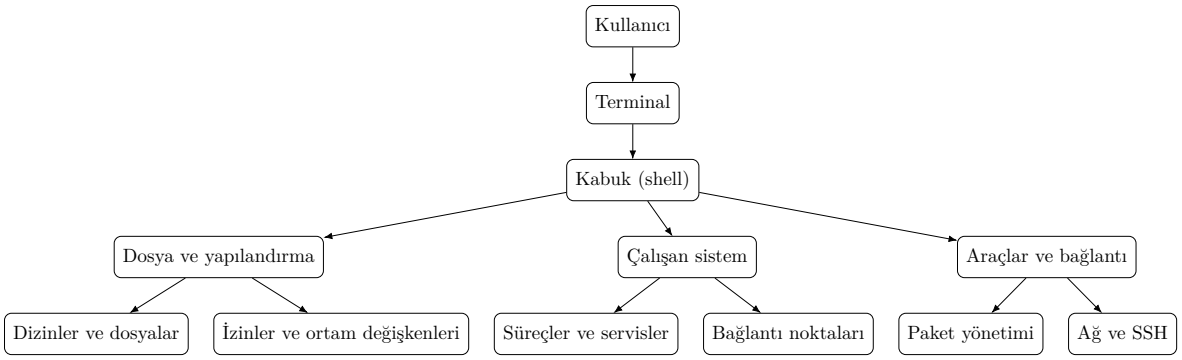
Özellikle Windows kullanıcıları için Unix-benzeri çalışma düzenine geçişte bazı kavramlar karışabilir. Windows Terminal, PowerShell, WSL, Ubuntu, Linux ve Bash aynı şey değildir. Benzer ekranda görünebilirler; fakat farklı katmanlarda çalışırlar. Bu bölümün önemli amaçlarından biri de bu katmanları sakin biçimde ayırmaktır.

Bu yolculukta tek yönlü bir sıra izlemeyeceğiz. Kullanıcı terminal üzerinden kabukla etkileşime girer; kabuk da verilen komuta göre işletim sisteminin farklı bileşenlerine ulaşır. Kullanıcıdan işletim sistemine uzanan bu zincir Şekil 2.1 ile özetlenmiştir.

Burada dikkat çeken nokta, terminalin tek başına bir hedef olmamasıdır. Terminal, geliştirme ortamının dosya, süreç, servis, izin, paket ve bağlantı tarafını anlayabilmek için kullandığımız bir geçittir.

2.1 İşletim Sistemi Nedir?

Bir uygulama geliştirdiğinizi düşünün. Kodunuz doğru görünüyor, gerekli paketleri kurduğunuzu sanıyorsunuz, fakat uygulama bir bilgisayarda çalışırken başka bir bil-



Şekil 2.1: Terminal üzerinden işletim sistemi bileşenleriyle etkileşim

gisayarda hata veriyor. Bu durumda sorun bazen kodda değil, kodun çalıştığı zeminde olabilir. Dosya yolları farklıdır, izin davranışı farklıdır, terminal komutları farklı çalışır veya kullanılan sistem kütüphanesi değişiktir.

Peki bu zemin nedir? Bu noktada işletim sistemi (operating system) kavramı devreye girer. İşletim sistemi, bilgisayarın donanımı ile uygulamalar arasında çalışan temel yönetim katmanıdır. Dosyaları saklar, programların çalışmasını düzenler, bellek ve işlemci gibi kaynakları paylaştırır, kullanıcıların sisteme erişimini yönetir ve uygulamaların cihazla konuşabilmesini sağlar.

Gündelik bir benzetmeyle düşünelim. Bir okul binasında sınıflar, kapılar, güvenlik görevlileri, panolar ve idari kurallar vardır. Binayı kullanan herkes bu düzen içinde hareket eder. İşletim sistemi de bilgisayarda benzer bir düzen kurar. Uygulamalar doğrudan diske, belleğe veya ağa istedikleri gibi davranmaz; işletim sisteminin sunduğu kurallar ve imkanlar üzerinden çalışır.

Bu kitapta işletim sistemini bütün akademik ayrıntılarıyla incelemeyeceğiz. Geliştirme ortamı açısından yanıt aradığımız soru şudur: Uygulamanın çalıştığı işletim sistemi, dosya yollarını, komutları, izinleri, paketleri ve sunucu davranışını nasıl etkiler? Özellikle Windows kullanan okuyucular için Unix-benzeri çalışma düzenini anlamak, Kısım 7 ve Kısım 8 içinde terminal, Docker, sunucu ve dağıtım konularını daha rahat takip etmeyi sağlar.

2.1.1 Donanım, işletim sistemi ve uygulama ilişkisi

Bir proje ekibi küçük bir raporlama uygulaması geliştiriyor olsun. Uygulama, dosyadan veri okuyor, ekrana tablo basıyor ve gerektiğinde veritabanına bağlanıyor. Bu işlemler bize uygulama davranışı gibi görünür; fakat arka planda donanım, işletim sistemi ve uygulama birlikte çalışır. Disk dosyayı saklar, ağ kartı bağlantıyı taşır, işletim sistemi bu kaynaklara erişimi düzenler, uygulama ise bu imkanları kullanarak işi gerçekleştirir.

Bu ilişkiyi anlamak özellikle hata ayıklarken önemlidir. Uygulama “dosyayı bulamadım” diyorsa bu çıktı tek başına sorunun kaynağını söylemez. Dosya yanlış konumda olabilir, kod hatalı bir yol oluşturabilir, süreci çalıştıran kullanıcıda okuma izni bulunmayabilir veya kullanılan yol gösterimi o işletim sisteminde geçerli olmayabilir. Ekip önce hatanın hangi katmanda oluştuğunu ayırmalıdır. Aksi halde sorunu çözmek yerine tahmin yürütmeye başlar. Bu yüzden donanım, işletim sistemi ve uygulama ilişkisini doğru okumak

ekibe zaman kazandırır ve gereksiz riski önler.

2.1.2 Kernel kavramı

Bir binada herkesin doğrudan elektrik panosuna, su vanalarına veya güvenlik sistemine müdahale ettiğini düşünün. Böyle bir düzen hızlı gibi görünse de çok risklidir. Bu yüzden kritik kaynaklara erişim belirli kurallar üzerinden yapılır. Bilgisayarda da uygulamaların donanıma ve temel sistem kaynaklarına doğrudan, sınırsız ve kontrolsüz biçimde erişmesi istenmez.

Bu düzeni sağlayan çekirdek katmana kernel adı verilir. Kernel, işletim sisteminin en temel parçasıdır; işlemci, bellek, disk, ağ ve cihazlar gibi kaynakların yönetiminde merkezi rol oynar. Bizim bu kitapta kernel ayrıntılarına derinlemesine girmemize gerek yok. Fakat Linux kernel, Ubuntu dağıtımı veya WSL gibi kavramlarla karşılaştığımızda, kernelin sistemin en alt ve kritik yönetim katmanı olduğunu bilmek yeterli bir başlangıç sağlar.

2.1.3 Kullanıcı ve Sistem Alanları

Gündelik kullanımda biz genellikle uygulamalarla çalışırız: Editörü açarız, tarayıcıyı kullanırız, terminalde komut yazarız. Bunlar çoğu zaman kullanıcı alanında (user space) gerçekleşen işlemlerdir. Kullanıcı alanı, uygulamaların ve kullanıcıya yakın araçların çalıştığı katman olarak düşünülebilir.

Buna karşılık sistem alanı (kernel space), işletim sisteminin daha kritik kaynakları yönettiği alandır. Uygulamalar bu alana doğrudan ve istedikleri gibi müdahale etmez; işletim sisteminin sunduğu mekanizmalar üzerinden istekte bulunur. Bu ayrımın pratik anlamı şudur: Bir program normal kullanıcı yetkisiyle çalışırken yapamadığı bazı işlemleri yönetici yetkisiyle yapabilir. Ancak bu yetki genişledikçe hata yapmanın bedeli de büyür. Bu yüzden 2.9.2 numaralı alt başlıkta `sudo` gibi komutları ele alırken, meselenin yalnızca "izin verildi" değil, "sorumluluk arttı" olduğunu da göreceğiz.

2.1.4 İşletim sisteminin geliştirici için önemi

Geliştirici için işletim sistemi, çoğu zaman fark edilmeden çalışan ama her işi etkileyen bir zemindir. Dosya yolları, komutların adı, paket kurulum biçimi, izin davranışı, servislerin arka planda çalışma şekli ve ağ portlarının kullanımı işletim sistemiyle doğrudan ilişkilidir. Bu yüzden bir proje Windows'ta, macOS'ta, Linux'ta veya WSL içinde küçük farklarla davranabilir.

Örneğin ekipte bir kişi dosya yolunu `C:\projeler\stok` şeklinde düşünürken, başka biri `/home/akadal/stok` yolunda çalışıyor olabilir. Bu yalnızca yazım farkı değildir; proje dokümantasyonunun, kurulum komutlarının ve hata mesajlarının nasıl okunacağını etkiler. İşletim sistemini geliştirici açısından önemli kılan şey de budur: Kodun çalıştığı zemini anlamadan, geliştirme ortamını güvenilir biçimde yönetmek zorlaşır.

Git, Docker ve çoğu paket yöneticisi bu farkı azaltmak için tasarlanmıştır; aynı komutlar farklı makinelerde benzer iş görür. Bu bir avantajdır. Dezavantajı ise farkın tamamen kaybolduğu yanılgısıdır. Satır sonu, büyük/küçük harf, yol ayracı veya izin modeli

hâlâ işletim sistemine bağlı kalabilir. Platformdan bağımsız araç, ortamı düşünmeyi bırakmayı değil, farkı daha geç ve daha pahalı görmeyi önlemeyi sağlar.

2.2 İşletim Sistemi Aileleri

Bilgisayar kullanırken çoğu zaman işletim sistemini bir tercih meselesi olarak görürüz. Kimi Windows kullanır, kimi macOS, kimi Linux. Günlük kullanımda bu tercih daha çok alışkanlık, cihaz, tasarım veya uygulama desteği üzerinden konuşulur. Ancak geliştirme ortamı açısından işletim sistemi ailesi, komutların, dosya yollarının, izinlerin ve sunucuya yakın çalışma biçiminin nasıl olacağını etkiler.

Bu kitapta tüm komutlar ve işlemler Unix ve Unix-benzeri sistemler özelinde anlatılmaktadır. Peki neden bu kitapta Unix-benzeri sistemlere özel önem veriyoruz? Çünkü modern sunucu dünyasında Linux çok yaygındır; macOS da Unix temeli üzerinde çalışır; Windows kullanıcıları ise WSL sayesinde Linux ortamına oldukça yakın bir geliştirme deneyimi elde edebilir. Bu durum, farklı işletim sistemleri kullanan okurların ortak bir komut ve çalışma dili kurmasını mümkün hale getirir.

Tablo 2.1, işletim sistemi ailelerine kuşbakışı bir başlangıç sunar.

Tablo 2.1: İşletim sistemi aileleri

Sistem	Günlük kullanımdaki yeri
Windows	Masaüstü, ofis uygulamaları ve geniş kullanıcı alışkanlığı
macOS	Unix temeli ve güçlü masaüstü deneyimi
Linux	Sunucu ve geliştirme dünyasında yaygın kullanım
WSL	Windows içinde Unix-benzeri geliştirme köprüsü

Tablo 2.1 uzman okurlar için tartışmaya açık olabilir. Mesela Linux dağıtımları da gündelik işler için işlevseldir. Ancak tarif edilen özellikler bu işletim sistemlerinin genellikle tercih edilme sebeplerini göstermektedir. Nihayetinde bir kullanıcı herhangi bir işletim sistemini seçerek tablodaki tüm faaliyetleri seçtiği işletim sistemiyle de gerçekleştirebilir.

Windows, macOS ve Linux geliştirme dünyasında farklı güçlü taraflarla karşımıza çıkar. Windows birçok okuyucunun alışık olduğu masaüstü ortamıdır; macOS Unix temeliyle terminal tarafında geliştiriciye rahatlık sağlar; Linux ise özellikle sunucu, konteyner ve bulut ortamlarında çok yaygın kullanılır. Bu sistemleri marka düzeyinde değil, proje sürecine etkileri üzerinden okuyacağız. Hangi sistemde çalışılırsa çalışılsın, ekip kurulum adımlarını ve ortam farklarını açıkça yönetebilmelidir.

Unix ve Unix-benzeri sistemler

Unix'in hikâyesi 1969'da Bell Laboratuvarlarında başladı.[1] Ken Thompson ve Dennis Ritchie'nin de aralarında bulunduğu küçük bir ekip, dönemin büyük ve karmaşık sistemlerine karşı daha sade bir işletim sistemi geliştirdi. Dosyaları ortak bir düzen içinde ele alma, küçük araçların tek işi iyi yapması ve bu araçların doğrudan birbirine bağlanması

[1] Ritchie, D. M. (1984). The evolution of the UNIX time-sharing system. *AT&T Bell Laboratories Technical Journal*, *63*(8), 1577–1593. https://doi.org/10.1002/j.1538-7305.1984.tb00054.x

gibi fikirler burada biçimlendi. Sistemin büyük bölümünün daha sonra C diliyle yeniden yazılması, Unix'in farklı bilgisayarlara taşınmasını kolaylaştırdı. Böylece başlangıçta kurum içindeki bir deney olan çalışma, üniversitelere ve araştırma merkezlerine yayılan etkili bir işletim sistemi ailesine dönüştü.

Bu yayılma tek bir çizgide ilerlemedi. Unix'ten türeyen sistemler, ticari ürünler ve aynı ilkeleri yeniden uygulayan bağımsız projeler zamanla farklı yönlere ayrıldı. Bu nedenle "Unix-benzeri" ifadesi, Unix'in dosya sistemi, süreç, kullanıcı ve izin modeli ile kabuk araçlarına benzeyen; ancak aynı kaynak kodundan gelmesi ya da resmî UNIX sertifikası taşıması gerekmeyen sistemleri anlatır. Linux bu grubun en bilinen örneklerinden biridir. Ortak kavramlar sayesinde bir sistemde kazanılan terminal alışkanlıklarının önemli bir bölümü diğerine taşınabilir.

Yine de UNIX ile Unix-benzeri kavramları eş anlamlı değildir. Büyük harfle yazılan UNIX, bugün Single UNIX Specification ile tanımlanan ve uygunluğu belgelendirilen sistemler için kullanılan bir markadır. Unix-benzeri sistemler ise aynı düşünme biçimini paylaşsa da paket yöneticileri, sistem klasörleri ve bazı komut seçenekleri bakımından ayrılabilir. Bu akrabalık, tam uyumluluk anlamına gelmez.

2.2.1 Linux kernel ve dağıtımlar

Linux denildiğinde aslında tek bir ürün değil, bir ekosistemden söz ederiz. Bunu bir eğitim kurumu gibi düşünebilirsiniz: Ortada temel kurallar ve altyapı vardır; fakat farklı okullar bu altyapıyı kendi düzenleri, araçları ve paketleriyle sunabilir. Linux kernel, bu ekosistemin çekirdek katmanıdır. Dağıtım (distribution) ise kernelin etrafına sistem araçları, paket yöneticisi, varsayılan ayarlar ve kullanıcı deneyimi ekleyen bütündür.

Ubuntu, bu dağıtımlardan biridir ve özellikle eğitim, sunucu ve WSL kullanımında sık karşımıza çıkar. Bu kitapta Ubuntu'yu tercih edilebilir bir ortak zemin olarak ele almamızın sebebi budur. Böylece Linux dünyasına düzenli ve yaygın kullanılan bir kapıdan girmeyi sağlayabiliriz.

2.2.2 Sunucularda Linux'un yaygın kullanılma nedenleri

Bir web uygulaması geliştirdiğinizde, uygulama en sonunda çoğu zaman bir sunucuda çalışır. Bu sunucu fiziksel bir makine, bulut sağlayıcısındaki sanal makine veya konteyner altyapısı olabilir. Bu ortamlarda Linux'un yaygın kullanılmasının sebepleri arasında kararlılık, güçlü komut satırı araçları, paket yönetimi, otomasyon kolaylığı ve geniş topluluk desteği bulunur.

Bu durum geliştirici için şu açıdan değerlidir: Kendi bilgisayarında Windows kullanıyor olsa bile, uygulamanın yayınlanacağı ortam Linux olabilir. Dolayısıyla terminal, dosya izinleri, servisler ve paket yönetimi gibi konuları Linux/Unix-benzeri bir zeminde öğrenmek, ekibi projenin taşınacağı gerçek sunucu ortamına önceden hazırlar.

2.3 Kullanılacak Unix-Benzeri Çalışma Düzeni

Bir sınıfta herkesin farklı bilgisayar kullandığını düşünelim. Bazı katılımcılar Windows, bazıları macOS, bazıları Linux kullanıyor olabilir. Eğer herkes yalnızca kendi işletim

sistemine özgü komutlarla ilerlerse ortak ders dili hızla dağılır. Birinin çalıştırdığı komut diğerinde çalışmaz, birinin klasör yolu diğerinde anlamlı olmaz, hata çıktıları karşılaştırılamaz.

Bu sebeple kitap boyunca Unix-benzeri bir çalışma düzenini temel alacağız. Windows kullanıcıları için bu düzenin en pratik yolu WSL üzerinden Ubuntu kullanmaktır. macOS ve Linux kullanıcıları ise kendi terminal ortamlarında büyük ölçüde benzer komutlarla ilerleyebilir. Böylece herkes aynı bilgisayarı kullanmasa bile, dosya, terminal, paket ve servis konularında ortak bir zemin oluşur.

Başlangıç için sık kullanılan çalışma yolları Tablo 2.2 içinde gösterilmiştir.

Tablo 2.2: Kitapta izlenecek çalışma yolu

İşletim sistemi	Önerilen çalışma yolu
Windows	PowerShell veya Komut İstemi üzerinden WSL ve Ubuntu
macOS	Sistemle gelen Terminal uygulaması
Linux	Dağıtımın varsayılan terminali
Uzak sunucu	SSH ile Linux kabuğu; kitaptaki komutlar orada da geçerlidir

Windows kullanıcıları için WSL, yani Linux için Windows Alt Sistemi (Windows Subsystem for Linux), Windows içinde Linux komutlarını ve araçlarını kullanmayı mümkün kılar. Örneğin Ubuntu'yu WSL üzerinden açtığınızda `/home` dizinini, `apt` paket yöneticisini, Bash komutlarını ve Linux'a yakın dosya/izin davranışını deneyimleyebilirsiniz. Pratik sınır şudur: WSL güçlü bir köprü sağlar; fakat Windows dosya sistemiyle Linux dosya sistemi arasındaki farklar, özellikle dosya yolu ve performans konusunda dikkat ister.

macOS ve Linux kullanıcıları ders örneklerinin büyük bölümünü doğrudan terminal üzerinden deneyebilir. Terminali, bilgisayardaki dosya ve programlarla metin tabanlı konuşma ekranı gibi düşünebiliriz. Bir klasöre gitmek, dosyaları görmek, paket kurmak veya çalışan servisi kontrol etmek için grafik arayüz yerine komutlardan faydalanırız.

Ancak macOS ve Linux tamamen aynı değildir. macOS çoğu zaman `zsh` ile gelir ve sistem paketleri için Homebrew ve MacPorts gibi araçlar kullanılır. Linux tarafında ise dağıtıma göre `apt`, `dnf`, `pacman` gibi paket yöneticileri karşımıza çıkabilir. Bu ayrım gözünüzü korkutmasın; bu kitapta komutlar ve işlevlerini ortak zeminde ele alıp sistemden sisteme değişen yerleri ayrıca belirteceğiz.

Ubuntu'nun kitap için tercih edilebilir olmasının temel nedeni, yaygın, iyi belgelenmiş ve yeni başlayanlar açısından erişilebilir bir Linux dağıtımı olmasıdır. WSL içinde kolay kurulabilir, sunucu dünyasında sık karşımıza çıkar ve `apt` gibi öğrenmesi görece sade bir paket yöneticisi sunar. Bu özellikler, sınıf içinde aynı komutların benzer çıktılar üretmesini kolaylaştırır.

2.3.1 WSL Kurulumu

WSL komutları PowerShell veya Komut İstemi üzerinden çalıştırılır. İlk kurulum için PowerShell'i yönetici yetkisiyle açıp şu komutu vermek yeterlidir:

```
PS> wsl --install
```

Komut gerekli Windows bileşenlerini etkinleştirir ve varsayılan olarak Ubuntu'yu kurar. İşlem tamamlandığında bilgisayarı yeniden başlatmak gerekebilir. Ubuntu ilk açılışta Linux ortamı için bir kullanıcı adı ve parola ister; bu bilgiler Windows hesabından bağımsızdır.

Kurulabilecek dağıtımları görmek ve Ubuntu'yu açıkça seçerek kurmak için şu iki komut kullanılabilir:

```
PS> wsl --list --online
PS> wsl --install -d Ubuntu
```

Birden fazla dağıtım veya Docker Desktop kurulduğunda WSL listesinde birden çok ad görülebilir. Kurulu dağıtımların çalışma durumunu ve WSL sürümünü okumak için `wsl --list --verbose`, genel yapılandırmayı görmek için `wsl --status` kullanılır:

```
PS> wsl --list --verbose
PS> wsl --status
```

Yalnızca `wsl` yazıldığında hangi dağıtımın açılacağını varsayılan dağıtım belirler. Aşağıdaki komut Ubuntu'yu varsayılan yapar; ikinci komut ise yeni kurulumlarda WSL 2 kullanılmasını sağlar:

```
PS> wsl --set-default Ubuntu
PS> wsl --set-default-version 2
```

Varsayılan dağıtımı açmak için `wsl`, belirli bir dağıtımı açmak için `wsl -d Ubuntu`, kullanıcı ev dizininden başlatmak için `wsl ~` yazılır. WSL bileşenlerini güncellemek ve çalışan bütün WSL ortamlarını kapatmak için gereken komutlar da şunlardır:

```
PS> wsl --update
PS> wsl --shutdown
```

Bir dağıtımı WSL'den kaldırmak için `wsl --unregister Ubuntu` komutu kullanılabilir. Bu işlem dağıtımdaki dosyaları, ayarları ve kurulu yazılımları kalıcı olarak siler. Bu nedenle komut ancak dağıtım adı `wsl --list --verbose` ile doğrulandıktan ve gerekli dosyalar yedeklendikten sonra çalıştırılmalıdır.

2.3.2 Kurulum doğrulama ve ilk deneme komutları

Kurulumdan sonra ilk yapmamız gereken şey, gerçekten hangi ortamda olduğumuzu görmektir. Bunun için birkaç küçük komut yeterlidir.

```
$ uname -a
Linux DESKTOP-42 5.15.153.1-microsoft-standard-WSL2 x86_64 GNU/Linux

$ pwd
/home/akadal

$ whoami
```

```
akadal
```

Kitaptaki ekran çıktıları, komutların hangi bilgiyi ürettiğini göstermek için hazırlanmış örneklerdir. Makine adı, kullanıcı adı, sürüm, tarih, süre, dosya boyutu ve nesne kimliği gibi alanlar kullanılan sisteme göre değişebilir. Uzun kurulum ve derleme (build) kayıtlarında yalnızca okunması gereken bölüm gösterilir.

Bu çıktıda `Linux` ifadesi Unix-benzeri bir ortamda olduğumuzu, `microsoft-standard-WSL2` ifadesi bunun Windows içindeki WSL olduğunu, `pwd` çıktısı o anda `/home/akadal` dizininde bulunduğumuzu, `whoami` ise komutları hangi kullanıcıyla çalıştırdığımızı gösterir. Üç küçük komut birlikte ortam, konum ve kullanıcı bilgisini verir; sistem hakkında ilk izlenimi bu kadar az yazarak elde edebiliriz.

2.4 Grafik Arayüz (GUI) ve Komut Satırı Arayüzü (CLI)

Bilgisayarla çalışırken çoğu zaman grafik arayüzleri kullanırız. Dosyayı sürükleriz, düğmeye tıklarız, menüden seçenek seçeriz. Bu kullanım biçimi öğrenmesi kolay ve görsel olarak anlaşılırdır. Bu yüzden günlük kullanımda grafik arayüz oldukça güçlüdür.

Ancak geliştirme ortamlarında her iş grafik arayüzle yürütülmez. Bazı işlemler tekrarlanabilir olmalıdır; bazıları uzak sunucuda yapılır; bazıları otomasyon sürecinin parçasıdır; bazıları da hata çıktısını doğrudan görmeyi gerektirir. Komut satırı arayüzü (command-line interface, CLI) burada devreye girer. CLI, bilgisayara metin komutlarıyla talimat vermemizi sağlar.[2]

Bu iki yaklaşımı karşılaştırırken “hangisi daha iyi?” diye sormak yerine “hangi iş için hangisi daha anlamlı?” diye sormak gerekir.

Tablo 2.3: Grafik arayüz ile komut satırı arayüzünün karşılaştırması

Ölçüt	Grafik arayüz (GUI)	Komut satırı arayüzü (CLI)
Öğrenme eşiği	Başlangıçta daha kolaydır	İlk anda yabancı gelebilir
Tekrarlanabilirlik	Tıklama adımları yazıya dökmek zor olabilir	Komutlar belgeye ve otomasyona kolay taşınır
Uzak sunucu kullanımı	Her zaman mümkün veya verimli değildir	Sunucularda yaygın ve hafiftir
Hata çıktısı	Bazen gizlenir veya sadeleştirilir	Çıktı doğrudan okunabilir
Ekip dokümantasyonu	Ekran görüntüsüne bağımlı kalabilir	Komutlar README içine açıkça yazılabilir

[2] Westerman, S. J. (1997). Individual differences in the use of command line and menu computer interfaces. *International Journal of Human-Computer Interaction*, *9*(2), 183–198. https://doi.org/10.1207/s15327590ijhc0902_6

Tabloda CLI'nin öne çıkması grafik arayüzü değersiz kıldığı anlamına gelmez; VS Code'un dosya gezgini veya bir veritabanı yönetim panelinin görsel sorgu ekranı günlük işi gerçekten hızlandırır. Fakat sunucuya uzaktan bağlanıp bir hatayı araştırdığınızda karşınızda çoğu zaman yalnızca komut satırı olur; o an CLI bilgisi bir tercih değil, gerekliliktir.

2.4.1 Grafik arayüz nedir?

Grafik arayüz (graphical user interface, GUI), bilgisayarla görsel öğeler üzerinden etkileşim kurduğumuz kullanım biçimidir. Klasör simgeleri, düğmeler, pencereler, menüler ve formlar bunun parçasıdır. Örneğin bir dosyayı masaüstünden başka bir klasöre sürüklemek grafik arayüzle yapılan bir işlemdir.

Geliştirme pratiğinde grafik arayüz çok işe yarar. VS Code, Git istemcileri, veritabanı yönetim araçları ve bulut panelleri birçok işi görünür hale getirir. Ancak grafik arayüzde yapılan her işlem kolayca tekrar edilebilir veya otomatikleştirilebilir olmayabilir. Bu yüzden GUI, öğrenmeyi kolaylaştıran güçlü bir araçtır; fakat geliştirme ortamının tamamını tek başına taşımaz.

2.4.2 Komut satırı arayüzü nedir?

Komut satırı arayüzü (command-line interface, CLI), bilgisayara metin yazarak talimat verdiğimiz arayüzdür. Bir düğmeye tıklamak yerine komutu yazar, çalıştırır ve çıktıyı okuruz. Bu ilk anda daha mekanik görünebilir; ancak komutların yazılı olması onları belgelemeyi, paylaşmayı ve otomasyon sürecine eklemeyi kolaylaştırır.

Basit bir örnekle bakalım:

```
$ ls
README.md  package.json  src
```

Burada `ls` komutu bulunduğumuz dizindeki dosya ve klasörleri listeler. Çıktı bize üç öğe gösteriyor: `README.md`, `package.json` ve `src`. Bu küçük çıktı bile proje hakkında ipucu verir. `README.md` dokümantasyon, `package.json` proje bağımlılıkları, `src` ise çoğu zaman kaynak kod klasörü olabilir.

2.4.3 Geliştirici neden CLI kullanır?

Geliştirici CLI kullanır çünkü geliştirme sürecinde birçok işlem tekrar eder. Projeyi kurmak, bağımlılıkları yüklemek, testleri çalıştırmak, Git durumunu görmek, konteyner başlatmak veya sunucuya bağlanmak komutlarla daha açık ve paylaşılabilir hale gelir. Bir ekipte "şu üç komutu çalıştır" demek, çoğu zaman uzun ekran tarifi yapmaktan daha güvenilirdir.

Örneğin teslim tarihine yaklaşan bir projede herkesin testleri aynı komutla çalıştırması gerekir:

```
$ npm test
12 tests passed
0 tests failed
```

Bu çıktı teknik olarak basit görünür; fakat ekip açısından anlamı büyüktür. Test komutu ortaksa ve çıktı herkes tarafından okunabiliyorsa, kalite kontrol kişisel yorumdan çıkıp ortak sürece dönüşür.

2.4.4 Sunucularda neden çoğunlukla CLI kullanılır?

Sunucularda çoğunlukla CLI kullanılır çünkü sunucunun görevi genellikle kullanıcıya masaüstü deneyimi sunmak değil, belirli servisleri güvenilir ve verimli biçimde çalıştırmaktır. Grafik arayüz kaynak tüketir, uzaktan yönetimi karmaşıklaştırabilir ve çoğu sunucu işleminde gerekli değildir. Bu yüzden sunucuya bağlandığımızda genellikle SSH protokolünü kullanırız ve bu durumda karşımıza terminal gelir.

Bir uygulama üretim ortamında hata verdiğinde geliştirici veya sistem sorumlusu sunucuda log dosyasına bakmak, servisin çalışıp çalışmadığını kontrol etmek veya portun açık olup olmadığını anlamak ister. Bu işlemler CLI ile hızlı ve kayıt altına alınabilir biçimde yapılır. Dolayısıyla CLI bilmek geliştiriciye günlük bir kolaylık sağlamanın ötesine geçer; iş sürekliliğini ve hızlı sorun gidermeyi doğrudan etkiler.

2.5 Terminal, Kabuk (Shell) ve Komut Satırı Kavramları

Terminali açtığımızda çoğu zaman tek bir şeyle karşılaştığımızı sanırız. Ekranda bir pencere vardır, içine komut yazarız ve sonuç görürüz. Ancak bu ekranın arkasında birkaç farklı kavram birlikte çalışır: terminal, shell ve command line. Bunları ayırmak başlangıçta gereksiz gibi görünebilir; fakat hata ayıklarken ve farklı sistemler arasında geçiş yaparken oldukça işe yarar.

Terminal, komut yazdığımız ve çıktıyı gördüğümüz pencere veya arayüzdür. Kabuk (shell), yazdığımız komutu yorumlayan programdır. Komut satırı (command line) ise komutları metin olarak yazdığımız etkileşim biçimini ifade eder. Yani terminal sahnedir, kabuk sahnede söylenen komutu anlayan yorumlayıcıdır, komut satırı ise bu iletişim tarzıdır.

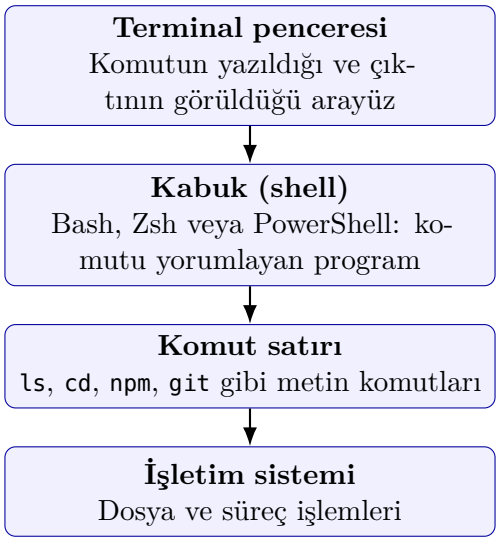
Bu katmanları Şekil 2.2 ile gösterebiliriz.

Windows kullanıcıları için bu ayrım özellikle önemlidir. Aynı Windows Terminal penceresi içinde PowerShell de açabilirsiniz, WSL Ubuntu içinde Bash de açabilirsiniz. Pencere benzer görünür; ancak komutları yorumlayan kabuk farklı olduğunda komutların davranışı da değişebilir.

2.5.1 Terminal nedir?

Terminali, bilgisayarın metin tabanlı çalışma masası gibi düşünebiliriz. Nasıl ki grafik arayüzde pencereler ve düğmeler üzerinden işlem yapıyorsak, terminalde de komutlar ve çıktılar üzerinden işlem yaparız. Terminal bize bir giriş alanı sunar; biz komutu yazarız, kabuk komutu yorumlar, sonuç yeniden terminalde görünür.

Bu ayrım pratikte şunu gösterir: Terminal yalnızca ekrandır; komutu asıl yorumlayan kabuktur. Bu yüzden aynı terminal uygulaması içinde farklı kabuk kullanıldığında



Şekil 2.2: Terminal, kabuk ve komut satırı arasındaki ilişki

davranış değişebilir. Okuyucunun "terminalim bozuldu" dediği durumda sorun bazen terminal uygulamasında değil, açılan kabuk veya çalışma dizininde olabilir.

2.5.2 Kabuk (shell) nedir?

Kabuk (shell), kullanıcının yazdığı komutları yorumlayan ve işletim sistemiyle iletişim kuran programdır. Restoranda sipariş verdiğinizi düşünün. Siz yemeğin adını söylersiniz; garson bu isteği mutfağın anlayacağı düzene taşır. Kabuk da buna benzer biçimde komutu alır, yorumlar ve işletim sistemine uygun işlem olarak iletir.

Örneğin `cd proje` yazdığınızda kabuk bunun bir dizin değiştirme isteği olduğunu anlar. `npm install` yazdığınızda ilgili programı bulmaya ve çalıştırmaya çalışır. Eğer komut bulunamazsa hata çıktısı verir. Bu yüzden kabuk, komut satırı deneyiminin merkezindedir; ancak tek başına işletim sisteminin tamamı değildir.

2.5.3 Bash, Zsh, Fish ve PowerShell

Bash, Zsh, Fish ve PowerShell farklı kabuk örnekleridir. Bash Linux ve sunucu dünyasında çok yaygındır. Zsh özellikle macOS tarafında varsayılan olarak karşımıza çıkar. Fish, etkileşimi ve tamamlama önerileriyle tanınır; bazı sözdizimi ayrıntıları Bash'ten ayrılır. PowerShell ise Windows odaklı güçlü bir komut ve otomasyon ortamıdır. Hepsi komut çalıştırmaya yarar; fakat sözdizimi, varsayılan komutlar ve çıktı davranışları farklı olabilir. Tablo 2.4 bu kabukların yaygın kullanım alanlarını ve dikkat edilmesi gereken noktaları özetler.

Tablo 2.4: Sık karşılaşılan kabuklar

Shell	Yaygın kullanım alanı	Dikkat noktası
Bash	Linux, WSL, sunucu ortamları	Ders örneklerinin çoğu buna yakın ilerler

Shell	Yaygın kullanım alanı	Dikkat noktası
Zsh	macOS terminali	Bash'e benzer; bazı ayar farkları olabilir
Fish	Kişisel geliştirme ortamları	Sözdizimi Bash ile her yerde aynı değildir
PowerShell	Windows yönetimi ve otomasyonu	Unix komutlarıyla birebir aynı düşünülmemelidir

Tek bir kabuğu mutlak doğru kabul etmek yerine hangi kabuk içinde olduğunuzu bilmek ve komut çıktısını buna göre yorumlamak gerekir.

2.5.4 Komut, argüman ve seçenek bayrağı (flag)

Komut satırında yazdığımız ifadeler genellikle üç parçadan oluşur: komut, argüman ve seçenek bayrağı (flag). Komut ne yapmak istediğimizi söyler. Argüman, bu işlemin hangi dosya, klasör veya değer üzerinde yapılacağını belirtir. Flag ise komutun nasıl davranacağını değiştirir.

Şu örneğe bakalım:

```
$ ls -la src
total 12
drwxr-xr-x  2 akadal akadal 4096 Jul  6 10:12 .
drwxr-xr-x 10 akadal akadal 4096 Jul  6 10:10 ..
-rw-r--r--  1 akadal akadal  842 Jul  6 10:12 app.js
```

Burada `ls` komuttur. `src` argümandır; yani listelemek istediğimiz dizindir. `-la` ise flag bölümüdür; daha ayrıntılı ve gizli dosyaları da içeren listeleme ister. Çıktıdaki satırlar dosya türünü, izinleri, sahibi, boyutu, tarih bilgisini ve dosya adını gösterir. İzinlerin ayrıntısına geçmeden de komutun parçaları ve ürettiği bilgi okunabilir.

2.5.5 Komut çıktısını okuyun!

Terminalde en önemli alışkanlıklardan biri, komutu yazdıktan sonra çıktıyı gerçekten okumaktır. Kullanıcı çoğu zaman komutu çalıştırır, hata görünce hemen başka komut denemeye geçer. Ancak terminal çıktısı çoğu durumda sorunun yerini, nedenini veya bir sonraki adımı gösterir.

Basit bir hata çıktısına bakalım:

```
$ cd reports
bash: cd: reports: No such file or directory
```

Bu çıktı bize "Change Directory" ifadesinin kısaltması ve çalışılan klasörü değiştirmek için kullanılan `cd` komutunun çalıştığını, fakat `reports` adında bir dizin bulunamadığını söyler. Yani sorun terminalin bozuk olması değildir; bulunduğumuz konumda böyle bir klasör yoktur veya adı farklıdır. Bu durumda daha doğru sonraki adım `ls` ile mevcut klasörleri görmek ya da `pwd` ile nerede olduğumuzu kontrol etmektir. Bu küçük örnek

bile çıktı okumanın değerini gösterir: deneme yanılmayı azaltan, küçük ama çok değerli bir çalışma alışkanlığıdır.

2.6 Dosya Sistemi

Bir projeyi terminalde çalıştırırken en sık karşılaşacağınız sorulardan biri şudur: "Ben şu anda neredeyim?" Grafik arayüzde klasörleri görerek ilerlediğimiz için bu soru çoğu zaman gözümüzün önündedir. Terminalde ise bulunduğumuz konumu, dosyaların nerede durduğunu ve hangi yolun hangi klasöre işaret ettiğini bilinçli olarak okumamız gerekir.

Unix-benzeri dosya sistemi düzeni, dosya ve dizinleri tek bir kök dizinden başlayarak ağaç biçiminde düzenler. Windows'ta `C:` ve `D:` gibi sürücü harflerine alışkın olabilirsiniz. Unix-benzeri sistemlerde ise her şey `/` ile gösterilen kök dizinden başlar. Kullanıcı dosyaları, sistem ayarları, programlar, log dosyaları ve geçici dosyalar bu ağacın farklı dallarında yer alır.

Basit bir görünüm şöyle düşünülebilir:

```
/
├── home
│   └── akadal
│       └── projects
├── etc
├── var
├── usr
└── tmp
```

Bu şema bize şunu söyler: Kullanıcının çalışma dosyaları genellikle `/home/akadal` gibi bir kullanıcı dizininde durur. Sistem ayarları `/etc`, değişken çalışma verileri ve loglar çoğu zaman `/var`, sistem programları ve kütüphaneleri ise `/usr` altında bulunabilir. Bu ayrım yalnızca teknik düzen değildir; hangi dosyanın paylaşılacağı, hangisinin sistem düzeyinde değiştirileceği ve hangisine dikkatli yaklaşılacağı konusunda da fikir verir.

Windows kullanıcıları için WSL tarafında küçük bir ek bilgi önemlidir. Linux ortamının kendi dosya sistemi vardır; Windows dosyalarına ise çoğu zaman `/mnt/c` gibi yollarla erişilir. Pratikte Linux projelerini mümkün olduğunca WSL'in kendi kullanıcı dizini içinde tutmak daha sağlıklı olabilir. Böylece izin, performans ve yol farklarından doğan bazı sorunlar azalır.

2.6.1 Kök dizin, kullanıcı dizini ve sistem dizinleri

Kök dizin `/`, dosya sisteminin başlangıç noktasıdır. Kullanıcı dizini ise çoğu zaman kendi çalışmalarınızı tuttuğunuz yerdir; örneğin `/home/akadal/projects/stok-app`. Sistem dizinleri ise işletim sisteminin ayarlarını, programlarını veya çalışma verilerini barındırır. Bir proje dosyasını kullanıcı dizininde tutmak normaldir; fakat sistem dizinlerinde gelişigüzel değişiklik yapmak riskli olabilir.

Örneğin bir ders projesi için makul çalışma konumu şöyle olabilir:

```
/home/akadal/projects/stok-app
├── README.md
├── package.json
└── src
```

Bu düzen ekip açısından anlaşılırdır. `README.md` sürüm kontrolüne girer ve paylaşılır. `src` kaynak kodu taşır. Ancak sistem ayarlarını değiştiren dosyalar proje klasörüne rastgele kopyalanmamalıdır. Hangi dosyanın projeye ait, hangisinin makineye ait olduğunu ayırmak bakım maliyetini azaltır.

2.6.2 `/etc`, `/var`, `/usr` gibi dizinlerin işlevi

Unix-benzeri sistemlerde bazı dizin adları sık karşımıza çıkar. `/etc` genellikle sistem yapılandırma dosyalarını, `/var` değişen verileri ve logları, `/usr` ise sistem programları ve paylaşılan kaynakları barındırır. Bu dizinleri işletim sisteminin çalışma düzenindeki roller olarak düşünmek faydalı olacaktır.

Örneğin bir web sunucusunun ayar dosyası `/etc/nginx/nginx.conf` altında durabilir; log çıktıları `/var/log/nginx/access.log` dosyasına yazılabilir; bazı program dosyaları `/usr/bin` altında bulunabilir. Bu dosyaların hepsi aynı türden değildir. Ayar dosyası dikkatli değiştirilmeli, log dosyası okunarak hata anlaşılmalı, program dosyası ise genellikle paket yöneticisi tarafından yönetilmelidir. Bu ayrım, "hangi dosyayı değiştirebiliriz, hangisini yalnızca okuyup yorumlamalıyız?" sorusunu doğru sormayı sağlar.

Mutlak yol ve göreli yol

Mutlak yol, dosyanın kök dizinden başlayarak tam konumunu gösterir: `/home/akadal/stok-app/README.md` gibi. Başındaki `/` işareti, tarifin o anda bulunduğumuz dizinden değil dosya sisteminin kökünden başladığını belirtir.

Göreli yol ise bulunduğumuz dizini başlangıç kabul eder. `.` mevcut dizini, `..` bir üst dizini gösterir. Örneğin `/home/akadal/projects/stok-app/src` içindeyken `./app.js`, aynı dizindeki `app.js` dosyasına gider. `../docs/kurulum.md` yolu önce `..` ile `src` dizininden `stok-app` dizinine çıkar, ardından `docs` dizinine girip `kurulum.md` dosyasına ulaşır. Her ek `../` ifadesi bir düzey daha yukarı çıkmak anlamına gelir.

Tilde (`~`) işareti kullanıcının ev dizinini temsil eder; `~/projects` yolu bu örnekte `/home/akadal/projects` ile aynı konuma karşılık gelir. Mutlak yol kesinlik sağlar, göreli yol ise proje başka bir kullanıcının dizinine taşındığında da çalışabilir. Bu yüzden proje dokümantasyonunda kişiye bağlı mutlak yollar yerine proje kökünü temel alan göreli yollar daha sağlıklıdır.

2.6.3 Gizli dosyalar ve konfigürasyon dosyaları

Unix-benzeri sistemlerde nokta ile başlayan dosyalar gizli dosya olarak kabul edilir. Örneğin `.env`, `.gitignore`, `.bashrc` veya `.config` gibi dosyalar normal listelemede görünmeyebilir. Bu dosyalar çoğu zaman ayar, kimlik bilgisi, kişisel tercih veya proje davranışı taşır. Gizli olmaları önemsiz oldukları anlamına gelmez; tam tersine bazen

en kritik bilgiler bu dosyalardadır. Örnek bir bulgu karşısında olası risk ve alınması gereken önlem Tablo 2.5 ile özetlenmiştir.

Tablo 2.5: Gizli dosyalar ve konfigürasyon dosyaları

Durum	Risk	Önlem
`.env` dosyasında veritabanı parolası var	Parola Git deposuna yanlışlıkla yüklenebilir	`.env` dosyasını Git dışında tut, `.env.example` paylaş
`.bashrc` içinde kişisel takma ad (alias) tanımlı	Komut başka bilgisayarda çalışmayabilir	Proje komutlarını kişisel ayara bağımlı yazma
`.gitignore` eksik	Geçici dosyalar veya gizli bilgiler depoya girebilir	Proje başında ignore kurallarını gözden geçir
Ortama özel ayarlar tek `.env` dosyasında karışık	Geliştirme ayarı yanlışlıkla üretim ortamına taşınabilir	Ortama özgü dosyalar kullan (`.env.development`, `.env.production`)

Tablo 2.5 gizli dosyaların yalnızca teknik ayrıntı olmadığını gösterir. Yanlış yönetilen bir konfigürasyon dosyası güvenlik, ekip uyumu ve bakım maliyeti üretir.

2.6.4 Dosya uzantıları ve işletim sistemi ilişkisi

Windows kullanıcıları dosya uzantılarına genellikle güçlü anlam yükler: `.docx`, `.xlsx`, `.exe`, `.txt` gibi. Unix-benzeri sistemlerde de uzantılar kullanılır; fakat dosyanın çalıştırılabilir olup olmaması çoğu zaman yalnızca uzantıya değil, izinlere ve dosya içeriğine de bağlıdır. Örneğin `deploy.sh` bir kabuk betiği olabilir; ancak çalıştırma izni yoksa doğrudan çalışmayabilir.

```
$ ls -l deploy.sh
-rw-r--r-- 1 akadal akadal 124 Jul  6 11:20 deploy.sh
```

Bu çıktıda dosya adı `.sh` ile bitiyor; fakat izin bölümünde çalıştırma izni olan `x` harfi görünmüyor. Dolayısıyla dosya bir betik gibi adlandırılmış olsa bile, sistem onu çalıştırılabilir olarak görmeyebilir. Bu ayrım, 2.9.1 numaralı alt başlıkta izinleri ele alırken daha anlamlı hale gelecek.

2.7 Dosya ve Dizinlerle Çalışma

Geliştirme ortamında dosyalar yalnızca bilgi saklamaz; aynı zamanda projenin nasıl çalışacağını da belirler. `package.json` bağımlılıkları anlatır, `README.md` kurulum bilgisini taşır, `.env` ortam ayarlarını içerir, `src` klasörü kaynak kodu barındırır. Bu yüzden dosya ve dizinlerle çalışmak, sadece klasör gezmek değil, projenin düzenini okumaktır.

Basit bir proje klasörü şöyle görünebilir:

```
stok-app
├── README.md
├── package.json
├── .env.example
```

```
├── src
│   └── app.js
└── docs
    └── kurulum.md
```

Bu düzen bize proje hakkında konuşma imkânı verir. Kurulum bilgisi nerede? `README.md` içinde. Kod nerede? `src` altında. Gizli bilgilerin örnek biçimi nerede? `.env.example` dosyasında. Dokümantasyon nerede? `docs` altında. Böyle bir düzen kurulmadığında dosya aramak, yanlış dosyayı düzenlemek ve bilgiyi ekip içinde kaybetmek kolaylaşır.

Ancak her düzen de kendi bakım maliyetini üretir. Bir klasör açmak kolaydır; fakat o klasörün amacı belirsizse zamanla çöplüğe dönüşebilir. Bu yüzden dosya ve dizin düzeninin sade, anlaşılır ve proje ihtiyacıyla uyumlu olması gerekir.

2.7.1 Konum öğrenme, listeleme ve dizin değiştirme

Terminalde dosyalarla çalışmanın ilk adımı konumumuzu bilmektir. `pwd` bulunduğumuz dizini gösterir, `ls` içeriği listeler, `cd` ise dizin değiştirir. Bu üç komut birlikte kullanıldığında terminalde kaybolma ihtimalimiz azalır.

```
$ pwd
/home/akadal/projects

$ ls
stok-app  crm-demo

$ cd stok-app

$ pwd
/home/akadal/projects/stok-app
```

Bu çıktıda önce `projects` klasöründe olduğumuzu görüyoruz. Sonra burada `stok-app` ve `crm-demo` dizinleri bulunduğunu anlıyoruz. `cd stok-app` ile proje klasörüne giriyoruz. Bu basit hareket, doğru proje üzerinde çalıştığımızı doğrular. Yanlış dizinde paket kurmak veya dosya silmek gibi hatalar çoğu zaman bu temel kontrol yapılmadığı için ortaya çıkar.

2.7.2 Dizin ve dosya oluşturma

Dizin ve dosya oluşturmak, proje düzenini bilinçli biçimde kurmanın ilk adımlarındandır. `mkdir` yeni dizin oluşturur, `touch` ise boş dosya oluşturmak için sık kullanılır. Oluşturulan her parçanın proje içinde neyi temsil ettiği de açık olmalıdır.

```
$ mkdir docs
$ touch docs/kurulum.md
$ ls docs
kurulum.md
```

Bu örnekte `docs` adında bir dokümantasyon klasörü oluşturduk ve içine `kurulum.md`

dosyası ekledik. Çıktı bize dosyanın gerçekten oluştuğunu gösteriyor. Böyle bir dosya ekip için kurulum bilgisini kalıcı hale getirebilir. Ancak dosya boş bırakılırsa yalnızca bir klasör düzeni olur; gerçek değer, içine anlaşılır bilgi yazıldığında oluşur.

Kopyalama için `cp`, taşıma ve yeniden adlandırma için `mv`, silme için `rm` komutları kullanılır. Bu komutlar güçlüdür; özellikle `rm` geri dönüşü zor sonuçlar doğurabilir. Örneğin `mv eski.md yeni.md` dosyayı yeniden adlandırırken, `rm yeni.md` dosyayı siler. Dosya işlemleri sürüm kontrolüyle birlikte düşünülmelidir. Bir sürüm kontrol sistemi altında izlenen dosyada hata yapılırsa geri dönüş mümkün olabilir; fakat izlenmeyen veya gizli bir dosyanın silinmesi ekipte kurulum ve güvenlik sorunlarına yol açabilir.

Şimdi küçük bir dosya işlemini komut ve çıktısıyla birlikte okuyalım.

```
$ mkdir reports
$ touch reports/july.txt
$ ls -l reports
-rw-r--r-- 1 akadal akadal 0 Jul  6 11:32 july.txt
```

İlk komut `reports` klasörünü oluşturur. İkinci komut bu klasör içinde `july.txt` adında boş bir dosya üretir. Üçüncü komut ayrıntılı listeleme yapar. Çıktıda dosya boyutunun `0` olması dosyanın henüz boş olduğunu gösterir. Dosya sahibi `akadal` olarak görünür. İzin bölümünü şimdilik tüm ayrıntısıyla okumamız gerekmiyor; ancak bu alanın dosyayı kimin okuyup yazabileceğini gösterdiğini bilmek yeterlidir.

2.8 Dosya İçeriği, Arama ve Çıktı Okuma

Bir projede dosyanın var olduğunu görmek yeterli değildir; bazen dosyanın içinde ne yazdığını, hangi satırda hata olabileceğini veya büyük bir log dosyasında aradığımız bilginin nerede geçtiğini de anlamamız gerekir. Grafik arayüzle dosya açmak mümkündür; ancak sunucu, konteyner ve büyük dosya senaryolarında içerik görüntüleme ve arama işini asıl pratik kılan yer terminaldir.

Şimdi aşağıdaki sorulara yanıt verebilmek üzere `cat`, `head`, `tail`, `less` ve `grep` gibi komutların çalışma biçimini ve örneklerini göreceğiz:

1. Dosyanın tamamına mı bakmam gerekiyor?
2. Sadece başındaki veya sonundaki bilgi yeterli mi?
3. Dosya çok büyükse sayfalı biçimde mi okumalıyım?
4. Belirli bir kelimeyi veya hata kodunu mu arıyorum?
5. Arama çıktısı bana hangi dosya, hangi satır ve hangi bağlam bilgisini veriyor?

Küçük dosyaların içeriğini görmek için `cat` komutu kullanılabilir. Örneğin bir `.env.example` dosyasına bakmak, projenin hangi ayarlara ihtiyaç duyduğunu anlamamızı sağlar.

```
$ cat .env.example
DATABASE_URL=postgres://user:password@localhost:5432/stok
APP_PORT=3000
```

Bu örnek dosya gerçek parola taşımamalıdır; yalnızca hangi alanların gerektiğini göstermelidir. `DATABASE_URL` veritabanı bağlantısını, `APP_PORT` uygulamanın çalışacağı portu

işaret eder. Bu dosya sürüm kontrolüne girebilir; fakat gerçek `.env` dosyası gizli bilgi taşıyorsa paylaşılmamalıdır.

Büyük dosyalarda `cat` kullanmak ekranı hızla doldurabilir. Bu durumda `head` dosyanın başını, `tail` sonunu, `less` ise sayfalı görünümü sağlar. Özellikle log dosyalarında son satırlar çoğu zaman en güncel hatayı gösterdiği için `tail` oldukça kullanışlıdır.

```
$ tail -n 3 app.log
2026-07-06 11:40:12 INFO Server started on port 3000
2026-07-06 11:42:03 ERROR Database connection failed
2026-07-06 11:42:03 INFO Retrying in 5 seconds
```

`tail` herhangi bir seçenek verilmediğinde dosyanın son on satırını gösterir. `-n 3` seçeneği bu sayıyı üçle sınırlar; `app.log` ise okunacak dosyadır. Burada üç satır seçmemizin nedeni son hatayı tek başına değil, hemen öncesi ve sonrasıyla birlikte görmektir: Uygulama önce başlamış, ardından veritabanı bağlantısı başarısız olmuş ve sistem beş saniye sonra yeniden deneyeceğini bildirmiştir. Gerçek bir incelemede bağlam yetersiz kalırsa sayı `-n 20` gibi daha büyük bir değere çıkarılabilir.

Dosya içinde belirli bir ifadeyi aramak için `grep` gibi araçlardan faydalanabiliriz. Örneğin log dosyasında yalnızca hata satırlarını görmek isteyebiliriz:

```
$ grep "ERROR" app.log
2026-07-06 11:42:03 ERROR Database connection failed
```

Bu çıktı bize büyük dosyanın tamamını okumadan hata satırını ayıklama imkanı verir. Ancak arama sonucu bağlamdan koparılmamalıdır. Hatanın hemen öncesi ve sonrası da önemli olabilir. Bu yüzden gerektiğinde `tail`, `less` veya daha ayrıntılı arama seçenekleriyle birlikte düşünmek gerekir.

Arama çıktısını anlamlandırmak, komutu çalıştırmaktan daha önemlidir. Bir proje klasöründe `TODO` ifadelerini aradığımızı düşünelim:

```
$ grep -R "TODO" src
src/report.js:// TODO: validate date range before query
src/user.js:// TODO: handle missing email
```

Bu çıktı iki dosyada tamamlanmamış iş olduğunu gösterir. `report.js` içinde tarih aralığı kontrolü, `user.js` içinde eksik e-posta durumunun ele alınması gerektiği yazıyor. Böyle bir arama, kod içine dağılmış küçük notları tek bakışta görünür kılar; ekip bunu teslimden önce hızlı bir kontrol listesi gibi kullanabilir.

2.9 Kullanıcılar, Gruplar ve Dosya İzinleri

Bir dosyanın bilgisayarda bulunması, herkesin o dosyayı okuyabileceği, değiştirebileceği veya çalıştırabileceği anlamına gelmez. Özellikle Unix-benzeri sistemlerde kullanıcı, grup ve izin modeli geliştirme ortamının önemli parçalarından biridir. Bu konu ilk anda teknik görünebilir; ancak yanlış izinler bir uygulamanın çalışmamasına, gizli dosyaların açığa çıkmasına veya ekipte gereksiz beklemeye yol açabilir.

Dosya izinlerini bir ofis dolabı gibi düşünebiliriz. Bazı kişiler dolabı sadece görebilir,

bazıları dosya ekleyebilir, bazıları ise belirli işlemleri yapabilir. Bilgisayarda da dosya sahibi, grup ve diğer kullanıcılar için okuma, yazma ve çalıştırma izinleri ayrı ayrı tanımlanabilir.

Örnek bir çıktı:

```
$ ls -l deploy.sh
-rwxr-x--- 1 akadal dev 124 Jul  6 12:10 deploy.sh
```

Bu satır dosyanın sahibinin `akadal`, grubunun `dev` olduğunu ve izinlerin farklı kullanıcı kümeleri için ayrı tanımlandığını gösterir. Bu bilgiyi okumak, "neden çalışmıyor?" sorusuna çoğu zaman hızlı bir başlangıç sağlar.

Windows kullanıcıları izinleri çoğu zaman "yönetici olarak çalıştır" ifadesi üzerinden tanır. Unix-benzeri sistemlerde ise izinler dosya düzeyinde daha görünür biçimde karşımıza çıkar. Bir dosyayı okuyabilir, değiştirebilir veya çalıştırabilirsiniz; fakat bu üç yetki aynı şey değildir. Bu ayrımı bilmek, özellikle WSL, Linux sunucu ve dağıtım ortamlarında karşılaşılan "permission denied" hatalarını anlamayı kolaylaştırır.

Unix-benzeri izin modelinde üç temel taraf vardır: kullanıcı (user), grup (group) ve diğerleri (others). Kullanıcı dosyanın sahibidir. Grup, belirli kullanıcıların ortak yetki kümesini temsil eder. Diğerleri ise bu ikisinin dışında kalan herkestir. Bu model ekip çalışmasında önemlidir; çünkü bir dosyayı yalnızca sahibinin değil, belirli bir geliştirici grubunun da okuyup yazması gerekebilir.

2.9.1 Okuma, yazma ve çalıştırma izinleri

Bir dosyayı görebiliyor, hatta içeriğini okuyabiliyor olmanız, onu değiştirebileceğiniz veya çalıştırabileceğiniz anlamına gelmez; Unix-benzeri sistemler bu yetkileri birbirinden ayrı tutar. Dosya izinlerinde üç temel eylem bulunur: okuma (read), yazma (write) ve çalıştırma (execute). Okuma dosyanın içeriğini görebilmeyi, yazma dosyayı değiştirebilmeyi, çalıştırma ise dosyayı program veya betik olarak çalıştırabilmeyi ifade eder.

```
$ ls -l script.sh
-rw-r--r-- 1 akadal akadal 80 Jul  6 12:15 script.sh
```

Bu çıktıda sahibin okuma ve yazma izni vardır; fakat çalıştırma izni yoktur. Dosya adı `script.sh` olsa bile doğrudan çalıştırıldığında hata alınabilir. Bu örnek bize şunu gösterir: Dosyanın adı bir niyet belirtir, izinleri ise sistemin gerçekten neye izin verdiğini gösterir.

2.9.2 İzin ve Sahiplik Değişikliklerini Güvenle Yönetmek

`chmod` izinleri, `chown` sahipliği değiştirir; `sudo` ise komutu daha yüksek yetkiyle çalıştırmak için kullanılır. Bu komutlar güçlüdür ve bu yüzden bağlamsız ezberlenmemelidir. Her biri "sistem bana izin vermedi, o halde zorlayayım" refleksiyle değil, "hangi yetki gerçekten gerekli?" sorusuyla kullanılmalıdır.

```
$ chmod +x script.sh
$ ls -l script.sh
```

```
-rwxr-xr-x 1 akadal akadal 80 Jul  6 12:15 script.sh
```

Burada `chmod +x` dosyaya çalıştırma izni ekler. Çıktıda `x` harflerinin görünmesi artık dosyanın çalıştırılabilir olduğunu gösterir. Ancak aynı mantıkla her dosyaya geniş izin vermek doğru değildir. Özellikle sunucu ve ekip ortamlarında gereğinden fazla izin, güvenlik riskine dönüşebilir.

2.9.3 Geliştirme ortamında sık görülen izin hataları

Geliştirme ortamında izin hataları çoğu zaman "permission denied" mesajıyla görünür. Örneğin bir betiği çalıştırmak istersiniz, fakat dosyada çalıştırma izni yoktur. Ya da bir klasöre dosya yazmak istersiniz, fakat klasör başka kullanıcıya aittir. Bu hatalar küçük görünse de teslim süresini uzatabilir; Tablo 2.6 sık karşılaşılan durumları, olası nedenleri ve çözüm yollarını bir arada gösterir.

Tablo 2.6: Geliştirme ortamında sık görülen izin hataları

Durum	Problem	Çözüm
`Permission denied` hatası	Betik veya dosya çalıştırılamaz	İzinleri `ls -l` ile kontrol et
Dosya `root` kullanıcısına ait	Normal kullanıcı güncelleme yapamaz	Sahiplik ihtiyacını değerlendir, gerekirse `chown` kullan
Herkese yazma izni verilmiş	Güvenlik ve yanlış değişiklik riski artar	En az gerekli izin ilkesini uygula
WSL'de Windows yolundan çalıştırma	İzin ve performans sürprizi çıkar	Projeyi Linux ev dizininde tut
`.ssh` izinleri geniş	Anahtar başkaları tarafından okunabilir	Klasör `700`, özel anahtar `600` olsun
Dizin izni yok, dosya izni var	İçeriğe ulaşılamaz	Üst dizinin arama iznini de kontrol et

Tablodaki örnekler ortak bir noktaya işaret eder: izin hatalarının çoğu dosyanın kendisinden değil, dosyanın bulunduğu bağlamdan kaynaklanır; sahiplik, üst dizin veya çalışma ortamı sorunun asıl kaynağıdır. Ekip bu tabloyu bir kontrol listesi gibi kullanarak "permission denied" gördüğünde zaman kaybetmeden doğru katmana bakabilir.

2.10 Sistem Düzeyinde Paket Yönetimi

Bir bilgisayara yeni bir araç kurmak gündelik hayatta çoğu zaman bir web sitesinden dosya indirip kurulum sihirbazını çalıştırmak anlamına gelir. Geliştirme ortamında ise özellikle Unix-benzeri sistemlerde bu iş çoğu zaman paket yöneticileri üzerinden yapılır. Paket yöneticisi, yazılımı bulur, indirir, kurar, günceller ve gerektiğinde kaldırır.

Sistem düzeyinde paket yönetimi, işletim sisteminin genel araçlarını yönetir. Örneğin `git`, `curl`, `nginx`, `postgresql` veya `python3` gibi araçlar sistem paket yöneticisiyle kurulabilir. Bu, proje bağımlılığı yönetiminden farklıdır. Bir Node.js projesinin `npm` paketleri

proje düzeyinde bağımlılıklardır; fakat `git` gibi araçlar çoğu zaman sistem düzeyinde kurulur.

Bu ayrım özellikle ekip çalışmasında önemlidir. Kurulum belgesinde "PostgreSQL kurun" yazmak yeterli olmayabilir. Hangi işletim sisteminde hangi paket yöneticisiyle, hangi sürümün kurulacağı ve kurulumdan sonra nasıl doğrulanacağı da belirtilmelidir.

Paket, kurulabilir yazılım birimidir. Paket yöneticisi, bu yazılımı kurmak ve yönetmek için kullanılan araçtır. Paket deposu ise paketlerin tutulduğu kaynaktır. Örneğin Ubuntu'da `apt` paket yöneticisi, Ubuntu depolarından paket indirir. Yazılım kurmak, kaynağa duyulan güveni, sürüm yönetimini ve bakım sorumluluğunu birlikte taşır.

2.10.1 Sistem Paket Yöneticileri

Aynı aracı farklı işletim sistemlerinde kurmak farklı komutlar gerektirebilir; bu yüzden ekibin hangi sistemde hangi paket yöneticisinin kullanıldığını bilmesi gerekir. Ubuntu tarafında `apt`, macOS tarafında ise Homebrew üzerinden gelen `brew` sık kullanılan sistem paket yöneticileridir. Örneğin Ubuntu'da `sudo apt install git`, macOS'ta `brew install git` komutuyla Git kurulabilir. Komutlar farklıdır; fakat düşünce benzerdir: Aracı güvenilir bir paket kaynağından kurmak ve gerektiğinde güncelleyebilmek. Bu yüzden dokümantasyonda işletim sistemine göre komut farklarını açık yazmak gerekir.

2.10.2 Paket kurmak, güncellemek ve kaldırmak

Paket kurmak, güncellemek ve kaldırmak sistemin çalışma ortamını değiştirir. Bu yüzden komutları çalıştırmadan önce neyi değiştirdiğimizi bilmek gerekir. Ubuntu'da basit bir kurulum ve doğrulama örneği şöyle olabilir:

```
$ sudo apt install git
...
Setting up git ...
...

$ git --version
git version 2.43.0
```

İlk komut Git'i sistem düzeyinde kurar. `sudo` kullanıldığı için işlem yönetici yetkisi ister. İkinci komut ise kurulumun gerçekten yapıldığını ve hangi sürümün çalıştığını gösterir. Kurulum belgesinde yalnızca "Git kurun" demek yerine bu doğrulama adımını da vermek, ekip içinde belirsizliği azaltır.

2.10.3 Sistem paketi ile proje bağımlılığı farkı

Kurulum belgesinde "Node.js kurun" ile "express paketini kurun" benzer adımlar gibi görünebilir; oysa ikisi farklı katmanlarda yönetilir. Sistem paketi ile proje bağımlılığı aynı şey değildir. Sistem paketi bilgisayarın genel çalışma ortamına kurulur. Proje bağımlılığı ise belirli bir projenin çalışması için gereken kütüphaneleri ifade eder. Bu ayrımı karıştırmak, kurulum belgelerinde ciddi belirsizlik üretir.

Tablo 2.7: Sistem paketi ile proje bağımlılığı farkı

Tür	Örnek	Nerede yönetilir?
Sistem paketi	`git`, `curl`, `postgresql`	İşletim sistemi paket yöneticisiyle
Proje bağımlılığı	`express`, `react`, `pytest`	Proje paket yöneticisiyle
Konfigürasyon	`.env`, `config.yml`	Proje veya ortam kurallarıyla
Çalışma zamanı (runtime)	Node.js sürümü, Python yorumlayıcısı	Sürüm yöneticisiyle (`nvm`, `pyenv` gibi)

Tablo 2.7 hangi aracın hangi düzeyde yönetileceğini gösterir. Örneğin `npm install express` proje bağımlılığıdır; `sudo apt install nodejs` ise sistem düzeyinde bir kurulumdur. Yanlış düzeyde yapılan kurulum, başka projeleri veya ekip üyelerini etkileyebilir.

2.10.4 Aynı makinede birden fazla ekosistemle çalışma riski

Aynı makinede birden fazla ekosistemle çalışmak geliştirme ortamlarında sık görülen bir durumdur. Bir bilgisayarda Python, Node.js, Java, Docker, PostgreSQL ve farklı paket yöneticileri aynı anda bulunabilir. Bu esneklik güçlüdür; ancak Tablo 2.8 ile özetlenen sürüm çakışması, yanlış komutun çalışması ve ortamın zamanla kirlenmesi gibi riskler de beraberinde gelir.

Tablo 2.8: Aynı makinede birden fazla ekosistemle çalışma riski

Bulgu	Risk	Aksiyon
Birden fazla Node.js sürümü var	Proje yanlış sürümle çalışabilir	Sürüm yöneticisi veya proje dokümantasyonu kullan
Hem sistem Python'u hem proje sanal ortamı var	Paket yanlış yere kurulabilir	Sanal ortamın aktif olduğunu doğrula
Aynı servis farklı portlarda çalışıyor	Uygulama beklenen servise bağlanmayabilir	Port ve servis listesini kontrol et
Paketler global olarak kurulmuş	Bir projede çalışan komut başka projede farklı davranabilir	Paketleri proje içine yerel kur, global kurulumdan kaçın

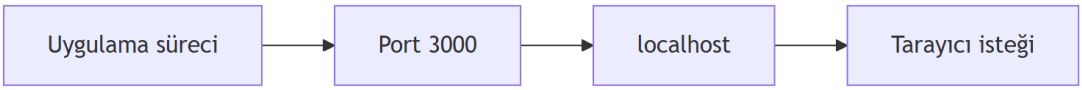
Bu başlığın temel dersi şudur: Geliştirme bilgisayarı zamanla karmaşıklaşır. Bu karmaşıklığı yok saymak yerine, sürümleri, paketleri ve servisleri görünür kılmak gerekir. Böylece teknik esneklik, proje riski haline gelmeden yönetilebilir.

2.11 Süreç (Process), Servis, Bağlantı Noktası (Port) ve Localhost

Bir web uygulamasını çalıştırdığınızda terminalde "Server started" benzeri bir çıktı görürsünüz. Tarayıcıya `http://localhost:3000` yazarsınız ve uygulama açılır. İlk bakışta bu oldukça basit görünür. Ancak arka planda çalışan bir program, dinlenen bir port, yerel makine adresi ve bazen de arka plan servisi vardır.

Bu başlıkta süreç (process), servis (service), bağlantı noktası (port) ve localhost kavramlarını birlikte ele alacağız. Çünkü geliştirme ortamında "uygulama çalışmıyor" dediğimizde aslında birkaç farklı şey bozulmuş olabilir: Program hiç başlamamış olabilir, başlamış ama yanlış portu dinliyor olabilir, port başka bir program tarafından kullanılıyor olabilir veya tarayıcı yanlış adrese gidiyor olabilir.

Süreç, bağlantı noktası ve `localhost` arasındaki bu ilişki Şekil 2.3 üzerinde sadeleştirilmiştir.



Şekil 2.3: Yerel uygulamaya erişimde süreç, port ve localhost ilişkisi

Bu ilişkide yalnızca kod yoktur; çalışan program, işletim sistemi, ağ adresi ve kullanıcı isteği birlikte davranır. Bu ilişkiyi okuyabilmek, hata ayıklamayı çok daha sistemli hale getirir.

2.11.1 Süreç (process) ve servis kavramları

Süreç (process), işletim sistemi üzerinde çalışan program örneğidir. Bir Node.js uygulamasını başlattığınızda, o uygulama bir süreç olarak çalışır. Terminali kapattığınızda süreç de durabilir. Servis ise çoğu zaman arka planda çalışan, sistem tarafından yönetilen ve gerektiğinde otomatik başlayabilen hizmettir. Örneğin PostgreSQL veritabanı bir servis olarak çalışabilir.

Küçük bir örnek düşünelim. Uygulamanız `localhost:3000` adresinde çalışıyor, veritabanınız ise arka planda PostgreSQL servisi olarak açık. Uygulama süreci kapanırsa sayfa açılmaz. Veritabanı servisi kapalıysa uygulama açılabilir ama veri çekemeyebilir. Bu yüzden "çalışıyor mu?" sorusunu sorarken hangi şeyden söz ettiğimizi netleştirmek gerekir: uygulama süreci mi, veritabanı servisi mi, yoksa ikisi arasındaki bağlantı mı?

2.11.2 Bağlantı noktası (port) dinlemek ne demektir?

Bir ofis binasında farklı birimlerin farklı dahili numaraları olduğunu düşünün. Ana binaya ulaştıktan sonra hangi birime gideceğiniz numarayla belirlenir. Bilgisayarda port fikri de buna benzer. Aynı makine üzerinde birden fazla uygulama çalışabilir; port, ağ üzerinden gelen isteğin hangi uygulamaya yönleneceğini ayırt etmeye yardım eder.

Port dinlemek, bir programın belirli bir bağlantı noktasından gelecek istekleri beklemesi anlamına gelir. Örneğin bir geliştirme sunucusu 3000 portunu dinliyorsa, tarayıcıdan

`http://localhost:3000` adresine gittiğinizde istek o programa ulaşır. Ancak aynı portu iki program aynı anda kullanamaz. Bu nedenle "port already in use" hatası aldığınızda, sorun çoğu zaman koddan önce çalışma ortamındaki çakışmadadır.[3]

2.11.3 Yerel ana makine (localhost) kavramı

Localhost, bilgisayarın kendisini işaret eden yerel ağ adıdır. Tarayıcıya `localhost` yazdığınızda internet üzerindeki uzak bir sunucuya değil, kendi makinenizde çalışan bir servise ulaşmaya çalışırsınız. Bu yüzden geliştirme ortamında `localhost` çok sık kullanılır. Henüz uygulamayı dünyaya açmadan, kendi bilgisayarınızda test etmenizi sağlar.

Ancak localhost kavramının sınırı unutulmamalıdır. Sizin bilgisayarınızda çalışan `localhost:3000`, arkadaşınızın bilgisayarında aynı uygulamanın çalıştığı anlamına gelmez. Her bilgisayarın kendi localhost'u vardır. Bu ayrım özellikle ekip içinde "bende açılıyor" türü konuşmalarda önemlidir.

`127.0.0.1`, aynı fikrin IPv4 adresidir. `localhost` ise çoğu makinede bu adrese çözülen addır. Pratikte ikisi aynı yere gider. Yine de birebir eş anlam değildir: ad çözümü `/etc/hosts` veya benzeri bir kayıttan gelir; bazı ortamlarda `localhost` IPv6 adresi olan `::1`'e düşer. Uygulama yalnızca `127.0.0.1` dinliyorsa `localhost` üzerinden gelen IPv6 isteği ulaşmayabilir. Hata ayıklarken "ad mı, adres mi, hangi protokol?" diye ayırmak bu yüzden işe yarar.

2.11.4 Çalışan süreci ve servisi tanımak

Çalışan süreçleri ve servisleri tanımak için sistemden bilgi istememiz gerekir. Örneğin bir portun kullanılıp kullanılmadığını görmek, hangi uygulamanın açık olduğunu anlamak için önemlidir.

```
$ ss -ltn
State   Recv-Q Send-Q Local Address:Port  Peer Address:Port
LISTEN  0      511    127.0.0.1:3000      0.0.0.0:*
LISTEN  0      128    127.0.0.1:5432      0.0.0.0:*
```

Bu örnek çıktıda `3000` ve `5432` portlarının dinlendiğini görüyoruz. Geliştirme dünyasında 3000 portu sıkça web uygulamaları için, 5432 portu ise PostgreSQL için karşımıza çıkabilir. Elbette bu kesin kural değildir; fakat çıktı bize hangi portların açık olduğunu ve yerel adreste dinlendiğini gösterir.

Basit servis testi, uygulamanın gerçekten cevap verip vermediğini anlamak için yapılır. Örneğin bir uygulamanın 3000 portunda çalıştığını düşünüyorsak `curl` ile küçük bir istek gönderebiliriz:

[3] TCP ve UDP port numarası 0–65535 aralığındadır. 0 çoğu sistemde "herhangi bir boş portu seç" anlamına gelir; uygulamanın bilinçli olarak seçtiği bir bağlantı noktası (kapı) değildir. 1–1023 arası geleneksel olarak ayrıcalıklı bağlantı noktalarıdır (kapılardır); 80 ve 443 gibi numaraları dinlemek genellikle yönetici yetkisi ister. Geliştirme sunucularının 3000, 8080 gibi numaraları tercih etmesinin sebebi budur.

```
$ curl http://localhost:3000/health
{"status":"ok"}
```

Bu çıktı uygulamanın sağlık kontrolü uç noktasının cevap verdiğini gösterir. Fakat hata çıktısı da en az başarı çıktısı kadar öğreticidir.

Tablo 2.9: Çalışan süreci ve servisi tanımak

Bulgu	Risk	Aksiyon
`Connection refused`	Uygulama o portta çalışmıyor olabilir	Sürecin açık olup olmadığını ve portu kontrol et
`404 Not Found`	Uygulama çalışıyor ama adres yanlış olabilir	URL yolunu ve yönlendirme (route) tanımını kontrol et
`500 Internal Server Error`	Uygulama içinde hata oluşmuştur	Log dosyasına ve ortam değişkenlerine bak

Tablo 2.9 teknik hatayı iş etkisine bağlamamıza yardım eder. Üretim ortamında bu hatalar kullanıcıya hizmet verememe, yanlış yönlendirme veya veri erişim sorunu olarak dönebilir.

2.12 İş Hattı (Pipeline) ve Yönlendirme

Terminalde güçlü olan şeylerden biri, küçük komutları bir araya getirerek daha anlamlı işler yapabilmektir. Gündelik hayatta bir raporu hazırlarken önce veriyi toplar, sonra filtreler, sonra özetler, en sonunda dosyaya kaydederiz. Terminalde pipe ve yönlendirme de buna benzer bir akış kurar.

İş hattı (pipeline), bir komutun çıktısını `|` (pipe) işaretiyle başka bir komutun girdisi haline getirir. Yönlendirme (redirection) ise çıktıyı ekrana basmak yerine dosyaya yazmak veya dosyadan girdi almak için kullanılır. Bu kavramlar ilk anda sembol ezberi gibi görünebilir; ancak asıl mesele komutları veri akışı olarak düşünebilmektir. İleride CI/CD bölümünde göreceğimiz iş hattı da aynı "çıktıyı bir sonraki adıma bağlama" fikrinin büyümüş halidir: orada adımlar komutlar değil, test, derleme ve dağıtım gibi kontrol aşamalarıdır.

Örneğin büyük bir log dosyasında yalnızca hata satırlarını bulup saymak istiyorsanız, bunu birkaç küçük komutu birleştirerek yapabilirsiniz. Böylece terminal yalnızca tek tek komut çalıştırdığınız yer olmaktan çıkar, küçük veri işleme adımlarını kurduğunuz bir çalışma alanına dönüşür. Örneğin log dosyasında hata satırlarını bulup saymak isteyelim:

```
$ grep "ERROR" app.log | wc -l
7
```

Burada `grep "ERROR" app.log` hata satırlarını bulur. `|` işareti bu çıktıyı `wc -l` komutuna aktarır. `wc -l` satır sayısını verir. Çıktıdaki `7`, log dosyasında 7 hata satırı bulunduğunu

gösterir. Bu örnek küçük görünür; fakat raporlama, hata analizi ve izleme süreçlerinde aynı düşünme biçimi çok işe yarar.

Komut çıktısını ekranda görmek her zaman yeterli değildir. Bazen çıktıyı dosyaya kaydetmek isteriz. `>` işareti dosyaya yazar ve varsa eski içeriği değiştirir. `>>` ise dosyanın sonuna ekleme yapar.

```
$ grep "ERROR" app.log > errors.txt
$ wc -l errors.txt
7 errors.txt
```

İlk komut hata satırlarını `errors.txt` dosyasına yazar. İkinci komut bu dosyada kaç satır olduğunu gösterir. Burada dikkat edilmesi gereken nokta şudur: `>` kullanmak dosyanın eski içeriğini silebilir. Eğer mevcut dosyanın sonuna ekleme yapmak istiyorsak `>>` tercih edilmelidir.

Bazı komutlar girdiyi doğrudan dosyadan alabilir. Bu, özellikle aynı işlemi tekrar tekrar yapmak istediğimizde faydalıdır. Örneğin bir liste dosyasındaki satır sayısını görmek isteyelim:

```
$ cat users.txt
akadal
mehmet
zeynep

$ wc -l < users.txt
3
```

Burada `users.txt` dosyası bir girdi kaynağıdır. Bu dosya proje verisi mi, test verisi mi, yoksa kişisel bilgi mi taşıyor? Bu soru önemlidir. Eğer dosyada gerçek kullanıcı bilgileri varsa, sürüm kontrolüne eklemek ve paylaşmak gizlilik riski doğurabilir.

Unix-benzeri çalışma düzeninin güçlü taraflarından biri, küçük komutların birlikte kullanılabilmesidir. Her komut tek başına sınırlı bir iş yapar; fakat birlikte kullanıldığında anlamlı bir işlem hattı oluşur.

```
$ grep "ERROR" app.log | sort | uniq -c
3 ERROR Database connection failed
4 ERROR Payment service timeout
```

Bu örnekte hata satırlarını buluyor, sıralıyor ve tekrar eden satırları sayıyoruz. Çıktı bize iki hata türü olduğunu ve kaç kez tekrarlandıklarını gösterir. Bu bilgi teknik ekip için hata önceliklendirmesi, yönetim tarafı için ise riskin yoğunlaştığı alanı görme açısından değerlidir.

2.13 Ortam Değişkenleri

Bir uygulama her ortamda aynı ayarlarla çalışmaz. Geliştirme ortamında yerel veritabanına bağlanabilir, test ortamında sahte servisler kullanabilir, üretim ortamında gerçek ödeme sistemiyle konuşabilir. Kod aynı kalsa bile, uygulamanın çalıştığı bağlam değişir. Bu bağlamı kodun içine sabitlemek çoğu zaman doğru değildir.

Ortam değişkenleri (environment variables), uygulamaya çalışma anında dışarıdan bilgi vermek için kullanılır. Veritabanı adresi, API[4] anahtarı, uygulama portu, çalışma modu veya gizli bilgiler bu yolla yönetilebilir. Böylece kod ile ortam ayarı birbirinden ayrılır.

Ancak ortam değişkenleri sihirli çözüm değildir. Yanlış isimlendirilmiş, eksik verilmiş veya yanlış ortamdan taşınmış bir değişken uygulamanın hatalı çalışmasına yol açabilir. Bu yüzden ortam değişkenleri hem teknik hem de operasyonel bir sorumluluktur.

Ortam değişkeni (environment variable), işletim sistemi veya çalışma ortamı tarafından programa sunulan isim-değer çiftidir. Bunu uygulamaya çalışma sırasında verilen küçük notlar gibi düşünebiliriz. Kod "hangi veritabanına bağlanacağım?" diye sorduğunda cevap bazen kod dosyasından değil, ortam değişkeninden gelir.

```
$ export APP_PORT=3000
$ echo $APP_PORT
3000
```

Bu örnekte `APP_PORT` adında bir ortam değişkeni tanımladık ve sonra değerini okuduk. Uygulama bu değeri okuyarak hangi portta çalışacağını belirleyebilir. Pratik sınır şudur: Bu değişken mevcut terminal oturumunda tanımlıdır; kalıcı hale getirmek veya proje düzeyinde yönetmek için ek düzen gerekir.

`PATH`, kabuğun komutları nerede arayacağını belirleyen özel bir ortam değişkenidir. Terminale `git` yazdığınızda kabuk bütün diski rastgele aramaz; `PATH` içinde belirtilen dizinlere bakar. Bu yüzden bir program kurulu olduğu halde terminal "command not found" diyorsa, sorun bazen programın kurulu olmaması değil, `PATH` içinde bulunmamasıdır.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/home/akadal/.local/bin
```

Bu çıktı, komutların hangi dizinlerde aranacağını iki nokta `:` ile ayrılmış biçimde gösterir. Ekip içinde "bende çalışıyor, sende çalışmıyor" türü sorunların bir kısmı `PATH` farklarından kaynaklanabilir. Bu nedenle `PATH` ayarını doğru kurmak, kurulumun ekipte tekrarlanabilir olmasını doğrudan etkiler.

Kabuk konfigürasyon dosyaları, terminal açıldığında uygulanacak ayarları taşır. Bash için `.bashrc`, Zsh için `.zshrc` gibi dosyalar sık karşımıza çıkar. Bu dosyalarda alias, PATH ekleme, ortam değişkeni veya kişisel terminal tercihleri bulunabilir.

```
export PATH="$HOME/.local/bin:$PATH"
alias ll="ls -la"
```

Bu satırlar kullanıcının kendi kabuk davranışını değiştirir. Ancak proje dokümantasyonunda kişisel alias'lara güvenmek doğru değildir. Örneğin `ll` sizin bilgisayarınızda çalışırken başka bir ekip üyesinde çalışmayabilir. Bu yüzden proje komutları kişisel kabuk ayarlarına bağımlı yazılmamalıdır.

`.env` dosyaları, proje için gerekli ortam değişkenlerini dosya biçiminde tutmak için kul-

[4] API (application programming interface), yazılımların birbirine belirli kurallarla istek gönderip yanıt almasını sağlayan arayüzdür. Web dünyasında bu konuşma çoğu zaman HTTP üzerinden, JSON gibi bir veri biçimiyle yapılır. Ancak API yalnızca JSON veya yalnızca web demek değildir; önemli olan sözleşmenin yazılı ve paylaşılabilir olmasıdır.

lanılır. Özellikle geliştirme ortamında veritabanı adresi, port, API anahtarı gibi bilgileri koddan ayırmaya yarar.

```
APP_PORT=3000
DATABASE_URL=postgres://user:password@localhost:5432/stok
NODE_ENV=development
```

Burada önemli bir güvenlik ayrımı vardır. Gerçek `.env` dosyası parola veya anahtar taşıyorsa Git deposuna eklenmemelidir. Bunun yerine `.env.example` gibi örnek dosya paylaşılır; hangi değişkenlerin gerektiği gösterilir ama gerçek gizli bilgiler paylaşılmaz. Bu küçük alışkanlık, ileride ciddi güvenlik sorunlarını önleyebilir.

Geliştirme (development), test ve üretim (production) ortamlarında değişkenler farklı değerler taşıyabilir. Örneğin geliştirme ortamında sahte ödeme sistemi kullanılırken, üretim ortamında gerçek ödeme servisi kullanılabilir. Yanlış değişkenin yanlış ortamda kullanılması ciddi sonuçlar doğurur.

2.14 SSH Temelleri

Geliştirme ortamı bir noktadan sonra yalnızca kendi bilgisayarınızla sınırlı kalmaz. GitHub'a güvenli bağlantı kurmak, bir VPS sunucusuna bağlanmak, uzak makinede log okumak veya dağıtım işlemi yapmak gerekebilir. Bu durumda "uzaktaki sisteme güvenli biçimde nasıl erişeceğiz?" sorusu ortaya çıkar.

SSH, yani güvenli kabuk (Secure Shell), uzak bir sisteme şifreli bağlantı kurmak için kullanılan yaygın bir yöntemdir. SSH sayesinde terminal üzerinden başka bir makineye bağlanabilir, komut çalıştırabilir ve bazı durumlarda dosya aktarımı yapabilirsiniz. Bu bağlantı özellikle sunucu yönetimi ve Git işlemlerinde sık karşımıza çıkar.

Ancak SSH yalnızca bir bağlantı komutu değildir. Kullanıcı adı, host, port, parola, açık anahtar, özel anahtar ve yetki gibi kavramları birlikte içerir. Bu yüzden SSH öğrenirken komutun detaylarının yanında, bağlantının hangi kimlikle, hangi makineye, hangi güven ilişkisiyle kurulduğunu anlamak güvenlik açısından değerlidir.

SSH'yi, uzak bir bilgisayarın terminaline güvenli bir kapıdan girmek gibi düşünebiliriz. Normalde kendi bilgisayarınızda komut yazarsınız. SSH ile başka bir makineye bağlanır ve o makinenin terminalindeymiş gibi komut çalıştırırsınız. Bağlantı şifreli olduğu için kullanıcı adı, komutlar ve oturum bilgileri açık biçimde taşınmaz.

Basit bağlantı biçimi şöyledir:

```
$ ssh akadal@example.com
```

Bu komut `example.com` adresindeki makineye `akadal` kullanıcısıyla bağlanmayı dener. Elbette gerçek bağlantı için sunucunun erişilebilir olması, kullanıcının var olması ve kimlik doğrulamanın başarılı olması gerekir.

2.14.1 Kullanıcı adı, host ve port

SSH bağlantısında üç temel bilgi sık görülür: kullanıcı adı, host ve port. Kullanıcı adı, uzak sistemde hangi kimlikle oturum açacağınızı belirtir. Host, bağlanılacak maki-

nenin adresidir. Port ise SSH servisinin hangi bağlantı noktasından dinlediğini gösterir. Varsayılan SSH portu çoğu sistemde 22'dir; ancak güvenlik veya yönetim tercihleri nedeniyle farklı olabilir.

```
$ ssh -p 2222 deploy@203.0.113.10
```

Bu örnekte `deploy` kullanıcı adıdır, `203.0.113.10` host adresidir, `-p 2222` ise varsayılan 22 yerine 2222 portunun kullanılacağını belirtir. Çıktı veya hata aldığınızda bu üç bilgiyi ayrı ayrı kontrol etmek gerekir.

SSH bağlantısında kimlik doğrulama parola ile veya anahtar çiftiyle yapılabilir. Parola yönteminde kullanıcı bildiği gizli değeri girer. Anahtar yönteminde ise birbiriyle ilişkili iki parça kullanılır: Biri bağlanılacak sunucuya veya GitHub gibi bir hizmete tanıtılır, diğeri kullanıcı bilgisayarında tutulur. Özellikle düzenli sunucu bağlantılarında ve Git işlemlerinde anahtar yöntemi sık tercih edilir.

Sunucuya tanıtılan parça açık anahtar (public key) olduğu için paylaşılabilir. Kimliği kanıtlayan özel anahtar (private key) ise kullanıcının bilgisayarından çıkarılmamalı ve parola gibi korunmalıdır. Bu ilişkiyi kapı kilidi üzerinden düşünebiliriz: Açık anahtar kapıya takılan kilit, özel anahtar ise yalnızca sahibinde kalan anahtardır. Kilidin başkaları tarafından görülmesi sorun değildir; fakat anahtarın paylaşılması o kimlikle bağlantı kurulmasına izin verebilir.

2.14.2 Uzak Sunucuda Projeyi Çalıştırmak

Yerel bilgisayardan bir VPS'ye bağlanıp daha önce sunucuya klonlanmış projeyi çalıştırdığımızı düşünelim. Aşağıdaki `203.0.113.10` adresi gerçek bir sunucu değil, dokümantasyon için ayrılmış örnek bir IP adresidir.

```
local$ ssh deploy@203.0.113.10
deploy@vps:~$ pwd
/home/deploy
deploy@vps:~$ cd /srv/stok-app
deploy@vps:/srv/stok-app$ git pull --ff-only
Already up to date.
deploy@vps:/srv/stok-app$ npm ci
added 142 packages in 5s
deploy@vps:/srv/stok-app$ npm test
18 tests passed
deploy@vps:/srv/stok-app$ APP_PORT=3000 npm start
Server listening on 0.0.0.0:3000
```

İlk satır yerel terminalde, sonraki satırlar VPS üzerinde çalışır. `pwd` uzak kullanıcının başlangıç konumunu doğrular; `cd` proje dizinine geçer; `git pull --ff-only` uzak depodaki değişiklikleri geçmişi birleştirme commit'i üretmeden alır. Ardından bağımlılıklar kilit dosyasına göre kurulur, testler çalıştırılır ve uygulama başlatılır.

Son komut olan `npm start` çoğu zaman terminali meşgul eder; imleç size geri dönmez çünkü süreç önde çalışmaya devam eder. Kısa bir deneme için bu doğaldır. Terminali başka iş için boşaltmak isterseniz komutun sonuna `&` koyarak süreci arka plana alabilirsiniz: `APP_PORT=3000 npm start &`. Kabuk işi bırakır, size yeni bir satır verir.

Çıkan süreç kimliğini not etmek gerekir; işi bitirmek için `kill` veya `fg` ile öne alıp durdurmak gerekir. Eğitimdeki gibi dışarıdan erişilen bir denemede süreç yöneticisi veya konteyner daha güvenlidir; `&` yalnızca o anki oturum için bir kolaylıktır.

Uygulamanın `0.0.0.0:3000` üzerinde dinlemesi ve VPS güvenlik duvarının 3000 numaralı porta izin vermesi halinde ikinci bir yerel terminalden `http://203.0.113.10:3000` adresine erişilebilir. Uygulama yalnızca `127.0.0.1:3000` üzerinde dinleseydi bu adres VPS'nin kendi içinden erişilebilir, dışarıdan erişilemezdi. Bu doğrudan bağlantı eğitim amaçlıdır; kalıcı bir serviste süreç yöneticisi veya konteyner ile birlikte HTTPS sağlayan bir ters vekil ve sınırlı güvenlik duvarı kuralları kullanılmalıdır.

2.15 Atölye

Şimdiye kadar öğrenilen konular teker teker ele alındığında son derece basit görünse de uygulamaya geçince kafa karışıklığına sebep olabilir. Bu sebeple bu konuların kavranabilmesi için en iyi yöntem uygulamak olacaktır. Kitap boyunca birkaç kez Atölye başlığıyla karşılaşacaksınız. Atölye başlığı altındaki uygulamalar, öğrenmenizin kalıcılığına fayda sağlamak içindir.

Bu atölye Ubuntu, WSL veya benzer bir Unix kabuğunda uygulanır. Komutlar aynı çalışma dizininde sırayla yürütülmelidir.

Görev 1: Ortamı tanı

Komut

```
$ pwd
$ whoami
$ uname -s
```

Beklenen çıktı

```
/home/akadal
akadal
Linux
```

Kullanıcı adı ve ev dizini değişebilir. macOS üzerinde son satır `Darwin` olur.

Görev 2: Atölye dizinini oluştur

Komut

```
$ mkdir -p ~/gelistirme-atolyesi/bolum-2
$ cd ~/gelistirme-atolyesi/bolum-2
$ pwd
```

Beklenen çıktı

```
/home/akadal/gelistirme-atolyesi/bolum-2
```

Görev 3: Dosya ve örnek log oluştur

Komut

```
$ touch bos.txt
$ printf 'terminal-atolyesi\n' > notlar.txt
$ printf 'INFO start\nERROR database\nINFO retry\n' > app.log
$ ls -la
```

Beklenen çıktı

```
.
..
app.log
bos.txt
notlar.txt
```

`ls -la` çıktısında izin, sahip, boyut ve tarih sütunları da görünür.

Görev 4: Kopyala ve yeniden adlandır

Komut

```
$ cp notlar.txt notlar.yedek
$ mv bos.txt tamamlandi.txt
$ ls
```

Beklenen çıktı

```
app.log  notlar.txt  notlar.yedek  tamamlandi.txt
```

Görev 5: Dosya içeriğini farklı ölçekte oku

Komut

```
$ cat app.log
$ head -n 1 app.log
$ tail -n 1 app.log
$ wc -l app.log
```

Beklenen çıktı

```
INFO start
ERROR database
INFO retry
INFO start
INFO retry
3 app.log
```

Görev 6: Metin ve dosya ara

Komut

```
$ grep -n 'ERROR' app.log
$ find . -type f -name '*.log'
```

Beklenen çıktı

```
2:ERROR database
./app.log
```

Görev 7: Yönlendirme ve iş hattı kur

Komut

```
$ printf 'ERROR api\nERROR database\n' >> app.log
$ grep 'ERROR' app.log | sort | uniq -c
```

Beklenen çıktı

```
      1 ERROR api
      2 ERROR database
```

Görev 8: Bir betiği çalıştırılabilir yap

Komut

```
$ printf '#!/usr/bin/env bash\nprintf "ok\n"\n' > check.sh
$ chmod +x check.sh
$ ls -l check.sh
$ ./check.sh
```

Beklenen çıktı

```
-rwxr-xr-x ... check.sh
ok
```

Görev 9: Bir süreci izle ve sonlandır

Komut

```
$ sleep 60 &
$ process_id=$!
$ ps -p "$process_id" -o pid=,comm=
$ kill "$process_id"
```

Beklenen çıktı

```
<PID> sleep
```

PID her çalıştırmada değişir. `kill` başarılı olduğunda ayrıca çıktı üretmez.

Görev 10: Ortam değişkeni tanımla

Komut

```
$ export ATOLYE_ENV=development
$ env | grep '^ATOLYE_ENV='
```

Beklenen çıktı

```
ATOLYE_ENV=development
```

Görev 11: Ağ araçlarını doğrula

Komut

```
$ curl -fsSI https://example.com | head -n 1
$ ssh -V 2>&1
```

Beklenen çıktı

```
HTTP/2 200
OpenSSH_9.x ...
```

HTTP protokolü ve OpenSSH sürümü sisteme göre değişebilir; ilk komutta `200`, ikinci komutta `OpenSSH` görülmesi yeterlidir.

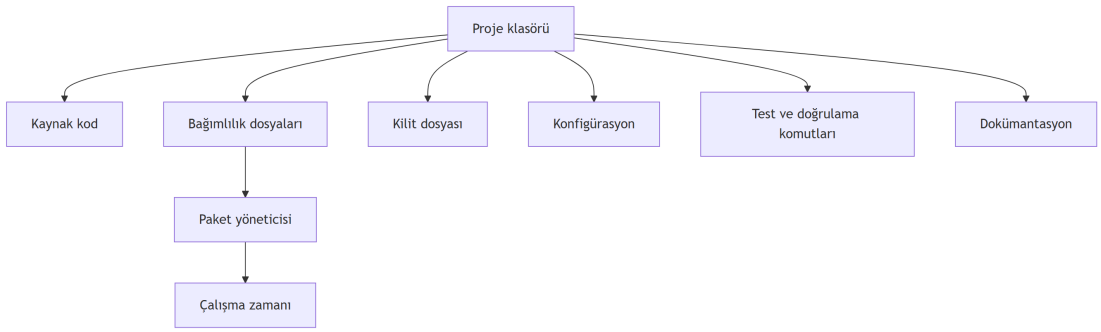
Bölüm 3

Proje, Çalışma Zamanı (Runtime) ve Bağımlılıklar

Bir yazılım projesini ilk kez açtığımızda çoğu zaman gözümüz doğrudan kaynak koda gider. `src` klasörü, bileşenler, fonksiyonlar, sınıflar veya ekran dosyaları bize projenin asıl gövdesi gibi görünür. Bu bakış başlangıç için anlaşılırdır; çünkü uygulamanın davranışını en görünür biçimde kod dosyaları üretir.

Ancak bir projenin çalışması yalnızca kaynak kodla açıklanamaz. Kodun hangi çalışma zamanıyla çalışacağı, hangi paketlere ihtiyaç duyduğu, hangi ortam değişkenlerini beklediği, hangi sürümlere bağımlı olduğu, nasıl test edileceği ve hangi komutla başlatılacağı da projenin parçasıdır. Bu bilgiler eksik olduğunda proje klasörü duruyor gibi görünür; fakat proje gerçekte yeniden kurulabilir bir çalışma düzeni sunmaz.

Bu bölümde projeyi bir dosya yığını olarak değil, çalışabilir ve sürdürülebilir bir bütün olarak ele alacağız. Bir proje klasörünün temel bileşenleri Şekil 3.1 ile kabaca özetlenmiştir.



Şekil 3.1: Çalışabilir bir projenin temel bileşenleri

Bu şema bize şu fikri verir: Kod tek başına anlamlıdır, fakat çalışabilir proje olmak için çevresindeki düzenle birlikte düşünülmelidir. Bölümün sonunda bu karmaşıklığın neden Docker gibi standartlaştırma araçlarına ihtiyaç doğurduğunu da daha net göreceğiz.

Bir arkadaşınız size bir proje klasörü gönderdi diyelim. Klasörü açtınız ve içinde kod dosyalarını gördünüz. İlk anda "tamam, proje bende" diye düşünebilirsiniz. Ancak projeyi çalıştırmaya kalktığınızda eksikler görünmeye başlar. Hangi Node.js sürümü

gerekiyor? Veritabanı nasıl kurulacak? Hangi paketler yüklenecek? Test komutu nedir? Ortam değişkenleri nereden gelecek?

İşte bu yüzden yazılım projesi sadece koddan ibaret değildir. Kod, projenin davranışını anlatır; fakat projenin çalışması için gerekli çevre bilgisi de en az kod kadar önemlidir. Bağımlılık dosyaları hangi paketlerin gerektiğini, kilit dosyaları hangi kesin sürümlerin kullanılacağını, konfigürasyon dosyaları uygulamanın hangi ayarlarla çalışacağını, dokümantasyon ise ekibin projeyi nasıl anlayacağını gösterir.

Bu ayrımı küçük bir teslim senaryosunda düşünelim. Geliştirici uygulamayı kendi bilgisayarında başarıyla çalıştırmış olabilir. Ancak öğretim elemanı veya ekip arkadaşı aynı projeyi çalıştıramıyorsa, sorun yalnızca "kod çalışıyor mu?" sorusu değildir. Daha doğru soru şudur: Bu proje başka bir ortamda yeniden kurulabilir mi? Eğer cevap hayırsa, kodun çalışması kişisel bilgisayara bağlı kalmış demektir.

Dolayısıyla projenin çalışma düzenini öğrenmek, yazılım geliştirme ortamını sürdürülebilir hale getirmenin ilk adımlarından biridir.

Proje klasöründeki dosya türleri

Bir proje klasörüne dikkatli bakarsanız dosyaların rastgele durmadığını görürsünüz. Bazı dosyalar uygulamanın davranışını belirler, bazıları kurulum bilgisini taşır, bazıları bağımlılıkları tarif eder, bazıları ise yalnızca belirli ortama ait ayarları içerir. Bu ayrımı görmek, proje okuryazarlığının önemli bir parçasıdır.

Tipik bir web projesi şu parçalardan oluşabilir:

```
stok-app
├── README.md
├── package.json
├── package-lock.json
├── .env.example
├── src
│   ├── app.js
│   └── db.js
└── tests
    └── app.test.js
```

Bu düzen ekosistemden ekosisteme değişebilir; dosya adları farklılaşabilir. İşleyiş yine de benzerdir. Kaynak kod uygulamanın davranışını taşır. Bağımlılık dosyası projenin hangi paketlere ihtiyaç duyduğunu anlatır. Kilit dosyası bu paketlerin kesin sürümlerini sabitler. `.env.example` gerekli ortam değişkenlerini gösterir. `README.md` ise projenin insan tarafından anlaşılmasını sağlar.

Bu parçaların her biri bakım maliyetiyle ilişkilidir. Güncel olmayan dokümantasyon yanlış kurulum doğurur. Eksik bağımlılık dosyası tekrarlanabilirliği bozar. Yanlış ortam değişkeni üretim ortamında hata üretebilir. Bu yüzden proje klasörünü okumak, yalnızca dosya isimlerine bakmak değil, projenin sürdürülebilirliğini değerlendirmektir.

Kaynak kod uygulamanın ne yaptığını, dokümantasyon ise projenin nasıl anlaşılacağını ve kullanılacağını gösterir. `src/app.js` dosyası uygulama davranışını taşıyabilir; fakat yeni gelen bir ekip üyesi hangi komutla başlayacağını `README.md` üzerinden öğrenir. Bu

yüzden dokümantasyon kodun süsü değildir. Ekip hafızasının parçasıdır. Ancak dokümantasyon güncel değilse güven kaybı üretir; yanlış bilgi bazen eksik bilgiden daha pahalıdır.

Konfigürasyon dosyaları, uygulamanın hangi ayarlarla çalışacağını belirler. Örneğin bir `config.yml` dosyasında varsayılan port, log seviyesi veya servis adresi bulunabilir:

```
server:
  port: 3000
logging:
  level: info
```

Bu tür dosyalar çoğu zaman Git'e eklenebilir; çünkü projenin beklenen davranışını tarif eder. Ancak içinde parola, API anahtarı veya gerçek müşteri verisi varsa durum değişir. O zaman dosyanın tamamı veya ilgili değerler gizli tutulmalıdır. Burada temel soru şudur: Bu dosya projeyi tekrar kurmak için gerekli bilgiyi mi taşıyor, yoksa gizli ortam bilgisini mi açığa çıkarıyor?

Bağımlılık dosyaları projenin hangi dış paketlere ihtiyaç duyduğunu gösterir. Kilit dosyaları ise bu paketlerin hangi kesin sürümlerle kurulduğunu kayda geçirir. Örneğin JavaScript dünyasında `package.json` bağımlılık tarifini, `package-lock.json` ise kurulumun ayrıntılı sürüm ağacını taşır.

Bu iki dosya genellikle Git'e eklenir; çünkü ekipte herkesin aynı bağımlılıkları kurmasını sağlar. Ancak `node_modules` gibi kurulan paket klasörleri genellikle Git'e eklenmez. Çünkü bu klasörler paket yöneticisi tarafından yeniden üretilebilir ve çok büyük olabilir. Yani paylaşılması gereken şey bağımlılıkların kendisi değil, bağımlılıkları yeniden kurmayı sağlayan tariftir.

Kurulumun doğrulanması ve ortam değişkenleri

Bir projenin sağlıklı aktarılabilmesi için nasıl çalıştırılacağı ve nasıl doğrulanacağı açık olmalıdır. "Bende çalışıyor" ifadesi yerine, herkesin aynı komutları çalıştırıp benzer çıktıları okuyabilmesi gerekir.

```
$ npm install
added 142 packages in 8s

$ npm test
18 tests passed
0 tests failed

$ npm run dev
Local: http://localhost:3000/
```

Bu çıktılar üç ayrı kontrol sağlar. İlk komut bağımlılıkların kurulduğunu, ikinci komut testlerin geçtiğini, üçüncü komut geliştirme sunucusunun başladığını gösterir. README dosyasında bu adımların yer alması, ekip içinde kurulum belirsizliğini azaltır.

Ortam değişkenleri, projenin farklı ortamlarda farklı ayarlarla çalışmasını sağlar. Veritabanı adresi, uygulama portu veya API anahtarı kodun içine yazılmak yerine dışarı-

dan verilebilir. Bu yaklaşım esneklik sağlar; ancak eksik veya yanlış değişken uygulamayı durdurabilir. Bu yüzden `.env.example` dosyasıyla hangi değişkenlerin beklendiğini göstermek, gerçek `.env` dosyasını ise gizli tutmak iyi bir ekip alışkanlığıdır.

3.1 Çalışma Zamanı (Runtime) Kavramı

Kod dosyasını görmek, o kodun kendi kendine çalışacağı anlamına gelmez. Bir JavaScript dosyası, Python betiği veya Java uygulaması belirli bir çalışma ortamına ihtiyaç duyar. Okuyucu çoğu zaman bu ortamı fark etmeden kullanır; çünkü bilgisayarında zaten kurulu olabilir. Ancak ekip arkadaşının bilgisayarında aynı ortam yoksa proje çalışmaz.

Bu noktada çalışma zamanı (runtime) kavramı devreye girer. Çalışma zamanı, programın çalışması için gereken yürütme ortamını ifade eder. JavaScript için Node.js, Python için Python yorumlayıcısı, Java için JVM, .NET için .NET runtime bu bağlamda düşünülebilir. Kod, çalışma zamanı tarafından okunur, yorumlanır, derlenir veya yürütülür.

Çalışma zamanı yalnızca yorumlanan dillerin işi değildir. Derlenen bir program da çalışırken başka kütüphanelere ihtiyaç duyabilir. Windows'ta bu çoğu zaman `.dll`, Linux'ta `.so`, macOS'ta `.dylib` dosyasıdır. İçinde artık toplama, matematik işlemi veya dile ait temel işlevler bulunabilir. Julia bu duruma iyi bir örnektir: kod derlenebilir; fakat sayısal işlemler için `libblas` gibi bir paylaşılan kütüphaneye bağlı kalabilir. Yani "derlendi, o halde ortam yok" demek her zaman doğru değildir. Ortam, programın yanında duran ve onun çalışmasını mümkün kılan parçaların bütünüdür.

Basit bir kontrol şöyle yapılabilir:

```
$ node --version
v20.11.1

$ python3 --version
Python 3.12.2
```

Bu çıktı yalnızca "program kurulu" demek değildir. Projenin hangi çalışma zamanı sürümüyle çalışacağını anlamamızı sağlar. Eğer proje Node.js 20 ile yazılmışsa Node.js 16 kullanan bir ekip üyesi beklenmeyen hatalarla karşılaşabilir.

Dolayısıyla çalışma zamanı bilgisi proje dokümantasyonunda açık olmalıdır. Bu bilgi eksikse kurulum kişisel tahmine kalır; kişisel tahmin ise sürdürülebilir geliştirme ortamının karşısındaki en yaygın risklerden biridir.

3.1.1 Programlama dili ile çalışma zamanı farkı

Programlama dili, kodu hangi sözdizimi ve kurallarla yazdığımızı anlatır. Çalışma zamanı ise bu kodun hangi ortamda çalışacağını belirler. JavaScript dili tarayıcıda da çalışabilir, Node.js üzerinde de. Python dili belirli bir Python yorumlayıcısıyla çalışır. Java kodu ise çoğu zaman JVM üzerinde yürütülür (Tablo 3.1).

Tablo 3.1: Programlama dili ile çalışma zamanı farkı

Ayrım	Ne ifade eder?	Örnek
Programlama dili	Kodu yazma biçimi	JavaScript, Python, Java
Çalışma zamanı	Kodun yürütüldüğü ortam	Node.js, Python interpreter, JVM
Sürüm	Davranışı etkileyen belirli çalışma zamanı düzeyi	Node.js 20, Python 3.12
Sürüm yöneticisi	Aynı makinede birden çok çalışma zamanı sürümünü yönetme aracı	nvm, pyenv, asdf

Bu ayrım önemlidir; çünkü ekip "JavaScript kullanıyoruz" dediğinde hâlâ eksik bilgi vardır. Nerede ve hangi sürümle çalıştırıyoruz sorusu ayrıca cevaplanmalıdır.

Node.js, Python, PHP, Java ve .NET örnekleri

Farklı ekosistemler çalışma zamanını farklı biçimlerde kurar. Node.js JavaScript'i sunucu tarafında çalıştırır. Python kendi yorumlayıcısı ve sanal ortamlarıyla kullanılır. PHP çoğu zaman web sunucusu veya PHP-FPM gibi çalışma düzenleriyle ilişkilidir. Java JVM üzerinde çalışır. .NET uygulamaları .NET runtime gerektirir. Proje teslimiinde yalnızca "hangi dili kullandık?" değil, "hangi çalışma zamanı, hangi sürüm ve hangi kurulum adımları gerekiyor?" soruları da sorulmalıdır.

3.1.2 Çalışma zamanı sürümünün geliştirme ortamına etkisi

Aynı tarifin farklı fırınlarda farklı sonuç vermesi gibi, aynı kod da farklı çalışma zamanı sürümlerinde farklı davranabilir. Özellikle dil özellikleri, paket uyumluluğu ve güvenlik güncellemeleri sürüme bağlıdır. Bu yüzden çalışma zamanı sürümü küçük bir ayrıntı değildir.

Örneğin `README.md` içinde şu bilgi yer alabilir:

```
Required runtime:
- Node.js 20.x
- npm 10.x
```

Bu bilgi ekip için bir kurulum standardı oluşturur. Ancak tek başına yazmak yeterli değildir; gerekirse `.nvmrc`, `.tool-versions` veya benzeri sürüm dosyalarıyla desteklenmelidir. Böylece proje, kişisel bilgisayara bağlı olmaktan bir adım daha uzaklaşır.

3.2 Derleyici, Yorumlayıcı ve Çalışma Zamanı

Kodun yazılması ile çalışması arasında her zaman aynı yol izlenmez. Bazı dillerde kod önce başka bir biçime dönüştürülür, bazı dillerde doğrudan yorumlanır, bazı ekosistemlerde ise ara bir model kullanılır. Bu ayrımın geliştirme ortamına etkisine odaklanacağız.

Peki bu ayrım proje pratiğinde neyi değiştirir? Derleme (build) adımı var mı? Uygulama çalışmadan önce derlenmeli mi? Hata geliştirme sırasında mı, derleme sırasında mı, çalışma sırasında mı ortaya çıkar? Dağıtım sürecine hangi dosyalar taşınır? Bu soruların cevabı kullanılan dil ve çalışma modeline göre değişir.

Derleyici, yorumlayıcı ve çalışma zamanı arasındaki iş bölümünü Tablo 3.2 başlangıç düzeyinde ayırır.

Tablo 3.2: Derleyici, Yorumlayıcı ve Çalışma Zamanı

Model	Temel fikir	Geliştirme ortamına etkisi
Derlenen diller	Kod çalışmadan önce başka biçime çevrilir	Derleme adımı önem kazanır
Yorumlanan diller	Kod çalışma anında yorumlanır	Çalışma zamanı ve bağımlılıklar daha görünürdür
Ara model	Kod ara biçime çevrilip çalışma zamanı üzerinde çalışır	Hem derleme hem çalışma zamanı gereksinimi olabilir
JIT derleme	Kod çalışırken parça parça makine koduna çevrilir	Hem yorumlayıcı hem derleyici izleri birlikte görülebilir

Bu ayrımı yapmaktaki asıl gerekçe dilleri sınıflandırıp bırakmak değildir; projenin neden belirli komutlara, dosyalara ve kurulum adımlarına ihtiyaç duyduğunu görmektir.

Derlenen diller

Derlenen dillerde kaynak kod genellikle çalıştırılmadan önce başka bir biçime çevrilir. Örneğin C veya Go gibi dillerde derleme çıktısı doğrudan çalıştırılabilir bir dosya olabilir. Bu yaklaşımda derleme adımı kritik hale gelir. Eğer derleme komutu, hedef işletim sistemi veya gerekli araçlar belgelenmemişse, proje başka bir makinede yeniden üretilemeyebilir.

Yorumlanan diller

Yorumlanan dillerde kod çoğu zaman çalışma anında yorumlayıcı tarafından yürütülür. Python ve JavaScript bu bağlamda sık verilen örneklerdir. Bu durum geliştirme sürecini hızlandırabilir; ancak yorumlayıcının sürümü, kurulu paketler ve ortam değişkenlerine dair özgül bilgilerin önemi daha görünür hale gelir. “Benim bilgisayarımda çalışıyor” problemi çoğu zaman bu çevre bilgilerinin farklı olmasından doğar.

3.2.1 Ara model kullanan ekosistemler

Bazı ekosistemler kaynak kodu doğrudan makine koduna çevirmek yerine ara bir biçim kullanır. Java tarafında `.class` dosyaları ve JVM, .NET tarafında ara dil (CIL - Common Intermediate Language) ve .NET runtime bu düşünceye örnek verilebilir. Bunu bir

belgenin önce ortak bir formata çevrilip sonra farklı cihazlarda okunması gibi düşünebiliriz.

Bu model geliştirme ortamında iki tür ihtiyacı birlikte doğurabilir: Kodun derlenmesi için gerekli araçlar ve ortaya çıkan çıktının çalışması için gerekli çalışma zamanı. Dolayısıyla proje dokümantasyonu yalnızca "Java kurun" demekle yetinmemeli; hangi JDK, hangi derleme aracı ve hangi çalıştırma komutu gerektiğini netleştirmelidir.

3.2.2 Geliştirme ortamı açısından farkları

Geliştirme ortamı açısından bu modellerin farkı, projenin hangi adımlarla çalıştırılacağını belirlemesidir. Yorumlanan bir projede "çalışma zamanı ve paketler doğru mu?" sorusu öne çıkarken, derlenen bir projede "derleme çıktısı üretildi mi?" sorusu önem kazanabilir. Tablo 3.3 bu soruları ve önemini bir arada sıralar.

Tablo 3.3: Geliştirme ortamı açısından farkları

Soru	Neden önemlidir?
Derleme gerekiyor mu?	Dağıtım sürecine hangi çıktının taşınacağını belirler
Çalışma zamanı gerekiyor mu?	Hedef ortamda hangi yazılımın kurulu olması gerektiğini gösterir
Sürüm sabit mi?	Ekipte aynı davranışın tekrar edilmesini sağlar
Test çıktısı ayrıldı mı?	Hangi kanıtın dağıtıma girdiğini gösterir
Ortam değişkenleri belgelendi mi?	Aynı kodun farklı davranış üretmesini önler

Bu yüzden hangi modelin daha iyi olduğundan çok, seçilen modelin geliştirme ve bakım sürecine ne yüklediğini anlamak gerekir.

3.3 Paket ve Bağımlılık Nedir?

Bir uygulama geliştirirken her şeyi sıfırdan yazmayız. Tarih biçimlendirme, HTTP isteği gönderme, veritabanına bağlanma, kullanıcı arayüzü oluşturma veya test yazma gibi birçok iş için hazır paketlerden faydalanırız. Bu, geliştirme sürecini hızlandırır ve yaygın çözümleri kullanmamızı sağlar.

Ancak hazır paket kullanmak, dışarıdan bir sorumluluğu projeye dahil etmek anlamına gelir. Paket güncellenecek mi? Güvenlik açığı çıkarsa ne olacak? Paketin kendi bağımlılıkları var mı? Lisansı uygun mu? Bakımı devam ediyor mu? Bu sorular özellikle sürdürülebilirlik açısından önemlidir.

Paket, belirli bir işlevi sağlamak üzere dağıtılan yazılım birimidir. Bağımlılık (dependency) ise projenizin çalışmak için ihtiyaç duyduğu dış paket veya bileşendir. Örneğin bir Node.js projesinde `express` paketi web sunucusu kurmak için kullanılan bir bağımlılık olabilir.

Bağımlılık kullanmak kötü değildir. Tam tersine modern yazılım geliştirme büyük ölçüde bağımlılıklar üzerine kuruludur. Ancak her bağımlılık projeye bakım, güncelleme ve güvenlik sorumluluğu ekler. Bu yüzden bağımlılık seçimi yalnızca teknik kolaylık değil, proje yönetimi kararıdır.

3.3.1 Kütüphane, framework ve bağımlılık farkı

Kütüphane, belirli işlevleri sizin çağırdığınız araçlar bütünü gibi düşünülebilir. Framework ise çoğu zaman uygulamanın genel yapısını belirler ve sizi kendi çalışma düzeni içine alır. Bağımlılık ise projenizin çalışmak için dışarıdan ihtiyaç duyduğu paketlerin genel adıdır; Tablo 3.4 bu üç kavramı karşılaştırır.

Tablo 3.4: Kütüphane, framework ve bağımlılık farkı

Kavram	İhtiyaç durumu	Örnek
Kütüphane	Siz ihtiyaç duyunca çağırırsınız	Tarih biçimlendirme paketi
Framework	Proje yapısını ve akışı belirler	Web uygulama framework'ü
Bağımlılık	Projenin ihtiyaç duyduğu dış bileşendir	Kütüphane veya framework olabilir
SDK (yazılım geliştirme kiti)	Birden çok kütüphane, araç ve dokümantasyonu bir arada sunar	AWS SDK, Firebase SDK

Bu ayrım net sınırlar koymaktan çok, hangi seçimin projeye ne kadar sorumluluk yükleyeceğini gösterir. Framework seçimi genellikle proje mimarisini, ekip öğrenme süresini ve bakım şeklini daha güçlü etkiler.

Dolaylı bağımlılık, sizin doğrudan kurmadığınız ama kurduğunuz paketin ihtiyaç duyduğu başka pakettir. Örneğin siz `express` kurarsınız; `express` de kendi çalışması için başka paketleri getirir. Böylece bağımlılık ağacı büyür. Bu durum normaldir; ancak güvenlik ve bakım açısından görünmez sorumluluklar oluşturabilir. Bir güvenlik açığı bazen sizin doğrudan seçtiğiniz pakette değil, onun alt bağımlılığında ortaya çıkar.

3.3.2 Bağımlılık sayısı neden hızla artar?

Bir proje ekibi işe "sadece birkaç paket kurduk" diye başlayabilir. Fakat `npm install` tamamlandığında yüzlerce paketin kurulduğunu görür. Bunun sebebi çoğu zaman dolaylı bağımlılıklardır. Bir paket başka paketlere, onlar da başka paketlere ihtiyaç duyar.

Bu durumu biraz da gülümseten bir örnekle görelim. JavaScript dünyasında bir sayının çift olup olmadığını soran `is-even` paketi vardır. Tek olup olmadığını soran `is-odd` ise çoğu zaman işi ona bırakır: tek sayı, çift olmayan sayıdır. Yani biri ötekine bağımlıdır. İki satırlık bir kontrol, iki paket ve bir dolaylı bağ haline gelir. Mizahın altında ciddi bir ders vardır: bağımlılık bazen gerçek bir işi çözer, bazen de çok küçük bir işi dışarı taşır.

Bu artış tek başına panik sebebi değildir; modern ekosistemlerin çalışma biçimi böyledir. Ancak bağımlılık sayısı arttıkça kurulum süresi, güvenlik taraması, sürüm uyumu ve hata ayıklama karmaşıklığı da artabilir. Bu yüzden bağımlılık eklemek, yalnızca "işimi kolaylaştırıyor" diye değil, "projeye hangi bakım sorumluluğunu ekliyor?" diye değerlendirilmelidir.

Bağımlılık seçimi kısa vadede hız kazandırabilir; fakat uzun vadede bakım maliyeti doğurur. Bu maliyet bazen sürüm güncellemesi, bazen güvenlik açığı, bazen de ekibin aracı öğrenme süresi olarak karşımıza çıkar. Bu yüzden bağımlılık kararı teknik olduğu kadar yönetsel bir karardır. Proje büyüdükçe bu kararların etkisi daha görünür hale gelir.

3.4 Paket Yöneticileri Neden Farklıdır?

Paket yöneticileri farklıdır; çünkü yazılım ekosistemleri aynı ihtiyaçları farklı tarihsel ve teknik yollarla çözmüştür. JavaScript dünyasında `npm`, Python tarafında `pip` ve Poetry, PHP tarafında Composer, Java tarafında Maven veya Gradle, Scala dünyasında sbt, Rust dünyasında ise Cargo gibi araçlarla karşılaşırız. Bunların hepsi paket yönetir; fakat dosya düzenleri, komutları ve bağımlılık çözme biçimleri farklıdır.

Bu fark ilk başta yorucu görünebilir. Ancak kavramsal düzeyde hepsi benzer sorulara cevap verir: Hangi paketler gerekli? Hangi sürümler kurulacak? Paketler nereden indirilecek? Kurulum nasıl tekrar edilecek? Hangi dosyalar Git'e eklenecek?

Tablo 3.5: Paket yöneticileri ve tipik dosyaları

Ekosistem	Paket yöneticisi / araç	Tipik dosya
JavaScript	npm, yarn, pnpm	`package.json`
Python	pip, Poetry	`requirements.txt`, `pyproject.toml`
PHP	Composer	`composer.json`
Java	Maven, Gradle	`pom.xml`, `build.gradle`
Rust	Cargo	`Cargo.toml`
Go	Go modules	`go.mod`
.NET	NuGet	`.csproj`
Sistem düzeyi	apt, brew	İşletim sistemi paket kayıtları

Tablo 3.5 yalnızca hangi ekosistemde hangi dosyayı arayacağınızı gösteren bir başlangıç haritasıdır.

Sistem paket yöneticileri işletim sistemi düzeyindeki araçları yönetir. Ubuntu'da `apt`, macOS'ta `brew` bu rolü üstlenebilir. Örneğin Git veya PostgreSQL gibi araçlar sistem düzeyinde kurulabilir. Bu paketler belirli bir projeye değil, makinenin genel çalışma ortamına etki eder. Bu yüzden sistem paketi kurmak, başka projeleri de etkileyebilecek bir karardır.

JavaScript projesinde `npm install`, Python projesinde `pip install -r requirements.txt`, PHP projesinde `composer install`, Java projesinde `mvn install`, Scala

projesinde `sbt run`, Rust projesinde `cargo run` gibi komutlarla karşılaşabilirsiniz. Komutlar farklıdır; fakat amaç benzerdir: Projeyi çalıştırmak için gereken bağımlılıkları kurmak. Bu yüzden yeni bir projeye girdiğinizde ilk bakacağınız şeylerden biri, ilgili ekosistemin bağımlılık dosyası ve kurulum komutudur.

3.4.1 Aynı makinede birden fazla paket yöneticisi kullanmanın riski

Aynı bilgisayarda birden fazla paket yöneticisi kullanmak normaldir; fakat sınırlar belirsizleşirse sorun çıkar. Örneğin Python paketini sistem Python'una mı, sanal ortama mı kurduğunuzu bilmezseniz proje başka makinede çalışmayabilir. JavaScript tarafında aynı projede hem `npm` hem `yarn` lock dosyası bulunması da ekipte kafa karışıklığı yaratabilir (Tablo 3.6).

Tablo 3.6: Aynı makinede birden fazla paket yöneticisi

Durum	Risk	Çözüm
Aynı projede iki kilit dosyası var	Ekip farklı bağımlılık ağaçları kurabilir	Tek paket yöneticisi standardı belirle
Paket sistem düzeyine kurulmuş	Başka projeleri etkileyebilir	Proje düzeyi ortam kullan
Sanal ortam aktif değil	Paket yanlış yere kurulabilir	Kurulum öncesi ortamı doğrula
`node` yolu beklenen sürüm değil	Uygulama başka çalışma zamanında açılır	Sürüm yöneticisini veya dokümandaki sürümü doğrula
Global ve yerel paket karışmış	"Bende var" başka makinede yok olur	Kurulu paketin hangi kapsamda olduğunu kontrol et
Aynı makinede iki Composer/PHP	Yanlış ikili çağrılabilir	`which` ile hangi aracın çalıştığını görün

Bu risklerin ortak noktası şudur: Paket yönetimi kişisel bilgisayar alışkanlığına bırakıldığında, ekip içinde tekrarlanabilirlik zayıflar.

3.5 Bağımlılık Dosyaları

Bağımlılık dosyaları, projenin dış dünyayla yaptığı teknik sözleşmeyi gösterir.[1] Hangi paketleri kullanıyoruz? Hangi sürüm aralıklarını kabul ediyoruz? Projeyi kurmak için hangi ekosistem aracına ihtiyaç var? Bu soruların cevabı çoğu zaman bağımlılık dosyalarında bulunur.

Örneğin bir JavaScript projesinde `package.json`, bir Python projesinde `requirements.txt` veya `pyproject.toml`, bir PHP projesinde `composer.json`, bir Java

[1] Decan, A., Mens, T., & Grosjean, P. (2019). An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, *24*(1), 381–416. https://doi.org/10.1007/s10664-017-9589-y

projesinde `pom.xml` bu rolü üstlenebilir. Bu dosyalar yalnızca paket listesi değildir; projenin çalışma koşullarını belgelendirir.

Basit bir JavaScript örneği:

```
{
  "scripts": {
    "dev": "node src/app.js",
    "test": "node --test"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

Bu dosya bize projenin `express` paketine ihtiyaç duyduğunu ve `dev`, `test` gibi komutların nasıl çalışacağını gösterir. Böylece ekip yalnızca kodu değil, kodu çalıştırma bilgisini de paylaşmış olur.

3.5.1 Proje bağımlılıklarının dosyada tarif edilmesi

Proje bağımlılıklarının dosyada tarif edilmesi, kurulum bilgisini kişisel hafızadan çıkarıp proje hafızasına taşır. Örneğin `requirements.txt` dosyasında şu satır yer alabilir:

```
fastapi==0.111.0
uvicorn==0.30.1
```

Bu dosya Git'e eklenebilir; çünkü gizli bilgi değil, kurulum bilgisidir. Başka bir ekip üyesi `pip install -r requirements.txt` komutuyla aynı paketleri kurabilir. Eğer bu bilgi dosyada değil de yalnızca bir kişinin bilgisayarında duruyorsa, proje sürdürülebilir olmaktan uzaklaşır.

Farklı ekosistemlerde dosya adları değişse de işlev benzerdir. `package.json` JavaScript projesinin paketlerini ve komutlarını, `requirements.txt` veya `pyproject.toml` Python bağımlılıklarını, `composer.json` PHP paketlerini, `pom.xml` ise Maven tabanlı Java projesinin bağımlılık ve derleme bilgisini taşır.

Bu dosyaları gördüğünüzde şunu sormalısınız: Bu dosya projeyi yeniden kurmak için gerekli bilgiyi içeriyor mu? Eğer evet, çoğu durumda Git'e eklenmelidir. Ancak dosyanın içine gizli anahtar, kişisel erişim bilgisi veya makineye özgü yol yazılmışsa bu ayrı bir risktir.

Bağımlılık tarif dosyaları Git'e eklenir; çünkü proje yalnızca kodla değil, kurulum tarifiyle birlikte paylaşılmalıdır. `package.json` dosyası olmadan JavaScript projesinin hangi paketlere ihtiyaç duyduğunu tahmin etmek gerekir. Bu da tekrarlanabilir kurulumu zayıflatır.

```
"dependencies": {
  "cors": "^2.8.5"
}
```

Bu satır projenin `cors` paketine ihtiyaç duyduğunu söyler. Ancak hangi alt bağım-

lılıkların hangi kesin sürümlerle kurulduğunu ayrıca kilit dosyası belirler. Bu yüzden bağımlılık tarif dosyası ve kilit dosyası birlikte düşünülmelidir.

3.6 Kilit Dosyası (Lock File)

Bağımlılık dosyası çoğu zaman hangi paketlerin gerekli olduğunu söyler; fakat kurulum sırasında bu paketlerin hangi kesin sürümlerle geleceği ayrıca önemlidir. Çünkü paket yöneticisi sürüm aralıklarını yorumlayabilir ve farklı tarihlerde farklı alt sürümler kurabilir. Bu durum ekip içinde "aynı dosyalar var ama davranış farklı" sorununa yol açabilir.

Kilit dosyası (lock file), kurulan bağımlılık ağacının kesin sürüm bilgisini kayda geçirir. JavaScript tarafında `package-lock.json`, `yarn.lock` veya `pnpm-lock.yaml`; PHP tarafında `composer.lock` gibi dosyalar bu işi yapar. Bu dosyalar büyük ve karmaşık görünebilir; fakat görevleri basittir: Kurulumun tekrar edilebilir olmasını sağlamak.

Küçük bir kilit dosyası parçası şöyle görünebilir:

```
"node_modules/express": {
  "version": "4.18.2",
  "resolved": "https://registry.npmjs.org/express/-/express-4.18.2.tgz"
}
```

Bu parça bize `express` paketinin kesin olarak `4.18.2` sürümüyle kurulduğunu gösterir. Bu bilgi olmadan paket yöneticisi izin verilen aralıktan farklı bir sürüm seçebilir.

3.6.1 Aynı bağımlılığın farklı versiyonları

Aynı bağımlılığın farklı sürümleri küçük davranış farkları üretebilir. Bir ekip üyesinde `4.18.2`, diğerinde `4.18.3` kurulmuş olabilir. Bu fark bazen sorun çıkarmaz; bazen de Tablo 3.7 içinde sıralanan türde beklenmeyen bir hataya yol açar.

Tablo 3.7: Aynı bağımlılığın farklı versiyonları

Durum	Olası sonuç
Kilit dosyası var	Herkes aynı sürüm ağacına daha yakın kurulum yapar
Kilit dosyası yok	Paket yöneticisi farklı tarihte farklı sürüm seçebilir
Kilit dosyası güncel değil	Kurulum ve kod beklentisi ayrışabilir
Kilit dosyası commit edilmemiş	CI ile yerel kurulum ayrışabilir
Farklı paket yöneticisi kullanılıyor	Aynı paket adı farklı çözülebilir

Bu yüzden kilit dosyası, ekip içinde "aynı projeyi gerçekten aynı bağımlılıklarla mı çalıştırıyoruz?" sorusunun cevabını güçlendirir.

3.6.2 Kilit dosyasıyla tekrarlanabilir kurulum

Tekrarlanabilir kurulum, aynı proje dosyalarından aynı çalışma düzenini tekrar oluşturabilmektir. JavaScript tarafında `npm ci` komutu bu düşünce için iyi bir örnektir; çünkü `package-lock.json` dosyasına bağlı kalarak temiz kurulum yapar.

```
$ npm ci
added 142 packages in 5s
```

Bu çıktı bize paketlerin kilit dosyası temel alınarak kurulduğunu gösterir. `npm install` geliştirme sırasında bağımlılık dosyalarını güncelleyebilirken, `npm ci` özellikle otomasyon ve CI ortamlarında daha kontrollü kurulum sağlar.

Takım içinde aynı bağımlılık ağacını kullanmak, hata ayıklamayı ve kalite kontrolü kolaylaştırır. Bir test bir geliştiricide geçiyor, diğerinde kalıyorsa ilk sorulardan biri bağımlılıkların aynı kurulup kurulmadığıdır. Kilit dosyası bu sorunun cevabını güçlendirir. Elbette tek başına yeterli değildir; çalışma zamanı sürümü ve ortam değişkenleri de aynı derecede önemlidir.

3.6.3 Kilit dosyasını silmenin sonuçları

Kilit dosyası silindiğinde paket yöneticisi bağımlılık ağacını yeniden çözebilir. Bu bazen bilinçli bir güncelleme adımıdır; fakat çoğu zaman farkında olmadan yapılırsa beklenmeyen sürüm değişiklikleri doğurur.

```
$ rm package-lock.json
$ npm install
```

Bu iki komut sonrasında yeni bir kilit dosyası oluşabilir; fakat bu dosya eski bağımlılık ağacıyla birebir aynı olmak zorunda değildir. Bu yüzden kilit dosyasını silmek basit temizlik işlemi gibi görülmemelidir. Ekip içinde gerekçesi, etkisi ve test sonucu birlikte değerlendirilmelidir.

3.7 Anlamsal Versiyonlama (Semantic Versioning)

Paket sürümlerini çoğu zaman `1.4.2` gibi üç parçalı sayılarla görürüz. İlk bakışta bunlar yalnızca teknik numaralar gibi durur. Ancak bu numaralar paketin nasıl değiştiğine dair bir sözleşme taşıyabilir. Bu sözleşmeye anlamsal versiyonlama (semantic versioning) denir.[2]

Anlamsal versiyonlamada sürüm genellikle ana.küçük.yama (`major.minor.patch`) düzeninde ifade edilir. Örneğin `2.5.1` sürümünde `2` ana (major), `5` küçük (minor), `1` ise yama (patch) kısmıdır. Bu sistem, güncellemenin ne kadar riskli olabileceği hakkında ilk fikir verir. Elbette bu sözleşmenin doğru uygulanması paket geliştiricisine bağlıdır; bu yüzden sürüm numarası tek başına garanti değildir.

[2] Raemaekers, S., van Deursen, A., & Visser, J. (2017). Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software*, *129*, 140–158. https://doi.org/10.1016/j.jss.2016.04.008

Bağımlılık yönetiminde bu kavram önemlidir. Çünkü `^4.18.2` gibi bir sürüm aralığı, paket yöneticisinin hangi güncellemeleri kabul edebileceğini belirler. Yani sürüm yazımı, projenizin gelecekte hangi değişiklikleri otomatik alabileceğini etkiler.

3.7.1 Ana, küçük ve yama sürümü (major, minor, patch)

Anlamsal versiyonlamada ana sürüm (major) genellikle uyumluluğu bozabilecek değişiklikleri, küçük sürüm (minor) geriye uyumlu yeni özellikleri, yama sürümü (patch) ise hata düzeltmelerini ifade eder.

Tablo 3.8: Ana, küçük ve yama sürümü (major, minor, patch)

Parça	Örnek	Genel anlam
Major	`2.0.0`	Uyumluluğu bozabilecek değişiklik
Minor	`2.1.0`	Geriye uyumlu yeni özellik
Patch	`2.1.1`	Hata veya güvenlik düzeltmesi
Ön sürüm (pre-release)	`2.0.0-beta.1`	Kararlı sürüm öncesi deneme sürümü

Tablo 3.8 düşünmeyi kolaylaştırır; ancak mutlak güvence değildir. Bazı paketler bu kurala sıkı uyar, bazıları daha gevşek davranır. Bu yüzden güncelleme yine test edilmelidir.

3.7.2 Versiyon aralıkları

Versiyon aralıkları, paket yöneticisine hangi sürümlerin kabul edilebilir olduğunu söyler. Örneğin `express` için `^4.18.2` yazıldığında, paket yöneticisi belirli kurallar içinde daha yeni `4.x` sürümlerini kabul edebilir. Tam sürüm yazmak ise daha sıkı kontrol sağlar; Tablo 3.9 bu üç yazımı ve taşıdıkları riski karşılaştırır.

Tablo 3.9: Versiyon aralıkları

Yazım	Genel fikir	Risk
`4.18.2`	Kesin sürüm	Güncellemeler otomatik gelmez
`^4.18.2`	Uyumlu kabul edilen yeni sürümler	Beklenmeyen küçük değişiklik alınabilir
`~4.18.2`	Daha dar güncelleme aralığı	Yama düzeyinde değişiklik alınabilir
`*` veya `latest`	Her sürüm kabul edilir	En yüksek risk, kontrolsüz güncelleme

Burada doğru seçim projenin ihtiyacına göre değişir. Kritik sistemlerde daha kontrollü güncelleme tercih edilebilir; hızlı gelişen küçük projelerde daha esnek aralıklar kullanılabilir.

3.7.3 Güncellemenin riski

Güncelleme hata düzeltmesi, yeni özellik veya güvenlik iyileştirmesi getirebilir; fakat çalışan bağımlılık ağacını da değiştirir. Doğrudan kullandığımız tek bir paketin yeni sürümü, onun getirdiği dolaylı bağımlılıkları da yenileyebilir. Major sürümlerde uyumluluğu bozan değişiklik olasılığı daha görünürdür; minor veya patch etiketi ise risksizlik garantisi vermez. Paket geliştiricisinin sürümleme hatası, kaldırılan bir davranış ya da çalışma zamanı uyumsuzluğu mevcut kodu etkileyebilir.

Güncelleme ayrı bir dalda ve sınırlı kapsamla yapılmalıdır. Paket değişiklik notları ile güvenlik duyuruları okunur, bağımlılık tarifindeki ve kilit dosyasındaki farklar incelenir. Ardından temiz kurulum yapılır; test, statik denetim (lint) ve derleme çalıştırılır; uygulamanın kritik akışları elden geçirilir. Çok sayıda paketi aynı anda güncellemek hata kaynağını belirsizleştirir; küçük ve anlamlı gruplar halinde ilerlemek hangi değişikliğin sonucu bozduğunu bulmayı kolaylaştırır. Sonuç sorunluysa bağımlılık tarifini ve kilit dosyasını birlikte geri almak, önceki ağaca dönmeyi sağlar.

Güncellemeyi hiç yapmamak da güvenli bir seçenek değildir. Bilinen güvenlik açığı bulunan bir paketi sırf mevcut kurulum çalışıyor diye bekletmek, özellikle internete açık sistemlerde daha büyük risk üretir. Bu nedenle güvenlik açığının ciddiyeti, paketin uygulamada gerçekten kullanıldığı yol ve sistemin dışarıya açıklığı birlikte değerlendirilir. Acil güvenlik düzeltmeleri önceliklendirilir; diğer güncellemeler ise planlı aralıklarla aynı doğrulama zincirinden geçirilir.

3.7.4 “Çalışıyordu, güncelledim bozuldu” vakası

“Çalışıyordu, güncelledim bozuldu” vakası bağımlılık yönetiminin en öğretici örneklerinden biridir. Diyelim ki ekip tüm paketleri güncelledi ve testler bozuldu:

```
$ npm update

$ npm test
3 tests failed
15 tests passed
```

Bu çıktı bize güncellemeden sonra en az üç testin beklenen davranışı doğrulamadığını gösterir. Bu durumda panikle tüm değişiklikleri silmek yerine, hangi paketlerin değiştiğini `git diff package-lock.json` ile incelemek gerekir. Eğer kilit dosyası Git’te tutuluyorsa, güncellemenin bağımlılık ağacına etkisi görülebilir. Böylece kilit dosyasının işlevi kurulumla sınırlı kalmaz: hangi paketin ne zaman değiştiğini gösteren bir kanıt kaydına da dönüşür.

3.8 Geliştirme, test ve üretim ortamları

Bir uygulamanın geliştirici bilgisayarında çalışması güzel bir başlangıçtır; fakat tek başına yeterli değildir. Uygulama test ortamında farklı veriyle denenebilir, üretim ortamında gerçek kullanıcılar ve gerçek verilerle çalışır. Bu ortamların aynı gibi düşünülmesi ciddi risk üretir.

Küçük bir vaka düşünelim. Ekip yeni raporlama özelliğini geliştiriyor. Geliştirme ortamında sahte verilerle test yapılıyor. Ancak yanlış ortam değişkeni nedeniyle test ortamı üretim veritabanına bağlanıyor. İlk bakışta uygulama çalışıyor gibi görünür; fakat aslında üretim verisine dokunma riski oluşmuştur. Burada bozulmuş varsayım şudur: "Test yapıyorum, gerçek sistemi etkilemiyorum."

Bu yüzden geliştirme (development), test ve üretim (production) ortamlarını ayırmak gerekir. Ayrım yalnızca teknik düzen değildir; veri gizliliği, müşteri güveni, iş sürekliliği ve operasyonel sorumlulukla ilgilidir.

Tablo 3.10: Geliştirme, test ve üretim ortamları

Ortam	Temel amaç	Kritik risk
Geliştirme	Özellik geliştirmek ve denemek	Kişisel ayarlara bağımlı kalmak
Test	Değişikliği kontrollü doğrulamak	Üretim verisine yanlışlıkla bağlanmak
Hazırlama	Üretime yakın son doğrulama	Gerçek veri veya gerçek entegrasyon sızıntısı
Üretim	Gerçek kullanıcıya hizmet vermek	Hatalı değişikliğin iş etkisi üretmesi

Tablo 3.10 ortam ayrımının neden yalnızca teknik tercih olmadığını gösterir.

Ortamların amacı

Geliştirme ortamı deneme ve yazım içindir; test ortamı doğrulama içindir; üretim ortamı gerçek kullanıcıya hizmet vermek içindir. Bu ayrım karar verme açısından önemlidir. Geliştirme ortamında hata öğrenme fırsatı olabilir; üretim ortamında aynı hata müşteri kaybına veya veri sorununa dönüşebilir. Bu yüzden ortam amacı net değilse risk yönetimi de zayıflar.

Ortama göre değişen ayarlar

Ortama göre veritabanı adresi, log seviyesi, ödeme sistemi, hata gösterimi, API anahtarı ve güvenlik ayarları değişebilir. Örneğin geliştirme ortamında `DEBUG=true` kabul edilebilir; fakat üretim ortamında bu ayar hassas hata bilgilerini açığa çıkarabilir. Bu yüzden ayarlar kod içine sabitlenmemeli, ortam değişkenleri ve konfigürasyon yönetimiyle kontrollü biçimde ayrılmalıdır.

3.8.1 `.env` ve `.env.example` kullanımı

`.env` dosyası gerçek ortam değerlerini taşıyabilir; `.env.example` ise hangi değerlerin gerektiğini gösteren paylaşılabilir örnektir. Bu ayrım küçük ama önemlidir.

```
# .env.example
APP_ENV=development
APP_PORT=3000
DATABASE_URL=postgres://user:password@localhost:5432/app
```

Bu dosya Git'e eklenebilir; çünkü gerçek parola değil, örnek yapı taşır. Gerçek `.env` dosyası ise genellikle Git dışında tutulmalıdır. Böylece ekip kurulum bilgisini paylaşır, fakat gizli bilgiyi paylaşmaz.

3.8.2 Üretim ortamına yanlışlıkla dokunma riski

Üretim ortamına yanlışlıkla dokunmak, geliştirme sürecindeki en ciddi operasyonel risklerden biridir. Yerelde çalıştırılan bir testin üretim veritabanına bağlanması gerçek veriyi silebilir veya kirletebilir. Üretim ödeme anahtarının geliştirme ortamında kullanılması gerçek işlem, kota ve maliyet doğurabilir. Hata ayıklamak için çalıştırılan sıradan bir script bile hedef ortamı yanlış seçtiğinde geri dönüşü zor bir iş olayına dönüşür.

Ortam adlarını farklı yazmak tek başına yeterli koruma sağlamaz. Geliştirme, test ve üretim için ayrı veritabanları, hizmet hesapları ve API anahtarları kullanılmalı; her kimliğe yalnızca ihtiyaç duyduğu yetki verilmelidir. Mümkünse üretim veritabanı geliştirici bilgisayarından doğrudan erişilebilir olmamalı, ağ ve güvenlik duvarı kuralları bu ayrımı desteklemelidir. Test ortamında gerçek müşteri verisi yerine sentetik veya uygun biçimde anonimleştirilmiş veri kullanılması da hatanın veri ihlaline dönüşmesini engeller.

Silme, toplu güncelleme ve şema değişikliği gibi işlemler hedefi açıkça göstermeli ve beklenmeyen ortamda güvenli biçimde durmalıdır. Üretim değişiklikleri yetkili bir dağıtım akışı, gözden geçirme ve gerektiğinde elle onay adımı üzerinden yürütülebilir. Uygulama başlarken etkin ortamın ve bağlandığı hizmetin adını gizli değerleri açığa çıkarmadan loglamak, yanlış bağlantıyı erken fark etmeyi kolaylaştırır. Yedek ve geri dönüş planı ise koruyucu kontrollerin yerini almaz; bu kontroller aşıldığında iş etkisini sınırlar.

Yanlış üretim erişimi fark edildiğinde işlem durdurulmalı, etkilenen veri ve hizmetler belirlenmeli, gerekiyorsa anahtarlar döndürülmeli ve olay kayda alınmalıdır. Ortam yönetimi bu nedenle yalnızca bir konfigürasyon ayrıntısı değil, veri güvenliği ve iş sürekliliği sorumluluğudur.

3.9 Çözüm olarak: Docker

Bu bölüm boyunca şunu gördük: Bir proje yalnızca kaynak koddan oluşmaz. Çalışma zamanı sürümü, paket yöneticisi, bağımlılık dosyaları, kilit dosyası, ortam değişkenleri, test komutları ve ortam ayrımları projenin çalışabilirliğini belirler. Başlangıçta bu parçalar ayrı ayrı yönetilebilir gibi görünür. Ancak ekip büyüdükçe ve proje farklı bilgisayarlarda çalıştırıldıkça karmaşıklık artar.

Docker bu noktada karşımıza çıkacak konulardan biridir. Docker'ı burada ayrıntılı anlatmayacağız; onu ilerleyen bölümde ele alacağız. Şimdilik bilmemiz gereken şey şudur: Docker, uygulamanın çalışması için gereken bazı ortam bilgilerini daha standart ve taşınabilir hale getirmeye yardım eder. Bu, "her şeyi çözer" anlamına gelmez; fakat "hangi çalışma zamanı, hangi bağımlılık, hangi servis, hangi ayar?" sorularını daha kontrollü bir paket içinde düşünmemizi sağlar.

Dolayısıyla Docker'a geçiş bir moda veya araç tercihi olarak görülmemelidir. Bu bölümde gördüğümüz karmaşıklığa verilen pratik bir yanıttır. Projenin kod dışındaki

parçalarını anlamadan Docker kullanmak, yalnızca karmaşıklığı başka bir dosyanın içine taşımak olur. Önce bu parçaları görmek, sonra standartlaştırma araçlarını değerlendirmek daha sağlıklı bir yoldur.

Bölüm 4

Sürüm Kontrol Sistemleri ve Git

Bir ekip projesinde kodun değişmesi en beklenen olaydır. Bugün çalışan bir ekran yarın yeni bir kuralla değişir. Bir dosyayı Emre düzenler, başka bir dosyayı Mehmet günceller, Zeynep hata düzeltir. Başlangıçta bu değişiklikleri dosya kopyalayarak veya mesajlaşma uygulamasından göndererek yönetmek mümkün gibi görünebilir.

Ancak proje büyüdükçe bu yöntem hızla dağılır. Hangi dosya güncel? Kim hangi satırı değiştirdi? Bir hata gelirse eski hale nasıl döneceğiz? İki kişi aynı dosyayı düzenlerse hangisi geçerli olacak? Geliştirmeyi tek kişi yapsa bile haftalar önceki bir kararın gerekçesini hatırlamak veya çalışan son duruma güvenle dönmek gerekir. Sürüm kontrolü bu yönüyle yalnızca ekip koordinasyonu değil, bireysel çalışma hafızası da sağlar. İşte sürüm kontrol sistemleri ve özellikle Git bu sorulara yanıt vermek için kullanılır.

Bu bölümde Git'i komutlar listesi olarak değil, takım hafızası olarak ele alacağız. Git, değişiklikleri kayda geçirir, geri dönmeyi mümkün kılar, farklı çalışma hatlarını yönetir ve ekip içinde değişikliklerin incelenebilir olmasını sağlar. GitHub ise bu hafızayı ekip çalışması, değişiklik isteği (pull request), inceleme (review) ve proje yönetimiyle daha görünür hale getirir.

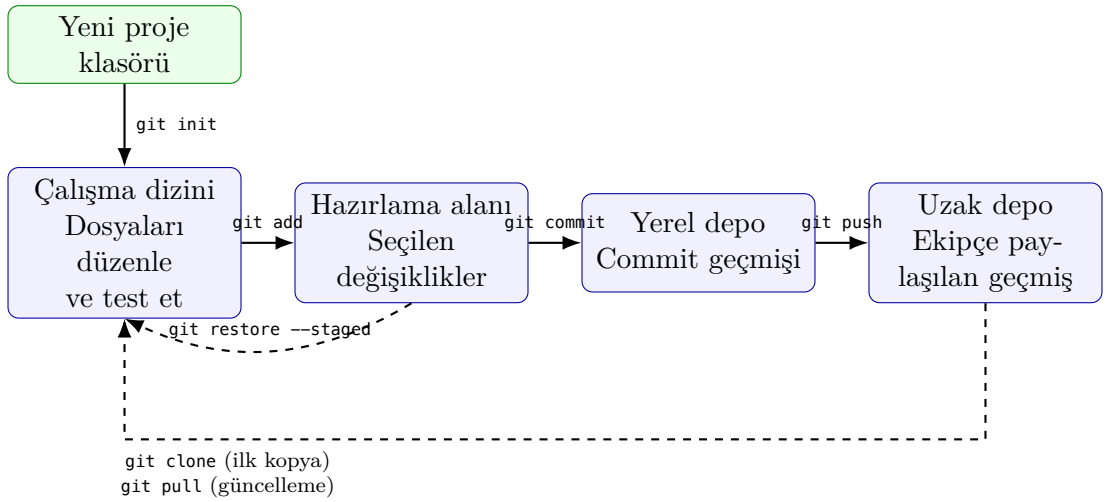
Git'in temel akışını başlangıçta şöyle düşünebiliriz:

Şekil 4.1 her adımı yalnızca teknik bir konum olarak göstermez. Çalışma dizini deneme yaptığınız alanı, hazırlama alanı hangi değişiklikleri kayda almak istediğinizi, yerel depo sizin bilgisayarınızdaki geçmişi, uzak depo ise ekiple paylaşılan ortak hafızayı temsil eder. Bu ayrımı kavramak, Git komutlarını ezberlemekten daha değerlidir.

4.1 Sürüm Kontrol Sistemi Nedir?

Bir ödev dosyasını düşünelim: `rapor.docx`, `rapor-son.docx`, `rapor-son-final.docx`, `rapor-final-gercek.docx`. Bu isimlendirme çoğumuza tanıdık gelir. Başlangıçta işe yarıyor gibi görünür; çünkü eski dosyayı kaybetmemek için yeni bir kopya oluştururuz. Ancak birkaç kişi aynı dosya üzerinde çalıştığında veya hangi değişikliğin ne zaman yapıldığını anlamamız gerektiğinde bu düzen yetersiz kalır.

Peki bu problemi daha sistemli nasıl çözeriz? Bu noktada sürüm kontrol sistemi (version control system, VCS) kavramı devreye girer. Sürüm kontrol sistemi, dosyalarda



Şekil 4.1: Git deposunun oluşturulması ve değişikliklerin uzak depoyla paylaşılma döngüsü

yapılan değişiklikleri zaman içinde kaydeden, bu değişiklikleri karşılaştırmayı, geri almayı ve ekip içinde paylaşmayı mümkün kılan sistemdir. Git, bugün yazılım geliştirme dünyasında en yaygın kullanılan sürüm kontrol sistemlerinden biridir.[1]

Burada dikkat edilmesi gereken nokta şudur: Git yalnızca dosya saklayan bir araç değildir. Git, değişikliklerin geçmişini ve bu değişikliklerin arkasındaki kararları izlenebilir hale getirir. GitHub ise Git depolarını internet üzerinde barındıran ve ekip çalışmasını kolaylaştıran bir platformdur. Git ile GitHub aynı şey değildir; fakat çoğu projede birlikte kullanılırlar.

Sürüm kontrolü, yalnızca yazılımcının komut satırında yaptığı teknik bir işlem değildir. Teslim kalitesi, ekip koordinasyonu, hata geri alma, sorumluluk takibi ve karar şeffaflığıyla doğrudan ilişkilidir. Bu sebeple teknik olduğu kadar yönetsel yönüyle de ele alınması gerekir.

Dosya kopyalayarak sürüm yönetmek küçük işlerde kısa süreli çözüm gibi görünür. Örneğin proje klasöründe şu dosyalar bulunabilir:

```
stok-app-final.zip
stok-app-final-2.zip
stok-app-sunum-oncesi.zip
stok-app-mehmet-duzenledi.zip
```

Bu liste ilk bakışta bir geçmiş tutuyor gibi görünür. Ancak hangi dosyada hangi değişiklik var, hangisi güvenilir, hangisi ekip tarafından onaylandı sorularına net cevap vermez. Üstelik gizli bilgi içeren dosyalar da bu kopyalar içinde dağılabilir. Bu yüzden dosya kopyaları geçmiş tutmak için değil, çoğu zaman belirsizlik üretmek için çalışır.

Değişiklik geçmişi, projenin zaman içinde nasıl evrildiğini gösterir. Bir dosyanın bugünkü halini görmek önemlidir; fakat bazen asıl soru bugünkü hale nasıl gelindiğidir.

[1] German, D. M., Adams, B., & Hassan, A. E. (2016). Continuously mining distributed version control systems: An empirical study of how Linux uses Git. *Empirical Software Engineering*, *21*(1), 260–299. https://doi.org/10.1007/s10664-014-9356-2

Git geçmişi bize hangi değişikliğin ne zaman yapıldığını, hangi mesajla kayda geçtiğini ve gerektiğinde hangi noktaya dönülebileceğini gösterir. Bu, ekip hafızası açısından çok değerlidir.

"Kim, neyi, ne zaman değiştirdi?" sorusu hata aramak için olduğu kadar sorumluluğu ve iletişimi anlamak için de önemlidir. Git bu soruya kişisel suçlama üretmek için değil, değişikliğin bağlamını görmek için cevap verir. Örneğin bir rapor ekranı bozulduysa, ilgili dosyada son değişikliği kimin yaptığı ve commit mesajında ne yazdığı sorunu daha hızlı anlamamıza yardım eder.

Yazılım geliştirmede her değişiklik doğru sonuç vermeyebilir. Bazen bir özellik bozulur, bazen yanlış dosya düzenlenir, bazen de karar değişir. Geri dönebilme ihtiyacı bu yüzden kritiktir. Git, değişiklikleri anlamlı kayıtlar halinde tuttuğu için belirli bir değişikliği incelemeyi, geri almayı veya yeni bir düzeltme ile etkisini tersine çevirmeyi mümkün kılar. Bu, proje riskini azaltan temel mekanizmalardan biridir.

4.2 Sürüm Kontrol Sistemi Türleri

Sürüm kontrol sistemleri farklı çalışma modelleriyle kurulabilir. Her model ekip koordinasyonunu farklı biçimde etkiler. Bazı sistemlerde geçmiş yalnızca yerel makinede tutulur, bazılarında merkezi bir sunucu temel alınır, bazılarında ise her geliştiricide tam geçmişin bir kopyası bulunur.

Bu ayrımı bir iş akışı olarak düşünelim:

1. Yerel modelde kişi kendi bilgisayarındaki dosyaların geçmişini tutar; ekip paylaşımı zayıftır.
2. Merkezi modelde ekip ortak bir sunucuya bağlı çalışır; merkez önem kazanır.
3. Dağıtık modelde her geliştirici geçmişin tam kopyasına sahiptir; çevrimdışı çalışma ve dallanma daha esnek hale gelir.

Git dağıtık sürüm kontrol sistemi olarak bu üçüncü modele aittir. Bu sayede geliştirici kendi bilgisayarında commit geçmişiyle çalışabilir, sonra değişikliklerini uzak depoya gönderebilir. GitHub gibi platformlar da bu uzak depo ve takım koordinasyonu tarafını güçlendirir.

Bu, Git'in yalnızca GitHub'a dosya göndermekten ibaret olmadığı anlamına gelir. Git bilgisayarınızda da çalışır; commit oluşturabilir, geçmişi inceleyebilir, dal açabilir ve değişiklikleri karşılaştırabilirsiniz. GitHub ise bu yerel geçmişin ekip içinde paylaşılmasını, incelenmesini ve yönetilmesini kolaylaştırır.

Git'i öğrenirken en çok kafa karıştıran noktalardan biri şudur: Dosyayı kaydettim, ama Git neden hâlâ başka bir şey istiyor? Çünkü Git'te dosyayı editörde kaydetmek ile değişikliği proje geçmişine almak aynı şey değildir. Git, değişikliğinizi birkaç aşamada işler.

Bu aşamalar çalışma dizini, hazırlama alanı, yerel depo ve uzak depo olarak özetlenebilir. Çalışma dizininde dosyalar üzerinde değişiklik yaparsınız. Hazırlama alanında hangi değişiklikleri bir sonraki kayda dahil edeceğinizi seçersiniz. Yerel depoda commit geçmişiniz oluşur. Uzak depo ise bu geçmişin ekip ile paylaşılan kopyasıdır. Bu dört alan arasındaki geçişler Şekil 4.2 üzerinde bir arada görülebilir.

Şekil 4.2: Git'in dört çalışma alanı

Bu modelin güzelliği şudur: Git size her değişikliği otomatik olarak geçmişe yazdırmaz. Önce seçme, sonra açıklama, sonra paylaşma imkanı verir. Bu da değişikliği teknik işlem olmaktan çıkarıp bilinçli bir karar haline getirir.

Çalışma dizini (working directory)

Çalışma dizini, proje dosyalarını düzenlediğiniz alandır. Editörde dosyayı değiştirirsiniz, yeni dosya oluşturursunuz veya bir dosyayı silersiniz. Bu değişiklikler henüz Git geçmişine alınmış değildir. Çalışma dizini deneme ve yazım alanıdır; burada yapılan her şeyin kayıt altına alınması gerekmez, fakat neyin kayda alınacağı bilinçli seçilmelidir.

Hazırlama alanı (staging area)

Hazırlama alanı, bir sonraki commit'e hangi değişikliklerin gireceğini seçtiğiniz ara bölgedir. `git add` komutu bu alanı doldurur. Bu alanın değeri özellikle karmaşık değişikliklerde görünür olur. Aynı anda hem hata düzeltip hem dokümantasyon güncellediyseniz, bunları ayrı commit'lere bölmek için hazırlama alanından faydalanabilirsiniz.

Yerel depo (local repository)

Yerel depo, sizin bilgisayarınızdaki Git geçmişidir. Commit oluşturduğunuzda bu kayıt önce yerel depoda oluşur. İnternet bağlantısı olmadan da geçmişi inceleyebilir veya yeni commit oluşturabilirsiniz. Ancak ekip çalışması için bu kayıtların uzak depoyla paylaşılması gerekir. Yerel depo kişisel çalışma hafızasıdır; tek başına takım hafızası değildir.

Uzak depo (remote repository)

Uzak depo, GitHub gibi bir platform üzerinde ekip tarafından paylaşılan depo kopyasıdır. `git push` ile yerel commit'leri uzak depoya gönderir, `git pull` ile uzak depodaki değişiklikleri alırsınız. Uzak depo ekip koordinasyonunun merkezidir; fakat Git'in kendisi yalnızca uzak depoda çalışan bir araç değildir.

4.2.1 Commit geçmişi

Commit geçmişi, projenin karar ve değişiklik kayıtlarını taşır. Bu geçmişi görmek için `git log` kullanılabilir:

```
$ git log --oneline
a3f21c8 Add login validation
9b4d112 Update README setup steps
61e0a7f Initial project structure
```

Bu çıktı üç commit gösterir. En üstteki en yeni kayıttır. Mesajlar bize yalnızca dosya değiştiğini değil, değişikliğin ne amaçla yapıldığını da anlatır. İyi commit mesajı geçmişi okunabilir hale getirir; kötü mesaj ise geçmişi teknik gürültüye dönüştürür.

4.3 İlk Repository ve Temel Git Döngüsü

Bir projede Git kullanmaya başladığınızda temel döngü oldukça sadedir: Depoyu oluşturur veya klonlarsınız, değişiklik yaparsınız, durumu kontrol edersiniz, değişikliği hazırlama alanına alırsınız, commit oluşturursunuz ve gerekiyorsa uzak depoya gönderirsiniz. Bu adımların her biri bir karar noktasıdır.

Temel döngüyü şöyle özetleyebiliriz:

1. Depoyu al veya oluştur: Projenin resmi başlangıç noktasını belirler.
2. Değişiklik yap: Çalışma dizininde dosyalar değişir.
3. Durumu kontrol et: Hangi dosyaların değiştiği görülür.
4. Değişikliği hazırla: Bir sonraki commit'e ne gireceği seçilir.
5. Commit oluştur: Değişiklik anlamlı bir kayıt haline gelir.
6. Uzak depoya gönder: Ekip hafızası güncellenir.

Bu döngü basit görünür; fakat disiplinli uygulandığında ekip kalitesini doğrudan etkiler. Hangi değişikliğin ne zaman ve neden yapıldığı görünür hale gelir.

Şekil 4.1, bu yaşam döngüsünün iki başlangıç yolunu birlikte gösterir: Yeni bir proje `git init` ile yerel depoya dönüştürülür; var olan proje ise `git clone` ile uzak depodan alınır. Her iki yol da değişikliklerin seçilip commit'e dönüştürüldüğü ve uzak depoyla paylaşıldığı aynı çalışma döngüsüne bağlanır.

4.3.1 Depo (repository) oluşturmak veya klonlamak

Depo oluşturmak, mevcut klasörü Git tarafından izlenebilir hale getirmek demektir. Klonlamak ise var olan uzak depoyu bilgisayarınıza almaktır. Ekip projelerinde genellikle klonlama ile başlanır; çünkü projenin resmi kaynağı GitHub gibi bir platformdadır.

Henüz uzak deposu olmayan yeni bir proje klasörü için `git init` kullanılır:

```
$ mkdir stok-app
$ cd stok-app
$ git init -b main
Initialized empty Git repository in /home/akadal/projects/stok-app/.git/
```

`git init -b main`, bulunduğumuz dizinde gizli `.git` dizinini oluşturur ve ilk dalın adını `main` olarak belirler. Proje dosyaları henüz commit edilmemiştir ve herhangi bir uzak depoyla bağlantı kurulmamıştır.

GitHub gibi bir platformda depo zaten varsa aynı hedef klasörde yeniden `git init` çalıştırmak yerine depo klonlanır:

```
$ git clone https://github.com/akadal/blockchain.git
Cloning into 'blockchain'...
```

`git clone` komutu `blockchain` dizinini kendisi oluşturur; proje dosyalarını, commit geçmişini ve varsayılan uzak depo bağlantısını birlikte getirir. Kısacası sıfırdan yerel başlayan proje için `init`, mevcut uzak depodan başlayan çalışma için `clone` seçilir.

Değişiklik durumunu görmek

Değişiklik durumunu görmek için `git status` kullanılır. Bu komut çalışma dizininde nelerin değiştiğini, nelerin hazırlama alanına alındığını ve nelerin henüz izlenmediğini gösterir. Git kullanırken `status` komutu küçük bir kontrol alışkanlığıdır. Ne yaptığınızı bilmeden commit oluşturmayı engeller.

4.3.2 Değişikliği hazırlamak ve commit'e çevirmek

Değişikliği commit'e çevirmek için önce hangi dosyaların kayda alınacağını seçeriz, sonra bu değişikliği açıklayan bir commit oluştururuz.

```
$ git add README.md
$ git commit -m "Add setup instructions"
[main 9b4d112] Add setup instructions
 1 file changed, 12 insertions(+)
```

İlk komut `README.md` dosyasını hazırlama alanına alır. İkinci komut bu değişikliği commit haline getirir. Çıktı bize commit'in hangi dal üzerinde oluştuğunu, kısa kimliğini ve kaç dosyada ne kadar değişiklik olduğunu gösterir. Bu küçük kayıt, proje geçmişinde kalıcı bir iz bırakır.

4.3.3 İlk commit'in anlamı

İlk commit, projenin Git geçmişindeki başlangıç noktasıdır. Çoğu zaman proje iskeleti, README, bağımlılık dosyaları ve temel klasör yapısı bu commit ile kayda geçer.

```
$ git commit -m "Initial project structure"
[main 61e0a7f] Initial project structure
 4 files changed, 38 insertions(+)
 create mode 100644 README.md
 create mode 100644 package.json
```

Bu çıktı bize projenin ilk yapısının kayda geçtiğini söyler. İyi bir ilk commit, sonraki değişikliklerin neyin üzerine kurulduğunu anlamayı kolaylaştırır.

4.4 Commit Kültürü

Commit oluşturmak teknik olarak kolaydır. Birkaç komutla değişiklik kayda alınır. Ancak iyi commit oluşturmak, ekip çalışması açısından ayrı bir kültür gerektirir. Çünkü commit yalnızca bugünkü işi kaydetmez; gelecekte bu değişikliği okuyacak kişiye de açıklama bırakır.

Kötü commit geçmişi, ekip hafızasını karıştırır. `update`, `fix`, `son hali`, `deneme` gibi mesajlar bir süre sonra hiçbir şey anlatmaz. İyi commit ise değişikliğin niyetini kısa ve açık biçimde gösterir.

İyi commit mesajı değişikliğin ne yaptığını anlaşılır biçimde söyler. Çok uzun olmak zorunda değildir; fakat belirsiz olmamalıdır.

```
$ git commit -m "Add stock report date filter"
```

Bu mesaj bize stok raporuna tarih filtresi eklendiğini söyler. "Update files" gibi bir mesaj ise aynı bilgiyi vermez. Gelecekte hata arayan biri için iyi mesaj zaman kazandırır.

4.4.1 Atomik commit

Atomik commit, tek bir anlamlı değişikliği içeren commit'tir. Örneğin hem giriş ekranını düzeltip hem veritabanı ayarlarını değiştirip hem README güncellemesini aynı commit'e koymak geçmişi okumayı zorlaştırır. Daha sağlıklı yaklaşım, ilişkili değişiklikleri bir arada tutmak ve farklı amaçları ayrı commit'lere bölmektir.

```
$ git status
modified: src/login.js
modified: README.md
```

Bu çıktı iki dosyanın değiştiğini gösterir. Eğer `login.js` hata düzeltmesi, `README.md` ise kurulum açıklamasıysa bunları ayrı commit'lere almak daha okunabilir bir geçmiş oluşturabilir.

4.4.2 Ne zaman commit oluşturulur, ne zaman oluşturulmaz?

Commit, anlamlı ve açıklanabilir bir değişiklik tamamlandığında oluşturulmalıdır. Kod henüz çalışmıyorsa, test edilmediyse veya değişiklik ne yaptığı belli olmayan bir deneme halindeyse commit için erken olabilir. Elbette uzun işlerde ara commit gerekebilir; fakat bu commit'ler de anlaşılır olmalıdır.

```
$ npm test
18 tests passed
0 tests failed
```

Bu çıktı commit öncesi iyi bir işaret olabilir. Değişiklik testlerden geçiyorsa ve commit mesajıyla açıklanabiliyorsa kayıt altına almak için uygun noktaya gelmiş olabiliriz.

4.4.3 Commit geçmişini okunabilir tutmak

Commit geçmişini okunabilir tutmak, gelecekteki hata ayıklama ve karar inceleme süreçlerini kolaylaştırır. Aşağıdaki çıktıya bakalım:

```
$ git log --oneline
f18a20c Fix report export encoding
9b4d112 Add setup instructions
61e0a7f Initial project structure
```

Bu geçmişte her satır değişikliğin amacı hakkında fikir verir. Eğer aynı geçmiş `fix`, `update`, `stuff` gibi mesajlardan oluşsaydı, hangi değişikliğin neyi etkilediğini anlamak çok daha zor olurdu. Okunabilir geçmiş, takımın zaman kazanmasını sağlar.

Commit'ten sürüm paketine

Tek tek commit'ler çalışma günlüğüdür; kullanıcıya dönük değer ise bu kayıtların anlamlı bir bütün halinde paketlenmesiyle ortaya çıkar. Ekip "bu dönem sonunda hangi değişiklikler kullanıcıya gidecek?" diye sorduğunda cevap commit geçmişinden okunur: biriken anlamlı değişiklik kayıtları belirli bir noktada etiketlenir ve bir sürüm paketi (release) olarak açıklanır. Sürüm numarasını seçerken anlamsal versiyonlamayı (Kısım 3.7) hatırlamak faydalıdır: hata düzeltmeleri yama (patch) sürümünü, geriye uyumlu yeni özellikler ara (minor) sürümü, geriye uyumsuz değişiklikler ise ana (major) sürümü artırır. Bu bakış commit disiplini ile sürüm yönetimini birbirine bağlar: mesajlar ne kadar anlaşılır olursa sürüm notunu yazmak ve numarayı seçmek o kadar kolaylaşır. Etiket, sürüm paketi ve değişiklik günlüğü konularını bu kısmın ilerleyen başlıklarında ayrıca ele alacağız.

4.5 Geçmişi ve Değişiklikleri İnceleme

Bir hata ortaya çıktığında çoğu zaman yalnızca bugünkü dosyaya bakmak yetmez. Hatanın ne zaman geldiğini, hangi değişiklikle başladığını ve hangi kararın sonucu olduğunu anlamak gerekir. Git geçmişi ve değişiklik inceleme komutları bu noktada devreye girer.

`git log` geçmişi gösterir, `git diff` henüz commit edilmemiş veya iki nokta arasındaki değişiklikleri gösterir, `git show` belirli bir commit'in içeriğini incelemeyi sağlar. Bu komutları ezberden çok okuma alışkanlığı olarak düşünmek gerekir.

Değişiklik inceleme, teknik ayrıntının ötesinde bir karar okuma pratiğidir. Hangi dosya değişmiş? Neden değişmiş? Mesaj ne söylüyor? Test veya dokümantasyon da değişmiş mi? Bu sorular proje kalitesi açısından önemlidir.

GitHub üzerindeki commit geçmişi de aynı bilgiyi çevrimiçi bir arayüzde gösterir. Şekil 4.3 her satırda mesaj, yazar, zaman ve kısa commit kimliğini birlikte okutur.

4.5.1 Log, diff ve show çıktısını okumak

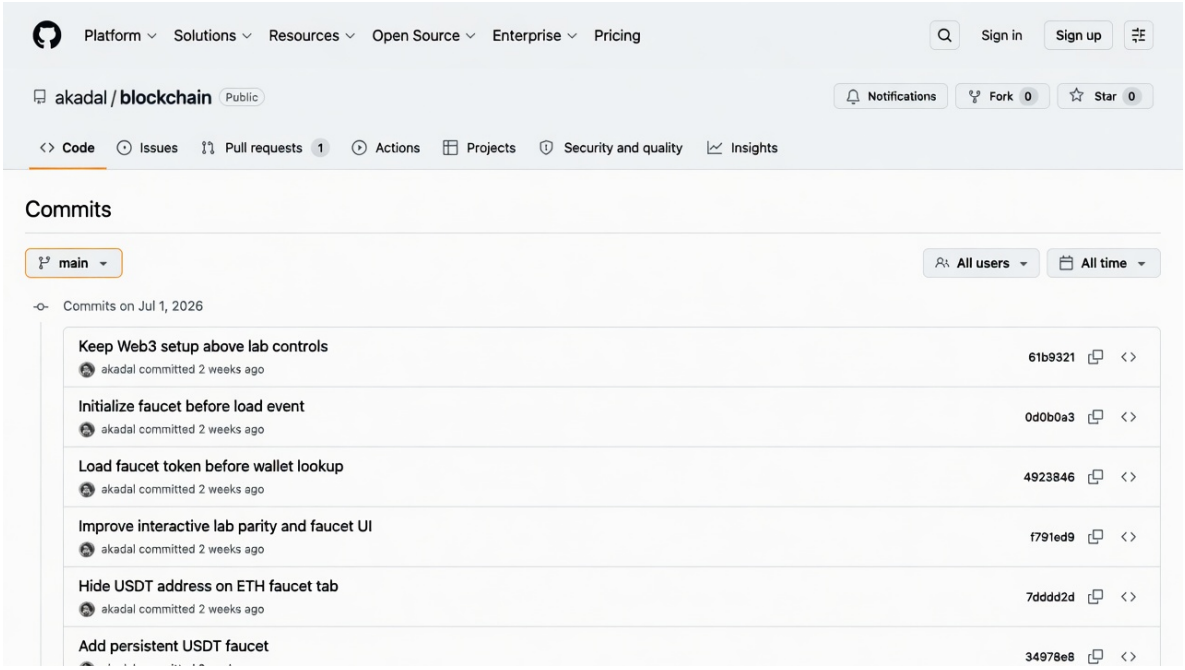
Küçük bir `git diff` çıktısı bile çok şey anlatır:

```
$ git diff
- const taxRate = 0.18
+ const taxRate = 0.20
```

Bu çıktı bir satırın değiştiğini gösterir. `-` eski satırı, `+` yeni satırı ifade eder. Teknik olarak vergi oranı değişmiştir; fakat karar açısından soru şudur: Bu değişiklik hangi mevzuat, iş kuralı veya müşteri talebi nedeniyle yapıldı? Commit mesajı ve varsa issue bağlantısı bu bağlamı tamamlamalıdır.

`git show` belirli bir commit'i incelemek için kullanılabilir:

```
$ git show f18a20c --stat
src/report.js | 4 ++--
1 file changed, 2 insertions(+), 2 deletions(-)
```



Şekil 4.3: `akadal/blockchain` deposunun commit geçmişinde mesaj, yazar ve kısa commit kimlikleri

Bu çıktı commit'in hangi dosyada ne kadar değişiklik yaptığını özetler. Ayrıntıya girmeden önce etkinin büyüklüğünü görmeyi sağlar.

Değişikliğin teknik değil karar geçmişi olarak görülmesi

Git geçmişini yalnızca satır değişiklikleri olarak görmek eksik olur. Her değişiklik bir kararın izidir: Bir hata düzeltilmiştir, bir iş kuralı değişmiştir, bir güvenlik önlemi eklenmiştir veya bir tasarım tercihi yapılmıştır. Git geçmişinin değeri buradadır. Geçmişi okuyabilen kişi, teknik değişiklik ile iş kararı arasındaki ilişkiyi daha sağlıklı kurabilir.

4.6 Dal (Branch) Kullanımı

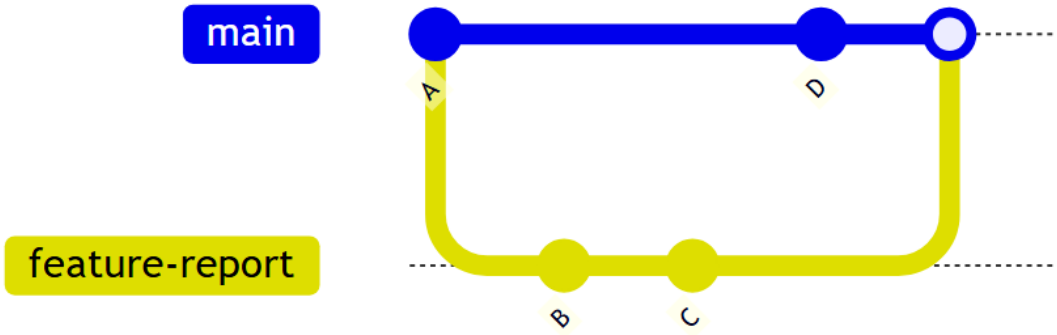
Bir proje üzerinde çalışırken her değişikliği doğrudan ana dala yapmak riskli olabilir. Yeni bir özellik geliştiriyorsunuz, fakat bu özellik henüz tamamlanmadı. Aynı anda başka biri hata düzeltiyor, bir başkası dokümantasyonu güncelliyor. Herkes aynı ana çizgide çalışırsa karışıklık çıkması kolaylaşır.

Dal (branch), bu problemi yönetmek için kullanılan çalışma hattıdır. Ana proje geçmişinden ayrılan geçici veya kalıcı bir çizgi gibi düşünülebilir. Bu çizgide değişiklik yapar, test eder, tartışır ve hazır olduğunda ana dala birleştirirsiniz.

Şekil 4.4 basit bir dal görünümünü gösterir.

Bu görselde `feature-report` dalı ana daldan ayrılır, kendi değişikliklerini taşır ve daha sonra ana dala birleşir. Bu yaklaşım, değişiklikleri izole etmeyi ve incelemeyi kolaylaştırır.

Dal, Git geçmişinde belirli bir noktadan başlayan ve değişiklikleri ayrı bir hatta taşıyan



Şekil 4.4: Bir özellik dalının ana daldan ayrılması ve yeniden birleşmesi

işaretçi gibi düşünülebilir. Gündelik bir örnekle, ortak raporun ana kopyasını bozmadan kendi taslak kopyanız üzerinde çalışmak gibidir. Ancak Git'te bu kopya bilinçli bir geçmiş ilişkisi taşır; ana daldan ne zaman ayrıldığı ve nasıl geri birleşeceği izlenebilir.

Dal kullanmak tek başına düzen garantisi vermez. Dalların ne için açıldığı, nasıl adlandırıldığı ve ne zaman ana dala alınacağı ekip kuralı haline gelmelidir.

Özellik dalı (feature branch) yaklaşımı

Özellik dalı yaklaşımında her yeni özellik veya anlamlı değişiklik ayrı bir dalda geliştirilir. Örneğin `feature/report-filter` dalında rapor filtresi geliştirilir, `fix/login-validation` dalında giriş doğrulama hatası düzeltilir. Bu yaklaşım ana dalı daha kararlı tutar ve değişiklik isteği üzerinden değişiklik incelemeyi kolaylaştırır. Ancak çok uzun yaşayan dallar ana daldan uzaklaşabilir; bu yüzden düzenli güncelleme ve küçük değişiklik alışkanlığı önemlidir.

Dal isimlendirme standardı

Dal isimleri ekibin çalışma düzenini görünür kılar. `feature/report-filter`, `fix/payment-timeout`, `docs/setup-guide` gibi adlar dalın amacını hızlıca anlatır. `deneme`, `son`, `akadal-branch` gibi adlar ise zamanla belirsizlik üretir. İsimlendirme standardı küçük bir ayrıntı gibi görünür; fakat ekip büyüdükçe hangi değişikliğin hangi amaçla yapıldığını anlamayı kolaylaştırır.

4.6.1 Dal değiştirme ve çalışma alanını koruma

Dal değiştirirken çalışma dizinindeki mevcut değişiklikler önemlidir. Henüz commit edilmemiş değişiklikler varsa Git bazen dal değiştirmenize izin vermez veya değişikliklerin yeni dalda da görünmesine neden olabilir. Bu davranış ilk anda şaşırtıcıdır; fakat Git aslında veri kaybını önlemeye çalışır.

Dal değiştirmeden önce `git status` çalıştırmak iyi bir alışkanlıktır:

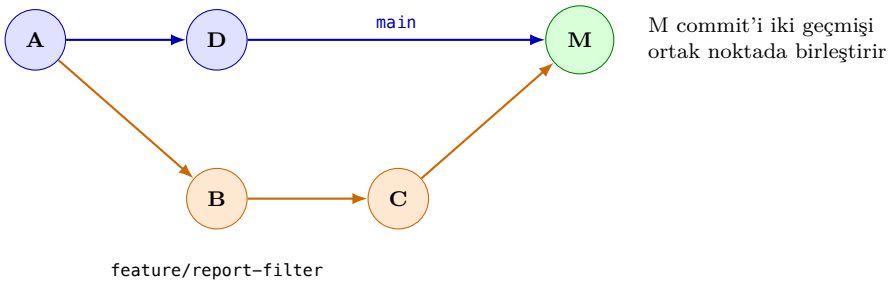
```
$ git status
Changes not staged for commit:
  modified: src/report.js
```

Bu çıktı bize çalışma alanında kayda alınmamış değişiklik olduğunu gösterir. Bu durumda önce commit oluşturmak, değişikliği geçici olarak saklamak veya bilinçli biçimde geri almak gerekir. Hangi yolun seçileceği ekip akışına ve değişikliğin durumuna bağlıdır.

4.7 Birleştirme (Merge) ve Çakışma (Conflict) Çözme

Dallar ayrı çalışma hatları sağladığı için bir noktada bu hatların tekrar birleşmesi gerekir. Bir özellik tamamlandığında, hata düzeltildiğinde veya dokümantasyon güncellendiğinde ilgili dal ana dala alınır. Bu işleme birleştirme (merge) denir.

Şekil 4.5, özellik dalının ana dal ilerlemişken nasıl birleştiğini gösterir.



Şekil 4.5: Ana dal ilerlemişken özellik dalının merge commit ile birleştirilmesi

Şekilde A commit'inden sonra `main` üzerinde D, özellik dalında ise B ve C commit'leri oluşmuştur. M commit'i bu iki hattı ortak geçmişte buluşturur ve iki ebeveyn taşır. Ana dal özellik dalı ayrıldıktan sonra ilerlememiş olsaydı Git, ayrı bir merge commit oluşturmadan dal işaretçisini ileri taşıyan *fast-forward* birleştirme de yapabilirdi.

Ancak iki kişi aynı dosyanın aynı bölümünü farklı biçimde değiştirdiyse Git hangi değişikliğin doğru olduğunu kendisi bilemeyebilir. Bu durumda çakışma (conflict) oluşur. Çakışma Git'in bozulduğu anlamına gelmez; Git'in insandan karar beklediği anlamına gelir.

Bir çakışma çözme süreci genellikle şu adımlarla ilerler:

1. Git hangi dosyada çakışma olduğunu gösterir.
2. Dosya açılır ve çakışma işaretleri okunur.
3. Hangi değişikliğin kalacağına veya nasıl birleştirileceğine karar verilir.
4. Çakışma işaretleri temizlenir.
5. Dosya test edilir.
6. Çözüm commit ile kayda alınır.

Bu süreç teknik olduğu kadar iletişimsel bir süreçtir. Bazen doğru çözüm kodu bilen kişiye sormayı, iş kuralını kontrol etmeyi veya test sonucuna bakmayı gerektirir.

4.7.1 Birleştirme (merge) nedir?

Merge, iki ayrı çalışma hattını ortak bir geçmişte birleştirme işlemidir. Örneğin `feature/report-filter` dalında geliştirilen rapor filtresi tamamlandıktan sonra `main` dalına alınabilir.

```
$ git checkout main
$ git merge feature/report-filter
```

Bu komutlar önce ana dala geçer, sonra özellik dalındaki değişiklikleri ana dala birleştirmeye çalışır. Merge işlemi, ayrı hatta yapılan çalışmayı ana proje geçmişine dahil eder.

4.7.2 Çakışma (conflict) neden çıkar?

Çakışma genellikle iki ayrı değişiklik aynı dosyanın aynı veya yakın bölgesine müdahale ettiğinde çıkar. Örneğin Emre rapor başlığını "Aylık Stok Raporu" yapmış, Mehmet aynı satırı "Stok Durumu Raporu" olarak değiştirmiş olabilir. Git iki değişikliği de görür; fakat hangisinin iş kararı olarak doğru olduğunu bilemez.

Bu durum özellikle teslim tarihi yaklaşırken stres üretir. Ancak çakışma kötü bir şey olmak zorunda değildir. Bazen ekip içinde aynı alan üzerinde koordinasyon eksikliği olduğunu görünür hale getirir. Bu yüzden çakışmayı yalnızca teknik hata değil, takım iletişimi sinyali olarak da okumak gerekir.

4.7.3 Çakışma dosyası nasıl okunur?

Çakışma olduğunda Git dosyanın içine özel işaretler koyar:

```
# <<<<<<< HEAD
title = "Aylık Stok Raporu"
# =======
title = "Stok Durumu Raporu"
# >>>>>>> feature/report-title
```

Burada başında `#` ile gösterdiğimiz satırlar, gerçek dosyada Git'in koyduğu çakışma işaretlerini temsil eder. `HEAD` bulunduğunuz dalın değişikliğini, alt bölüm ise gelen dalın değişikliğini gösterir. Çözüm, bu işaretleri silip doğru son hali bırakmaktır. Örneğin iş kuralına göre başlık "Aylık Stok Durumu Raporu" olmalıysa dosya şu hale getirilir:

```
title = "Aylık Stok Durumu Raporu"
```

Bu işlemden sonra dosyanın çalıştığı test edilmeli ve çözüm commit edilmelidir. Çakışma çözmek yalnızca metin seçmek değil, doğru iş kararını dosyaya yansıtmaktır.

Takım içinde conflict azaltma yöntemleri

Takım içinde çakışmaları azaltmak için küçük ve sık commit yapmak, uzun yaşayan dallardan kaçınmak, aynı dosya üzerinde çalışan kişilerin iletişim kurması ve değişiklik isteklerini çok büyütmemek gerekir. Ayrıca kodun mantıklı parçalara ayrılması da

çakışma riskini azaltır. Burada asıl mesele teknik değildir: Çakışma sayısı, takım koordinasyonu ve görev paylaşımı hakkında önemli bir göstergedir.

4.8 Rebase Kavramı

Merge, iki çalışma hattını birleştirirken dallanmayı geçmişte görünür bırakır. Uzun süre çalışan bir dalda bu dallanma birikince geçmişi okumak zorlaşabilir; bazı ekipler bu yüzden daha düz, tek çizgi halinde ilerleyen bir geçmiş tercih eder. Rebase, Git'te geçmişi düzenli tutmak için kullanılan güçlü ama dikkat isteyen bir işlemdir. Basitçe söylemek gerekirse, bir dalda yaptığınız commit'leri başka bir tabanın üzerine yeniden yerleştirir. Bu, geçmişi daha düz gösterebilir; fakat geçmişi değiştirme riski taşıdığı için bilinçli kullanılmalıdır.

Örneğin siz bir özellik dalında çalışırken `main` dalında yeni commit'ler oluşmuş olabilir. Rebase ile kendi commit'lerinizi güncel `main` üzerine yeniden oturtabilirsiniz. Ancak bu işlem commit kimliklerini değiştirebilir. Bu yüzden özellikle başkalarıyla paylaşılmış dallarda rebase yapmak dikkat ister.

4.8.1 Rebase neyi değiştirir?

Rebase, commit'lerinizi yeni bir tabanın üzerine yeniden uygular. Bu nedenle geçmiş çizgisi daha düz görünebilir.

```
$ git rebase main
Successfully rebased and updated refs/heads/feature/report-filter.
```

Bu çıktı, bulunduğunuz dalın `main` üzerine yeniden oturtulduğunu gösterir. Ancak burada önemli bir ayrıntı vardır: Rebase sonrasında commit kimlikleri değişebilir. Bu yüzden işlem yalnızca "temiz geçmiş" üretmez; aynı zamanda geçmişi yeniden yazar.

4.8.2 Merge ile farkı

Merge ve rebase benzer bir problemi çözer: Ayrılmış çalışma hattını güncel ana dalla ilişkilendirmek. Fakat Tablo 4.1 içinde gösterildiği gibi geçmişe etkileri farklıdır.

Tablo 4.1: Merge ve rebase işlemlerinin geçmişe etkisi ve dikkat noktaları

İşlem	Geçmişe etkisi	Dikkat noktası
Merge	Birleştirme commit'i oluşturabilir	Geçmiş dallanmayı açık gösterir
Rebase	Commit'leri yeni tabana taşır	Geçmişi yeniden yazabilir
Paylaşılmış dalda rebase	Ortak geçmiş bozulabilir	Önce ekip kuralı gerekir
Henüz gönderilmemiş yerel dal	Rebase geçmişi sadeleştirebilir	Push edilmeden daha güvenlidir

Hangisinin daha iyi olduğu proje kuralına bağlıdır. Bazı ekipler açık merge geçmişini sever, bazıları daha düz geçmiş ister. Önemli olan ekip içinde tutarlı davranmaktır.

Ne zaman kullanılmalı, ne zaman uzak durulmalı?

Rebase, çoğunlukla kendi yerel dalınızı güncel ana dala yaklaştırmak için kullanılabilir. Ancak başkalarının da kullandığı, uzak depoya gönderilmiş ve üzerine çalışma yapılmış dallarda rebase risklidir; çünkü başkalarının geçmiş algısını bozabilir. Ders ve küçük takım projelerinde rebase kullanımı gerekiyorsa önce ekip kuralı belirlenmeli, özellikle `main` gibi ortak dallarda geçmiş değiştiren işlemlerden uzak durulmalıdır.

4.9 Geri Alma Operasyonları

Git kullanırken hata yapmak normaldir. Yanlış dosyayı değiştirebilir, eksik commit oluşturabilir veya üretime gitmemesi gereken bir değişikliği fark edebilirsiniz. Bu durumda önemli olan paniğe kapılmadan hangi aşamada olduğunuzu anlamaktır. Değişiklik çalışma alanında mı, hazırlama alanında mı, commit geçmişinde mi, yoksa uzak depoya gönderildi mi?

Geri alma operasyonlarında temel ayrım şudur: Bazı işlemler geçmişe dokunmadan yeni bir düzeltme kaydı oluşturur, bazıları ise geçmişi değiştirir. Ekip çalışmasında genellikle geçmişi değiştirmeyen yollar daha güvenlidir. Özellikle uzak depoya gönderilmiş commit'lerde dikkatli olmak gerekir.

4.9.1 Çalışma alanındaki değişikliği geri almak

Çalışma alanındaki değişikliği geri almak, henüz commit edilmemiş bir düzenlemeyi iptal etmek anlamına gelir. Örneğin bir dosyada deneme yaptınız ve bu denemeyi istemiyorsunuz. Bu durumda önce `git status` ile hangi dosyanın değiştiğini görmek gerekir.

```
$ git status
modified: src/report.js
```

Bu çıktı yalnızca `src/report.js` dosyasının değiştiğini gösterir. Bu değişikliği geri almak veri kaybı doğurabilir; çünkü dosyadaki son düzenlemeleri silersiniz. Bu yüzden geri alma komutları çalıştırmadan önce gerçekten neyi kaybedeceğinizi bilmek gerekir.

4.9.2 Commit geçmişine dokunmadan geri dönmek

Commit geçmişine dokunmadan geri dönmenin en güvenli yollarından biri `git revert` kullanmaktır. Revert, geçmişteki bir commit'in etkisini tersine çeviren yeni bir commit oluşturur.

```
$ git revert f18a20c
[main 2c4d991] Revert "Fix report export encoding"
```

Bu çıktı bize eski commit'in silinmediğini, onun etkisini geri alan yeni bir commit oluştuğunu gösterir. Ekip çalışmasında bu yaklaşım genellikle daha güvenlidir; çünkü herkesin gördüğü geçmiş korunur.

4.9.3 Commit geçmişini değiştiren komutların riski

Commit geçmişini değiştiren komutlar güçlüdür; fakat ekip çalışmasında risklidir. Özellikle uzak depoya gönderilmiş geçmiş üzerinde bilinçsiz değişiklik yapmak, diğer ekip üyelerinin yerel geçmişiyle çakışma yaratabilir.

`git reset --hard` bu ailenin en sert örneğidir. Çalışma dizinini ve hazırlama alanını belirtilen commit'teki haline zorla döndürür; kayda alınmamış düzenlemeler silinir. Yerel bir denemeyi tamamen atmak için kullanılabilir. Paylaşılmış bir dalda veya henüz yedeklenmemiş iş varken çalıştırılmamalıdır. Ders için kural sadedir: emin değilseniz `--hard` kullanmayın; önce `git status` ile neyi kaybedeceğinizi görün.

Hangi durumda hangi yaklaşım güvenlidir?

Genel ilke şudur: Değişiklik yalnızca sizin çalışma alanınızdaysa yerel geri alma düşünülebilir; commit oluşmuş ama paylaşılmamışsa daha fazla seçenek vardır; commit uzak depoya gönderildiyse geçmişi koruyan `revert` çoğu zaman daha güvenli yaklaşımdır.

4.10 `.gitignore` ve Depo Temizliği

Bir projede her dosyanın Git'e eklenmesi gerekmez. Hatta bazı dosyaların Git'e eklenmesi ciddi sorun üretir. Kurulan bağımlılık klasörleri, derleme (build) çıktıları, geçici dosyalar, kişisel editör ayarları ve gizli anahtarlar depo geçmişinde yer almamalıdır. Git'in neyi izlemeyeceğini söylemek için `.gitignore` dosyası kullanılır.

`.gitignore`, proje temizliği açısından küçük ama kritik bir dosyadır. Bu dosya sayesinde yeniden üretilebilir dosyalar depoya eklenmez, gizli bilgiler yanlışlıkla paylaşılmaz ve proje geçmişi gereksiz dosyalarla şişmez.

Basit bir örnek:

```
node_modules/
dist/
.env
*.log
```

Bu kurallar `node_modules` klasörünü, derleme çıktısı olan `dist` klasörünü, gerçek `.env` dosyasını ve log dosyalarını Git dışında tutar. Ancak `.gitignore` geleceği korur; geçmişte commit edilmiş bir gizli bilgiyi kendiliğinden temizlemez.

Git'e neler eklenmez?

Git'e genellikle yeniden üretilebilen bağımlılık klasörleri, derleme çıktıları, geçici dosyalar, loglar, işletim sistemi artıkları ve gizli bilgi içeren dosyalar eklenmez. Buna karşılık kaynak kod, bağımlılık tarif dosyaları, kilit dosyası, README ve proje dokümantasyonu çoğu zaman eklenir. Buradaki temel soru şudur: Bu dosya proje hafızası için gerekli mi, yoksa makineye veya kişiye özgü geçici bir çıktı mı?

4.10.1 Bağımlılık klasörleri, derleme çıktıları ve geçici dosyalar

Bağımlılık klasörleri ve derleme çıktıları çoğu zaman paket yöneticisi veya derleme komutu tarafından yeniden üretilebilir. Bu nedenle Git'e eklemek depoyu büyütür ve geçmişi gereksiz yere kirletir.

```
$ git status --short
?? node_modules/
?? dist/
?? .env
```

Bu çıktı üç izlenmeyen öğe gösterir. `node_modules/` ve `dist/` yeniden üretilebilir olabilir. `.env` ise gizli bilgi taşıyabilir. Bu durumda `.gitignore` dosyasıyla bu öğeleri dışarıda tutmak gerekir. Böylece depo yalnızca proje hafızası için gerekli dosyaları taşır.

4.10.2 `.env` dosyaları ve gizli bilgi sızıntısı

`.env` dosyaları veritabanı parolası, API anahtarı veya gizli token içerebilir. Bu dosya yanlışlıkla Git'e eklenirse gizli bilgi depo geçmişine girebilir. Depo özel olsa bile bu risk ciddidir; çünkü erişimi olan herkes bu bilgiye ulaşabilir. Önemli ayrım şudur: `.env` gizli değerleri taşır, `.env.example` ise paylaşılabilir yapıyı gösterir.

4.10.3 Gizli bilgi daha önce commit edildiyse ne değişir?

Gizli bilgi daha önce commit edildiyse yalnızca dosyayı silmek yeterli olmayabilir. Çünkü bilgi Git geçmişinde kalmış olabilir. Bu durumda ilk yapılacak iş, sızan anahtarı veya parolayı geçersiz kılmak ve yenisini üretmektir. Geçmiş temizliği ayrı ve dikkatli yönetilmesi gereken bir işlemdir (Tablo 4.2).

Tablo 4.2: Gizli bilgi daha önce commit edildiyse ne değişir?

Durum	Neden kritik?	Eylem planı
Parola commit edilmiş	Geçmişten okunabilir	Parolayı değiştir
API anahtarı paylaşılmış	Kullanım maliyeti veya veri riski doğabilir	Anahtarı iptal et
Sadece dosya silinmiş	Geçmiş hâlâ bilgiyi taşıyabilir	Güvenlik olayı gibi ele al
Depo herkese açık (public)	Sızıntı, arama araçlarınca hızlıca fark edilebilir	Anahtarı acilen iptal et, olayı kayıt altına al

Bu yüzden gizli bilgi sızıntısı yalnızca Git temizliği değil, güvenlik ve operasyon yönetimi meselesidir.

4.11 GitHub

Git'i kendi bilgisayarınızda kullanabilirsiniz; ancak ekip çalışmasında değişikliklerin ortak bir yerde görünmesi gerekir. GitHub bu noktada devreye girer. GitHub, Git depo-

larını internet üzerinde barındıran, değişiklikleri ekip içinde tartışmayı, incelemeyi ve yönetmeyi kolaylaştıran bir platformdur.

GitHub'ı yalnızca "kod yüklediğimiz site" olarak görmek eksik olur. Repository ana sayfası, konu kayıtları (issue), değişiklik istekleri, inceleme yorumları, dal korumaları ve sürüm (release) notları bir araya geldiğinde GitHub proje koordinasyon alanına dönüşür. Bu alan teknik olduğu kadar yönetseldir.

Bir projede GitHub sayesinde ekip şunları görebilir: kim hangi işi üstlenmiş, hangi değişiklik inceleme bekliyor, testler geçmiş mi, ana dala doğrudan değişiklik yapılmış mı, teslimden önce hangi issue'lar açık kalmış. Bu yüzden GitHub, Git geçmişini takım çalışmasına bağlayan önemli bir katmandır.

GitHub üzerinde sık karşılaşacağımız kavramları bu aşamada kısaca ayıralım:

Depo (repository) Projenin dosyalarını, Git geçmişini ve çevrimiçi çalışma alanını bir arada tutar. Depo ana sayfasında dallar, son commit, dosyalar ve README gibi temel bilgiler görünür.

İş kaydı (issue) Yapılacak işi, hatayı, öneriyi veya tartışılması gereken konuyu kayıt altına alır. Bir issue; açıklama, sorumlu, etiket ve durum bilgisiyle işin neden var olduğunu ve kim tarafından izleneceğini görünür hale getirir.

Dal (branch) Bir değişikliğin ana çalışma hattından ayrılarak geliştirilmesini sağlar. Dal kullanıldığında henüz tamamlanmamış iş doğrudan `main` dalını etkilemeden commit edilebilir ve test edilebilir.

Değişiklik isteği (pull request) Bir daldaki commit'lerin başka bir dala alınması için açılan inceleme ve birleştirme talebidir. Değişiklik isteği yalnızca kod farkını değil, açıklamayı, test sonuçlarını, yorumları ve onay kararını da bir araya getirir.

Kod inceleme (review) Değişiklik isteği içindeki değişikliğin ekip tarafından değerlendirilmesidir. İnceleme sırasında hata, kapsam, okunabilirlik, güvenlik ve test yeterliliği tartışılır; gerekirse değişiklik istenir.

4.11.1 Git ve GitHub farkı

Git ve GitHub aynı şey değildir. Git sürüm kontrol aracıdır; bilgisayarınızda çalışır ve değişiklik geçmişini yönetir. GitHub ise Git depolarını barındıran ve ekip çalışması özellikleri sunan platformdur. Tablo 4.3 bu ayrımı örnekleriyle özetler.

Tablo 4.3: Git ve GitHub farkı

Kavram	Temel görevi	Örnek
Git	Değişiklik geçmişini yönetir	`git commit`, `git log`
GitHub	Depoyu ve takım akışını çevrimiçi yönetir	Değişiklik isteği, konu kaydı, kod inceleme
İş kaydı (issue)	İşin nedenini ve durumunu tutar	GitHub issue; Git tek başına iş listesi değildir

Kavram	Temel görevi	Örnek
Uzak depo (`origin`)	Yerel geçmişi ekiple paylaşır	`git remote -v`

Bu ayrımı bilmek önemlidir. GitHub'a erişemediğiniz bir anda Git yerel olarak çalışmaya devam edebilir; fakat ekip koordinasyonu için değişiklikleri uzak depoya göndermek gerekir.

4.11.2 Uzak depo (remote repository) ve `origin` kavramı

Uzak depo, projenin GitHub gibi bir platform üzerinde duran paylaşılan kopyasıdır. `origin` ise yerel Git deposunda bu uzak depoya verilen varsayılan addır. Bunu bir ekibin ortak proje klasörü gibi düşünebiliriz. Her ekip üyesi kendi bilgisayarında çalışır; fakat ortak hafızaya gönderilen değişiklikler `origin` üzerinden paylaşılır.

```
$ git remote -v
origin  https://github.com/akadal/blockchain.git (fetch)
origin  https://github.com/akadal/blockchain.git (push)
```

Bu çıktı, `origin` adının hangi GitHub deposunu işaret ettiğini gösterir. `fetch` uzak depodan bilgi almayı, `push` ise yerel değişiklikleri uzak depoya göndermeyi ifade eder.

4.11.3 Clone, push, pull ve fetch akışı

GitHub ile çalışırken birkaç temel hareket sık tekrar eder:

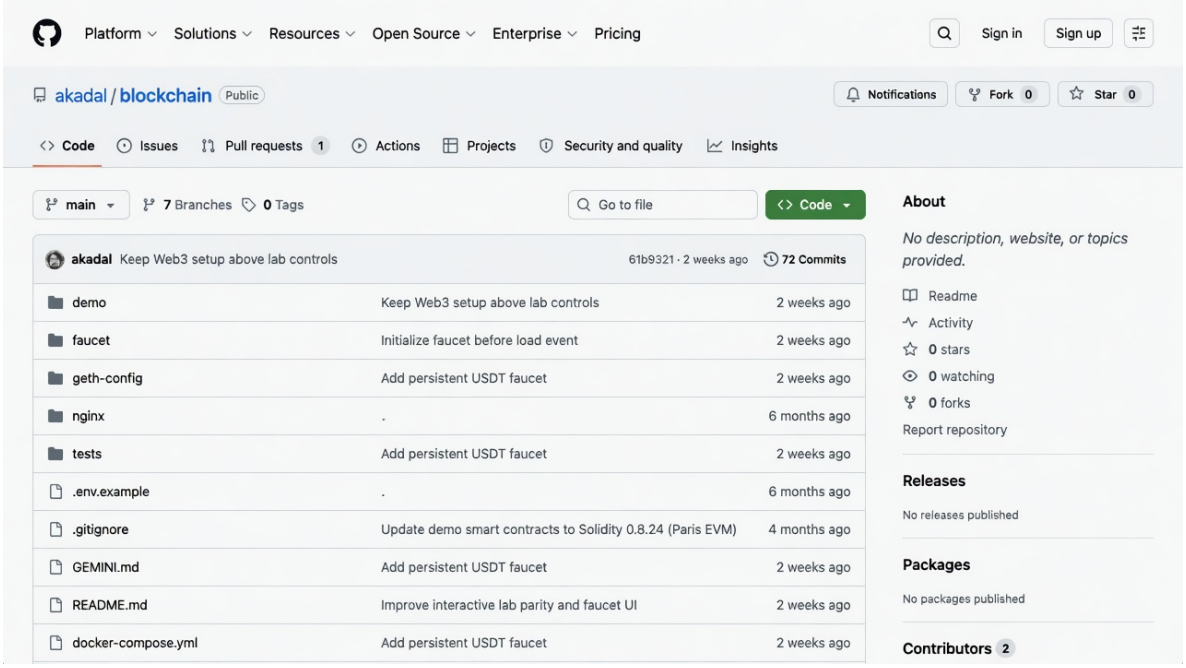
1. `clone`: Uzak depoyu bilgisayara alır; ekipte ortak başlangıç noktası sağlar.
2. `fetch`: Uzak depodaki yeni bilgileri getirir; çalışma alanını doğrudan değiştirmeden durumu görmeyi sağlar.
3. `pull`: Uzak değişiklikleri alır ve yerel dala uygular; yerel ortamı güncel tutar.
4. `push`: Yerel commit'leri uzak depoya gönderir; ekip hafızasını günceller.

Bu komutlar dosya taşıma komutları gibi görünse de aslında ekip koordinasyonunu yönetir. Kimin neyi ne zaman paylaştığı bu akışla görünür hale gelir.

Repository ana sayfasını okumak

GitHub'da repository ana sayfası projenin giriş kapısıdır. Dosya listesi, README, dal bilgisi, son commit, konu kaydı ve değişiklik isteği sayıları burada görünür. Bu sayfayı okumak, projeye katılmadan önce genel durumu anlamayı sağlar. README güncel mi, son değişiklik ne zaman yapılmış, açık değişiklik isteği var mı, konu kayıtları takip ediliyor mu? Bu sorular yalnızca geliştirici için değil, proje koordinasyonu için de önemlidir.

Şekil 4.6 böyle bir depo ana sayfasını gösterir.



Şekil 4.6: `akadal/blockchain` deposunun ana sayfasında dal, dosya ve commit bilgileri

4.12 GitHub Üzerinde Takım Çalışması

GitHub üzerinde takım çalışması, herkesin aynı depoya rastgele kod göndermesi anlamına gelmez. Sağlıklı bir ekip akışında iş önce tanımlanır, sonra dal açılır, değişiklik yapılır, değişiklik isteği oluşturulur, inceleme alınır ve onaydan sonra ana dala birleştirilir. Bu akış, küçük ekiplerde bile düzen sağlar.

Bir ders projesinde bu düzen abartılı görünebilir. Ancak teslim tarihi yaklaştığında kimin hangi işi yaptığı, hangi değişikliğin hazır olduğu ve hangi hatanın beklediği net değilse ekip zaman kaybeder. GitHub bu görünürlüğü sağlamak için kullanılabilir.

Bu sebeple bu bölümde ortak depo kurmaktan dal açıp değişiklik isteği oluşturmaya, oradan inceleme almaya kadar somut bir akış üzerinden ilerleyeceğiz.

4.12.1 Ortak repository oluşturma

Ortak repository oluşturmak, ekibin resmi proje kaynağını belirlemek anlamına gelir. Herkesin kendi klasöründe çalışması yerine ortak bir uzak depo üzerinden ilerlenir. Yerel bilgisayara almak için şu komut kullanılabilir:

```
$ git clone https://github.com/akadal/blockchain.git
Cloning into 'blockchain'...
```

Bu çıktı, ekibin ortak deposunun yerel bilgisayara alındığını gösterir. Bundan sonra yapılacak değişiklikler bu ortak geçmişle ilişkilidir.

Katılımcı (collaborator) ve erişim yönetimi

GitHub'da collaborator, depoya belirli yetkilerle katkı verebilen kişidir. Herkese aynı yetkiyi vermek kolay görünür; ancak risklidir. Kim ana dala doğrudan yazabilir, kim issue açabilir, kim inceleme onayı verebilir, kim ayarları değiştirebilir? Bu sorular erişim yönetiminin parçasıdır. Yetki, güven ve sorumluluk birlikte düşünülmelidir.

4.12.2 Konu kaydından (issue) dala, daldan değişiklik isteğine akış

Basit bir takım akışı şöyle kurulabilir:

1. Issue açılır; yapılacak iş görünür hale gelir.
2. Issue için dal açılır; değişiklik ana daldan izole edilir.
3. Commit'ler bu dalda birikir; karar geçmişi oluşur.
4. Değişiklik isteği açılır; değişiklik ekip incelemesine sunulur.
5. İnceleme ve testlerden sonra birleştirme yapılır; ana dal güncellenir.

Bu akış, her değişikliği bir issue'ya, bir dala ve bir incelemeye bağlayarak işi izlenebilir ve geri dönebilir kılar.

Yerel ortamı güncel tutmak

Yerel ortamı güncel tutmak, başkalarının yaptığı değişikliklerden kopmamak için gereklidir. Uzun süre `main` dalını güncellemeden çalışmak, birleştirme çakışması ve tekrar iş yapma riskini artırabilir. Bu yüzden ekip içinde düzenli olarak uzak depodaki değişiklikleri almak ve kendi dalını güncel bağlamda test etmek önemlidir.

4.13 Değişiklik İsteği (Pull Request)

Değişiklik isteği, bir dalda yapılan değişikliklerin ana dala alınması için açılan inceleme isteğidir. Adı teknik görünse de işlevi oldukça insanidir: "Ben şu değişikliği yaptım, lütfen inceleyin, tartışalım ve uygunsa birleştirelim."

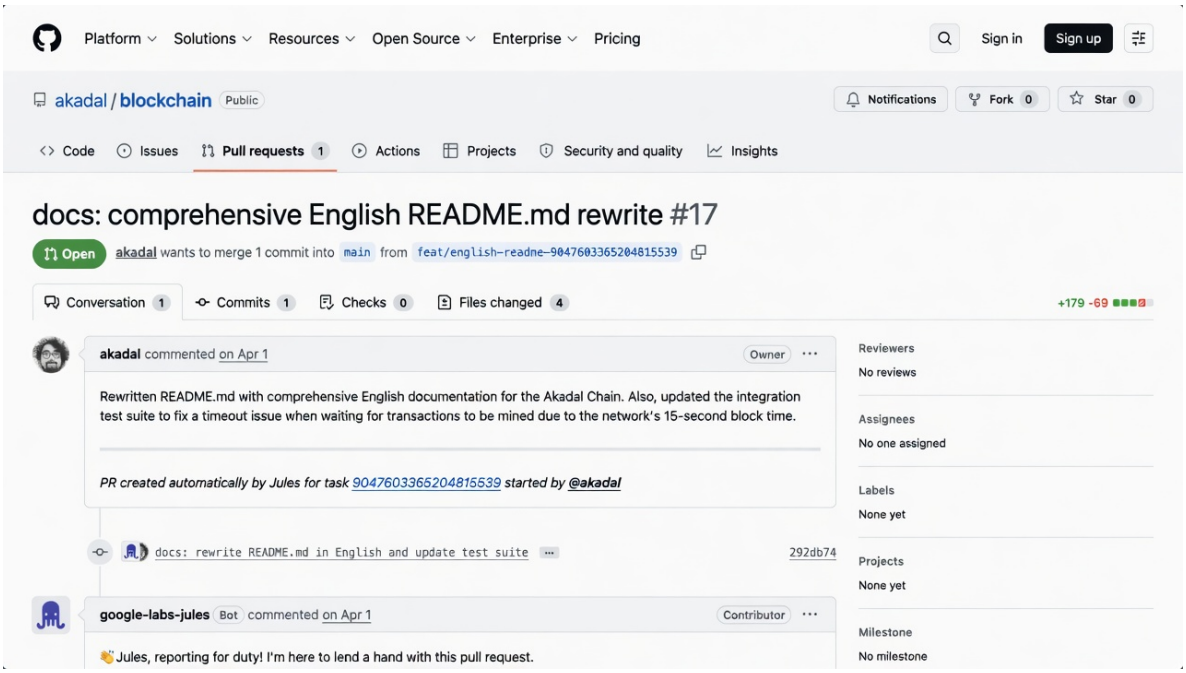
Değişiklik isteği yalnızca kod taşımaz. Açıklama, commit geçmişi, değişen dosyalar, inceleme yorumları, test sonuçları ve karar tartışması aynı yerde toplanır. Bu yüzden PR, takım hafızasının önemli parçalarından biridir.[2]

Şekil 4.7 açık bir değişiklik isteğinin nasıl göründüğünü özetler.

Basit bir PR akışı:

1. Dalda değişiklik yapılır.
2. Commit'ler oluşturulur.
3. Dal GitHub'a gönderilir.
4. PR açılır ve açıklama yazılır.

[2] McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E. (2016). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, *21*(5), 2146–2189. https://doi.org/10.1007/s10664-015-9381-9 Davila, N., & Nunes, I. (2021). A systematic literature review and taxonomy of modern code review. *Journal of Systems and Software*, *177*, Article 110951. https://doi.org/10.1016/j.jss.2021.110951



Şekil 4.7: `akadal/blockchain` deposunda açık bir değişiklik isteğinin dal, commit ve değişiklik özeti

5. İnceleme ve testler tamamlanır.
6. PR birleştirilir veya revizyon istenir.

PR neden vardır?

PR, tek başına çalışır görünen bir değişikliğin fark edilmeden başka bir ekranı bozmasını veya eksik dokümantasyonla ana dala girmesini önler. Bir proje ekibinde raporlama ekranı değiştirildi diyelim. Bu değişiklik tek başına çalışıyor olabilir; fakat başka bir ekranı bozmuş olabilir veya README güncellemesi gerektirebilir. PR sayesinde değişiklik ana dala girmeden önce ekip tarafından görülür. Bu, kalite kontrolü ve ortak sorumluluk açısından önemlidir.

4.13.1 PR açıklaması nasıl yazılır?

İyi PR açıklaması değişikliğin ne yaptığını, neden yapıldığını ve nasıl test edildiğini söyler. Örneğin:

```
## Ne değişti?
- Stok raporuna tarih filtresi eklendi.

## Neden?
- Kullanıcıların aylık rapor alabilmesi gerekiyor.

## Nasıl test edildi?
- `npm test` çalıştırıldı.
- Rapor ekranında tarih aralığı denendi.
```

Bu açıklama inceleyenin yalnızca koda değil, karar bağlamına da bakmasını sağlar.

Küçük PR neden daha iyidir?

Küçük PR daha kolay incelenir, daha hızlı geri bildirim alır ve hata riskini daraltır. Büyük PR ise birçok değişikliği aynı anda taşıdığı için inceleyenin neye odaklanacağını zorlaştırır. Teslim tarihine yakın büyük PR açmak, ekibin hem teknik hem iletişim yükünü artırır. Bu yüzden küçük, anlamlı ve test edilebilir PR'lar takım akışını hızlandırır.

İncelenebilir değişiklik hazırlamak

İncelenebilir değişiklik, amacı belli, kapsamı sınırlı, açıklaması yazılmış ve test edilmiş değişikliktir. Dosya biçimlendirme, büyük yeniden yapılandırma (refactor) ve işlev değişikliğini aynı PR içine koymak incelemeyi zorlaştırır. PR açmadan önce "Bu değişiklik inceleyen tarafından makul sürede anlaşılabilir mi?" sorusunu sormak iyi bir alışkanlıktır.

4.14 Kod İnceleme (Code Review) Süreci

PR açıldığında değişiklik ana dala kendiliğinden girmez; değişikliği yazan kişi hangi kararı neden verdiğini bilir, fakat projeyi sonradan okuyacak veya bakımını üstlenecek kişi aynı bilgiye sahip değildir. Bu farkı kapatan adım kod incelemedir. Kod inceleme, bir kişinin yazdığı değişikliğin başka biri tarafından değerlendirilmesidir. Bu süreç yalnızca hata bulmak için yapılmaz. Kodun okunabilirliği, iş kuralına uygunluğu, test durumu, güvenlik etkisi ve bakım maliyeti de inceleme kapsamına girer.

İyi inceleme kültürü ekipte güven üretir. Amaç kişiyi eleştirmek değil, değişikliği iyileştirmektir. Bu yüzden yorumların açık, gerekçeli ve saygılı olması gerekir. Bir yorum "bu yanlış" demek yerine "bu durumda boş e-posta geldiğinde ne olacağını da kontrol edebilir miyiz?" şeklinde yazıldığında öğrenme alanı açar.

Kod inceleme, karar kalitesini artıran bir kontrol noktasıdır. Değişiklik yalnızca yazılmış olmaz; ekip tarafından anlaşılmış ve kabul edilmiş hale gelir.

İnceleyen kişi (reviewer) neye bakar?

İnceleyen yalnızca kodun çalışıp çalışmadığına bakmaz. Değişikliğin amacına, kapsamına, testine, okunabilirliğine, güvenlik etkisine ve dokümantasyon ihtiyacına bakar. Küçük projelerde bile bu kontrol değerlidir; çünkü ekip üyeleri birbirinin kararlarını görür ve ortak kalite standardı oluşur.

Kod değil karar incelemek

Kod incelemenin güçlü tarafı, kod satırının arkasındaki kararı görünür kılmasıdır. Neden bu kütüphane seçildi? Neden bu hata burada ele alındı? Neden bu veri kullanıcıya gösteriliyor? Bu sorular yalnızca yazılımcı ilgisi değildir; iş kuralı, güvenlik ve kullanıcı deneyimiyle ilgilidir. Yönetim Bilişim Sistemleri okuru inceleme sürecinde bu karar ilişkisini okumayı öğrenmelidir.

Yorum yazma kültürü

İnceleme yorumları kısa, açık ve gerekçeli olmalıdır. Kişiye değil değişikliğe odaklanmak gerekir. "Bunu böyle yapma" yerine "Bu durumda kullanıcı boş değer gönderirse hata alabiliriz; burada doğrulama eklemek daha güvenli olur" demek daha öğreticidir. Böyle bir dil ekip içinde savunma refleksini azaltır ve kaliteyi artırır.

İnceleme sonrasında yapılacaklar

İnceleme sonrasında geliştirici yorumları değerlendirir, gerekli değişiklikleri yapar, yeni commit gönderir ve gerekirse açıklama yazar. İnceleyen de değişikliğin gerçekten ele alınıp alınmadığını kontrol eder. Bu süreç uzlaşma gerektirir. Amaç PR'ı hızlıca kapatmak değil, ana dala girecek değişikliğin ekipçe anlaşılmış olmasını sağlamaktır.

4.15 Dal ve Birleştirme Stratejileri

Git'te dal ve birleştirme (merge) stratejileri ekibin çalışma biçimini doğrudan etkiler. Küçük bir ders projesi ile büyük bir ürün ekibi aynı karmaşıklıkta dal modeli kullanmak zorunda değildir. Hatta gereğinden karmaşık model, küçük ekiplerde ilerlemeyi yavaşlatabilir.

Bu yüzden strateji seçerken "en profesyonel görünen model hangisi?" diye değil, "ekibin büyüklüğü, teslim döngüsü, inceleme kapasitesi ve risk düzeyi ne?" diye sormak gerekir (Tablo 4.4).

Tablo 4.4: Dal ve Birleştirme Stratejileri

Strateji	Ne sağlar?	Ne zaman dikkat ister?
Özellik dalı (feature branch)	Değişikliği izole eder	Dal çok uzun yaşarsa çakışma artar
Merge commit	Dallanma geçmişini açık gösterir	Geçmiş kalabalık görünebilir
Squash merge	PR'ı tek commit'e indirger	Ara commit ayrıntıları kaybolabilir
Rebase merge	Daha düz geçmiş sağlar	Geçmiş yeniden yazma mantığı iyi anlaşılmalıdır

Ders projelerinde çoğu zaman sade bir özellik dalı, PR ve inceleme akışı yeterlidir.

4.15.1 `main` dalı ve özellik dalı (feature branch)

`main` dalı projenin ana ve güvenilir çalışma hattı olarak düşünülmelidir. Özellik dalları ise belirli değişikliklerin geliştirildiği geçici çalışma alanlarıdır. Bu ayrım ana dalı korur ve değişiklikleri PR üzerinden incelemeyi kolaylaştırır. Ancak `main` dalının güvenilir kalması için doğrudan yazma yerine inceleme ve test süreciyle güncellenmesi gerekir.

4.15.2 Merge commit, squash merge ve rebase merge

Merge türleri geçmişin nasıl görüneceğini belirler. Örneğin GitHub üzerinde bir PR birleştirirken ekip merge commit, squash merge veya rebase merge seçeneklerinden birini kullanabilir. Seçim, yalnızca teknik görünüm değil, geçmiş okuma biçimi kararıdır.

```
Merge commit:   PR geçmişini dallanma ile gösterir.
Squash merge:   PR'daki değişiklikleri tek commit'e indirir.
Rebase merge:   Commit'leri ana dalın üzerine düz biçimde taşır.
```

Bu seçeneklerin hiçbiri tek başına her durumda en iyi değildir; seçilen strateji ekibin çalışma sözleşmesine yazılmalı ve GitHub'daki birleştirme ayarlarına da yansıtılmalıdır.

4.15.3 Ders için önerilen basit takım akışı

Ders projeleri için basit bir akış çoğu zaman yeterlidir:

1. Her iş için issue açılır; görev görünür olur.
2. Issue için özellik dalı açılır; ana dal korunur.
3. Küçük commit'lerle değişiklik yapılır; geçmiş okunabilir kalır.
4. PR açılır; ekip değişikliği inceler.
5. Testler geçtikten ve inceleme tamamlandıktan sonra birleştirilir.

Bu akış basit görünür; fakat ekip düzeni, geri dönebilme ve kalite kontrol açısından güçlü bir başlangıç sağlar.

4.15.4 Karmaşık dal modellerinden neden kaçınabiliriz?

Karmaşık dal modelleri büyük ekiplerde ve belirli sürüm çıkarma süreçlerinde anlamlı olabilir. Ancak küçük ekiplerde, kısa teslim döngülerinde ve sınırlı inceleme kapasitesinde bu modeller gereksiz yük üretebilir; Tablo 4.5 bu durumları örnekler.

Tablo 4.5: Karmaşık dal modellerinden neden kaçınabiliriz?

Durum	Basit akış neden yeterli olabilir?
3-5 kişilik ders ekibi	İletişim doğrudan kurulabilir
Kısa dönem projesi	Uzun sürüm dalları gerekmeyebilir
Sınırlı inceleme zamanı	Küçük PR akışı daha sürdürülebilir olur
Sık ve küçük teslim	Uzun sürüm dalları bakım maliyeti üretir
Tek üretim hattı	Ek katmanlar görünürlüğü azaltabilir

Basit akış ilkel olmak zorunda değildir. Çoğu ders ve küçük proje için bilinçli olarak sade tutulmuş bir akış daha sağlıklı sonuç verir.

4.16 Konu Kaydı (Issue), Proje Panosu (Project) ve Sürüm Paketi (Release) Yönetimi

GitHub yalnızca kod değişikliklerini değil, yapılacak işleri ve yayın kararlarını da takip etmeye yardım eder. Konu kayıtları (issue) yapılacak iş veya hata kayıtlarını, proje panoları işin durumunu, sürüm paketi (release) ise belirli bir sürümde kullanıcıya ne sunulduğunu gösterir.

Bir ekip projesinde "kim ne yapıyor?" sorusu yalnızca toplantıda konuşulursa unutulabilir. Konu kaydı ve proje panosu kullanımı bu bilgiyi görünür hale getirir. Sürüm paketi ve değişiklik günlüğü (changelog) ise kullanıcıya veya öğretim elemanına hangi değişikliklerin teslim edildiğini anlatır.

Bu düzen teknik görünse de proje yönetimiyle doğrudan ilişkilidir. İşin tanımı, durumu ve teslim notu aynı ekosistem içinde tutulur.

4.16.1 Basit iş listesi (backlog)

Backlog, yapılacak işlerin biriktiği listedir. Küçük bir projede backlog birkaç issue'dan oluşabilir:

```
#12 Stok raporuna tarih filtresi ekle
#13 Login ekranında boş e-posta kontrolü yap
#14 README kurulum adımlarını güncelle
```

Bu liste bize teknik işlerin yanında dokümantasyon ve kalite işlerinin de görünür olduğunu gösterir. Backlog yalnızca yazılımcı işi listesi değildir; proje tesliminin hafızasıdır.

Yapılacak, devam eden ve tamamlanan işler

To do, in progress ve done gibi durumlar işin akışını görünür kılar. Türkçe düşünürsek yapılacak, devam eden ve tamamlanan işler ayrımıdır. Bu ayrım basit görünür; fakat ekip içinde aynı anda çok işe başlanmasını, bir işin sahipsiz kalmasını veya tamamlandı sanılan bir işin aslında inceleme beklemesini önlemeye yardım eder.

4.16.2 Etiket (tag), sürüm paketi (release) ve değişiklik günlüğü (changelog)

Etiket (tag), Git geçmişinde belirli bir noktayı işaretlemek için kullanılır. Sürüm paketi (release) bu noktayı kullanıcıya sunulan sürüm olarak açıklamaya yarar; değişiklik günlüğü (changelog) ise bu sürümde neyin değiştiğini anlatır. Bir sürüm paketi çoğu zaman önceki paketten bu yana birikmiş commit'lerin kullanıcıya dönük özetidir; bu yüzden anlamlı commit mesajları ve atomik kayıtlar doğrudan sürüm notlarının kalitesini belirler.

```
$ git tag v1.0.0
$ git tag
v1.0.0
```

Bu çıktı `v1.0.0` etiketinin oluştuğunu gösterir. Ancak etiket tek başına kullanıcı için yeterli değildir. Kullanıcı veya proje paydaşı bu sürümde hangi hataların düzeldiğini, hangi özelliklerin eklendiğini ve hangi değişikliklerin önemli olduğunu görmek ister.

Kullanıcıya değişikliği anlatmak

Kullanıcıya değişikliği anlatmak, commit mesajı yazmaktan farklıdır. Kullanıcı "src/report.js güncellendi" bilgisini değil, "stok raporları artık tarih aralığına göre filtrelenebiliyor" bilgisini duymak ister. Bu yüzden sürüm paketi notları teknik değişikliği iş değerine çevirir.

4.17 Takım Projesinde GitHub Kuralları

Takım projesinde GitHub kuralları baştan belirlenmezse, proje ilerledikçe herkes kendi alışkanlığıyla çalışmaya başlar. Bir kişi doğrudan `main` dalına push eder, başka biri PR açar, bir başkası inceleme bekler, biri testler başarısızken birleştirir. Bu durumda teknik araç aynı olsa bile ekip düzeni dağılır.

Bu yüzden küçük projelerde bile basit GitHub kuralları belirlemek faydalıdır. Ana dal korunacak mı? PR zorunlu mu? En az bir inceleme gerekiyor mu? Testler geçmeden birleştirme yapılabilir mi? Issue açmadan işe başlanacak mı? Bu sorular çalışma sözleşmesinin parçasıdır.

Kurallar ana dalı korur, kalite kontrolünü görünür kılar ve herkesin aynı beklentiyle çalışmasını sağlar.

Dal koruması (branch protection)

Dal koruması, özellikle `main` gibi önemli dallara kontrolsüz değişiklik gitmesini engeller. Örneğin doğrudan push kapatılabilir, PR zorunlu tutulabilir veya testlerin geçmesi istenebilir. Bu kural küçük ekiplerde bile değerlidir; çünkü ana dalın her zaman çalışır durumda kalmasına yardım eder.

PR ve inceleme zorunluluğu

PR ve inceleme zorunluluğu, ana dala girecek değişikliğin en az bir kişi tarafından görülmesini sağlar. Bu, hataları tamamen ortadan kaldırmaz; fakat kararların görünür olmasını sağlar. Ekip içinde "kimse bakmadan birleştirilmiş" durumunu azaltır. Böylece sorumluluk kişisel hafızadan ortak sürece taşınır.

4.17.1 Birleştirme için sürekli entegrasyon (CI) gerekliliği

Sürekli entegrasyon (CI; İngilizce continuous integration), değişikliğin otomatik kontrollerden geçmesini sağlar. Testler başarısızsa değişikliğin ana dala alınmaması iyi bir kuraldır.

```
Checks
[OK] lint
[OK] unit tests
[X] integration tests
```

Bu örnek çıktı bize bazı kontrollerin geçtiğini, fakat entegrasyon testlerinin başarısız olduğunu gösterir. Böyle bir durumda birleştirmek, bilinen bir riski ana dala taşımak anlamına gelir. Bu karar yalnızca teknik değil, kalite ve teslim güvenilirliği kararıdır.

Takım çalışma sözleşmesi

Takım çalışma sözleşmesi, ekibin GitHub üzerinde nasıl çalışacağını kısa ve açık biçimde belirler. Örneğin "her iş için issue açılır", "her değişiklik özellik dalı üzerinde yapılır", "main dalına doğrudan push edilmez", "en az bir inceleme alınır", "CI geçmeden birleştirme yapılmaz" gibi kurallar yazılabilir. Bu sözleşme uzun olmak zorunda değildir; önemli olan herkesin aynı çalışma beklentisini paylaşmasıdır.

4.18 Atölye

Önceki atölyede konum, dosya ve süreç okumayı denediniz. Şimdi aynı disiplini değişiklik kaydına taşıyoruz: bir klasörü depoya çevirmek, iki anlamlı commit oluşturmak ve geçmişi okumak. GitHub hesabı gerekmez; her şey yerel kalır. Amaç, komutu ezberlemek değil, "ne kayda girdi?" sorusunu çıktıdan cevaplayabilmektir.

Görev 1: Yerel depoyu başlat

Komut

```
$ mkdir -p ~/gelistirme-atolyesi/bolum-4
$ cd ~/gelistirme-atolyesi/bolum-4
$ mkdir stok-app
$ cd stok-app
$ git init -b main
$ git config user.name "Atolye Kullanici"
$ git config user.email "atolye@example.com"
```

Beklenen çıktı

```
Initialized empty Git repository in <yol>/stok-app/.git/
```

Kimlik ayarları yalnızca bu depoya yazılır.

Görev 2: İlk dosyaları kaydet

Komut

```
$ printf '# Stok Uygulamasi\n' > README.md
$ printf 'title=Stok Raporu\n' > config.txt
$ git status --short
$ git add README.md config.txt
$ git commit -m "Proje iskeletini ekle"
```

Beklenen çıktı

```
?? README.md
?? config.txt
[main <kimlik>] Proje iskeletini ekle
```

Görev 3: Farkı incele ve ikinci commit'i oluştur

Komut

```
$ printf '\nKurulum: npm ci\n' >> README.md
$ git diff -- README.md
$ git add README.md
$ git commit -m "Kurulum notunu ekle"
$ git log --oneline -2
```

Beklenen çıktı

```
+Kurulum: npm ci
<kimlik> Kurulum notunu ekle
<kimlik> Proje iskeletini ekle
```

Görev 4: İki dalda aynı satırı değiştir

Komut

```
$ git switch -c feature/report
$ printf 'title=Aylik Stok Raporu\n' > config.txt
$ git add config.txt
$ git commit -m "Aylik rapor basligini ekle"
$ git switch main
$ printf 'title=Stok Durum Raporu\n' > config.txt
$ git add config.txt
$ git commit -m "Rapor basligini guncelle"
```

Beklenen çıktı

```
Switched to a new branch 'feature/report'
Switched to branch 'main'
[main <kimlik>] Rapor basligini guncelle
```

Görev 5: Kontrollü çakışmayı gör

Komut

```
$ git merge feature/report
$ git status --short
$ cat config.txt
```

Beklenen çıktı

```
CONFLICT (content): Merge conflict in config.txt
UU config.txt
 <<<<<<< HEAD
title=Stok Durum Raporu
 =======
title=Aylik Stok Raporu
 >>>>>>> feature/report
```

Birleştirme komutunun başarısız durum koduyla bitmesi bu görevde beklenen sonuçtur.

Görev 6: Çakışmayı çöz ve geçmişi oku

Komut

```
$ printf 'title=Aylik Stok Durum Raporu\n' > config.txt
$ git add config.txt
$ git commit -m "Rapor basligi cakismasini coz"
$ git log --oneline --graph --all -6
```

Beklenen çıktı

```
*   <kimlik> Rapor basligi cakismasini coz
|\
| * <kimlik> Aylik rapor basligini ekle
* | <kimlik> Rapor basligini guncelle
```

Görev 7: Sürümü etiketle

Komut

```
$ git tag -a v1.0.0 -m "Ilk atolye surumu"
$ git tag
$ git show --no-patch --oneline v1.0.0
```

Beklenen çıktı

```
v1.0.0
<kimlik> Rapor basligi cakismasini coz
```

Görev 8: Yerel uzak depo kur ve paylaş

Komut

```
$ cd ~/gelistirme-atolyesi/bolum-4
$ git init --bare -b main remote.git
$ git -C stok-app remote add origin ../remote.git
$ git -C stok-app push -u origin main --tags
$ git clone remote.git kontrol-kopyasi
$ git -C kontrol-kopyasi log --oneline -1
```

Beklenen çıktı

```
To ../remote.git
 * [new branch]      main -> main
 * [new tag]         v1.0.0 -> v1.0.0
Cloning into 'kontrol-kopyasi'...
<kimlik> Rapor basligi cakismasini coz
```

Bölüm 5

Teknik Dokümantasyon

Bir proje üzerinde çalışırken ilk bakışta asıl işin kod yazmak olduğunu düşünebiliriz. Ekran çalışıyorsa, veritabanı kayıt alıyorsa, kullanıcı giriş yapabiliyorsa proje ilerliyor gibi görünür. Bu sezgi bütünüyle yanlış değildir; çünkü yazılım sonunda çalışan bir sistem üretmek için geliştirilir.

Ancak ekip çalışması başladığında başka bir gerçek ortaya çıkar. Kodun kendisi bize her şeyi anlatmaz. Proje neden bu şekilde kurulmuş, hangi servis hangi amaçla kullanılmış, uygulama ilk kez nasıl çalıştırılır, hangi kararlar daha önce tartışılmış, hangi ayarlar gizli tutulmalıdır, hangi API yanıtı hangi anlamı taşır? Bu soruların yanıtı yalnızca bir kişinin zihninde duruyorsa, proje kırılgan hale gelir.

Bu bölümde teknik dokümantasyonu yalnızca "belge yazmak" olarak değil, kodun yanında yaşayan bir çalışma pratiği olarak ele alacağız.[1] Yönetim Bilişim Sistemleri açısından dokümantasyon; ekip içi bilgi aktarımı, kurulum ve bakım maliyeti, karar izlenebilirliği, kalite kontrolü ve yapay zekâ (YZ; İngilizce *artificial intelligence*, AI) araçlarına bağlam verme gibi konularla doğrudan ilişkilidir. Projenin bugün ve yarın anlaşılmasını sağlayacak doğru bilgiyi, gerektiği kadar ve doğru yerde tutmak gerekir.

5.1 Geliştirici İçin Dokümantasyon

Dokümantasyonun en basit cevabı şudur: geliştirici ne yaptığını ve başkasının ne yaptığını anlayabilsin diye dokümantasyon gerekir. Küçük bir ders projesinde bile bu ihtiyaç hemen görünür hale gelir. Üç kişilik bir ekip düşünelim. Bir kişi veritabanını kurar, bir kişi arayüzü hazırlar, bir kişi de API tarafını yazar. Herkes kendi bölümünü biliyor olabilir; fakat projenin tamamı birleştirileceği zaman "hangi komutla çalıştırıyorduk?", "`.env` dosyasında ne olmalıydı?", "bu endpoint neden böyle tasarlandı?" gibi sorular ortaya çıkar.

Ancak dokümantasyonu yalnızca unutkanlığa karşı alınmış küçük bir not gibi görmek eksik olur. Dokümantasyon aynı zamanda iş sürekliliği aracıdır. Ekipten biri derse gelemezse, mezun olursa, projeden ayrılırsa veya iki ay sonra aynı projeye geri dönülürse,

[1] Lethbridge, T. C., Singer, J., & Forward, A. (2003). How software engineers use documentation: The state of the practice. *IEEE Software*, *20*(6), 35. https://doi.org/10.1109/MS.2003.1241364

sistemin tekrar anlaşılabilmesi gerekir. Bu durum yazılım geliştirmede teknik olduğu kadar organizasyonel bir meseledir.

Bir bilgi sistemi yalnızca kod dosyalarından oluşmaz; insanlar, süreçler, kararlar, yetkiler ve kullanım beklentileriyle birlikte çalışır. Dokümantasyon bu parçalar arasında ortak dil üretir. İyi yazılmış bir README, yeni gelen birinin projeyi çalıştırmasını sağlar. Kısa bir karar kaydı, ileride "bunu neden böyle yapmıştık?" sorusunu cevaplar. Güncel bir API dokümanı, hem ekip üyelerinin hem de YZ araçlarının projeyi yanlış varsayımlarla yorumlamasını azaltır.

Bu yüzden dokümantasyonun değeri sayfa sayısıyla ölçülmez. Değeri, doğru kişinin doğru anda doğru bilgiye ulaşabilmesiyle ölçülür.

Bilgiyi ekip içinde dolaştırmak

Dokümantasyon, bilginin tek bir kişide sıkışmasını engeller. Örneğin projede ödeme entegrasyonunu yalnızca bir ekip üyesi biliyorsa, diğerleri o parçaya dokunmaktan çekinir. Kısa bir `PAYMENT.md` dosyası veya README içindeki birkaç açıklama, hangi ortam değişkenlerinin gerektiğini, test modunun nasıl çalıştığını ve gerçek ödeme bilgisinin neden repoya yazılmaması gerektiğini gösterebilir. Böylece herkesin her ayrıntıyı ezberlemesine gerek kalmadan sorumluluk devrine yetecek ortak zemin oluşur.

5.1.1 Kurulum adımlarını netleştirmek

Kurulum adımları yalnızca "şu komutu yazın" listesi değildir. Kurulum, projenin hangi bağımlılıklara, hangi konfigürasyona ve hangi başlangıç verisine ihtiyaç duyduğunu gösterir. Bu bilgi yazılmadığında proje aslında çalışır durumda olsa bile başka bir bilgisayarda yeniden çalıştırılamayabilir.

Örneğin README içinde şu kadar küçük bir örnek bile yeni başlayan kişi için çok şey ifade eder:

```
$ npm install
added 143 packages in 8s
```

Bu çıktı bize komutun yalnızca çalıştığını değil, projenin Node.js paket bağımlılıklarının yüklendiğini gösterir. Eğer bu adım başarısız oluyorsa sorun uygulama kodunda değil, çoğu zaman paket yöneticisi, internet bağlantısı, Node.js sürümü veya kilit dosyasıyla ilgilidir. Dolayısıyla kurulum dokümantasyonu, hata ayıklamanın nereden başlayacağını da belirler.

Kararları kaydetmek

Projelerde bazı tercihler yalnızca teknik beğeniyle yapılmaz; zaman, ekip bilgisi, ders kapsamı, güvenlik beklentisi ve bakım maliyeti birlikte değerlendirilir. "SQLite mı kullanalım, PostgreSQL mi?", "kimlik doğrulama hazır servisle mi olsun, kendimiz mi yazalım?" gibi kararlar ileride tekrar gündeme gelebilir. Bu kararları kısa notlarla kaydetmek, ekibi aynı tartışmayı defalarca yapmaktan kurtarır. Sınır şudur: her küçük tercihi belgelemek gerekmez; fakat projenin yönünü, riskini veya maliyetini etkileyen kararlar görünür olmalıdır.

YZ araçlarına bağlam vermek

Yapay zekâ araçları kod yazarken veya kodu açıklarken bağlama ihtiyaç duyar. Eğer projede klasör yapısı, iş kuralları, API beklentileri ve kararlar yazılı değilse, araç çoğu zaman genel yazılım bilgisine dayanarak tahmin yürütür. Bu tahmin bazen işe yarar, bazen de projeye uymayan öneriler üretir. README, ADR, API dokümanı ve küçük mimari notlar bu yüzden yalnızca insanlara değil, YZ araçlarına da "bu projenin gerçekleri bunlardır" diyen bağlam dosyalarıdır.

5.1.2 Dokümantasyonun eskiyebilme problemi

Dokümantasyonun önemli olması, her yazılan metnin sonsuza kadar faydalı olacağı anlamına gelmez. Aksine, güncel olmayan doküman bazen hiç doküman olmamasından daha tehlikelidir. Bir ekip düşünelim: README hâlâ `npm start` komutunu gösteriyor, fakat proje Vite'a taşındığı için artık `npm run dev` ile çalışıyor. Teslim haftasında yeni gelen ekip üyesi eski komutu dener, hata alır ve sorunu kendi bilgisayarında aramaya başlar.

Buradaki maliyet yalnızca birkaç dakika kaybı değildir. Yanlış dokümantasyon güveni azaltır. İnsanlar bir süre sonra "dokümana bakmaya gerek yok, zaten eski" demeye başlar. Bu sebeple dokümantasyon yaşayan bir varlık gibi düşünülmelidir: kod değiştiğinde, kurulum değiştiğinde, karar değiştiğinde doküman da güncellenmelidir. İyi dokümantasyonun değeri güncelliğiyle birlikte ortaya çıkar.

5.2 Kod Gibi Dokümantasyon Yaklaşımı

Eskiden birçok ekip dokümantasyonu ayrı bir klasörde, paylaşılan bir sürücüde veya bir ofis belgesinde tutardı. Bu yöntem başlangıçta kolay görünür. Bir Word dosyası açılır, kurulum adımları yazılır, ekran görüntüleri eklenir ve dosya bir yere bırakılır. Ancak kod değiştikçe belge başka bir yerde kaldığı için güncelliğini hızla kaybedebilir.

Kod gibi dokümantasyon (Documentation as Code) yaklaşımı bu problemi azaltmak için dokümantasyonu kodla aynı çalışma düzeninin parçası haline getirir. Dokümanlar çoğu zaman Markdown gibi düz metin formatlarında yazılır, repository içinde tutulur, Git ile versiyonlanır ve değişiklik isteği sürecinde incelenir. Böylece dokümantasyon ayrı bir bürokrasi değil, geliştirme akışının doğal parçası olur.

Peki bunun pratik sonucu nedir? Bir özellik değiştiğinde yalnızca kod dosyası değil, o özelliği anlatan README, API dokümanı veya ADR de değişebilir. İnceleyen, PR içinde hem kodu hem de açıklamayı birlikte görebilir. Bu durum ekip için kalite kontrol sağlar; YZ aracı için de daha güncel bağlam üretir.

Dokümantasyonu repository içinde tutmak

Dokümantasyonu repository içinde tutmak, projenin bilgisini projenin kendisiyle aynı yerde saklamak anlamına gelir. Örneğin `README.md`, `docs/API.md`, `docs/ADR/0001-database-choice.md` gibi dosyalar kodla birlikte görünür. Bu yaklaşım yeni gelen kişinin "belge nerede?" sorusunu azaltır. Ancak gizli bilgiler bu dosyalara yazılmamalıdır; API

anahtarı, gerçek müşteri verisi veya parola gibi bilgiler dokümantasyon konusu olabilir, fakat değerleri repoya konulmamalıdır.

Dokümantasyonu versiyonlamak

Dokümantasyon Git ile versiyonlandığında, yalnızca son halini değil, nasıl değiştiğini de görebiliriz. Bu küçük gibi görünür; fakat proje yönetimi açısından önemlidir. Bir kurulum adımı ne zaman değişti, bir API alanı ne zaman eklendi, eski karar hangi commit ile terk edildi? Bu soruların yanıtı dokümanın geçmişinde bulunabilir. Böylece belge, yalnızca açıklama değil, proje hafızası haline gelir.

5.2.1 PR ile dokümantasyon güncellemek

Değişiklik isteği ile dokümantasyon güncellemek, değişikliğin ekip tarafından görünür biçimde incelenmesini sağlar. Örneğin bir geliştirici yeni raporlama endpoint'i eklediyse, aynı PR içinde `docs/API.md` dosyasını da güncellemelidir. Böylece inceleyen kişi yalnızca kodun çalışıp çalışmadığına değil, başkasının bu özelliği anlayıp anlayamayacağına da bakar.

Basit bir durum şöyle görünebilir:

```
$ git diff -- docs/API.md
+ GET /reports/monthly
+ Aylık satış özetini döndürür. `month` alanı YYYY-MM biçiminde gönderilir.
```

Bu birleştirme, dokümantasyon değişikliğinin kod değişikliğiyle aynı inceleme sürecine girdiğini gösterir. Yönetim Bilişim Sistemleri açısından bu, kalite güvencesinin yalnızca testlerden değil, bilginin doğru aktarılmasından da geçtiğini hatırlatır.

Kod değişince dokümanı da değiştirmek

Kod değişince dokümanı da değiştirmek, ek iş gibi görünse de aslında bakım maliyetini düşürür. Yeni bir ortam değişkeni eklendiyse `.env.example` güncellenmeli, yeni bir kullanıcı rolü geldiyse yetki açıklaması değiştirilmelidir. Bu alışkanlık kurulmadığında bilgi önce sohbetlerde, sonra kişilerin hafızasında dağılır. Bir yardımcı fonksiyonun iç değişkeni yeniden adlandırıldığında doküman sessiz kalabilir; fakat `.env.example` dosyasına yeni bir ortam değişkeni eklendiğinde veya bir kullanıcı rolü yetki sınırını değiştirdiğinde doküman da aynı anda güncellenir, çünkü bu değişiklikler dışarıdan bakan birinin projeyi yanlış kurmasına yol açabilir.

5.3 Markdown Temelleri

Dokümanı repository içinde tutup Git ile versiyonlamak, doğru biçimde yazılmadığı sürece tek başına yeterli olmaz; hem hızlıca düzenlenebilen hem de okunaklı görüntülenen bir dile ihtiyaç vardır. Markdown, teknik dokümantasyonda sık kullanılan sade bir işaretleme dilidir.[2] Onu özel yapan şey, hem düz metin gibi okunabilmesi hem de

[2] Krewinkel, A., & Winkler, R. (2017). Formatting open science: Agilely creating multiple document formats for academic manuscripts with Pandoc Scholar. *PeerJ Computer Science*. https://doi.org/10.7717/peerj-cs.112

GitHub gibi platformlarda biçimlendirilmiş belge olarak görüntülenebilmesidir. Yani aynı dosya hem geliştirici için hızlıca düzenlenebilir, hem de ekip için okunabilir bir sayfaya dönüşebilir.

Basit bir proje dokümantasyonu şu yapıyla başlayabilir:

```
project/
├─ README.md
├─ .env.example
├─ docs/
│  ├─ API.md
│  └─ ADR/
│     └─ 0001-database-choice.md
└─ src/
```

Bu klasör yapısı bize şunu söyler: README projenin ana kapısıdır, `.env.example` gerekli ayarları gösterir, `docs/API.md` servis sözleşmesini açıklar, ADR klasörü ise önemli kararları kayda geçirir. Ancak Markdown dosyaları çoğaldıkça bakım maliyeti de artar. Bu sebeple her dosyanın bir soruya cevap vermesi gerekir; cevap vermeyen dosya, zamanla yalnızca klasör kalabalığı üretir.

Başlık, paragraf, liste ve link kullanımı

Markdown'da başlıklar `#`, `##`, `###` gibi işaretlerle kurulur; paragraflar düz metin olarak yazılır; listeler ise adımları veya kontrol noktalarını görünür hale getirir. Bir link, dış kaynağa ya da repo içindeki başka bir dosyaya yönlendirme sağlar. Örneğin README içindeki `[API dokümanı](docs/API.md)` bağlantısı, okuyucuyu doğru bilgiye taşır. Bu küçük yapı, ekip içinde "nereden bakacağım?" sorusunu azaltır.

Gerçek bir örnek olarak https://github.com/akadal/blockchain deposunun README dosyasındaki küçük bir parçaya bakalım:

```
# Akadal Chain

[Docker](./docker-compose.yml)

## Run Locally
### Prerequisites
- Docker and Docker Compose
- Node.js 18+
```

Tek `#` belge başlığını, iki ve üç işaret alt başlık düzeylerini kurar. `[Docker](./docker-compose.yml)` ifadesindeki `./`, bağlantının aynı depo içindeki dosyaya göreli olduğunu gösterir. Altındaki tireler ise gereksinimleri listeye dönüştürür. Aynı README, çalıştırma komutlarını kod bloğunda; servis adreslerini ve portları tabloda; güvenlik uyarısını ise alıntı bloğunda sunar. Biçim seçimi böylece bilginin işlevine bağlanır.

Kod bloğu, inline code, tablo ve alıntı

Teknik metinde komutlar, dosya adları ve örnek çıktılar açık seçik görünmelidir. Bu yüzden `npm run dev` gibi kısa ifadeler inline code olarak, daha uzun komutlar veya çıktı örnekleri kod bloğu olarak yazılır. Tablo, kararları karşılaştırmak için; alıntı ise bir

kuralı veya önemli uyarıyı vurgulamak için kullanılabilir. Ancak biçimlendirme amacı süs değildir. Okurun hangi bilginin komut, hangi bilginin açıklama, hangi bilginin karar olduğunu ayırt etmesini sağlamaktır.

5.3.1 GitHub'ın Zenginleştirilmiş Markdown'ı

GitHub'ın Zenginleştirilmiş Markdown'ı (GitHub Flavored Markdown), GitHub üzerinde Markdown'a eklenen bazı pratik özellikleri ifade eder. Görev listeleri, tablolar ve kod bloklarında dil belirtme bunların en sık kullanılanlarıdır. Örneğin bir PR açıklamasında şu liste görülebilir:

```
- [x] README güncellendi
- [x] API örneği eklendi
- [ ] Ekran görüntüsü eklenecek
```

Bu liste yalnızca biçimsel bir ayrıntı değildir. Ekip, dokümantasyon işinin hangi parçasının tamamlandığını ve hangisinin beklediğini aynı yerde görür. Elbette bu dosyalar repoda paylaşılacağı için gizli bilgi içermemelidir; görev listesi süreç bilgisini taşımalı, sır veya kişisel veri taşımamalıdır.

Teknik metinde okunabilirlik

Teknik metinde okunabilirlik, cümlelerin güzel görünmesinden daha fazlasıdır. Okuyucu hangi sırayla ilerleyeceğini, hangi komutu çalıştıracağını ve hata alırsa nereye bakacağını anlamalıdır. Uzun paragraflar, açıklanmayan kısaltmalar ve amacı belirsiz listeler dokümantasyonu zorlaştırır. Şu iyi bir ölçüt olacaktır: başka bir ekip üyesi bu metinle projeyi çalıştırabiliyor, temel kararı anlayabiliyor ve yanlış yerde vakit kaybetmiyorsa metin yeterince okunabilirdir.

5.4 README Yazımı

README, projenin ana kapısıdır. Bir kişi repoyu ilk kez açtığında çoğu zaman önce README dosyasına bakar. Bu dosya projenin ne yaptığını, nasıl kurulacağını, nasıl çalıştırılacağını ve katkı sürecinde nelere dikkat edileceğini gösterir. İyi bir README, projeyi açıklayan afiş değildir; projeye giriş rehberidir.

Küçük bir README iskeleti şu şekilde kurulabilir:

```
# Stok Takip Uygulaması

Bu proje, küçük işletmeler için ürün stok hareketlerini takip eden bir web
↪ uygulamasıdır.

## Gereksinimler
- Node.js 20
- PostgreSQL 16

## Kurulum
1. `.env.example` dosyasını `.env` olarak kopyalayın.
2. Veritabanı bağlantı bilgilerini doldurun.
```

```
3. Bağımlılıkları yükleyin.

## Çalıştırma
Komut: npm run dev

## Temel Kullanım
- Ürün ekleme
- Stok girişi
- Aylık stok raporu alma

## Hesap kuralı
Kritik stok şu eşitsizlikle oluşur:
$stok < yenidenSiparisDuzeyi$

## Katkı
Her değişiklik için issue açın, özellik dalı oluşturun ve PR gönderin.
```

Gördüğünüz gibi bu iskelet uzun değildir. Fakat proje hakkında ilk bakışta bilinmesi gereken ana soruları cevaplar: ne yapıyor, neye ihtiyaç duyuyor, nasıl başlıyor, nasıl kullanılıyor, nasıl katkı alıyor? README'nin gücü burada ortaya çıkar. Kısa ama doğru bir README, hem ekip arkadaşına hem danışmana hem de YZ aracına projenin çerçevesini verir.

Ancak README'yi her şeyi taşıyan dev bir dosyaya dönüştürmek doğru değildir. API ayrıntıları, mimari kararlar veya uzun test senaryoları büyüdükçe ayrı dokümanlara taşınmalıdır. README kapıdır; bütün bina değildir.

Projenin ana kapısı olarak README

Repository kök dizinindeki `README.md`, projenin ilk karşılaşma noktasıdır. Bu dosyada "Stok Takip Uygulaması, küçük işletmelerin ürün giriş ve çıkışlarını izlemek için geliştirilmiştir" gibi bir cümle bile okuyucunun zihninde bağlam oluşturur. README paylaşılabilir ve versiyonlanabilir olmalıdır; fakat gerçek bağlantı şifreleri, kişisel kimlik numaraları, müşteri listeleri veya özel servis anahtarları burada yer almamalıdır.

5.4.1 Kurulum, çalıştırma ve konfigürasyon

Kurulum bölümü, projenin bilgisayara nasıl yerleşeceğini; çalıştırma bölümü, uygulamanın nasıl çalıştırılacağını; konfigürasyon bölümü ise hangi ayarların dışarıdan verilmesi gerektiğini anlatır. Bu ayrımı yapmak önemlidir; çünkü hata alındığında sorunun hangi katmanda aranacağını belirlemeyi kolaylaştırır.

Örneğin `.env.example` dosyasında şu satır bulunabilir:

```
DATABASE_URL=postgresql://user:password@localhost:5432/stock_app
```

Bu satır gerçek parola vermek için değil, beklenen ayar biçimini göstermek için vardır. Okuyucu kendi bilgisayarındaki kullanıcı adı, parola ve veritabanı adını buraya uyarlayacağını anlar. Böylece dokümantasyon hem yol gösterir hem de gizli bilginin repoya yazılmasını engeller.

Kullanım örnekleri ve katkı bilgisi

README içinde kısa kullanım örnekleri, projenin yalnızca nasıl kurulduğunu değil, ne amaçla kullanılacağını da gösterir. "Ürün eklemek için Yönetim > Ürünler sayfasına gidin" gibi bir cümle, özellikle ders projesi sunumlarında yararlıdır. Katkı bilgisi ise ekibin çalışma düzenini sadeleştirir: issue açma, dal oluşturma, PR gönderme ve inceleme beklentileri burada kısaca yazılabilir. Bu bölüm uzun bir yönetmelik değil, ortak çalışma hatırlatıcısı olmalıdır.

5.4.2 Sık yapılan README hataları

Sık yapılan hata, README'yi ya çok boş ya da gereksiz ayrıntılarla dolu bırakmaktır. Bir ekip yalnızca "Bu proje stok takip uygulamasıdır" yazıp bırakırsa yeni gelen kişi projeyi çalıştıramaz. Başka bir ekip ise her ekranın uzun açıklamasını README'ye koyar; bu kez dosya büyür ve kimse okumaz (Tablo 5.1).

Tablo 5.1: README'de sık görülen boşluklar ve giderilme yolu

Bulgu	Risk	Aksiyon
Kurulum komutu yok	Yeni ekip üyesi projeyi çalıştıramaz	Gereksinim ve çalıştırma adımları ekle
`.env` örneği yok	Gizli bilgiler repoya yazılabilir	`.env.example` oluştur
Her ayrıntı README'de	Ana kapı okunamaz hale gelir	Uzun içeriği `docs/` altına taşı
Katkı süreci yazılı değil	Yeni katkıda bulunan issue veya PR akışını bilmez	Katkı bölümüne issue, dal ve PR adımları eklenir

README hatalarının asıl maliyeti belge estetiği değil, ekip zamanı ve teslim güvenilirliğidir.

5.5 Teknik Karar Kayıtları

Projelerde bazı kararlar vardır; kodun içine bakarak neden alındığını anlamak zordur. Örneğin ekip PostgreSQL yerine SQLite kullanmış olabilir. Kod bize SQLite kullanıldığını gösterir, fakat nedenini göstermez. Belki proje küçük tutulmak istenmiştir, belki ders kapsamında kurulum kolaylığı önceliklidir, belki de ekipte PostgreSQL deneyimi yoktur.

Teknik karar kaydı (Architecture Decision Record, ADR), bu tür kararları kısa ve düzenli biçimde kaydetmek için kullanılan bir doküman türüdür.[3] ADR genellikle `docs/ADR/` klasöründe tutulur ve her dosya tek bir önemli kararı anlatır. ADR, uzun bir

[3] Tyree, J., & Akerman, A. (2005). Architecture decisions: Demystifying architecture. *IEEE Software*, *22*(2), 19. https://doi.org/10.1109/MS.2005.27 Buchgeher, G., Schoeberl, S., Geist, V., Dorninger, B., Haindl, P., & Weinreich, R. (2023). Using architecture decision records in open source projects—An MSR study on GitHub. *IEEE Access*, *11*, 63725–63740. https://doi.org/10.1109/ACCESS.2023.3287654

mimari rapor yerine kararın bağlamını, seçilen yolu ve beklenen sonuçlarını kısaca kaydeder; böylece karar gelecekte de anlaşılır kalır.

Basit bir yapı şöyle olabilir:

```
docs/
└─ ADR/
   ├─ 0001-veritabani-secimi.md
   └─ 0002-kimlik-dogrulama-yaklasimi.md
```

Bu dosyalar zamanla proje hafızasına dönüşür. Ancak her küçük karar için ADR yazmak da bakım maliyeti üretir. Hangi buton renginin seçildiği ADR konusu değildir; veritabanı, kimlik doğrulama, mimari katman, dış servis veya veri saklama kararı çoğu zaman ADR konusu olabilir.

Bir proje ekibi dönem başında "veritabanını bulutta mı tutalım, yerel mi çalıştıralım?" diye tartışmış olsun. O an herkes gerekçeleri hatırlar. Ancak üç hafta sonra bağlantı sorunları çıktığında aynı tartışma yeniden başlar. ADR, bu tartışmayı kısa bir karar notuna dönüştürür.

Mimari karar kaydı, projenin teknik yönünü etkileyen bir kararın bağlamını, seçilen çözümü ve sonuçlarını kaydeden kısa belgedir. "Mimari" kelimesi burada gözünüzü korkutmasın; Yönetim Bilişim Sistemleri projelerinde bu çoğu zaman sistemin nasıl kurulacağı, hangi servisle konuşacağı veya hangi veriyi nerede tutacağı gibi pratik kararları ifade eder. Peki ADR bir kullanım kılavuzu mudur? Hayır; ADR kararın kendisini ve gerekçesini kaydeder, adım adım nasıl kullanılacağını anlatmaz — o bilgi README'nin veya API dokümanının işidir.

5.5.1 Karar neden kayıt altına alınır?

Karar kaydı, ekibi geçmişteki gerekçeye geri götürür. Teslim tarihine iki gün kala bir servis yavaş çalışıyorsa, ekip "neden bu servisi seçmiştik?" sorusunu tekrar tartışmak zorunda kalabilir. Eğer ADR varsa, kararın o günkü şartları görünür olur: ücretsiz plan yeterli görülmüştür, ekipte deneyim vardır, alternatif servislerin kurulum maliyeti yüksektir.

Bu kayıt kararın her zaman doğru kalacağını garanti etmez. Fakat yanlış ya da eskimiş bir karar bile kayıtlıysa daha sağlıklı değiştirilebilir. Önemli olan, ekibin teknik tercihi hafızaya değil, izlenebilir bir belgeye dayandırmasıdır.

5.5.2 Bağlam, karar ve sonuçlar

Bir ADR genellikle üç ana parçaya dayanır: bağlam, karar ve sonuçlar. Bağlam, o gün hangi sıkışmanın yaşandığını anlatır. Karar, neyin seçildiğini tek cümleyle söyler. Sonuçlar ise kazancı ve ödenen bedeli yan yana koyar.

Küçük bir stok projesinde örnek şöyle durabilir:

> **Bağlam.** Ekip dönem içinde hızlı ilerlemek istiyor; henüz eşzamanlı yazma yükü yok.
> **Karar.** İlk sürümde SQLite kullanılacak.

Sonuçlar. Kurulum sadeleşir. Buna karşılık sonraki dönemde eşzamanlı yazma veya uzak erişim gerekirse PostgreSQL'e geçiş ayrı bir karar olur.

Bu üç parça, kararı ezberden değil, o günkü şartlardan okumayı sağlar.

Değişen kararları yönetmek

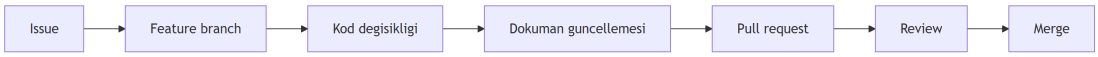
Kararlar değişebilir. Örneğin dönem başında SQLite yeterliyken, proje büyüdükçe PostgreSQL'e geçmek gerekebilir. Bu durumda eski ADR silinmek zorunda değildir; "yerine geçti" veya "geçerliliğini yitirdi" gibi bir durum bilgisi eklenebilir ve yeni karar ayrı ADR olarak yazılabilir. Böylece ekip yalnızca son kararı değil, kararın evrimini de görür. Bu yaklaşım özellikle proje tesliminden sonra bakım yapacak kişiler için değerlidir.

5.6 Diyagramlarla Dokümantasyon

Bazı konular metinle anlatılabilir; bazıları ise ilişki görmeden zor anlaşılır. Bir siparişin kullanıcıdan arayüze, arayüzden API'ye, API'den veritabanına nasıl gittiğini uzun uzun yazabiliriz. Ancak çoğu zaman küçük bir diyagram, bu akışı daha hızlı görünür hale getirir.

Diyagramın işi süslemek değil, ekibin aynı sistem resmine bakmasını sağlamaktır. Özellikle veri akışı, sorumluluk sınırı, servisler arası iletişim ve hata noktaları diyagramla daha kolay tartışılır. Bu, bilgi sistemini yalnızca ekranlardan değil, süreç ve aktör ilişkilerinden okumayı sağlar.

Basit bir GitHub dokümantasyon akışı Şekil 5.1 ile örneklenmiştir.



Şekil 5.1: Kod değişikliğiyle birlikte ilerleyen dokümantasyon akışı

Bu diyagram bize şunu gösterir: dokümantasyon en sonda hatırlanan ayrı bir iş değil, kod değişikliğinin doğal parçasıdır. Ancak diyagram da doküman gibi güncel tutulmalıdır. Eski bir diyagram, ekipte yanlış sistem resmi oluşturabilir.

Metinle anlatmanın sınırları

Metin, gerekçeyi ve ayrıntıyı anlatmakta güçlüdür; fakat ilişki yoğunlaştığında okuru yorabilir. "Kullanıcı formu doldurur, arayüz API'ye istek atar, API stok servisini çağırır, servis veritabanını günceller" cümlesi anlaşılırdır; ancak üç servis daha eklendiğinde zihinde takip zorlaşır. Diyagram bu noktada metnin yerine geçmez, metni tamamlar.

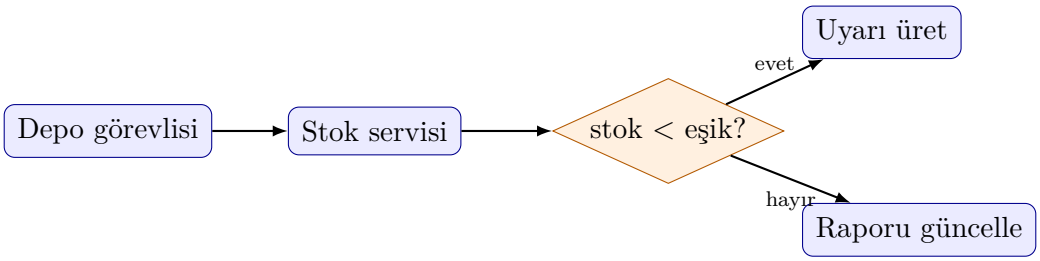
Mermaid'e giriş

Mermaid, diyagramları metin olarak yazmayı sağlayan bir araçtır. GitHub gibi ortamlarda Mermaid kodu görsel diyagrama dönüştürülebilir. Bu yaklaşımın avantajı, diyagramın da kod gibi versiyonlanabilmesidir. Bir ekip `docs/architecture.md` içinde Mermaid diyagramı tuttuğunda, sistem akışı değiştiğinde diyagram satırları da PR içinde görülebilir. Bu, görsel bilginin de proje hafızasına katılması anlamına gelir.

Stok uyarısı için kısa bir akış örneği:

```
flowchart LR
  A[Depo görevlisi] --> B[Stok servisi]
  B --> C{stok < eşik?}
  C -->|evet| D[Uyarı üret]
  C -->|hayır| E[Raporu güncelle]
```

Aynı metin GitHub'da Şekil 5.2 gibi bir diyagrama dönüşür. Kod kısa kaldığı sürece ekip, görseli değil satırları tartışır.



Şekil 5.2: Mermaid akışının derlenmiş hali

Akış ve sıralama diyagramları

Akış diyagramı (flowchart), adımların hangi sırayla ilerlediğini göstermek için kullanışlıdır. Sıralama diyagramı (sequence diagram) ise farklı aktörlerin veya servislerin birbirleriyle hangi sırayla konuştuğunu gösterir.

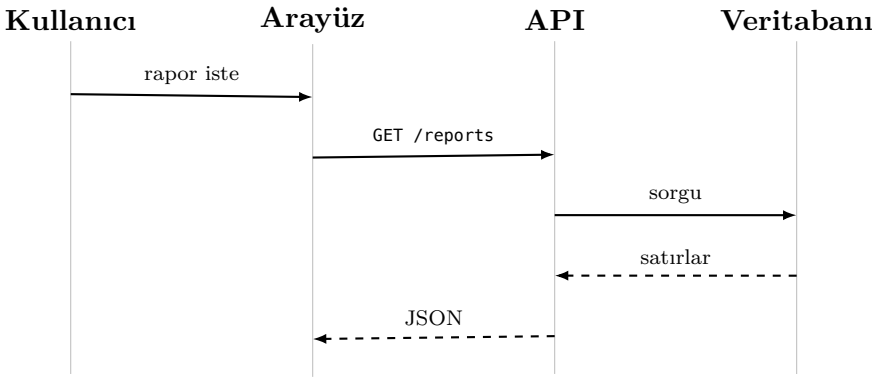
Rapor isteği için kısa bir sıralama örneği:

```
sequenceDiagram
  Kullanici->>Arayuz: rapor iste
  Arayuz->>API: GET /reports
  API->>Veritabani: sorgu
  Veritabani-->>API: satirlar
  API-->>Arayuz: JSON
  Arayuz-->>Kullanici: tablo
```

Şekil 5.3 aynı konuşmayı gösterir. Akış diyagramı süreç sırasını, sıralama diyagramı iletişim sırasını görünür kılar.

Diyagramı kodla birlikte versiyonlamak

Diyagram ayrı bir görsel dosyada ve kimsenin güncellemediği bir klasörde duruyorsa hızla eskir. Mermaid gibi metin tabanlı diyagramlar bu sorunu azaltır; çünkü değişiklik Git diff içinde görünür. Örneğin yeni bir "Bildirim Servisi" eklendiğinde diyagramdaki satır da değişir ve inceleyen bunu PR içinde fark eder. Bunun bedeli, diyagramın her sınıfı ve her fonksiyonu göstermemesidir; bu yüzden diyagram yalnızca karar vermeyi kolaylaştıracak seviyede tutulur, ayrıntı metne ve koda bırakılır.



Şekil 5.3: Sıralama diyagramı: rapor isteğinin servisler arası konuşması

5.7 API ve Veri Dokümantasyonu

Bir uygulamada arayüz ile sunucu, sunucu ile başka servisler veya iki farklı ekip çoğu zaman API üzerinden haberleşir. Bu haberleşmenin nasıl yapılacağı yazılı değilse, taraflar birbirinden farklı beklentiler geliştirebilir. Arayüz geliştirici `customerName` beklerken API `name` döndürüyor olabilir. Bu küçük fark, teslim haftasında büyük zaman kaybına dönüşebilir.

API ve veri dokümantasyonu, sistemin hangi uç noktaları sunduğunu, hangi isteği beklediğini, nasıl yanıt verdiğini ve verinin hangi alanlardan oluştuğunu açıklar. Küçük projelerde bu bilgi `docs/API.md` içinde tutulabilir. Daha büyük projelerde OpenAPI gibi araçlar kullanılabilir; fakat bu kitapta asıl mesele araç değil, sözleşme fikridir.

Basit bir API dokümanı şu soruları cevaplamalıdır:

```
docs/API.md
- Hangi endpoint var?
- Hangi method kullanılıyor?
- Hangi request alanları gerekiyor?
- Başarılı response nasıl görünüyor?
- Hata durumunda hangi status code döner?
```

Bu dosya güncel değilse bakım maliyeti üretir. Çünkü ekip yanlış sözleşmeye göre kod yazar. Bu yüzden API dokümantasyonu, kodla birlikte değişmesi gereken canlı bir anlaşma gibi düşünülmelidir.

Uç nokta, istek ve yanıt örnekleri

Uç nokta (endpoint), API'nin hangi adresten hangi işlemi sunduğunu gösterir. İstek (request), istemcinin ne gönderdiğini; yanıt (response), sunucunun ne döndürdüğünü anlatır. Örneğin `GET /products/42` isteği, 42 numaralı ürünü sorgulamak için kullanılabilir. Dokümandaki küçük bir JSON yanıtı, bütün olasılıkları sıralamadan doğru sözleşmeyi görünür kılar ve arayüz geliştiricinin kullanacağı alanları gösterir.

Durum kodu açıklamaları

Durum kodları (status codes), API yanıtının nasıl yorumlanacağını gösterir. `200` başarılı

sonucu, `400` hatalı isteği, `401` yetkisiz erişimi, `404` bulunamayan kaydı ifade edebilir. Bu kodları dokümana yazmak yalnızca teknik düzen değil, kullanıcı deneyimi ve destek süreci açısından da önemlidir. Arayüz `404` aldığında "ürün bulunamadı" mesajı gösterebilir; `500` aldığında kullanıcıyı suçlamadan sistem hatası bildirimi yapmalıdır.

Sık karşılaşılan kodlar Tablo 5.2 ile özetlenebilir.

Tablo 5.2: Sık kullanılan HTTP durum kodları

Kod	Anlamı
200	İstek başarılı; yanıt kullanılabilir.
201	Kayıt oluşturuldu; çoğu kez yeni kaynağın adresi de döner.
204	İşlem oldu, gövde boştur; silme sonrası sık görülür.
301	Kalıcı yönlendirme; eski adres bir daha kullanılmamalıdır.
400	İstek bozuk veya eksik; istemci tarafını kontrol edin.
401	Kimlik doğrulanmadı; oturum veya token gerekir.
402	Ödeme gerekli; yeni sayılabilecek, ücretli API'lerde görülür.
403	Kimlik var ama yetki yok.
404	Kaynak yok veya bu kullanıcıya gösterilmiyor.
429	Çok sık istek; bir süre bekleyip yeniden denemek gerekir.
500	Sunucu iç hatası; kullanıcıyı suçlamadan log bakılır.

Tablodaki en sık karıştırılan ayrım 401 ile 403'tür: 401 kimliğin hiç doğrulanmadığını, 403 ise kimliğin bilindiğini fakat yetkinin yetmediğini gösterir. Bu ayrımı doğru kaydetmek, ekibin bir hata raporunda oturum sorununu mu yoksa yetki sorununu mu aradığını daha ilk bakışta ayırt etmesini sağlar.

Durum odaklı API belgesi: Bruno örneği

Uç nokta listesi sözleşmeyi gösterir; fakat ekip bazen başka bir şeye ihtiyaç duyar: "şu özellik, baştan sona nasıl işler?" Bruno gibi araçlar isteği koleksiyon olarak tutar. Burada asıl fikir ürün adı değil, *durum* odaklı belgedir. "Kritik stok uyarısı" bir uç nokta değil, bir senaryodur: görevli giriş yapar, eşiğin altındaki ürünü sorar, 200 ile uyarı listesini alır; yetkisiz kullanıcı 401, olmayan ürün 404 görür.

Aynı senaryo koleksiyonda birkaç istek olarak durabilir. Ortam değişkeni geliştirme ve hazırlama adresini ayırır; örnek yanıt beklenen gövdeyi kilitler. Bruno'da bu durum kabaca şöyle durur:

```
koleksiyon: kritik-stok-uyarisi
ortam: baseUrl=http://localhost:3000

1. giris
  POST /auth/login
  {"email":"depo@ornek.com"}
  beklenen: 200

2. kritik liste
  GET /products?critical=true
```

```
  header: Authorization
  beklenen: 200, stockQuantity < reorderLevel

3. yetkisiz deneme
  GET /products?critical=true
  header yok
  beklenen: 401
```

Arayüz ekibi, test yazan kişi ve YZ aynı durumu izole okur. OpenAPI'nin yerini almaz; sözleşmenin yanında, özelliğin nasıl yaşandığını anlatır.

5.7.1 Veri modeli ve alan açıklamaları

Şöyle bir formu ele alalım. Formda "ürün adı", "stok miktarı" ve "kritik seviye" alanları varsa, ekip bu alanların ne anlama geldiği konusunda aynı fikirde olmalıdır. `stock` alanı depodaki mevcut miktarı mı, satışa uygun miktarı mı, yoksa toplam girişi mi ifade ediyor? Bu ayrım yazılmadığında raporlar yanlış yorumlanabilir.

Veri modeli, sistemdeki temel nesnelerin hangi alanlardan oluştuğunu ve bu alanların ne anlama geldiğini açıklar. Örneğin `Product` modeli için `id`, `name`, `stockQuantity`, `reorderLevel` gibi alanlar tanımlanabilir. `id` alanının veritabanında nasıl indekslendiğini son kullanıcı dokümanına yazmak gerekmez; fakat `stockQuantity` alanının depodaki fiziksel miktarı mı yoksa satışa açık miktarı mı ifade ettiği mutlaka görünür olmalıdır, çünkü bu ayrım raporu okuyan herkesin aynı sayıyı aynı şekilde yorumlamasını sağlar.

5.7.2 API dokümanını YZ için bağlam olarak kullanmak

YZ aracından arayüz kodu üretmesini istediğinizde, API dokümanı güçlü bir bağlam dosyasıdır. Örneğin istem (prompt) içinde "lütfen `docs/API.md` dosyasındaki `GET /products` sözleşmesine göre ürün listeleme ekranını oluştur" demek, aracın uydurma alan adları üretmesini azaltır. Küçük bir satır bile işe yarar: `GET /products -> [{ "id": 1, "name": "Kalem", "stockQuantity": 120 }]`.

Ancak API dokümanı repoda paylaşılacağı için gerçek token, özel müşteri verisi veya kurum içi gizli uç noktalar dikkatle ele alınmalıdır. YZ için bağlam üretmek, gizli bilgiyi gereksiz yere çoğaltmak anlamına gelmez.

5.8 Markdown, LaTeX ve Wiki Seçimi

Her dokümantasyon aynı araçla yazılmak zorunda değildir. Proje içindeki kısa teknik notlar için Markdown çok uygundur. Akademik rapor, tez veya biçimsel çıktı gerektiren metinlerde LaTeX anlamlı olabilir. Kurumsal bilgi tabanlarında ise Wiki düzeni daha kullanışlı hale gelebilir. Araç seçimini popülerlikten çok bilginin kim tarafından, ne sıklıkla ve hangi amaçla kullanılacağı belirler.

Bu seçimin dayandığı ölçütler Tablo 5.3 üzerinde yer alır.

Tablo 5.3: Markdown, LaTeX ve Wiki Seçimi

Araç	Güçlü olduğu yer	Sınırlılık
Markdown	Repo içi teknik not, README, API açıklaması	Büyük ve karmaşık yayın düzeni için sınırlı kalabilir
LaTeX	Akademik rapor, formül, kaynakça, düzenli çıktı	Öğrenme eşiği ve işbirliği maliyeti daha yüksek olabilir
Wiki	Kurumsal bilgi tabanı, süreç dokümanı, sık erişilen rehber	Kodla aynı versiyon akışında olmayabilir
Otomatik araçlar	API referansı veya doküman üretimi	Yanlış kaynak verilirse yanlış dokümanı hızla çoğaltabilir

Aynı projede birden fazla araç birlikte kullanılabilir. Örneğin ekip README ve ADR dosyalarını Markdown ile tutarken, dönem sonu raporunu LaTeX ile hazırlayabilir. Kurum içinde yaşayan operasyonel bilgiler ise Wiki'de durabilir.

Peki doğru seçim nasıl yapılır? Şu soruyu sormak çoğu zaman yeterlidir: Bu belgeyi kim güncelleyecek, kim okuyacak, hangi değişiklikle birlikte güncel kalması gerekecek? Cevap kodla birlikte değişiyorsa Markdown ve repository daha uygun görünür. Cevap akademik çıktıysa LaTeX öne çıkar. Cevap kurum içi süreç paylaşımıysa Wiki daha doğal olabilir.

Proje içi dokümantasyon için Markdown

Proje içi dokümantasyon için Markdown'ın avantajı sadeliğidir. `docs/SETUP.md` dosyasına "Geliştirme veritabanını başlatmak için Docker Compose kullanılır" gibi bir not yazmak kolaydır; aynı dosya Git içinde değişiklik geçmişiyle birlikte saklanır. Bu dosyalar ekip ve YZ araçları için paylaşılabilir bağlam üretir. Ancak gizli konfigürasyon değerleri yine dışarıda tutulmalı, yalnızca örnek biçimler gösterilmelidir.

Akademik rapor için LaTeX

LaTeX, özellikle akademik raporlarda düzen, kaynakça, tablo, şekil ve numaralandırma konusunda güçlüdür. Yönetim Bilişim Sistemleri alanındaki bir okuyucu dönem projesinin teknik raporunu, literatür özetini ve kaynakçasını düzenli biçimde vermek istiyorsa LaTeX yararlı olabilir. Ancak her kısa proje notunu LaTeX'e taşımak gereksiz bir yük oluşturur. LaTeX'i proje hafızası için değil, biçimli ve teslim edilebilir akademik çıktı için düşünmek daha sağlıklıdır.

Kurumsal bilgi tabanı için Wiki

Wiki, kurum içi süreçlerin ve sık sorulan bilgilerin paylaşılmasında güçlüdür. Örneğin "yeni kullanıcı hesabı nasıl açılır?", "destek talebi hangi sırayla ele alınır?", "raporlama takvimi nedir?" gibi bilgiler Wiki'de daha görünür olabilir. Ancak Wiki kodla aynı PR sürecinden geçmiyorsa, teknik ayrıntılar hızla koddan kopabilir. Bu yüzden kodla birebir ilişkili bilgiler repository içinde, daha genel süreç bilgileri Wiki'de tutulabilir.

5.8.1 Hangi durumda hangisi?

Araç seçimini "hangisi en iyi?" diye sormak yerine "hangi bilgi nerede yaşayacak?" diye sormak gerekir. Kodla birlikte değişen bilgi Markdown ile repoda, biçimli teslim çıktısı LaTeX ile raporda, kurum genelinde sık erişilen süreç bilgisi Wiki'de daha anlamlı olabilir.

Kısa ayrım şöyle yapılabilir: Markdown geliştirme akışına yakındır; LaTeX akademik çıktı için uygundur ve Beamer gibi paketler sayesinde sunum da hazırlanabilir; Wiki kurumsal paylaşım için kullanılabilir. Bu ayrım mutlak değildir, fakat karar vermeyi kolaylaştırır.

5.9 Dokümantasyonun GitHub İş Akışına Bağlanması

Dokümantasyonun gerçekten yaşaması için geliştirme iş akışına bağlanması gerekir. Aksi halde belge güncellemek iyi niyetli ama unutulmaya açık bir ek iş olarak kalır. GitHub iş akışı bu konuda doğal bir zemin sağlar; çünkü issue, dal, commit, PR ve inceleme zaten ekip çalışmasının izlenebilir parçalarıdır.

Basit bir süreç şöyle kurulabilir:

1. Issue içinde yapılacak değişiklik ve beklenen dokümantasyon etkisi yazılır.
2. Özellik dalı üzerinde kod ve ilgili doküman birlikte değiştirilir.
3. PR açıklamasında hangi dokümanın güncellendiği belirtilir.
4. İnceleyen, kodun yanında README, API veya ADR değişikliğine de bakar.
5. CI ve inceleme geçtikten sonra birleştirme yapılır.

Bu akışın amacı bürokrasi üretmek değildir. Amaç, bilginin koddan kopmasını engellemektir. Örneğin yeni bir rapor ekranı eklendiğinde README'deki kullanım bölümü, API dokümanı ve gerekiyorsa karar kaydı aynı akış içinde güncellenir. Böylece ekip, proje bilgisini sonradan toparlamak zorunda kalmaz.

Bu yaklaşım kalite, risk ve sorumluluk yönetimiyle doğrudan ilişkilidir. Dokümantasyonun PR içinde görünmesi, "kim neyi değiştirdi, neden değiştirdi, başkası bunu nasıl anlayacak?" sorularını cevaplanabilir hale getirir.

Dokümantasyon PR'ı, yalnızca belge değişikliği içeren veya kod değişikliğiyle birlikte belgeyi de güncelleyen değişiklik isteği olabilir. Örneğin proje kurulum adımları değiştiyse küçük bir PR yalnızca README güncellemesi taşıyabilir. Bu tür PR'lar önemsiz görülmemelidir; çünkü yeni ekip üyesinin projeye giriş maliyetini doğrudan etkiler. Sınır ise açıktır: belge değişikliği incelenebilir, kısa ve gerekçeli olmalıdır.

5.9.1 Birleştirme (merge) öncesinde README güncellemesi

Yeni bir özellik kullanıcı davranışını, kurulum adımını veya çalışma komutunu etkiliyorsa README güncellenmeden birleştirmek bilgi borcu üretir. Bu borç hemen görünmeyebilir; fakat bir sonraki ekip üyesi projeyi çalıştırırken veya danışman projeyi incelerken ortaya çıkar.

PR açıklamasındaki küçük bir kontrol listesi bu alışkanlığı güçlendirebilir:

```
## Kontrol
- [x] Kod değişikliği tamamlandı
- [x] Test edildi
- [x] README güncellendi
```

Bu liste bize şunu hatırlatır: özellik yalnızca koddan ibaret değildir. Kullanılabilir ve sürdürülebilir hale gelmesi için açıklamasının da güncel olması gerekir.

ADR olmadan büyük teknik karar almamak

Büyük teknik kararlar, kod yazılmadan önce veya en geç PR içinde ADR ile görünür hale getirilmelidir. Örneğin `docs/ADR/0003-auth-provider.md` dosyası, kimlik doğrulama için neden hazır bir servis seçildiğini açıklayabilir. Bu dosya paylaşılabilir bir karar kaydıdır; fakat servis anahtarları veya gizli yönetim paneli bağlantıları burada yer almamalıdır. ADR'nin görevi sırrı saklamak değil, gerekçeyi saklamaktır.

5.9.2 Issue, PR ve dokümantasyon ilişkisi

Issue yapılacak işi tanımlar, PR yapılan işi gösterir, dokümantasyon ise bu işin nasıl anlaşılacağını kalıcı hale getirir. Bir proje ekibi "rapor ekranı eklenecek" issue'su açtıysa, PR içinde ekran kodu, API değişikliği ve README kullanım notu birlikte görülebilir. Teslim günü hata çıktığında ekip yalnızca koda değil, issue ve PR geçmişine de bakarak değişikliğin bağlamını anlayabilir.

Bu ilişki önemlidir; çünkü proje yönetimi yalnızca görevleri kapatmak değil, kapatılan görevlerden sonra sistemin anlaşılır kalmasını sağlamaktır.

5.10 İyi Dokümantasyonun Sınırları

Dokümantasyonun faydasını gördükten sonra doğal bir eğilim oluşabilir: o halde her şeyi yazalım. İlk bakışta bu güvenli görünür. Ne kadar çok yazarsak o kadar az şey unutulur gibi düşünebiliriz. Ancak pratikte fazla dokümantasyon da en az eksik dokümantasyon kadar sorun çıkarabilir.

İyi dokümantasyon, okurun işini kolaylaştıran dokümantasyondur. Gereksiz ayrıntı, güncel olmayan bilgi ve belirsiz örnekler okuru yorar. Bu yüzden dokümanın yeterliliğini sayfa sayısıyla değil, cevapladığı sorularla değerlendirmek gerekir: proje nasıl çalışır, hangi karar neden alınmıştır, hangi veri ne anlama gelir, hangi hata durumunda nereye bakılır?

İyi ve kötü dokümantasyon arasındaki fark, küçük bir karşılaştırmada daha görünür olur:

Tablo 5.4: İyi Dokümantasyonun Sınırları

Durum	Zayıf dokümantasyon	Daha iyi dokümantasyon
Kurulum	"Projeyi çalıştırın"	Gereksinimler, komut ve beklenen çıktı yazılır
API	"Ürünleri getirir"	Endpoint, request, response ve hata kodu gösterilir
Karar	"PostgreSQL seçildi"	Neden seçildiği ve hangi bedeli olduğu yazılır
Güncellik	Eski komutlar kalır	Kod değişikliğiyle aynı PR içinde güncellenir

Tablo 5.4 içindeki karşılaştırma özünde tek bir fikri tekrarlar: iyi dokümantasyon daha uzun olan değil, okurun kararını ve eylemini daha doğru yönlendiren dokümantasyondur.

Gerektiği kadar açıklamak

Gerektiği kadar açıklamak, az yazmak anlamına gelmez; doğru soruyu cevaplayacak kadar yazmak anlamına gelir. Örneğin `npm run dev` komutunun yanına "geliştirme sunucusunu başlatır" demek yeterli olabilir. Fakat aynı komut özel bir port, özel bir veritabanı veya ek servis gerektiriyorsa bu bilgi de eklenmelidir. Yönetim Bilişim Sistemleri açısından burada denge önemlidir: doküman ekip zamanını korumalı, fakat ekibin bakım yükünü gereksiz yere büyütmemelidir.

5.10.1 Güncel olmayan dokümanın riski

Kısa bir örnek üzerinden gidelim. Ekip, geliştirme veritabanını Docker Compose ile çalıştırmaya geçmiş, fakat README hâlâ yerel PostgreSQL kurulumu anlatıyor. Yeni ekip üyesi eski adımları izliyor, farklı portta çalışan bir veritabanı kuruyor ve uygulama yanlış veritabanına bağlanıyor. Sorun teknik gibi görünür; fakat asıl problem güncel olmayan bilginin ekip davranışını yanlış yönlendirmesidir (Tablo 5.5).

Tablo 5.5: Güncel olmayan dokümanın yol açtığı riskler ve giderilme yolu

Bulgu	Risk	Aksiyon
README eski kurulum anlatıyor	Yeni geliştirici yanlış ortam kurar	README ve `.env.example` aynı PR içinde güncellenir
API dokümanı eski alan adı kullanıyor	Arayüz yanlış veri bekler	Response örneği test edilen çıktıyla eşleştirilir
ADR güncel kararı göstermiyor	Ekip eski gerekçeye göre tartışır	Yeni ADR eklenir, eski kararın durumu belirtilir
Diyagram gerçek akışı yansıtmıyor	Ekip yanlış sistem resmine göre karar alır	Mermaid diyagramı ilgili PR ile birlikte güncellenir

Bu riskin iş etkisi açıktır: zaman kaybı, yanlış teslim, ekip içi güven kaybı ve bakım maliyeti.

Okurun işini kolaylaştıran örnekler vermek

İyi örnek, okurun bir sonraki adımı daha az tahminle atmasını sağlar. Bir API dokümanında yalnızca "ürün döner" demek yerine küçük bir JSON örneği vermek, arayüz geliştiricinin alan adlarını doğru kullanmasını sağlar. Bir kurulum dokümanında yalnızca komutu değil, beklenen çıktıyı göstermek hata ayıklamayı kolaylaştırır. Ancak örnekler gerçek kişisel veri, gerçek müşteri bilgisi veya gizli anahtar içermemelidir. Okurun işini kolaylaştırmak, güvenliği gevşetmek anlamına gelmez.

5.11 Atölye

Bu bölümde kodun değişmesiyle belgenin de değişmesi gerektiğini, dokümantasyonun ayrı bir dosya yığını değil kodla aynı geçmişte yaşayan bir çalışma pratiği olduğunu konuştuk. Atölye bu fikri küçük bir depoda somutlaştırır: gerçek bir proje kurmak yerine README, bir API notu ve bir ADR'den oluşan minik bir dokümantasyon iskeleti hazırlayıp bunu Git ile versiyonlayacağız. Depo küçük tutulur ki adımlar hızlı denenebilsin; fakat aynı mantık gerçek bir projede de geçerlidir. README ve ADR'nin neden commit'lendiğine dikkat edin: ikisi de kod gibi geçmişe, diffe ve incelemeye tabi olur, böylece kim neyi ne zaman yazdığı görünür kalır. İki commit yeterlidir; ilki dokümantasyon iskeletini oluşturur, ikincisi mevcut bir dosyaya küçük bir ekleme yaparak kelime düzeyindeki bir değişikliğin nasıl izlendiğini gösterir.

Görev 1: Dokümantasyon iskeletini kur

Komut

```
$ mkdir -p ~/gelistirme-atolyesi/bolum-5/docs-app
$ cd ~/gelistirme-atolyesi/bolum-5/docs-app
$ git init -b main
$ git config user.name "Atolye Kullanici"
$ git config user.email "atolye@example.com"
$ mkdir -p docs/adr docs/diagrams
$ touch README.md docs/API.md
$ touch docs/adr/0001-veritabani.md
$ touch docs/diagrams/README.md
```

Beklenen çıktı

Komutlar hata vermeden tamamlanır; `touch` ve `mkdir` başarılı olduğunda çıktı üretmez.

Görev 2: README, API ve ADR içeriğini yaz

Komut

```
$ printf '%s\n' '# Docs App' '' '## Kurulum' \
  'Komut: npm ci' > README.md
$ printf '%s\n' '# API' '' '## GET /health' \
  'Basarili yanit: 200 OK' > docs/API.md
```

```
$ printf '%s\n' '# ADR 0001: Veritabani' '' \
  '## Karar' 'PostgreSQL kullanilacak.' \
  > docs/adr/0001-veritabani.md
$ printf '%s\n' '# Diyagramlar' \
  'Mimari diyagramlar bu dizinde tutulur.' \
  > docs/diagrams/README.md
```

Beklenen çıktı

Komutlar çıktı üretmez; `>` hedef dosyaların içeriğini oluşturur.

Görev 3: Yapıyı ve metin büyüklüğünü incele

Komut

```
$ find . -type f | grep -v '/.git/' | sort
$ wc -w README.md docs/API.md docs/adr/0001-veritabani.md
```

Beklenen çıktı

```
./README.md
./docs/API.md
./docs/adr/0001-veritabani.md
./docs/diagrams/README.md
<sayi> README.md
<sayi> docs/API.md
<sayi> docs/adr/0001-veritabani.md
```

Görev 4: Başlıkları ve bağlantı adaylarını tara

Komut

```
$ rg -n '^#{1,3} ' README.md docs
$ printf '\n## Belgeler\n- [API](docs/API.md)\n' >> README.md
$ rg -o '\x5b[^]]+\]\([^)]+\)' README.md
```

Beklenen çıktı

```
README.md:1:# Docs App
README.md:3:## Kurulum
docs/API.md:1:# API
...
[API](docs/API.md)
```

Görev 5: Eksik başlık ve geçici not ara

Komut

```
$ rg --files-without-match '^# ' README.md docs/API.md docs/adr/*.md docs/diagrams/*.md
$ rg -n 'TODO|FIXME' README.md docs
```

Beklenen çıktı

Her dosyanın ana başlığı varsa ve geçici not yoksa iki komut da çıktı üretmez.

Görev 6: İlk dokümantasyon sürümünü kaydet

Komut

```
$ git add README.md docs
$ git diff --cached --check
$ git diff --cached --stat
$ git commit -m "Dokumantasyon iskeletini ekle"
```

Beklenen çıktı

```
 README.md                          | ...
 docs/API.md                        | ...
 docs/adr/0001-veritabani.md        | ...
[main <kimlik>] Dokumantasyon iskeletini ekle
```

`git diff --cached --check` başarılıysa çıktı üretmez.

Görev 7: Kelime düzeyindeki değişikliği incele

Komut

```
$ printf '\nKurulumdan sonra npm test calistirin.\n' >> README.md
$ git diff --word-diff -- README.md
$ git diff --check
$ git status --short
$ git add README.md
$ git commit -m "Test adimini belgeye ekle"
```

Beklenen çıktı

```
{+Kurulumdan sonra npm test calistirin.+}
 M README.md
[main <kimlik>] Test adimini belgeye ekle
```

Bölüm 6

Yapay Zekâ ile Kontrollü Geliştirme

Yapay zekâ araçları şüphesiz yazılım geliştirme sürecine insan hızıyla kıyaslanması zor bir hız getirdi. Artık bir fonksiyon iskeleti, bir test taslağı, bir hata açıklaması veya bir README bölümü için uzun süre boş sayfaya bakmak zorunda kalmayabiliyoruz. Bu cazibe gerçektir. Özellikle öğrenme aşamasındaki bir okuyucu için YZ, doğru kullanıldığında iyi bir açıklayıcı, deneme arkadaşı ve üretkenlik destekçisi olabilir.

Ancak aynı hız, kontrol edilmediğinde kolayca yüzeysel üretime dönüşür. Kod çalışıyor gibi görünebilir ama iş kuralını yanlış anlamış olabilir. Test var gibi görünebilir ama kritik senaryoyu sınamıyor olabilir. Dokümantasyon akıcı olabilir ama projedeki gerçek dosya adlarını veya API sözleşmesini uyduruyor olabilir. Bu yüzden YZ ile geliştirme, yalnızca daha hızlı kod yazma meselesi değildir; bağlam verme, kabul kriteri tanımlama, test etme, denetleme ve sorumluluk alma meselesidir.

Bu bölümde YZ kullanımına ne aşırı hayranlıkla ne de bütünüyle korkuyla yaklaşacağız. Yönetim Bilişim Sistemleri perspektifinden daha dengeli bir soru soracağız: YZ, ekip çalışmasını, proje hafızasını, kalite güvencesini ve akademik dürüstlüğü zayıflatmadan nasıl kullanılabilir? Cevap, kontrolsüz üretimde değil, kontrollü geliştirme akışındadır.

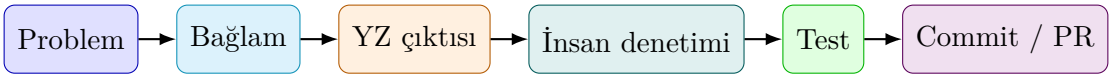
6.1 Yapay Zekâ Destekli Geliştirme Nedir?

Bir ders projesinde çoğu zaman aynı anda birçok iş üst üste gelir. Ekran tasarlanacak, API yazılacak, veritabanı kurulacak, hata ayıklanacak, test hazırlanacak ve teslim raporu düzenlenecektir. Bu yoğunluk içinde "keşke birisi şu kodu açıklasa", "şu fonksiyon için test taslağı çıkarsa", "bu hatanın muhtemel sebebini söylese" demek oldukça doğaldır.

Yapay zekâ destekli geliştirme, yazılım geliştirme sürecinde YZ araçlarından planlama, kod yazma, kod açıklama, hata ayıklama, test üretme, dokümantasyon hazırlama veya PR açıklaması yazma gibi işlerde destek almak anlamına gelir. Buradaki ana kelime "destek"tir. YZ geliştiricinin yerine geçen sihirli bir mekanizma değil, doğru bağlam verildiğinde üretimi hızlandırabilen bir yardımcıdır.[1]

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, *30*. Han, S., Wang, M., Zhang, J., Li, D., & Duan, J. (2024). A review of large language

Peki bu destek nerede başlar, nerede biter? Şekil 6.1 bu sırayı gösterir.



Şekil 6.1: YZ destekli geliştirmede denetimli akış

Bu akışta YZ çıktısı tek başına son ürün değildir. Çıktı, insan tarafından okunur, projeye uygunluğu kontrol edilir, test edilir ve gerekiyorsa düzeltilir.

YZ kod yazabilir mi, yazılım geliştirebilir mi?

YZ kod yazabilir; hatta çoğu zaman hızlı ve ikna edici kod parçaları üretebilir. Ancak yazılım geliştirmek yalnızca kod yazmak değildir. Problem alanını anlamak, kullanıcı ihtiyacını ayırmak, doğru veri modelini seçmek, güvenlik riskini görmek, test etmek ve ekip kararlarını yönetmek de geliştirme sürecinin parçasıdır. Bu sebeple "YZ yazılım geliştirdi" demek yerine "YZ geliştirme sürecinin bazı parçalarına destek oldu" demek daha doğru ve daha sağlıklı bir ifadedir.

6.1.1 Yardımcı, kodlama asistanı, ajan ve otomasyon farkları

YZ araçlarını tek bir kategori gibi düşünmek kafa karıştırabilir. Yardımcı, genellikle soru cevaplar veya küçük metin/kod üretir. Kodlama asistanı (copilot), editör içinde çalışarak geliştiricinin yanında öneriler sunar.[2] Ajan, daha geniş bir hedef alıp dosyaları okuyabilir, plan yapabilir ve değişiklik önerebilir. Otomasyon ise belirli işi kurala bağlar; örneğin test çalıştırmak veya formatlamak gibi (Tablo 6.1).

Tablo 6.1: Dört YZ yaklaşımının uygunluk anı ve sınırları

Yaklaşım	Ne zaman uygun?	Sınır
Yardımcı	Kavram açıklama, küçük örnek	Bağlamı siz taşırsınız
Kodlama asistanı	Kod yazarken anlık destek	Öneriler denetlenmelidir
Ajan	Birden çok dosyaya yayılan iş	Yetki ve kapsam açık olmalıdır
Otomasyon	Tekrarlanan kontrol işleri	Karar vermez, kural uygular

Tablo 6.1 araç seçmek için değil, her yaklaşıma ne kadar denetim eşlik etmesi gerektiğini hatırlamak içindir; yetki ajan ve otomasyona doğru genişledikçe insan gözetimi de aynı oranda artmalıdır.

YZ'nin güçlü olduğu işler

YZ özellikle taslak üretme, açıklama yapma, alternatifleri listeleme, hata mesajını

models. *Electronics*, *13*(24). https://doi.org/10.3390/electronics13245040

[2] Dakhel, A. M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., & Jiang, Z. M. (2023). GitHub Copilot AI pair programmer: Asset or liability? *Journal of Systems and Software*, *203*, Article 111734. https://doi.org/10.1016/j.jss.2023.111734

yorumlama, test senaryosu önermek ve dokümantasyon ilk sürümü çıkarmak gibi işlerde güçlüdür. Örneğin "bu fonksiyonun ne yaptığını başlangıç düzeyinde açıkla" dediğinizde, kodu anlamaya başlamanıza yardımcı olabilir. Ancak güçlü olduğu alanlar bile denetim gerektirir; çünkü akıcı açıklama doğru açıklama anlamına gelmez.

6.1.2 YZ'nin zayıf olduğu işler

YZ'nin zayıf olduğu yer çoğu zaman örtük bağlama, güncelliğe ve sorumluluğa dayanan işlerdir. Küçük bir stok sistemi düşünelim. Bu sistemde ürünün güncel bakiyesi negatif olamaz; ancak stok hareketleri giriş ve çıkış yönünü göstermek için pozitif veya negatif değer taşıyabilir. Örneğin depodan üç ürün çıkışı `-3` hareketiyle kaydedilir, ardından yeni bakiye hesaplanır. İş kuralı, hareketin işaretini yasaklamak değil işlem sonundaki bakiyenin sıfırın altına düşmesini engellemektir.

Ekip YZ'den "stok miktarı negatif olmasın" diyerek doğrulama kodu istediğinde araç bu cümleyi bütün sayısal alanlara uygulayabilir. Ortaya temiz görünen ve birim testi de bulunan bir kod çıkar; fakat negatif hareketleri reddettiği için ürün çıkışı ve düzeltme kayıtları çalışmaz. Kullanıcılar işlemi tamamlamak için yanlış türde pozitif kayıt girmeye başlarsa teknik hata, stok raporlarını bozan bir veri kalitesi sorununa dönüşür. YZ söylenmeyen "bakiye" ile "hareket" ayrımını güvenilir biçimde çıkaramaz.

Bu hata daha iyi bir bağlam ve kabul kriteriyle azaltılabilir: "Hareket miktarı negatif olabilir; işlem sonrası kullanılabilir stok negatif olamaz." Ardından elde 5 ürün varken `-3` hareketinin kabul edildiği, `-6` hareketinin reddedildiği testlerle sınır görünür hale getirilir. Yine de YZ çıktısının güncel iş kuralına, veri modeline ve mevcut kod davranışına uyduğunu doğrulamak ekipte kalır.

Sorumluluğun geliştiricide kalması

YZ bir öneri üretir; fakat o öneriyi projeye ekleyen kişi geliştiricidir. Dönem projesinde teslimden önce YZ'nin yazdığı kodu anlamadan ekleyen biri, hata çıktığında "bunu YZ yazdı" diyerek sorumluluktan çıkamaz. Çünkü kod repoya girdiği anda ekip kararının parçası haline gelir. Bu sebeple YZ kullanımında en temel kural şudur: anlamadığınız, test etmediğiniz ve savunamayacağınız çıktıyı projeye dahil etmeyin.

6.2 Sezgisel Kodlama (Vibe Coding)

Bazen okuyucu aklındaki fikri hızlıca görmek ister. "Bir ürün listeleme ekranı olsun, arama kutusu bulunsun, stok azsa kırmızı görünsün" diye tarif eder ve YZ aracı birkaç dakika içinde çalışan bir taslak üretebilir. Bu deneyim motive edicidir; çünkü fikir ile ekran arasındaki mesafe bir anda kısalır.

Bu yaklaşıma kabaca sezgisel kodlama veya vibe coding denebilir. Burada geliştirici ayrıntılı teknik plan yazmadan, doğal dille fikrini tarif eder ve YZ'den hızlıca çalışan bir taslak üretmesini ister. Özellikle prototip aşamasında bu yöntem faydalıdır. Ekran akışı, kullanıcı deneyimi veya fikir doğrulama için hızlı sonuç verir.

Ancak sezgisel kodlamanın tehlikesi de tam burada başlar. Hızlı oluşan kod, sistemin geri kalanıyla uyumlu olmayabilir. Dosya yapısını bozabilir, test eklemeyebilir, güvenlik

sınırlarını atlayabilir veya gereksiz bağımlılık getirebilir. İlk bakışta "çalışıyor" görünen şey, daha sonra bakım yüküne dönüşebilir.

Bu yüzden bu kitapta sezgisel kodlamayı tamamen reddetmeyeceğiz; onu kontrollü hale getirmeye çalışacağız. Fikirden taslağa geçmek değerlidir, fakat taslaktan projeye geçmek için bağlam, kabul kriteri, test ve inceleme gerekir.

Hızlı prototipleme, özellikle belirsizlik yüksekken değerlidir. Bir ekip rapor ekranının nasıl görünmesi gerektiğinden emin değilse, YZ ile iki farklı taslak üretip kullanıcı akışını tartışabilir. Bu, haftalarca yanlış ekrana yatırım yapmaktan daha sağlıklı olabilir. Ancak prototipin adı üstünde taslak olduğu unutulmamalıdır. Prototipi doğrudan final kod sanmak, hızlı başlangıcı yavaş bakıma çevirebilir.

Fikirden çalışan taslağa geçişte YZ'nin katkısı, boş sayfa etkisini azaltmasıdır. "Stok seviyesi azalan ürünleri listeleyen basit bir tablo oluştur" gibi bir istek, tasarım tartışmasını somutlaştırır. Sınır ise şudur: çalışan taslak, doğrulanmış ürün değildir; veri kaynağı, yetki, hata durumu ve test hâlâ ayrıca ele alınmalıdır.

Bir ekip teslim haftasında YZ'ye "bize yönetici paneli yap" der ve çalışan bir panel alır. Panel güzel görünür, fakat bütün kullanıcılar aynı sayfaya erişebilir. Çünkü istem içinde rol ve yetki kuralı yazılmamıştır. Bu durumda teknik problem aslında iş problemine dönüşür: yetkisiz kişi stok raporlarını görebilir.

Kontrollü sezgisel kodlama, hızlı taslak üretme isteğini proje disipliniyle birleştirir. Önce küçük hedef yazılır, sonra dosya sınırı verilir, ardından kabul kriteri ve test beklentisi belirtilir. YZ'den çıkan sonuç doğrudan ana dala gönderilmez; çalıştırılır, okunur, gerekiyorsa sadeleştirilir ve PR içinde incelenir. Bu yaklaşım hızdan vazgeçmez, fakat hızı sorumlulukla dengeler.

6.3 YZ ile Geliştirmeden Önce Proje Hafızası Kurmak

YZ ile çalışırken yapılan en yaygın hatalardan biri, bütün bağlamı sohbet (chat) penceresine bırakmaktır. İlk gün proje amacını uzun uzun anlatırız. İkinci gün başka bir konuya geçeriz. Üçüncü gün aynı araca "şunu da düzelt" deriz. Fakat proje büyüdükçe sohbet geçmişi dağılır, hangi kararın güncel olduğu belirsizleşir.

Bu yüzden proje hafızası sohbet geçmişinde değil, repository içinde yaşamalıdır. README, PRD, BACKLOG, DOMAIN, ARCHITECTURE, ADR ve TESTING gibi dosyalar insana açık bağlam sağlar; AGENTS.md ise aynı görevi YZ ajanı için üstlenir. Bu ayrım gelişigüzel değildir: README bir insanın projeyi ilk kez açtığında okuduğu dosyadır, AGENTS.md ise bir YZ ajanının oturuma başlarken okuduğu dosyadır. Üstelik AGENTS.md bir yönlendirici (router) gibi çalışır: ajan hangi dosyayı hangi sırayla okuyacağını, hangi komutları çalıştıracağını ve hangi sınırlara uyacağını buradan öğrenir; ayrıntı gerektiğinde diğer Markdown dosyalarına buradan yönlendirilir. Böylece "YZ bilsin" demek yerine "repo söylesin" yaklaşımı kurulur.

Küçük bir proje hafızası haritası şöyle olabilir:

```
project/
├─ README.md
├─ AGENTS.md
├─ PRD.md
├─ BACKLOG.md
├─ .env.example
├─ docs/
│  ├─ DOMAIN.md
│  ├─ ARCHITECTURE.md
│  ├─ API.md
│  ├─ TESTING.md
│  └─ ADR/
└─ src/
```

Bu dosyalar başlangıçta iş yavaşlatıyor gibi görünebilir. Ancak doğru yazıldığında tersine hız kazandırır. Çünkü YZ'ye her seferinde aynı bağlamı yeniden anlatmak yerine, dosyaları kaynak gösterirsiniz. Ekip üyeleri de aynı belgeler üzerinden konuşur. Bu farkın nerede ortaya çıktığını görmek için sohbet kapsamlı bilgiyle repo kapsamlı bilgiyi yan yana koymak faydalıdır.

Tablo 6.2: Sohbet kapsamlı ve repo kapsamlı bağlamın karşılaştırması

Ölçüt	Sohbet kapsamlı	Repo kapsamlı
Paylaşılabilirlik	Kişiye veya araca bağlı kalabilir	Ekip tarafından görülebilir
Versiyon geçmişi	Belirsizdir	Git ile izlenir
YZ bağlamı	Sohbet içinde dağılabilir	Dosya olarak referans verilir
Denetim	Zorlaşır	PR ve inceleme ile yapılabilir

Tablo 6.2 şunu gösterir: mesele daha çok dosya üretmek değil, projenin gerçek bağlamını kalıcı ve denetlenebilir hale getirmektir.

6.3.1 Neden önce Markdown dosyaları?

Markdown dosyaları kolay yazıldığı için değil, proje bilgisini sade ve versiyonlanabilir biçimde taşıdığı için değerlidir. Örneğin `PRD.md` içinde şu kadar küçük bir bölüm bile YZ'ye ve ekibe yön verir:

```
## Problem
Küçük işletmeler stok seviyesini Excel dosyalarıyla takip ediyor.
Bu yöntem güncel stok bilgisinin ekip içinde paylaşılmasını zorlaştırıyor.

## Hedef
Ürün stok giriş ve çıkışlarını web üzerinden izlenebilir hale getirmek.
```

Bu metin kod değildir; fakat kodun hangi probleme hizmet edeceğini belirler. Markdown sadedir, ancak bakım sorumluluğu gerektirir: problem değişirse dosya da değişmelidir.

Sohbet geçmişi hızlı düşünmek için iyidir; repository hafızası ekipçe çalışmak için daha güvenilirdir. Sohbet içinde yapılan açıklama o oturuma bağlı kalabilir. Repository içindeki dosya ise commit geçmişiyle birlikte durur, PR'da incelenir ve başka araçlar tarafından da okunabilir. Ayırt etmek gerekirse: sohbet geçici düşünme alanı, repo kalıcı proje hafızasıdır. İkisini birlikte kullanmak mümkündür, fakat proje kararını yalnızca sohbette bırakmak risklidir.

Proje kararlarını dosyalaştırmak, kararın sohbetten çıkıp izlenebilir belgeye dönüşmesidir. Örneğin `docs/ADR/0002-auth.md` dosyasında "kimlik doğrulama için hazır servis kullanılacak; çünkü ekip güvenli parola saklama ayrıntılarını bu proje kapsamında yönetmeyecek" yazılabilir. Bu bilgi paylaşılabilir ve versiyonlanabilir. Ancak servis anahtarı veya yönetici parolası bu dosyaya yazılmaz; kararın gerekçesi saklanır, gizli değerler saklanmaz.

"YZ bilsin" yaklaşımı, bağlamı aracın hatırlamasına bırakır. "Repo söylesin" yaklaşımı ise bağlamı dosyalara koyar ve YZ'ye bu dosyaları referans verir. Örneğin "lütfen `DOMAIN.md` ve `API.md` dosyalarına göre stok azaltma fonksiyonunu düzenle" demek, "stok sistemimizi biliyorsun" demekten daha güvenlidir. Böylece yapılan, bilginin kişide veya araçta değil, sistemli kayıt içinde tutulmasıdır.

Proje hafızası bir kez yazılıp bırakılırsa hızla eskir. Ekip kapsamı daralttıysa PRD, yeni API alanı eklediyse API dokümanı, teknik karar değiştirdiyse ADR güncellenmelidir. Teslim haftasında "doküman başka, kod başka" durumuna düşen ekip, hem danışmana açıklama yapmakta hem de hatayı bulmakta zorlanır. Bu yüzden proje hafızasının bakımı, kod bakımının doğal parçası olarak görülmelidir.

6.4 YZ Araçlarında Bağlantı Standartları

Bir uygulamanın başka bir uygulamayla konuşması yeni bir konu değildir. Daha önce API dokümantasyonundan söz ederken gördüğümüz gibi, sistemler çoğu zaman belirli uç noktalar, istek biçimleri, yanıt örnekleri ve yetki kuralları üzerinden birbirine bağlanır. Bir stok uygulaması ödeme servisine, e-posta sağlayıcısına veya harita hizmetine bağlanabilir. Bu bağlantıların yönetilebilir olması için API sözleşmesi, kimlik doğrulama, hata kodları ve loglama gibi konular önem kazanır.

YZ araçları geliştirme ortamına girdikçe benzer bir ihtiyaç başka bir biçimde karşımıza çıkar. Artık yalnızca bir uygulamanın başka bir uygulamaya istek göndermesinden değil, YZ aracının proje dosyalarını okuyabilmesinden, issue listesini görebilmesinden, veritabanı şemasına bakabilmesinden, terminal komutu çalıştırabilmesinden veya bir dış servisten bilgi alabilmesinden söz ediyoruz. Bu durumda soru şuna dönüşür: YZ hangi bağlama erişiyor, hangi aracı çağırabiliyor ve bu işlemler hangi yetkiyle gerçekleşiyor?

Bu noktada bağlantı standardı fikri önem kazanır. Bağlantı standardı, farklı araçların birbirine her seferinde özel ve dağınık çözümlerle değil, daha ortak bir sözleşmeyle bağlanmasını amaçlar. Web dünyasında API, GraphQL, webhook ve OAuth gibi yaklaşımlar bu ihtiyacın farklı parçalarına yanıt verir. YZ tarafında ise benzer arayışlar

modelin dış dünyayla daha düzenli ve denetlenebilir ilişki kurması etrafında gelişmektedir.

Model Context Protocol, kısa adıyla MCP, YZ uygulamalarının dış sistemlere bağlanması için geliştirilen açık bir bağlantı standardı olarak düşünülebilir. Burada dış sistem dediğimiz şey yerel dosya sistemi, veritabanı, GitHub, issue takip sistemi, arama motoru, hesaplama servisi veya kurumsal bir API olabilir. MCP'nin önemi, tek tek her YZ aracı için ayrı entegrasyon yazma fikrini azaltıp, bağlam ve araç erişimini daha standart bir yapıya taşıma denemesidir.

Ancak MCP'yi "YZ her şeye bağlanacak" heyecanıyla okumak sağlıklı değildir. Asıl soru şudur: Bu bağlantı bize hangi iş problemini çözüyor ve hangi riski getiriyor? Örneğin bir YZ aracı repository içindeki `AGENTS.md`, `API.md` ve `TESTING.md` dosyalarını okuyabiliyorsa daha iyi öneri üretebilir. Fakat aynı araç gerçek `.env` dosyasına, üretim veritabanına veya dağıtım anahtarına erişiyorsa artık yalnızca üretkenlikten değil, yetki ve güvenlikten söz ederiz.

MCP anlatılırken sık karşılaşılan üç kavram kaynak (resource), araç (tool) ve istem şablonu (prompt template) kavramlarıdır. Kaynak, YZ'nin bağlam olarak okuyabileceği bilgi parçasıdır; örneğin proje dokümanı, dosya, veritabanı şeması veya API açıklaması. Araç, YZ'nin belirli bir işi yaptırmak için çağırabileceği fonksiyondur; örneğin veritabanı sorgulamak, bir API'den veri çekmek veya hesaplama yapmak. İstem şablonu ise belirli bir iş akışını başlatmak için kullanılan yapılandırılmış talimat olabilir.

Bu ayrım küçük görünür, fakat denetim açısından önemlidir. Bir kaynağı okumak ile bir aracı çalıştırmak aynı risk düzeyinde değildir. YZ'nin proje dokümanını okuması çoğu zaman düşük risklidir; fakat "müşteri kaydını sil" gibi etkisi olan bir aracı çağırması onay, log ve geri alma planı gerektirir. Bu yüzden bağlantı standardı konuşurken yalnızca "bağlandı mı?" diye değil, "neye erişti, ne yaptı, kim onayladı?" diye sormak gerekir.

MCP tek başına bütün entegrasyon dünyasının yerine geçmez. API hâlâ sistemlerin veri alışverişi için temel yollarından biridir. Webhook, bir olay gerçekleştiğinde başka sisteme haber vermek için kullanılır. Eklenti veya uzantı yapıları, belirli bir ürünün kendi ekosistemi içinde yetenek eklemeye yarar. Skill veya benzeri yapılar ise çoğu zaman YZ aracına belirli bir iş alanı, iş akışı veya kodlama davranışı hakkında talimat vermek için kullanılır.

Tablo 6.3, bu üç kavram arasındaki kısa ayrımı yan yana koyar.

Tablo 6.3: API, webhook, eklenti ve skill

Yaklaşım	Temel işlev	Sorulması gereken soru
API	Sistemler arasında veri ve işlem sözleşmesi kurar	Hangi veri gidiyor, kim yetkili, hata nasıl ele alınıyor?
Webhook	Bir olay olduğunda başka sisteme haber verir	Hangi olay tetikleniyor, tekrar çalışırsa ne olur?
Eklenti veya uzantı	Belirli bir ürün içinde yeni yetenek sağlar	Bu yetenek hangi izinleri istiyor?

Yaklaşım	Temel işlev	Sorulması gereken soru
MCP	YZ aracını dış kaynak ve araçlara standart biçimde bağlamaya çalışır	YZ neyi okuyabilir, neyi çalıştırabilir, kim onaylar?
Skill veya iş akışı talimatı	YZ'ye belirli görev için yöntem ve bağlam verir	Bu talimat güncel mi, proje kurallarıyla uyumlu mu?

Gördüğünüz gibi bu teknolojiler aynı problemi çözmez; fakat birbirine yakın bir alanı paylaşır. Hepsi sistemler arası bağlantı, bağlam taşıma veya iş akışı kurma meselesine dokunur. Bu yüzden YZ ile geliştirme ortamı kurarken yalnızca "hangi model?" sorusu yeterli değildir. "Model hangi bilgiye, hangi araçla, hangi sınır içinde erişiyor?" sorusu daha yönetilebilir bir bakış sağlar.

Bu kitapta MCP sunucusu yazmayı, ileri seviye plugin geliştirmeyi veya skill paketlemeyi öğretmeyeceğiz. Bu konular ayrı bir teknik derinlik gerektirir. Temel okuryazarlık şuradan başlar: YZ aracı bir geliştirme ortamına bağlandığında yeni bir veri, yetki ve denetim katmanı oluşur.

Bu katmanı değerlendirirken basit bir kontrol listesi yeterli bir başlangıç sağlar:

1. YZ hangi kaynakları okuyabiliyor?
2. Hangi araçları çalıştırabiliyor?
3. İşlem öncesinde kullanıcı onayı gerekiyor mu?
4. Gizli bilgi, token veya müşteri verisi erişim dışında mı tutuluyor?
5. Araç çağrıları loglanıyor mu?
6. Hatalı işlem için geri alma planı var mı?

Toparlamak gerekirse, MCP ve benzeri bağlantı standartları bu kitapta geliştirme ortamı okuryazarlığının doğal uzantısı olarak yer almaktadır. YZ'nin ne kadar akıllı göründüğünden önce, hangi bağlamla çalıştığını ve hangi yetkilerle hareket ettiğini anlamak gerekir.

6.5 Önerilen Proje Dokümantasyon Şablonu

YZ ile çalışırken dokümantasyon yapısı bir klasör gösterisi haline gelmemelidir. Her projeye aynı sayıda dosya koymak doğru değildir. Küçük bir ders alıştırması için README ve birkaç not yeterli olabilir. Bir dönem projesi, ekip çalışması ve YZ destekli geliştirme içeriyorsa daha düzenli bir bağlam haritasına ihtiyaç duyabilir.

Örnek bir şablon Tablo 6.4 içinde yer alır.

Tablo 6.4: Proje dokümantasyon dosyalarının amacı ve gerekliliği

Dosya	Amacı	Ne zaman gerekir?
`README.md`	Projeye giriş, kurulum ve çalıştırma (insan için)	Neredeyse her projede

Dosya	Amacı	Ne zaman gerekir?
`AGENTS.md`	YZ ajanı için yönlendirici: sınır, kural, test komutu ve okuma sırası	YZ ajanı projeye dokunuyorsa
`PRD.md`	Problem, kullanıcı, kapsam ve başarı ölçütü	Ürün davranışı belirsizse
`BACKLOG.md`	İş parçaları ve öncelik	Ekip işi veya sprint varsa
`.env.example`	Gerekli ayarların örnek biçimi	Ortam değişkeni kullanılıyorsa
`DOMAIN.md`	İş kavramları ve kurallar	İş alanı karmaşıksa
`ARCHITECTURE.md`	Sistem bileşenleri ve veri akışı	Birden fazla bileşen varsa
`docs/ADR/`	Önemli karar kayıtları	Kararın bedeli ve alternatifi varsa

Belirsizlik arttıkça doğru soru, ilgili dosyada görünür kılınmalıdır.

6.5.1 Asgari Proje İskeleti

Bir dönem projesine başlarken beş dosya çoğu zaman iyi bir çekirdek oluşturur. README projenin nasıl çalışacağını, PRD neyin yapılacağını, BACKLOG hangi işlerin sırada olduğunu, `.env.example` hangi ayarların beklendiğini gösterir; `AGENTS.md` ise projeye dokunan bir YZ ajanının ilk okuyacağı yönlendirici dosyadır (Tablo 6.5).

Tablo 6.5: Asgari proje iskeleti

Dosya	Fayda
`README.md`	Projeyi nasıl kurar ve çalıştırırım?
`AGENTS.md`	YZ ajanı hangi kurallarla, hangi sırayla ve hangi sınırlarla çalışacak?
`PRD.md`	Hangi problemi, kimin için çözüyoruz?
`BACKLOG.md`	Sıradaki işler ve öncelikler neler?
`.env.example`	Hangi konfigürasyon değerleri gerekiyor?

Bu beş dosyanın ilk dördü öncelikle insan okuyucuyu hedefler; bir ekip arkadaşı projeye ilk kez baktığında ihtiyaç duyduğu asgari bilgiyi taşır. `AGENTS.md` ise aynı görevi YZ ajanı için üstlenir ve bir yönlendirici (router) dosya gibi çalışır: ajan projeye girdiğinde önce bu dosyayı okur, buradaki kural ve komutları uygular, ayrıntı gerektiğinde README, PRD veya TESTING gibi dosyalara buradan yönlendirilir. Bu yüzden YZ destekli geliştirmede `AGENTS.md` isteğe bağlı bir eklenti değil, asgari iskeletin parçasıdır. README'nin insana, AGENTS.md'nin ajana hitap etmesi rastgele bir isim tercihi değildir: biri kuruluma dair bir anlatı biçiminde yazılırken, diğeri doğrudan komut ve sınır listesi olarak yazılır. Bu farkı ilerleyen bir başlıkta ele alacağız.

6.5.2 Proje hafızası

`PROJECT-MEMORY.md` veya `MEMORY.md`, projede sık unutulan ama YZ'ye de verilmesi gereken kısa bilgileri taşıyabilir. Örneğin "stok hareketleri silinmez, iptal hareketiyle düzeltilir" gibi bir not, iş kuralı açısından çok değerlidir. Bu dosya paylaşılabilir proje hafızasıdır; fakat kişisel veri veya gizli anahtar içermez. İçeriği kısa, güncel ve karar odaklı tutulmalıdır.

6.5.3 Domain ve mimari

`DOMAIN.md` iş kavramlarını, `ARCHITECTURE.md` sistemin ana parçalarını, `ADR/` ise önemli kararları tutar. Örneğin `DOMAIN.md` içinde "kritik stok seviyesi, ürünün yeniden sipariş edilmesi gereken eşik değerdir" cümlesi bulunabilir. Bu bilgi hem geliştiriciye hem YZ'ye alanın dilini öğretir. Mimari ve domain bilgisi paylaşılabilir olmalıdır; ancak kurum içi özel şema, gizli adres veya gerçek müşteri verisi gerektiğinde soyutlanmalıdır.

6.5.4 Uygulama ayrıntısı

Uygulama ayrıntısı dosyaları, YZ'ye ve ekibe kodun nasıl davranması gerektiğini anlatır. `API.md` hangi uç noktanın ne döndürdüğünü, `DATA-MODEL.md` alanların anlamını, `TESTING.md` ise hangi testlerin nasıl çalıştırılacağını gösterir.

```
$ npm test
[OK] unit tests
[OK] api tests
```

Bu çıktı yalnızca testlerin geçtiğini söylemez; aynı zamanda `TESTING.md` dosyasındaki beklentinin doğrulanabilir olduğunu gösterir. Eğer test komutu değiştiyse, doküman da değişmelidir.

6.5.5 Çalışma kuralları

YZ ajanlarıyla, yani dosyaları kendi başına okuyup değişiklik önerebilen araçlarla çalışırken README'nin yetmediği bir nokta vardır. README bir insanın gözünü ekrana dikip birkaç dakika ayırarak okuyacağı bir anlatıdır. Bir YZ ajanı ise oturuma her başladığında projeyi baştan "tanımaya" çalışır ve bunu kısa, doğrudan ve makine tarafından kolayca ayrıştırılabilecek bir dosyadan yapmak ister. `AGENTS.md`, farklı YZ araçları arasında yaygınlaşan ve tek bir şirkete bağlı olmayan böyle bir kural dosyasıdır: çoğu araçta oturumun ilk okuduğu kapıdır ve bir yönlendirici (router) gibi çalışır; YZ ajanı buradan başlar ve gerekiyorsa diğer Markdown dosyalarına buradan yönlendirilir. `SECURITY.md` gizli bilgi ve güvenlik beklentilerini, `TECH-DEBT.md` ise bilinen teknik borçları taşır. Bu üç dosya paylaşılabilir kural üretir; gizli değerleri içermez.

README ile AGENTS.md'nin ayrı dosyalar olması gelişigüzel bir tercih değildir. README bir insanın merakını karşılamak için yazılır: bu proje neden var, nasıl kurulur, ne işe yarar. AGENTS.md ise bir ajanın davranışını sınırlamak için yazılır: hangi dizine dokunabilir, hangi komutu çalıştırmalı, hangi kurala uymalı. İkisini tek dosyada birleştirmek, insana yönelik anlatıyı ajana yönelik kısa komutlarla karıştırır;

sonuçta ne insan hızlıca bağlam kurabilir ne de ajan kuralları güvenilir biçimde ayrıştırabilir.

`AGENTS.md` şişirilmez. İçine sınır, dokunulmayacak dizin, test komutu ve PR kuralı yeter. Uzun mimari anlatı `ARCHITECTURE.md` veya `DOMAIN.md` içinde kalır. Küçük bir örnek:

```
# AGENTS.md
Önce README.md, PRD.md ve docs/API.md oku.
Gerçek .env ve anahtar yazma.
Yalnızca src/ ve tests/ değiştir.
İş bitince npm test çalıştır.
PR açıklamasında test sonucunu belirt.
```

Böyle bir dosya, "stok sistemimizi biliyorsun" demekten daha güvenilirdir. YZ ajanına önce bunu, sonra ilgili belgeyi işaret etmek, sohbet geçmişine bel bağlamamayı kolaylaştırır. Bu yönlendirme, ajanla insanın aynı gerçek kaynaklara (README, PRD, API) farklı sırayla ve farklı derinlikte ulaşmasını sağlar: insan README'den başlar, ajan AGENTS.md'den başlar; ikisi de sonunda aynı repoyu okur.

Her projede her dosya gerekmez. Tek kişilik küçük bir ödev için README ve kısa backlog yeterli olabilir; YZ ajanı kullanılmıyorsa AGENTS.md da gerekmez. Ekip büyüdükçe, YZ ajanları daha fazla kullanıldıkça ve iş alanı karmaşıklaştıkça hem dokümantasyon hem de çalışma kuralları dosyaları büyür.

6.6 PRD Yazımı

Bir ekipte proje hakkında konuşulduğunda herkes farklı bir şey hayal edebilir; biri basit bir giriş-çıkış ekranı düşünürken diğeri raporlama ve uyarı üreten kapsamlı bir sistem bekleyebilir. Bu farklı beklentiler yazılı hale gelmezse, YZ'ye verilen görev de aynı belirsizliği taşır. Ürün gereksinim dokümanı (Product Requirements Document, PRD), projenin ne yapacağını ve neden yapacağını açıklayarak bu belirsizliği azaltır. YZ'ye "stok takip sistemi yap" demek ile "küçük işletme çalışanlarının ürün giriş-çıkışlarını izleyebileceği, kritik stok uyarısı üretecek bir sistem tasarla" demek arasında büyük fark vardır. PRD bu farkı görünür kılar.

Küçük bir PRD şu başlıklarla başlayabilir:

1. Problem tanımı
2. Hedef kullanıcılar
3. Kullanıcı ihtiyaçları
4. Kapsam içi özellikler
5. Kapsam dışı özellikler
6. Başarı kriterleri
7. Varsayımlar ve riskler

Bu başlıklar yalnızca belge düzeni değildir. YZ'ye verilen görevin sınırını çizer, ekip içinde yanlış beklentiyi azaltır ve akademik dürüstlük açısından okuyucunun problemi gerçekten anladığını gösterir.

Problem tanımı

Problem tanımı, projenin hangi rahatsızlığı veya ihtiyacı ele aldığını söyler. "Stok uygulaması yapılacak" bir problem tanımı değildir; "küçük işletmeler ürün giriş-çıkışlarını farklı Excel dosyalarında tuttuğu için güncel stok bilgisini ekipçe göremiyor" daha iyi bir tanımdır. Bu cümle, YZ'nin de ekibin de çözmesi gereken asıl durumu anlamasını sağlar.

Hedef kullanıcılar

Hedef kullanıcılar, sistemin kim için tasarlandığını açıklar. Depo görevlisi, işletme sahibi ve satış temsilcisi aynı ekrana aynı ihtiyaçla bakmaz. Depo görevlisi hızlı giriş yapmak isterken, işletme sahibi özet rapor görmek isteyebilir. Bu ayrım yazılmadığında YZ genel ekranlar üretir; yazıldığında ise rol ve ihtiyaç ilişkisini daha doğru kurabilir.

Kullanıcı ihtiyaçları

Kullanıcı ihtiyaçları, özellik listesinden daha temeldir. "Excel'e aktar" bir özellik olabilir; ihtiyaç ise "işletme sahibi aylık stok durumunu muhasebe görüşmesine götürmek istiyor" cümlesinde görünür. Proje ekibi bu ihtiyacı yazmadığında, YZ doğru görünen ama iş değeri zayıf özellikler önerebilir. Bu yüzden ihtiyaçları açık yazmak, hem kapsamı hem de önceliği daha sağlıklı belirler.

Kapsam içi ve kapsam dışı özellikler

Kapsam içi özellikler yapılacak işleri, kapsam dışı özellikler ise bilinçli olarak yapılmayacak işleri belirtir. Örneğin "ürün ekleme, stok girişi ve aylık rapor kapsam içindedir; barkod entegrasyonu bu sürümde kapsam dışıdır" demek ekibi korur. YZ'ye de sınır verir. Aksi halde araç, istenmeyen entegrasyonları veya gereksiz ekranları önererek kapsamı büyütebilir.

6.7 Alan Odaklı Düşünmeye Giriş

Bir bilgi sistemi çoğu zaman bir iş alanının dilini taşır. Stok projesinde ürün, depo, hareket, kritik seviye; randevu sisteminde hasta, doktor, slot, iptal; öğrenci bilgi sisteminde ders, kayıt, not, dönem gibi kavramlar vardır. Kod bu kavramları doğru anlamazsa ekran çalışsa bile sistem yanlış davranır.

Alan odaklı düşünme (domain-driven thinking), yazılımı yalnızca teknik nesneler üzerinden değil, iş alanının kavramları ve kuralları üzerinden düşünme yaklaşımıdır. Burada derin bir yazılım mimarisi teorisine girmeyeceğiz. İhtiyacımız olan şey; YZ'ye ve ekibe, projenin alan dilini doğru öğretmektir.

Peki bu neden önemlidir? Çünkü YZ genel bir "stok" kavramı üretebilir; fakat sizin projenizde "rezerve stok", "satılabilir stok" ve "fiziksel stok" farklı anlamlara geliyor olabilir. Bu fark yazılmazsa üretilen kod iş kuralını bozabilir.

Bu yüzden domain bilgisi, YZ ile kontrollü geliştirmenin temel bağlamlarından biridir.

Domain nedir?

Bir restoran sipariş sistemi düşünelim. Masalar, siparişler, ürünler, mutfak durumu ve ödeme adımları bu sistemin dünyasını oluşturur. İşte bu dünyaya alan veya domain diyebiliriz. Domain, yazılımın hizmet ettiği iş bağlamıdır. Pratik sınır şudur: domain öğrenmek, her sektörü uzman seviyesinde bilmek değildir; projeyi doğru modelleyecek kadar kavram ve kuralı anlamaktır.

Aktör, kullanım senaryoları ve iş kuralı

Aktör, sistemle etkileşime giren kişi veya rolü ifade eder; örneğin depo görevlisi. Kullanım senaryosu (use case), aktörün sistemle ne yapmak istediğini anlatır; örneğin stok girişi yapmak. İş kuralı ise bu işlemin hangi sınırlarla yapılacağını belirler; örneğin "stok miktarı onaylanmadan rapora yansımaz." Bu üçlü, YZ'ye yalnızca ekran değil, davranış üretmesi için bağlam verir.

6.7.1 Varlık ve değer nesnesi fikrine kısa bakış

Derin teknik ayrıntıya girmeden şu ayrımı bilmek faydalıdır. Varlık (entity), kimliği olan şeydir; ürün, kullanıcı veya sipariş gibi. Değer nesnesi (value object), kimliğinden çok değeriyle anlamlıdır; para tutarı, tarih aralığı veya adres parçası gibi. Küçük bir akışla düşünelim:

1. Önce iş alanındaki kavramları listeleyin.
2. Hangilerinin kendi kimliği olduğunu ayırın.
3. Hangilerinin yalnızca değer taşıdığını belirleyin.
4. Bu ayrımı `DOMAIN.md` içinde kısa açıklamayla yazın.

Bu ayrım kod düzeni kadar ekip dilini de netleştirir.

Ortak dil: ubiquitous language

Bir ekipte biri "stok düşme", diğeri "stok hareketi", üçüncüsü "ürün satıldı" diyorsa ve hepsi farklı şeyi kastediyorsa karışıklık kaçınılmazdır. Ortak dil (ubiquitous language), ekip, kullanıcı ve kod arasında aynı kavramların aynı anlamla kullanılmasıdır. Bu dili `DOMAIN.md` içinde küçük bir sözlük olarak tutmak, hem geliştiricilerin hem de YZ'nin yanlış terim üretmesini azaltır.

Alan bilgisi, teknik sistem ile iş ihtiyacı arasındaki köprüdür. Bir muhasebe raporundaki "tahsil edildi" durumu ile "fatura kesildi" durumu aynı şey değildir. Bu ayrımı bilmeyen sistem, doğru çalışan kodla yanlış yönetim bilgisi üretebilir. Bu yüzden domain bilgisi yalnızca analiz aşamasında değil, YZ'ye görev verirken de önemlidir.

YZ'ye domain öğretmek, uzun bir sektör ansiklopedisi yazmak değildir. Projedeki temel kavramları, rolleri ve iş kurallarını kısa ve açık biçimde vermektir. Örneğin "kritik stok, `stockQuantity < reorderLevel` olduğunda oluşur; fakat rezerve ürünler satılabilir stoktan düşülür" cümlesi YZ için çok güçlü bağlamdır. Bunun bedeli süreklidir: domain bilgisi tek seferlik yazılıp bırakılırsa, iş kuralı değiştikçe YZ hâlâ eski kurala göre çözüm üretmeye devam eder; bu yüzden `DOMAIN.md` dosyasını güncel tutmak, kural değiştiği her seferde tekrar eden bir bakım görevidir.

6.8 Mimari Dosyası Yazımı

Bir proje birkaç haftadır sürüyorsa ekip üyeleri sistemin hangi parçalardan oluştuğunu zamanla farklı hatırlamaya başlayabilir; biri veritabanının arayüze doğrudan bağlı olduğunu sanır, biri API katmanının varlığını unutur. Bu karışıklığı önleyen belge mimari dosyasıdır. Mimari dosyası, projenin teknik haritasını kısa ve anlaşılır biçimde gösterir. Büyük bir kurumsal rapora dönüşmeden "bu sistem hangi parçalardan oluşuyor, veri nereden nereye gidiyor, hangi dış servislere bağlıyız, hangi varsayımlarla çalışıyoruz?" sorularına cevap verir.

YZ ile çalışırken `ARCHITECTURE.md` özellikle değerlidir. Çünkü YZ'ye yalnızca tek dosya gösterdiğinizde, sistemin geri kalanını tahmin edebilir. Mimari dosyası bu tahmini azaltır. Örneğin "frontend React, backend Express, veritabanı PostgreSQL, kimlik doğrulama Supabase üzerinden" gibi bir açıklama, önerilerin proje gerçeklerine yaklaşmasını sağlar.

Kısa bir mimari dosyası şu akışla yazılabilir:

1. Sistem amacı ve sınırı
2. Ana bileşenler
3. Veri akışı
4. Dış bağımlılıklar
5. Konfigürasyon varsayımları
6. Bilinen riskler

Bu başlıklar yalnızca teknik düzen değil, denetlenebilir bağlam üretir.

Sistem amacı, uygulamanın neyi çözdüğünü; sistem sınırı ise neyi çözmediğini açıklar. Örneğin "Bu sistem stok hareketlerini izler; muhasebe fişi üretmez" cümlesi çok değerlidir. YZ'ye bu sınır verilmezse, araç projeye gereksiz muhasebe ekranları eklemeyi önerebilir. Sınır yazmak, kapsam yönetiminin teknik karşılığıdır.

6.8.1 Ana bileşenler ve veri akışı

Ana bileşenler ve veri akışı, sistemin nasıl nefes aldığını gösterir. Kısa bir akış yeterli olabilir:

1. Kullanıcı arayüzde stok hareketi formunu doldurur.
2. Frontend isteği API'ye gönderir.
3. API iş kuralını kontrol eder.
4. Veritabanına hareket kaydı yazılır.
5. Rapor ekranı güncel veriyi okur.

Bu akış ekip düzeni için önemlidir; çünkü hata çıktığında hangi katmana bakılacağı daha hızlı anlaşılır.

Bir proje yalnızca kendi kodundan oluşmayabilir. E-posta servisi, ödeme sağlayıcısı, kimlik doğrulama sistemi, harita API'si veya bulut veritabanı dış bağımlılık olabilir. Bu bağımlılıklar yazılmadığında YZ yanlış paket önerebilir veya ekip hangi servisin kritik olduğunu gözden kaçırabilir. Mimari dosyasında dış bağımlılığın adı, amacı ve hata durumundaki etkisi kısa yazılmalıdır.

6.8.2 Yapılandırma ve dağıtım varsayımları

Yapılandırma (configuration) ve dağıtım (deployment) varsayımları, sistemin hangi ortamda nasıl çalışacağını gösterir. Örneğin:

1. Geliştirme ortamı `.env` dosyasından ayar okur.
2. Üretim ortamında gizli değerler panel üzerinden verilir.
3. Veritabanı bağlantısı ortam değişkeniyle belirlenir.
4. Uygulama varsayılan olarak `3000` portunda çalışır.

Bu bilgi yazıldığında YZ ve ekip, ayarları kod içerisine doğrudan yazmak yerine doğru yerde tutmayı öğrenir.

6.9 Backlog ve İş Parçalama

Bir ekip aynı toplantıda ürün ekleme ekranını, rapor sayfasını ve kritik stok uyarısını birden konuşmaya başladığında, hangi işin bu hafta bitmesi gerektiği çoğu zaman netleşmez; herkes kendi önceliğini konuşmuş olur. Backlog, yapılacak işlerin gelişigüzel bir liste olmaktan çıkıp öncelik ve kapsam kazandığı yerdir. YZ'ye büyük ve belirsiz görev vermek çoğu zaman dağınık sonuç üretir. "Bütün stok sistemini yap" yerine "ürün listeleme ekranında arama filtresi ekle" demek daha güvenlidir. Çünkü küçük görev daha kolay anlaşılır, test edilir ve incelenir.

Backlog, bir yönetim aracı taklidinden çok işi parçalara ayırarak hem ekibi hem YZ'yi daha denetlenebilir hale getirmelidir. Küçük bir `BACKLOG.md` şu yapıyı taşıyabilir:

```
## Sprint 1
- [ ] Ürün listeleme ekranı
- [ ] Ürün ekleme formu
- [ ] Kritik stok filtresi

## Teknik borç
- [ ] Form doğrulama mesajlarını sadeleştir
```

Bu liste, yalnızca yapılacak işleri değil, işlerin hangi seviyede ele alınacağını da gösterir.

6.9.1 Epic, feature, user story ve task farkı

Backlog seviyelerini ayırmak, YZ'ye görev verirken de faydalıdır. Epic büyük hedefi, feature kullanılabilir özelliği, user story kullanıcı ihtiyacını, task ise yapılabilir küçük işi ifade eder (Tablo 6.6).

Tablo 6.6: Epic, feature, user story ve task farkı

Seviye	Örnek
Epic	Stok yönetimi
Feature	Kritik stok uyarıları
User story	Depo görevlisi kritik ürünleri görmek ister
Task	`LowStockTable` bileşenine filtre ekle

YZ için en güvenli seviye çoğu zaman task seviyesidir.

Bug ve technical debt

Hata (bug), beklenen davranışın bozulmasıdır. Teknik borç (technical debt) ise bugün bilinçli veya bilinçsiz bırakılan ve ileride bakım maliyeti çıkarabilecek eksiktir. Örneğin "stok filtresi yanlış sonuç gösteriyor" bug olabilir; "form doğrulama kuralları üç farklı dosyada tekrar ediyor" teknik borç olabilir. YZ'ye görev verirken bu ayrımı yazmak, çözümün doğru yöne gitmesini sağlar.

YZ'ye küçük görev vermek neden daha güvenlidir?

Küçük görev, hatayı küçük tutar. Bir geliştirici YZ'ye "tüm backend'i düzenle" derse, çıkan değişikliği anlamak ve test etmek zorlaşır. "Sadece `products.service.ts` dosyasında kritik stok hesaplamasını düzelt" dediğinde ise değişikliğin sınırı bellidir. Teslim haftasında küçük görevler, hem geri alma hem de inceleme açısından daha güvenlidir.

6.9.2 Kabul kriteri yazmak

Kabul kriteri, bir işin ne zaman tamam sayılacağını belirler. YZ'ye görev verirken kabul kriteri yazmak, çıktıyı değerlendirmeyi kolaylaştırır. Örneğin:

```
## Kabul kriterleri
- Stok miktarı kritik seviyenin altındaysa ürün "Kritik" etiketiyle görünür.
- Stok miktarı eşitse "Kritik" sayılmaz.
- Liste ürün adına göre filtrelenebilir.
```

Bu örnek bize beklenen davranışı netleştirir. Böylece YZ'nin ürettiği kod yalnızca "güzel görünüyor" diye değil, ölçülebilir kriterlere göre değerlendirilir.

6.10 Test Bilgisini Markdown ile Tarif Etmek

YZ'ye test yazdırmak mümkündür; fakat test beklentisi verilmezse araç çoğu zaman yüzeysel testler üretir. Bu yüzden test bilgisini Markdown ile tarif etmek faydalıdır. `TESTING.md` dosyası hangi test türlerinin kullanıldığını, hangi komutların çalıştırılacağını ve kritik senaryoların neler olduğunu anlatabilir.

Küçük bir test dokümanı şu sorulara cevap vermelidir:

1. Hangi test komutu çalıştırılır?
2. Hangi iş kuralları mutlaka test edilir?
3. Hangi senaryolar duman testiyle kontrol edilir?
4. YZ'den test isterken hangi dosyalar bağlam verilir?

Bu bilgiler yazıldığında YZ'nin test üretimi de daha anlamlı hale gelir. Çünkü araç yalnızca kodu değil, beklenen kalite ölçüsünü de görür.

Test stratejisi, her şeyi test etmeye çalışmak değil, hangi riskin hangi testle azaltılacağını belirlemektir. Küçük bir projede birim testleri kritik hesaplamaları, entegrasyon testleri

API-veritabanı ilişkisini, uçtan uca testler ise temel kullanıcı akışını kontrol edebilir. Test sayısından çok önemli hataların erken yakalanması değer taşır.

6.10.1 Birim, entegrasyon ve uçtan uca test beklentileri

Birim testi (unit test), küçük bir fonksiyonun davranışını sınar. Entegrasyon testi, parçaların birlikte çalışıp çalışmadığına bakar. Uçtan uca test (E2E), kullanıcı akışını baştan sona dener. Bu ayrım YZ'ye de yazılmalıdır; çünkü "test yaz" demek belirsizdir.

```
$ npm test
[OK] stock calculation unit tests
[OK] products api integration tests
[X] login e2e test
```

Bu çıktı bize testlerin yalnızca sayısını değil, hangi katmanda sorun çıktığını da gösterir. Login akışı bozuksa sorun birim hesaplama testlerinde değil, kullanıcı akışında aranmalıdır.

6.10.2 Duman Testi

Duman testi (smoke test), sistemin temel olarak çalışıp çalışmadığını kontrol eden hızlı testtir. Detaylı doğrulama yapmaz; "en kritik yerler tamamen kırılmış mı?" sorusuna bakar. Örneğin:

```
$ npm run smoke
[OK] app starts
[OK] login page opens
[OK] products page loads
```

Buradaki üç satır, projenin ana kapılarının açık olduğunu doğrular. Duman testi geçiyor diye sistem bütünüyle doğru sayılmaz; fakat duman testi geçmiyorsa daha derin testlere geçmeden önce temel sorun çözülmelidir.

Kabul kriterleri

Kabul kriterinin ne işe yaradığını 6.9.2 numaralı alt başlıkta gördük; test yazarken bu kriter, iş kuralını sınayan somut bir senaryoya dönüşür. "Ürün ekleme çalışıyor" yerine "zorunlu alanlar boşsa kayıt yapılmaz, başarılı kayıtta ürün listede görünür, hata mesajı kullanıcıya açık gösterilir" demek, YZ'den yalnızca teknik değil davranış odaklı test istemeyi sağlar. YZ ile test başlatırken yalnızca test dosyasını değil, ilgili iş kuralını ve kabul kriterini de vermek gerekir. Örneğin:

```
DOMAIN.md içindeki kritik stok kuralına ve BACKLOG.md içindeki kabul kriterlerine göre
`stock.service.ts` için birim testleri öner.
```

Bu istek, YZ'nin neyi doğrulaması gerektiğini açıklar. Böylece testler yalnızca fonksiyon çağıran boş kabuklar olmaktan çıkar, iş kuralını denetleyen örneklere dönüşür.

6.11 YZ İçin İyi İstem Yazımı

YZ'den alınan çıktının kalitesi çoğu zaman verilen istemin kalitesiyle ilişkilidir.[3] Ancak iyi istem yazımı sihirli kelime bulmak değildir. İyi istem, bağlamı, görevi, sınırı, kabul kriterini, test beklentisini ve çıktı formatını açık yazar. Böylece YZ'nin tahmin alanı daralır.

Zayıf ve daha iyi istem arasındaki fark Tablo 6.7 ile somutlaşır.

Tablo 6.7: YZ için iyi istem yazımı

Zayıf istem	Daha iyi istem
"Stok ekranı yap"	"`src/pages/Products.tsx` içinde ürün listeleme ekranına isim filtresi ekle. API sözleşmesi `docs/API.md` içinde. Kabul kriterleri: boş filtre tüm ürünleri gösterir, filtre büyük/küçük harfe duyarsızdır. Önce plan yaz, sonra patch öner."
"Test yaz"	"`stock.service.ts` için `DOMAIN.md` içindeki kritik stok kuralını kapsayan birim testleri yaz. Eşitlik durumunun kritik sayılmadığını test et."
"README düzelt"	"README'nin kurulum bölümünü `.env.example` ve mevcut `package.json` komutlarına göre güncelle. Gizli bilgi ekleme."
"Bu hatayı düzelt"	"Hata çıktısını, ilgili dosyayı ve beklenen davranışı vererek düzeltme öner. Gizli değer yazdırma."
"Dokümantasyon üret"	"README ve `DOMAIN.md` hangi kararı cevaplıyor, onu yaz; uydurma komut ekleme."

Gördüğünüz gibi iyi istem daha uzun olmak zorunda değildir; fakat daha sorumludur. YZ'ye ne yapacağını söylediği kadar, neye göre doğru sayılacağını da söyler.

Bağlam ver

Bağlam vermek, YZ'ye projenin gerçeklerini göstermektir. "Bu bir stok uygulaması" demek başlangıçtır; fakat "kritik stok, mevcut miktar yeniden sipariş seviyesinin altına düştüğünde oluşur" demek daha değerlidir. Bağlam dosyalarla da verilebilir: AGENTS, DOMAIN, API, TESTING. Bağlam verilmezse YZ genel bilgilerle tahmin yürütür.

Görevi küçült

Görevi küçültmek, YZ çıktısını denetlenebilir hale getirir. "Tüm projeyi düzenle" yerine

[3] Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., & Neubig, G. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, *55*(9). https://doi.org/10.1145/3560815

"ürün listeleme bileşenindeki filtre davranışını düzelt" demek daha güvenlidir. Küçük görev hem okuyucunun öğrenmesini kolaylaştırır hem de inceleme sürecini sadeleştirir. Büyük görev gerekiyorsa önce plan istenmeli, sonra küçük adımlarla uygulanmalıdır.

Dosya sınırı belirt

Dosya sınırı belirtmek, YZ'nin gereksiz yerlere dokunmasını engeller. Örneğin "`src/services/stock.service.ts` ve `tests/stock.service.test.ts` dışında dosya değiştirme" demek kapsamı netleştirir. Bu sınır paylaşılabilir proje davranışıdır; PR'da da kontrol edilebilir. Ancak sınır yanlışsa YZ çözümü yapay biçimde zorlayabilir. Bu yüzden dosya sınırı, projenin gerçek yapısına göre verilmelidir.

6.11.1 Kabul kriteri ve test beklentisi yaz

6.9.2 numaralı alt başlıkta ele alınan kabul kriteri mantığı istem yazarken de aynı şekilde işler: YZ çıktısının hangi ölçüte göre değerlendirileceği önceden yazılır, test beklentisi de bu ölçütün üzerine eklenir. Örneğin:

```
Kabul kriteri:
- Stok miktarı 5, kritik seviye 10 ise "Kritik" döner.
- Stok miktarı 10, kritik seviye 10 ise "Normal" döner.

Test:
- Bu iki durum için birim test ekle.
```

Bu örnek, YZ'nin ürettiği kodun yalnızca biçimsel olarak değil, iş kuralı açısından da kontrol edilmesini sağlar.

6.11.2 Çıktı formatı belirt

Çıktı formatı belirtmek, YZ'den ne tür cevap beklediğinizi netleştirir. Bazen plan, bazen patch, bazen tablo, bazen test listesi istersiniz. Örneğin:

```
Önce 5 maddelik plan yaz. Ardından yalnızca değiştirilmesi gereken dosyaları ve
↪ gerekçesini listele.
Henüz kod yazma.
```

Bu istek, YZ'nin doğrudan geniş değişiklik yapmasını engeller. Özellikle öğrenme aşamasında önce plan görmek, okuyucunun kontrolünü artırır.

Önce plan, sonra uygulama yaklaşımı

Önce plan, sonra uygulama yaklaşımı YZ ile çalışırken aceleyi azaltır. Plan aşamasında yanlış varsayım, fazla kapsam veya riskli dosya değişikliği görülebilir. Uygulama aşamasında ise daha dar ve denetlenebilir adım atılır. Bu yöntem özellikle ekip projelerinde faydalıdır; çünkü herkes önce niyet edilen değişikliği görür, sonra kodu inceler.

6.12 YZ ile Kod Üretme ve Değiştirme

YZ ile kod üretme ve değiştirme, çoğu okuyucunun ilk denediği kullanım biçimidir. Bir hata mesajını yapıştırmak, bir bileşen istemek veya var olan fonksiyonu sadeleştirmesini rica etmek kolaydır. Bu kolaylık, öğrenme ve üretim açısından gerçekten faydalı olabilir.

Ancak kod değişikliği, projenin en riskli alanlarından biridir. Çünkü yanlış değişiklik yalnızca metin hatası üretmez; veri kaybına, güvenlik açığına veya yanlış iş sonucuna yol açabilir. Bu yüzden YZ ile kod üretirken şu üç soru her zaman sorulmalıdır: Hangi bağlama göre üretti? Hangi kabul kriterini karşılıyor? Hangi testle doğrulandı?

Bu yaklaşım YZ kullanımını yavaşlatmak için değil, üretilen hızın projeyi kırmaması için gereklidir.

Yeni özellik geliştirme

Yeni özellik geliştirirken YZ'ye doğrudan "özelliği yap" demek yerine, ilgili PRD, API ve kabul kriterleri verilmelidir. Örneğin kritik stok filtresi eklenecekse, önce bu davranışın iş kuralı yazılmalı, sonra hangi dosyada değişiklik beklendiği belirtilmelidir. YZ hız kazandırabilir; fakat özelliğin gerçekten kullanıcı ihtiyacını karşılayıp karşılamadığını ekip değerlendirmelidir.

Var olan kodu açıklatma

Var olan kodu açıklatmak, YZ'nin en yararlı öğrenme desteklerinden biridir. Özellikle miras alınan bir ders projesinde "bu servis hangi veriyi nereden alıyor?" sorusunu YZ'ye sorabilirsiniz. Ancak açıklamayı doğru kabul etmeden önce kodla karşılaştırmak gerekir. YZ bazen niyeti tahmin eder; oysa kod gerçekten başka bir şey yapıyor olabilir.

Yeniden düzenleme yaptırma

Yeniden düzenleme (refactoring), kodun davranışını değiştirmeden yapısını iyileştirmektir. YZ tekrar eden kodu sadeleştirmede veya isimleri daha anlaşılır hale getirmede yardımcı olabilir. Ancak refactoring adı altında davranış değişmemelidir. Bu yüzden "mevcut testler geçmeli, dış API değişmemeli, yalnızca okunabilirlik iyileştirilmeli" gibi sınırlar isteme yazılmalıdır.

6.12.1 Hata ayıklama

Hata ayıklamada YZ, hata mesajını yorumlayarak muhtemel sebepleri sıralayabilir. Fakat yanlış teşhis de koyabilir. Bir ekip "veritabanı bağlantısı yok" hatasını YZ'ye sorar; YZ paket yeniden kurmayı önerir. Oysa sorun `.env` dosyasındaki yanlış bağlantı adresidir. Yanlış öneri zaman kaybettirir. Hata ayıklamada amaç hatayı susturmak değil, nedenini anlamak ve kalıcı bir iyileştirme yapmaktır.

6.12.2 Test ve dokümantasyon güncelletme

YZ'den kod değişikliğine eşlik eden test ve dokümantasyon güncellemesi istemek iyi bir alışkanlıktır. Örneğin bir API alanı değiştiyse test de API dokümanı da güncellenmelidir.

```
$ git diff --name-only
src/services/products.service.ts
tests/products.service.test.ts
docs/API.md
```

Bu çıktı bize değişikliğin yalnızca üretim kodunda kalmadığını gösterir. Test ve doküman aynı PR içinde yer aldığında inceleme daha güvenilir hale gelir.

6.12.3 Commit mesajı ve PR açıklaması hazırlatma

YZ, commit mesajı ve PR açıklaması taslağı hazırlamada faydalıdır; çünkü değişiklikleri toparlayabilir. Ancak mesajın doğru olması için diff'i ve niyeti bilmesi gerekir.

```
feat: add low stock filter

- add product name and low stock filters
- update API documentation
- add unit tests for critical stock rule
```

Bu örnek, değişikliğin ne yaptığını kısa ve izlenebilir biçimde anlatır. Yine de son kontrol geliştiricide olmalıdır; YZ bazen değişmeyen bir dosyayı değişmiş gibi yazabilir.

6.13 YZ Çıktısını Denetleme

YZ çıktısını denetlemek, YZ'ye güvenmemek anlamına gelmez. Daha doğru ifade şudur: YZ çıktısını proje standardına göre doğrulamak gerekir. Nasıl ki bir ekip arkadaşının PR'ı inceleniyorsa, YZ yardımıyla üretilen değişiklik de incelenmelidir.

Kullanılabilir bir denetim listesi şöyle olabilir:

```
## YZ çıktı denetimi
- [ ] Değişiklik istenen dosya sınırları içinde mi?
- [ ] Kod çalışıyor mu?
- [ ] İlgili testler geçiyor mu?
- [ ] Kabul kriterleri karşılandı mı?
- [ ] Mimari ve domain notlarına uyuyor mu?
- [ ] Gereksiz soyutlama veya fazla kapsam var mı?
- [ ] Gizli bilgi, token veya özel veri eklenmiş mi?
- [ ] Yeni bağımlılık gerekiyorsa gerekçesi var mı?
- [ ] Dokümantasyon güncellendi mi?
- [ ] PR olarak okunabilir ve geri alınabilir mi?
```

Bu listeyi her küçük değişiklikte aynı ağırlıkla kullanmak gerekmez. Ancak özellikle YZ ile üretilen kod ana dala yaklaşırken, bu kontrol ekip ve kalite açısından koruyucu bir alışkanlık oluşturur.

Kod çalışıyor mu?

"Kod çalışıyor mu?" sorusu basit görünür ama ilk kontroldür. YZ'nin ürettiği kod söz dizimi olarak doğru olmayabilir, eksik import içerebilir veya mevcut dosya yapısıyla

uyuşmayabilir. Çalıştırmadan güvenmek, teslim günü sürprizlerine kapı açar. Yönetim Bilişim Sistemleri açısından çalışan kod, karar güvenilirliğinin teknik tabanıdır; fakat tek başına yeterli değildir.

Test geçiyor mu?

Testin geçmesi, değişikliğin beklenen davranışla çelişmediğini gösteren önemli bir sinyaldir. Örneğin:

```
$ npm test
[OK] 24 tests passed
```

Bu çıktı rahatlatıcıdır; fakat hangi testlerin olduğu da önemlidir. Eğer kritik stok kuralı için test yoksa, 24 testin geçmesi o kuralın doğru olduğunu kanıtlamaz. Bu yüzden test sonucu, test kapsamıyla birlikte okunmalıdır.

Mimariye uyuyor mu?

YZ çıktısı çalışsa bile mimariye uymayabilir. Örneğin proje API çağrılarını `services/` katmanında topluyorken YZ doğrudan bileşen içinde fetch yazmış olabilir. Bu küçük tercih büyüdükçe bakım sorununa dönüşür. Mimari uyum, kodun yalnızca bugünkü çalışmasını değil, yarınki anlaşılabilirliğini korur.

Gereksiz soyutlama veya fazla kapsam var mı?

YZ bazen küçük bir iş için gereğinden büyük yapı kurabilir. Tek filtre eklemek için genel bir filtre motoru, yeni klasör düzeni ve fazladan bağımlılık önerebilir. Bu durum ilk bakışta profesyonel görünebilir; fakat küçük bir projede bakım maliyeti yaratır. Soru şudur: Bu soyutlama bugünkü gerçek problemi sadeleştiriyor mu, yoksa yalnızca büyük görünmesini mi sağlıyor?

6.13.1 Güvenlik açığı, gizli bilgi veya gereksiz bağımlılık ekledi mi?

YZ'nin ürettiği kod bazen örnek API anahtarını gerçekmiş gibi dosyaya koyabilir veya gereksiz bir paket ekleyebilir. Bir geliştirici, deneme için aldığı servis token'ını isteme yapıştırıp sonra `.env` yerine README'ye koyarsa, gizli bilgi yayılmış olur (Tablo 6.8).

Tablo 6.8: YZ çıktısında sık görülen güvenlik durumları

Durum	Risk	Eylem
Token repoya eklendi	Yetkisiz erişim ve hesap maliyeti	Token iptal edilir, geçmiş temizleme süreci değerlendirilir
Gereksiz paket eklendi	Güvenlik ve bakım riski	Paket gerekçesi kontrol edilir
Özel veri isteme kondu	Gizlilik ihlali	Veri maskeleme yapılır

Durum	Risk	Eylem
Sahte anahtar gerçekmiş gibi yazıldı	Karışıklık, gerçek değerle karıştırılma riski	Örnek olduğu açıkça belirtilir veya `.env.example` kullanılır
Lisansı kontrol edilmeyen paket önerildi	Hukuki ve uyumluluk riski	Paketin lisansı ve bakım durumu doğrulanır

Bu riskler teknik görünür; fakat iş sürekliliği, maliyet ve güven açısından doğrudan sonuç üretir.

PR olarak incelenebilir mi?

YZ çıktısı PR olarak incelenebilir büyüklükte olmalıdır. Çok fazla dosyaya yayılan, neyi neden değiştirdiği belirsiz bir çıktı inceleme sürecini zayıflatır. İyi değişiklik, açıklaması olan, test sonucu görülen ve gerekirse geri alınabilen değişikliktir. Bu yüzden YZ ile çalışırken küçük PR'lar, ekip güvenliği açısından daha sağlıklı bir tercihtir.

6.14 YZ ve Akademik Dürüstlük

YZ araçları ders projelerinde önemli bir akademik dürüstlük sorusu doğurur. Bir katılımcı takıldığı kodu açıklatabilir, test fikri alabilir veya README taslağı hazırlatabilir. Bunlar öğrenmeyi destekleyebilir. Ancak aynı kişi projeyi anlamadan bütünüyle YZ'ye yaptırıp teslim ederse, ortaya çıkan şey artık öğrenme ürünü olmaktan uzaklaşır.

Küçük bir vaka düşünelim. Ekipten biri YZ ile büyük bir modül üretir ve repoya ekler. Kod çalışır, fakat ekipte kimse modülün nasıl çalıştığını açıklayamaz. Sunumda soru gelince belirsizlik ortaya çıkar. Buradaki sorun yalnızca kaynak göstermemek değildir; öğrenme sorumluluğunun devredilmesidir.

Bu yüzden akademik dürüstlükte temel ayrım şudur: YZ'den yardım almak başka, işi YZ'ye devretmek başkadır. Dersin öğretim elemanı farklı kurallar koyabilir; fakat hangi politika geçerli olursa olsun, teslim sahibinin yaptığı işi anlaması, açıklayabilmesi ve sorumluluğunu alması gerekir.

6.14.1 Yardım alma ve işi devretmenin farkı

Yardım almak, öğrenme sürecini destekler. İşi devretmek ise öğrenme sürecini atlatır. Bu ayrım ders politikasına göre daha sıkı veya daha esnek uygulanabilir; fakat öğrenmeden teslim etmek her durumda sorunludur.

Kaynak gösterme

Dersin kuralı YZ kullanımının belirtilmesini gerektiriyorsa, kullanılan yardım açıkça yazılmalıdır. Örneğin raporun sonunda "YZ aracı hata mesajlarının açıklanması, test senaryosu önerileri ve README taslağı için kullanılmış; nihai kod ekip tarafından incelenmiş ve düzenlenmiştir" gibi bir açıklama yapılabilir. Kaynak göstermek yalnızca biçimsel bir zorunluluk değil, çalışmanın nasıl üretildiğini dürüstçe görünür kılma davranışıdır.

Öğrenmeden teslim etme problemi

Öğrenmeden teslim edilen kod, kısa vadede not kazandırıyor gibi görünebilir; fakat bakım anında sorun çıkarır. Teslimden bir gün önce küçük bir hata çıktığında ekip kodu anlamıyorsa düzeltemez. Sunumda "bu fonksiyon neden böyle?" sorusu geldiğinde cevap veremez. Bu yüzden YZ kullanımı öğrenmeyi hızlandırmalı, öğrenmenin yerine geçmemelidir.

Kodun sorumluluğunu almak

Kodun sorumluluğunu almak, "bu kodu ben yazdım" demekten daha geniştir. Kodun ne yaptığını açıklayabilmek, hangi testle doğrulandığını gösterebilmek ve hata çıktığında müdahale edebilmek gerekir. YZ'nin katkısı olsa bile repoya giren kod ekip ürünüdür. Bu sorumluluk alınmadığında proje yönetilebilir olmaktan çıkar.

Ders içi kullanım politikası

Ders içi kullanım politikası, YZ'nin hangi sınırlarla kullanılabileceğini açıkça belirtmelidir. Örneğin "YZ açıklama ve test önerisi için kullanılabilir; nihai kod teslim sahibi tarafından anlaşılmalı ve raporda kullanım beyan edilmelidir" gibi bir kural belirsizliği azaltır. Bu politika yalnızca etik değil, süreç yönetimi aracıdır; ekipler neyin kabul edilebilir olduğunu önceden bilir.

6.15 YZ ve Güvenlik Riskleri

YZ ile çalışırken güvenlik riski çoğu zaman iyi niyetli bir hız anında ortaya çıkar. Geliştirici hata alır, servis bağlantısını çözmek için `.env` dosyasını veya gerçek API anahtarını isteme yapıştırır. YZ yardımcı olmaya çalışır; fakat özel bilgi artık gereksiz yere dışarı taşınmıştır.

Başka bir durumda YZ, sorunu çözmek için bilinmeyen bir paket önerir. Geliştirici paketi kurar, fakat paketin bakım durumu, lisansı veya güvenilirliği kontrol edilmez. Kod çalışıyor gibi görünür; fakat projeye yeni bir bağımlılık ve potansiyel risk girmiştir.

Bu yüzden YZ güvenlik açısından yalnızca kod üreten araç değil, veri paylaşımı ve karar verme sınırı olan bir ortamdır. İsteme ne koyduğunuz, repoya ne eklediğiniz ve hangi öneriyi kabul ettiğiniz güvenlik davranışıdır (Tablo 6.9).

Tablo 6.9: Güvenlik Riskleri

Risk	Olası etki	Güvenli davranış
Gizli bilgi paylaşımı	Yetkisiz erişim, maliyet	Değerleri maskele, token'ı paylaşma
Özel veri paylaşımı	Gizlilik ihlali	Örnek ve anonim veri kullan
Gereksiz paket	Güvenlik ve bakım riski	Paket kaynağını ve lisansını kontrol et
Uydurma API	Zaman kaybı, kırık kod	Resmi doküman veya mevcut kodla doğrula

Güvenlik, YZ kullanımından sonra yapılan ek kontrol değil, istem yazarken başlayan bir sorumluluktur.

Gizli bilgi, token ve key paylaşımı

Gizli bilgi, token ve key gibi değerler sisteme erişim yetkisi verir. Bir geliştirici "şu API neden çalışmıyor?" diyerek gerçek token'ı YZ aracına gönderirse, sorunu çözmeye çalışırken erişim bilgisini paylaşmış olur. Bu durumda yalnızca kod değil, hesap güvenliği de riske girer. Doğru davranış, değeri maskelemek ve yalnızca hata mesajının gerekli kısmını paylaşmaktır.

Özel veri ve müşteri verisi paylaşımı

Özel veri veya müşteri verisi, YZ istemine konulmadan önce mutlaka sorgulanmalıdır. Gerçek ad, e-posta, telefon, sipariş geçmişi veya finansal bilgi örnek veri diye paylaşılmamalıdır. Bunun yerine anonimleştirilmiş veya uydurma veri kullanılabilir. Teknik olarak küçük görünen bu davranış, gizlilik ve hukuki sorumluluk açısından büyük sonuçlar doğurabilir.

Kurumsal kod paylaşımı

Kurumsal kod paylaşımı, kurum politikası olmadan yapılmamalıdır. Bazı kurumlar belirli YZ araçlarına kod gönderilmesine izin vermez, bazıları yalnızca kurumsal hesaplar ve özel ayarlar üzerinden izin verir. Burada mesele yalnızca teknik gizlilik değildir; fikri mülkiyet, müşteri güveni ve sözleşme sorumluluğu da devrededir.

6.15.1 Lisans riski

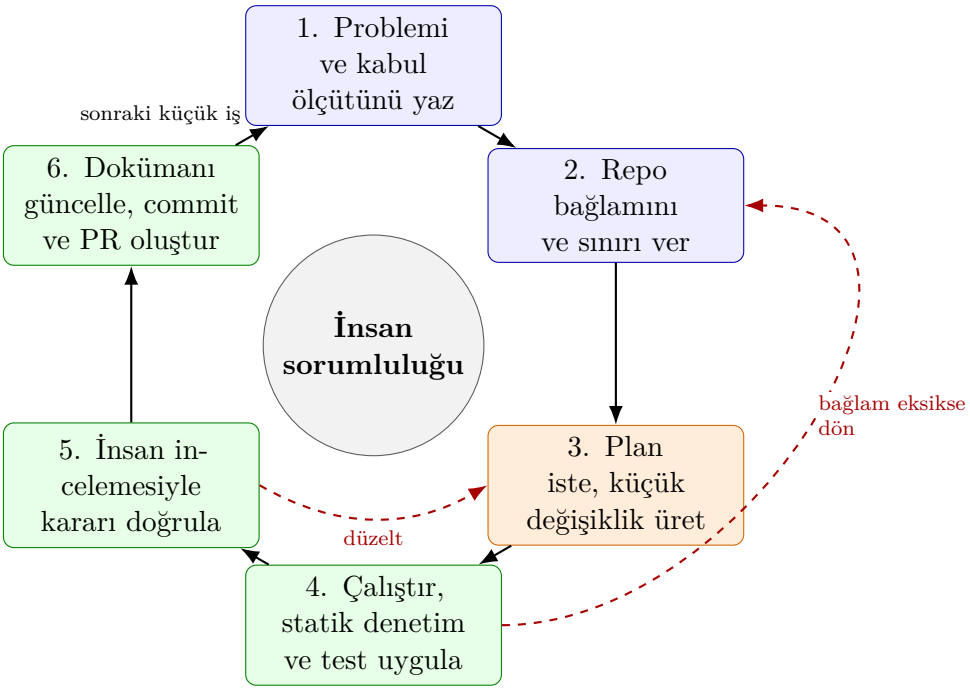
YZ bir paket veya kod parçası önerdiğinde lisans riski doğabilir. Okuyucu önerilen paketi kurar, fakat paketin lisans koşullarını veya kullanım sınırlarını kontrol etmez. Ders projesinde bu çoğu zaman fark edilmeyebilir; ancak kurumsal projelerde lisans uyumsuzluğu ciddi sonuçlar doğurabilir.

YZ bazen de var olmayan API metotları veya paketler önerebilir. Bu durum özellikle güncel kütüphanelerde ve az bilinen servislerde ortaya çıkabilir. Çözüm, öneriyi resmi dokümanla veya mevcut proje dosyalarıyla doğrulamaktır. Örneğin `package.json` içinde olmayan bir paketi kullanmasını istiyorsa, önce bu paketin gerçekten gerekli olup olmadığı sorulmalıdır. `TECH-DEBT.md` veya PR açıklamasında yeni bağımlılığın gerekçesi yazılabilir.

6.16 Önerilen Kontrollü Geliştirme Akışı

YZ ile kontrollü geliştirme, tek bir komutla her şeyi yaptırmak yerine süreci küçük ve denetlenebilir adımlara bölmektir. Bu yaklaşım hızdan vazgeçmez; hızı proje hafızası, test ve insan sorumluluğuyla birlikte kullanır.

Akışı şöyle düşünebiliriz:



Şekil 6.2: YZ ile kontrollü geliştirmede üretim, doğrulama ve geri besleme döngüsü

Şekil 6.2 için önemli olan sıralamadır. YZ, bağlamdan önce devreye girmez; testten önce son kabul yapılmaz; dokümantasyon güncellenmeden iş tamam sayılmaz. Böylece YZ çıktısı ekip çalışmasının denetlenebilir bir parçası haline gelir.

Bu akış, teknoloji kullanımını yönetim disipliniyle birleştirir. Hangi bilgi nerede tutuluyor, kim neyi onaylıyor, hangi test geçti, hangi doküman güncellendi? Bu sorular cevaplanabiliyorsa YZ kullanımı daha güvenli hale gelir.

Problemi Markdown ile yaz

İlk adım problemi Markdown ile yazmaktır. Örneğin `BACKLOG.md` içinde "Depo görevlisi kritik stok altındaki ürünleri tek ekranda görmek istiyor" cümlesi bulunabilir. Bu dosya paylaşılabilir ve versiyonlanabilir bağlam üretir. Ancak gerçek müşteri adı veya özel veri gerekiyorsa anonimleştirilmelidir. Problem yazılmadan YZ'ye görev vermek, hedefi belirsiz bırakır.

6.16.1 Domain, mimari ve backlog bilgisini oluştur

Domain, mimari ve backlog bilgisi YZ'nin tahmin alanını daraltır. Bu bilgiler olmadan araç genel bir çözüm üretir. Dosyalar hazır olduğunda basit bir kontrol yapılabilir:

```
$ ls docs
ARCHITECTURE.md  DOMAIN.md  TESTING.md  ADR
```

Bu listeleme, bağlam dosyalarının repoda bulunduğunu doğrular. Elbette dosyanın varlığı yetmez; içeriğin güncel ve kısa olması gerekir.

Küçük görev seç

Küçük görev seçmek, YZ ile güvenli çalışmanın temelidir. “Stok modülünü yap” büyük ve belirsizdir. “Kritik stok filtresini ürün listeleme ekranına ekle” daha küçük ve test edilebilirdir. Küçük görev, hatayı sınırlı tutar; inceleme sürecini kolaylaştırır; okuyucunun ne öğrendiğini görmesini sağlar.

YZ'den plan iste

YZ'den önce plan istemek, yanlış yöne giden kodu baştan durdurma imkanı verir. Plan içinde hangi dosyaların değişeceği, hangi testlerin ekleneceği ve hangi risklerin olduğu görülebilir. Eğer plan gereksiz kapsam büyütüyorsa, uygulamaya geçmeden önce daraltılır. Bu alışkanlık özellikle ekipli projelerde güven sağlar.

Kodu üret veya değiştir

Kod üretme veya değiştirme aşamasında YZ'nin önerisi uygulanabilir; fakat değişiklik küçük tutulmalıdır. Kodun mevcut stile, mimariye ve dosya sınırına uyup uymadığı kontrol edilir. Anlaşılmayan satırlar varsa YZ'den açıklama istenir, fakat açıklama kodun yerine geçmez. Geliştirici sonunda kodu kendisi okuyabilmelidir.

6.16.2 Çalıştır, test et ve gözden geçir

Çalıştırmak, test etmek ve gözden geçirmek YZ çıktısını proje gerçeğiyle karşılaştırır.

```
$ npm run lint
[OK] no lint errors
$ npm test
[OK] 31 tests passed
```

Bu çıktı olumlu bir sinyaldir; fakat son adım değildir. Kod hâlâ iş kuralına, güvenlik sınırına ve kapsam beklentisine göre okunmalıdır. Testler geçse bile yanlış şeyi test ediyor olabiliriz.

6.16.3 Commit, PR ve dokümantasyon akışını tamamla

Son adım değişikliği izlenebilir hale getirmektir. Kısa bir akış yeterlidir:

1. İlgili kod, test ve dokümantasyon dosyalarını birlikte stage edin.
2. Commit mesajında değişikliğin amacını yazın.
3. PR açıklamasında bağlamı, test sonucunu ve doküman güncellemesini belirtin.
4. İnceleme sonrasında gerekirse düzeltme yapın.
5. Birleştirme sonrası backlog veya issue durumunu güncelleyin.

Bu akış, YZ ile üretilen işi ekip çalışmasının normal kalite kapılarından geçirir. Böylece hız, denetim ve sorumluluk aynı süreçte buluşur.

6.17 Atölye

Önceki atölyede dokümantasyonu kodla aynı geçmişte tutmayı denediniz; bu atölyede aynı disiplini YZ'ye verilen görevlere taşıyoruz. YZ aracının kendisi gerekmez, çünkü buradaki alıştırma aracın ne ürettiğini değil, aracın okuyacağı zemini sizin nasıl kurduğunuzu öğretir: küçük bir bağlam iskeleti, kısa ve sınırlı bir istem dosyası, ardından denetlenebilir bir değişiklik. İstem dosyasının kendisi de `prompts/` altında repoya girer; böylece hangi görevin hangi bağlamla ve hangi sınırla istendiği commit geçmişinde kalır ve bir sonraki oturumda yeniden anlatılması gerekmez.

Görev 1: Proje hafızası yapısını kur

Komut

```
$ mkdir -p ~/gelistirme-atolyesi/bolum-6/context-app
$ cd ~/gelistirme-atolyesi/bolum-6/context-app
$ git init -b main
$ git config user.name "Atolye Kullanici"
$ git config user.email "atolye@example.com"
$ mkdir -p docs prompts src tests
$ touch README.md AGENTS.md PRD.md BACKLOG.md TESTING.md
$ touch docs/ARCHITECTURE.md .env.example
```

Beklenen çıktı

```
Initialized empty Git repository in <yol>/context-app/.git/
```

Görev 2: Kısa bağlam dosyalarını doldur

Komut

```
$ printf '%s\n' '# Context App' \
  'Kritik stok kurallarini dener.' > README.md
$ printf '%s\n' '# AGENTS.md' \
  'Once README.md ve PRD.md oku.' \
  'Gercek .env dosyasina dokunma.' \
  'Yalnizca src/ ve tests/ degistir.' \
  'Is bitince node --test calistir.' > AGENTS.md
$ printf '%s\n' '# Problem' \
  'Dusuk stok erken gorulmelidir.' > PRD.md
$ printf '%s\n' '# Backlog' \
  '- Kritik stok kuralini ekle.' > BACKLOG.md
$ printf '%s\n' '# Test' \
  'Komut: node --test --test-reporter=tap' > TESTING.md
$ printf '%s\n' '# Mimari' \
  'Tek Node.js modulu ve birim testi.' > docs/ARCHITECTURE.md
$ printf '%s\n' 'APP_ENV=development' \
  'DATABASE_URL=postgres://user:password@localhost/app' \
  > .env.example
```

Beklenen çıktı

Komutlar çıktı üretmez. Gerçek `.env` dosyası oluşturulmaz ve örnek değerler ekrana

yazdırılmaz.

Görev 3: Bağlamın kapsamını denetle

Komut

```
$ find . -type f | grep -v '/.git/' | sort
$ wc -w README.md AGENTS.md PRD.md BACKLOG.md TESTING.md docs/ARCHITECTURE.md
$ rg -n '^#|^-' README.md AGENTS.md PRD.md BACKLOG.md TESTING.md docs
```

Beklenen çıktı

```
./.env.example
./AGENTS.md
./BACKLOG.md
./PRD.md
./README.md
./TESTING.md
./docs/ARCHITECTURE.md
...
```

Görev 4: Denetlenebilir istem dosyası yaz

Komut

```
$ printf '%s\n' '# Gorev' \
  'Stok kritik mi hesaplayan bir fonksiyon ekle.' '' \
  '# Baglam' 'PRD.md ve BACKLOG.md dosyalarini kullan.' '' \
  '# Kabul' 'stock < level ise true, esit ise false.' '' \
  '# Sinir' 'Yalnizca src ve tests dizinlerini degistir.' \
  > prompts/kritik-stok.md
$ cat prompts/kritik-stok.md
```

Beklenen çıktı

```
# Gorev
Stok kritik mi hesaplayan bir fonksiyon ekle.
...
# Sinir
Yalnizca src ve tests dizinlerini degistir.
```

Görev 5: Bağlamı başlangıç commit'iyle sabitle

Komut

```
$ git add README.md AGENTS.md PRD.md BACKLOG.md TESTING.md docs prompts .env.example
$ git commit -m "Proje baglamini ekle"
$ git status --short
```

Beklenen çıktı

```
[main <kimlik>] Proje baglamini ekle
```

Commit sonrasında `git status --short` çıktı üretmez.

Görev 6: Küçük değişikliği üret ve farkı oku

Komut

```
$ printf '%s\n' \
  'exports.isCritical = (stock, level) => stock < level;' \
  > src/stock.js
$ printf '%s\n' \
  "const test = require('node:test');" \
  "const assert = require('node:assert/strict');" \
  "const { isCritical } = require('../src/stock');" \
  "test('critical stock', () => {" \
  "  assert.equal(isCritical(5, 10), true);" \
  "  assert.equal(isCritical(10, 10), false);" \
  "});" > tests/stock.test.js
$ git status --short
$ git add -N src/stock.js tests/stock.test.js
$ git diff --stat
$ git diff
```

Beklenen çıktı

```
?? src/
?? tests/
 src/stock.js        | ...
 tests/stock.test.js | ...
```

Görev 7: Proje testiyle davranışı doğrula

Komut

```
$ node --test --test-reporter=tap
```

Beklenen çıktı

```
ok 1 - critical stock
# pass 1
# fail 0
```

Süre ve ayrıntı satırları Node.js sürümüne göre değişebilir.

Görev 8: Sır, biçim ve commit kapsamını denetle

Komut

```
$ rg -n '(sk_live_|ghp_|BEGIN [A-Z ]*PRIVATE KEY)' \
  --glob '!.git/**' --glob '!.env.example' .
$ git diff --check
$ git add src/stock.js tests/stock.test.js
$ git diff --cached --stat
```

Beklenen çıktı

```
 src/stock.js        | ...
 tests/stock.test.js | ...
```

İlk iki denetim çıktı üretmemelidir. Son çıktı yalnızca istemde izin verilen `src` ve `tests` kapsamını göstermelidir.

Bölüm 7

Docker ve Konteyner Tabanlı Geliştirme Ortamları

Bir proje kendi bilgisayarımızda çalıştığında doğal olarak rahatlarız. Komutları biliyoruzdur, veritabanı kuruludur, paketler yerindedir, portlar çakışmıyordur. Ancak aynı proje ekip arkadaşımızın bilgisayarında, CI ortamında veya sunucuda çalıştırılmak istendiğinde küçük farklar büyük sorunlara dönüşebilir.

Docker ve konteyner tabanlı geliştirme ortamları bu soruya yanıt arar: Aynı proje farklı makinelerde daha tahmin edilebilir biçimde nasıl çalışır? Bu bölümde Docker'ı komut ezberi olarak değil, geliştirme ortamını tekrarlanabilir hale getiren bir düzenleme aracı olarak ele alacağız.

Burada üretim orkestrasyonu, Kubernetes derinliği veya ileri seviye imaj optimizasyonu anlatmayacağız. Öncelikli olan şudur: uygulama, veritabanı, port, volume, ortam değişkeni ve gizli bilgi gibi parçalar ekip çalışması içinde nasıl yönetilir? Docker bu soruya pratik bir zemin sağlar; fakat sınırları ve riskleriyle birlikte anlaşılmalıdır.

7.1 "Benim Bilgisayarımda Çalışıyor" Problemi

"Benim bilgisayarımda çalışıyor" cümlesi yazılım ekiplerinde çok tanıdıktır. Bu cümle çoğu zaman kötü niyetli değildir. Geliştirici gerçekten kendi makinesinde projeyi çalıştırmıştır. Hata, başka bir ortamda ortaya çıkmıştır. İlk bakışta bu durum küçük bir kurulum farkı gibi görünür.

Ancak problem bundan daha derindir. Yazılım yalnızca kod dosyalarından oluşmaz. İşletim sistemi, runtime sürümü, paketler, ortam değişkeni değerleri, veritabanı durumu, port ayarları ve servis bağlantıları da çalışma ortamının parçasıdır. Bu parçalardan biri farklı olduğunda aynı kod farklı davranabilir.

Üç kişilik bir proje ekibi düşünelim. Bir kişi Windows kullanıyor, diğeri macOS, üçüncüsü Linux. Birinin bilgisayarında Node.js 18, diğerinde Node.js 20 var. Veritabanı birinde yerel PostgreSQL, diğerinde Docker ile çalışıyor. Kod aynı repodan gelse bile ortam aynı değildir. Bu yüzden hata bazen kodda değil, kodun çalıştığı çevrede ortaya çıkar.

Konteyner yaklaşımı bu problemi tamamen yok etmez; fakat çalışma ortamını tarif edilebilir hale getirir. "Şu sürüm Node.js, şu bağımlılıklar, şu port, şu veritabanı servisi" gibi bilgileri dosyalaştırır. Yönetim Bilişim Sistemleri açısından bu, teknik kolaylıktan daha fazlasıdır: ekip zamanını, teslim güvenilirliğini ve operasyonel riski yönetme meselesidir.

Farklı işletim sistemleri

Farklı işletim sistemleri dosya yolları, satır sonları, terminal komutları ve bazı paket davranışları açısından farklılık gösterebilir. Örneğin Windows'ta çalışan bir yol ifadesi Linux ortamında aynı şekilde çalışmayabilir. Docker, işletim sistemlerini yarıştırmadan bu farkların proje çalışmasına etkisini azaltan daha tutarlı bir Linux ortamı sunabilir; fakat Windows ve macOS üzerinde kendisinin de ek katmanlarla çalıştığı unutulmamalıdır.

Farklı runtime ve paket versiyonları

Runtime, uygulamanın çalışmasını sağlayan yürütme ortamıdır. Node.js, Python veya Java sürümü değiştiğinde aynı kod farklı davranabilir. Paket sürümleri de benzer risk taşır. Bir ekip üyesinde bağımlılıkların eski sürümü varken diğerinde yeni sürüm olabilir. Dockerfile içinde runtime sürümünü belirtmek ve bağımlılık kurulumunu aynı adımlarla yapmak, bu belirsizliği azaltır. Yine de kilit dosyaları ve güncelleme politikası ayrıca yönetilmelidir.

Farklı ortam değişkeni, port ve servis ayarları

Ortam değişkeni, port ve servis ayarları çoğu zaman görünmeyen farklar üretir. Bir bilgisayarda `DATABASE_URL` doğruyken diğerinde boş olabilir. Birinde uygulama `3000` portunda çalışırken diğerinde aynı port dolu olabilir. Birinde Redis açıktır, diğerinde hiç kurulmamıştır. Bu farklar kodun dışındadır; fakat uygulamanın davranışını belirler. Docker ve Docker Compose, bu ayarları dosyada görünür hale getirerek ekip içinde tekrarlanabilirliği artırır.

7.2 Konteyner Nedir?

Bir uygulamanın çalışması için yalnızca kodu kopyalamak yetmez. Çalışma zamanı, sistem paketleri, ortam değişkenleri, dosya yapısı ve başlangıç komutu da gerekir. Bu parçaları her bilgisayarda elle kurmak hem zaman alır hem de hata üretir. Peki bu ortamı paketleyebilir miyiz?

Konteyner, bir uygulamanın çalışması için gereken kullanıcı alanı dosyalarını, bağımlılıkları ve ayarları izole bir çalışma ortamında bir araya getiren yapıdır. Basitçe söylemek gerekirse, uygulamanın "kendi küçük çalışma odası" vardır. Bu oda host işletim sistemiyle bazı kaynakları paylaşır; fakat uygulamanın dosya sistemi, süreçleri ve ağ ayarları belirli ölçüde ayrılır.

Konteyner, sanal makine değildir. Kendi başına tam bir işletim sistemi açmaz. Bu sebeple çoğu durumda daha hızlı başlar ve daha az kaynak kullanır. Ancak bu aynı

zamanda sınırını da gösterir: konteynerler izolasyon sağlar, fakat güvenlik ve operasyon açısından her şeyi otomatik çözmez.

Yönetim Bilişim Sistemleri açısından konteynerin değeri, proje kurulumunu kişisel bilgisayar alışkanlıklarından ayırmasıdır. Böylece "bende çalışıyor" yerine "tarif edilen ortamda çalışıyor" demeye yaklaşırız.

İzole çalışma ortamı, uygulamanın kendi bağımlılıkları ve ayarlarıyla diğer uygulamalardan ayrılmasıdır. Örneğin bir projede PostgreSQL 16 gerekirken başka projede farklı sürüm gerekebilir. Konteyner kullanıldığında bu servisler birbirini daha az etkiler. Ancak izolasyon mutlak değildir; port, volume, gizli bilgi ve ağ ayarları yanlış yapılırsa projeler yine birbirine karışabilir.

Bir apartmanda her dairenin kendi kapısı, eşyası ve elektrik sayacı olduğunu düşünelim. Daireler aynı binayı paylaşır; fakat günlük kullanımda birbirinden ayrıdır. Konteyner de buna benzer biçimde host kaynaklarını paylaşırken süreçleri, dosya sistemini ve ağ görünümünü ayırır.

Süreç izolasyonu, konteyner içindeki çalışan programların ayrı görünmesini sağlar. Dosya sistemi izolasyonu, konteynerin kendi dosya yapısını kullanmasına imkan verir. Ağ izolasyonu ise hangi portların dışarı açılacağını ve konteynerlerin birbirini nasıl göreceğini belirler. Bu ayrımlar pratikte kurulum düzeni sağlar; fakat güvenlik garantisi gibi düşünülmemelidir.

Konteynerin geliştiriciye sağladığı temel fayda, kurulum bilgisini kodla birlikte tarif edebilmesidir. Yeni ekip üyesi "PostgreSQL'i nasıl kuracağım, hangi sürüm Node.js lazım, Redis nereden gelecek?" sorularıyla boğulmak yerine Docker dosyalarını çalıştırabilir. Bu, öğrenmeyi ortadan kaldırmaz; fakat kurulumu daha tekrarlanabilir hale getirir. Sınır şudur: Docker dosyaları kötü yazılmışsa, aynı karmaşa bu kez konteyner içinde yaşanır.

7.3 Konteyner ve Sanal Makine Farkı

Konteyner ile sanal makine sık sık karşılaştırılır. Bunun sebebi ikisinin de izolasyon sağlamasıdır. Ancak aynı problemi aynı yöntemle çözmezler. Sanal makine, ayrı bir işletim sistemi çalıştırır. Konteyner ise host işletim sisteminin çekirdeğini paylaşır ve uygulama ortamını izole eder.[1]

Bu farkın karar anında hangi soruları gündeme getirdiği Tablo 7.1 ile özetlenmiştir.

Tablo 7.1: Konteyner ve Sanal Makine

Ölçüt	Sanal makine	Konteyner
İşletim sistemi	Kendi işletim sistemini çalıştırır	Host çekirdeğini paylaşır
Başlama süresi	Daha yavaş olabilir	Genellikle daha hızlıdır

[1] Mavridis, I., & Karatza, H. (2019). Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing. *Future Generation Computer Systems*, *94*, 674–696. https://doi.org/10.1016/j.future.2018.12.035

Ölçüt	Sanal makine	Konteyner
Kaynak kullanımı	Daha ağırdır	Daha hafiftir
İzolasyon türü	Daha kalın sınır	Uygulama ortamı odaklı sınır
Geliştirme ortamı	Tam sistem gerektiğinde anlamlıdır	Servisleri hızlı çalıştırmada güçlüdür

Bazı durumlarda sanal makine daha doğru tercih olabilir. Ancak geliştirme ortamında uygulama, veritabanı ve cache gibi servisleri hızlı ve tekrarlanabilir çalıştırmak istediğimizde konteynerler pratik bir avantaj sağlar.

Sanal makine

Sanal makine, fiziksel bilgisayarın içinde ayrı bir bilgisayar varmış gibi çalışır. Kendi işletim sistemi, kendi disk alanı ve kendi sistem servisleri bulunur. Bu yaklaşım güçlü izolasyon sağlar; fakat geliştirme sırasında hızlı açılıp kapanması gereken küçük servisler için gereğinden ağır olabilir. Sanal makineyi tam ortam gerektiğinde, konteyneri ise uygulama servislerini taşınabilir hale getirmek istediğimizde düşünmek daha doğrudur.

Konteyner

Konteyner yaklaşımı, uygulamanın ihtiyaç duyduğu ortamı hafif bir paket halinde çalıştırmaktır. Örneğin bir web uygulaması için Node.js, uygulama dosyaları ve başlatma komutu bir imaj içinde tarif edilir; çalıştırıldığında bu imajdan konteyner oluşur. Bu düzen geliştiriciye hız kazandırır, fakat host işletim sistemi ve Docker altyapısına bağımlılığı bütünüyle ortadan kaldırmaz.

Kaynak kullanımı ve başlama süresi

Konteynerler genellikle sanal makinelere göre daha hızlı başlar ve daha az kaynak kullanır. Çünkü tam bir işletim sistemi başlatmak yerine uygulama sürecini izole biçimde çalıştırırlar. Bu, geliştirme ortamında önemli bir avantajdır. Ancak kaynak tüketimi yine de sıfır değildir; çok sayıda konteyner, büyük imajlar ve gereksiz servisler bilgisayarı yavaşlatabilir.

Geliştirme ortamında tercih nedenleri

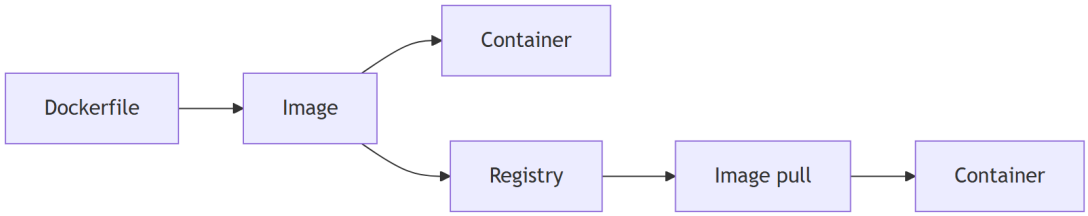
Geliştirme ortamında konteyner tercihinin en pratik nedeni, ekibin aynı servisleri benzer şekilde çalıştırabilmesidir. Bir ekip üyesi “PostgreSQL kuramadım” diye saatler kaybetmek yerine `docker compose up` ile veritabanını başlatabilir. Teslim haftasında bu fark önemlidir; çünkü zaman koddan çok kurulum sorunlarına harcanırsa proje riske girer.

7.4 İmaj, Konteyner ve Kayıt Deposu

Docker öğrenirken üç kavram sürekli karşımıza çıkar: imaj, konteyner ve kayıt deposu.[2] Bu kavramlar karışırsa komutlar da ezbere dönüşür. O yüzden önce aralarındaki ilişkiyi netleştirelim.

İmaj (image), çalıştırılacak ortamın şablonudur. Konteyner, bu şablondan çalışan örnektir. Kayıt deposu (registry), imajların saklandığı ve paylaşıldığı yerdir. Docker Hub en bilinen kayıt depolarından biridir.

İlişkiyi şöyle görebiliriz:



Şekil 7.1: Dockerfile, imaj, konteyner ve kayıt deposu ilişkisi

Şekil 7.1 şunu gösterir: Dockerfile imaj üretir, imaj konteyner olarak çalışır, imaj kayıt deposu üzerinden başka ortamlara taşınabilir. Geliştirme ortamının tekrarlanabilirliği bu ilişki sayesinde güçlenir.

İmaj ve konteyner farkı

İmaj ile konteyner farkını tarif ve yemek benzetmesiyle düşünebiliriz. Tarif imaj gibidir; hangi malzemenin hangi sırayla kullanılacağını söyler. Pişmiş yemek ise bu tariften üretilen konteynerdir. Aynı tarifle iki tabak yemek hazırladığımızda başlangıçları benzerdir; fakat servis edildikten sonra tabaklardan birine tuz eklemek yalnızca o tabağı değiştirir. Tarif ve diğer tabak bundan etkilenmez.

Konteynerlerde de her çalışan örnek, imajın üzerine kendisine ait yazılabilir bir katman ekler. Bir konteyner içinde oluşturulan geçici dosya veya yapılan değişiklik, imajı ve aynı imajdan başlayan diğer konteynerleri kendiliğinden değiştirmez. Konteyner kaldırıldığında bu yazılabilir katmandaki veriler de kaybolabilir. Kalıcı uygulama değişikliği için çalışan konteyneri elle özelleştirmek yerine Dockerfile güncellenip yeni imaj üretilir; kalması gereken uygulama verisi ise volume gibi ayrı bir kalıcılık mekanizmasında tutulur.

Temel imaj ve imaj etiketi

Temel imaj (base image), kendi imajımızı üzerine kurduğumuz başlangıç imajıdır. Örneğin `node:20-alpine`, Node.js 20 içeren daha küçük bir Linux tabanlı imajdır. Etiket (tag) ise imaj sürümünü veya çeşidini belirtir. `node:latest` kolay görünür; fakat

[2] Casalicchio, E., & Iannucci, S. (2020). The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency and Computation: Practice and Experience*, *32*(17). https://doi.org/10.1002/cpe.5668

zaman içinde değişebilir. Bu yüzden ekip projelerinde daha belirli bir etiket kullanmak, tekrarlanabilirlik açısından daha güvenlidir.

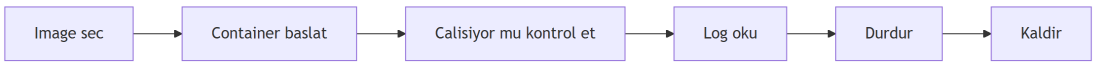
Kayıt Deposu ve Docker Hub

Kayıt deposu, imajların saklandığı ve indirildiği depodur. Docker Hub, GitHub'ın kod için yaptığına benzer biçimde imajlar için merkezi paylaşım noktası gibi düşünülebilir. Ancak her imaj güvenilir kabul edilmemelidir. Ekip, kullandığı imajın kaynağını, etiket bilgisini ve güncelliğini kontrol etmelidir. Yönetim Bilişim Sistemleri açısından bu, tedarik zinciri riskinin küçük ama önemli bir parçasıdır.[3]

7.5 Konteyner Yaşam Döngüsü

Konteyner çalıştırmak tek bir anlık işlem değildir. Başlatma, izleme, log okuma, durdurma ve kaldırma gibi bir yaşam döngüsü vardır. Bu döngüyü anlamak, Docker komutlarını ezberlemekten daha faydalıdır; çünkü hata çıktığında hangi adımda olduğunuzu bilirsiniz.

Şekil 7.2 bu döngüyü gösterir.



Şekil 7.2: Konteyner yaşam döngüsü

Bu akış ekip açısından da önemlidir. Bir servis çalışıyor mu, hangi portu kullanıyor, hata logu ne söylüyor, eski konteyner hâlâ ayakta mı? Bu sorulara yanıt vermeden Docker kullanımı karışık hale gelir.

7.5.1 Konteyner başlatmak, durdurmak ve kaldırmak

Konteyner başlatmak, imajdan çalışan bir örnek oluşturmaktır. Durdurmak, çalışan süreci kapatır. Kaldırmak ise konteyner kaydını siler. Örneğin:

```
$ docker run -d --name web-demo -p 8080:80 nginx:alpine
$ docker stop web-demo
web-demo
$ docker rm web-demo
web-demo
```

İlk komut `nginx:alpine` imajından `web-demo` adlı konteyneri arka planda başlatır ve host üzerindeki `8080` portunu konteyner içindeki `80` portuna bağlar. Tarayıcıdan `localhost:8080` adresine gidildiğinde konteyner içindeki web sunucusuna ulaşılır. Son iki komut önce çalışan konteyneri durdurur, ardından konteyner kaydını kaldırır; imaj yerinde kalır.

[3] Enck, W., & Williams, L. (2022). Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Security & Privacy*, *20*(2), 96–100. https://doi.org/10.1109/MSEC.2022.3142338

Çalışan konteynerleri görmek

Çalışan konteynerleri görmek, bilgisayarda hangi servislerin ayakta olduğunu anlamaya benzer. Bir ekip üyesi "uygulama açılmıyor" dediğinde ilk sorulardan biri web konteynerin gerçekten çalışıp çalışmadığıdır. `docker ps` komutu çalışan konteynerleri listeler. Sınır şudur: yalnızca çalışanları gösterir; durmuş konteynerleri görmek için farklı seçenekler gerekir.

7.5.2 Log okumak

Log okumak, konteyner içindeki uygulamanın ne söylediğini görmektir. Çalışmayan bir uygulamada tahmin yürütmek yerine loga bakmak gerekir.

```
$ docker logs web-demo
192.168.65.1 - - [18/Jul/2026:21:13:41 +0000] "GET / HTTP/1.1" 200 896 ...
```

Bu erişim kaydı, ana makineden gelen `GET /` isteğinin `200` durum koduyla yanıtlandığını gösterir. Uygulama hata üretseydi aynı komut hata kaydını da görünür kılabilirdi. Buna karşılık host üzerindeki `8080` portu zaten doluysa hata konteyner sürecinden değil Docker motorundan gelir ve `docker run` çıktısında `port is already allocated` ifadesi görülür; böyle bir hata `docker logs` içinde aranmaz.

7.5.3 Geçici konteyner ile kalıcı servis farkı

Geçici konteyner kısa bir iş için çalıştırılır ve işi bitince kapatılır. Kalıcı servis ise geliştirme süresince ayakta kalması beklenen veritabanı veya web uygulaması gibi yapılardır (Tablo 7.2).

Tablo 7.2: Geçici konteyner ile kalıcı servis farkı

Kullanım	Örnek	Dikkat
Geçici konteyner	Tek seferlik komut çalıştırma	Veri kalıcılığı beklenmez
Kalıcı servis	PostgreSQL, Redis, web app	Volume, port ve restart davranışı düşünülür
Etkileşimli inceleme	`docker run -it --rm ubuntu bash`	Konteyner kapanınca içindeki değişiklikler silinir
Adlandırılmış volume ile veri	PostgreSQL verisi `docker volume` üzerinde tutulur	Konteyner silinse bile veri volume'da kalır
Sağlık kontrollü servis	`healthcheck` tanımlı veritabanı	"Ayakta" olmak "hazır" olmak anlamına gelmeyebilir

Her konteyner kalıcı servis gibi yönetilmez; yaşam döngüsü kullanım biçimine göre belirlenir.

7.6 Docker CLI Çıktısını Okumak

Docker kullanırken terminal çıktısını okumak komut yazmak kadar önemlidir. Çünkü Docker çoğu zaman bize sorunun nerede olduğunu söyler: imaj mı bulunamadı, port mu dolu, konteyner mi durmuş, build sırasında dosya mı eksik? Çıktıyı okumadan komutları tekrar etmek zaman kaybettirir.

Basit bir okuma alışkanlığı şöyle kurulabilir:

1. Komut hangi nesneyle çalışıyor: imaj mı, konteyner mi, volume mü?
2. Hata build aşamasında mı, çalıştırma aşamasında mı?
3. Çıktı port, dosya, yetki veya ağ sorunu mu söylüyor?
4. Bir sonraki en küçük kontrol komutu ne olabilir?

Bu sorular, Docker'ı ezberden çıkarıp hata ayıklama aracına dönüştürür.

7.6.1 Komut kalıbı, argüman ve flag

Docker komutları genellikle bir eylem, bazı seçenekler ve hedef nesneden oluşur.

```
$ docker run --name api -p 3000:3000 my-api:dev
```

Burada `run` eylemdir, `--name api` konteyner adını belirler, `-p 3000:3000` port eşlemesidir, `my-api:dev` ise çalıştırılacak imajdır. Komutu bu parçalara ayırdığınızda, hata çıktığında hangi kısmı kontrol edeceğinizi daha kolay görürsünüz.

Konteyner ve imaj listelerini yorumlamak

Konteyner listesi çalışan veya daha önce oluşturulmuş servis örneklerini, imaj listesi ise kullanılabilir şablonları gösterir. Bir depoda hem kullanılmaya hazır ürünler hem de o ürünlerden üretilmiş aktif siparişler olduğunu düşünün. İmaj şablondur; konteyner çalışan örnektir. Yanlışlıkla imaj silmeye çalışırken konteynerin hâlâ onu kullandığını görmek bu ayrımın pratik sonucudur.

7.6.2 Hata mesajından sonraki adımı çıkarmak

Okuyucu `docker compose up` çalıştırır ve uygulama açılmaz. Panikle Docker'ı yeniden kurmak yerine hata mesajı okunmalıdır (Tablo 7.3).

Tablo 7.3: Hata mesajından sonraki adımı çıkarmak

Hata mesajı	Muhtemel anlam	Sonraki adım
`port is already allocated`	Host portu dolu	Çalışan servisleri ve port eşlemesini kontrol et
`no such file or directory`	Bağlam içinde dosya yok	Dockerfile ve dosya yolunu kontrol et
`connection refused`	Servis hazır değil veya adres yanlış	Servis adı, port ve başlama sırasını kontrol et

Bu okuma alışkanlığı teknik problemi iş etkisine bağlar: `connection refused` hatasını kod hatası sanıp saatlerce fonksiyon didiklemek yerine servis adını ve başlama sırasını kontrol etmek, aynı teslim gününde ekibe saatler kazandırabilir.

7.7 Dockerfile Yazımı

Dockerfile, bir imajın nasıl üretileceğini tarif eden dosyadır. Bir projenin çalışması için gereken adımları elle hatırlamak yerine dosyaya yazmamızı sağlar. Bu yönüyle Dockerfile, geliştirme ortamının tarifidir.

Basit bir Node.js uygulaması için Dockerfile şöyle olabilir:

```
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
EXPOSE 3000
CMD ["npm", "run", "dev"]
```

Bu dosyadaki her satır bir gerekçe taşır. Temel imaj seçilir, çalışma dizini belirlenir, bağımlılıklar kurulur, uygulama dosyaları kopyalanır ve başlatma komutu tarif edilir. Satırlar ezberlenecek bir kalıptan çok çalışma ortamının nasıl kurulduğunu gösteren kayıtlar olarak okunmalıdır.

Dockerfile katman düzeniyle çalışır:

```
FROM node:20-alpine
  -> WORKDIR /app
    -> COPY package*.json
      -> RUN npm ci
        -> COPY .
          -> CMD npm run dev
```

Bu sıralama cache davranışını da etkiler; sık değişmeyen adımların önce yazılması build süresini azaltabilir.

Dockerfile nedir?

Dockerfile genellikle projenin kök dizininde veya ilgili servis klasöründe bulunur. Örneğin `api/Dockerfile`, API servisinin nasıl imaj haline getirileceğini anlatabilir. Bu dosya paylaşılabilir ve versiyonlanabilir olmalıdır; çünkü ekip aynı imaj tarifini kullanır. Ancak gizli bilgiler Dockerfile içine yazılmamalıdır. Dockerfile ortamı tarif eder, gizli bilgiyi taşımaz.

Temel imaj seçmek

Temel imaj seçimi, projenin üzerine kurulacağı başlangıç ortamını seçmektir. `node:20` daha genel, `node:20-alpine` daha küçük olabilir. Daha küçük imajlar avantaj sağlayabilir; fakat bazı sistem paketleri eksik olabilir. Bu yüzden seçim “en küçük olan en

iyidir" diye yapılmaz. Projenin ihtiyaç duyduğu paketler, ekip deneyimi ve hata ayıklama kolaylığı birlikte düşünülür.

7.7.1 Çalışma dizini, dosya kopyalama ve bağımlılık kurulumu

`WORKDIR`, konteyner içinde komutların hangi klasörde çalışacağını belirler. `COPY`, host tarafındaki dosyaları imaj içine taşır. `RUN npm ci` gibi bir adım ise bağımlılıkları kurar. Sıralama önemlidir:

```
COPY package*.json ./
RUN npm ci
COPY . .
```

Bu düzen, önce bağımlılık tanımlarını kopyalayıp paketleri kurar; sonra uygulama kodunu taşır. Kod sık değiştiğinde bağımlılık kurulumunun her seferinde yeniden yapılmasını azaltabilir.

7.7.2 Uygulamayı başlatma komutunu tarif etmek

`CMD`, konteyner başladığında hangi komutun çalışacağını belirtir. Örneğin:

```
CMD ["npm", "run", "dev"]
```

Bu satır, konteynerin yalnızca dosya taşıyan bir kutu olmadığını, başladığında hangi uygulama sürecini çalıştıracağını gösterir. Yanlış CMD verilirse imaj başarıyla build edilse bile konteyner hemen kapanabilir.

Katman ve cache kavramları

Dockerfile'daki adımlar katmanlar oluşturur. Docker daha önce değişmemiş katmanları cache'ten kullanabilir. Bu, build süresini azaltır. Ancak cache bazen eski davranışı saklıyormuş gibi de görünebilir. Bu yüzden build sırasında hangi adımın yeniden çalıştığını okumak önemlidir.

7.8 İmaj oluşturmak

Dockerfile yalnızca bir tariftir; kendi başına çalışmaz. Bu tarifi çalıştırılabilir bir ortama dönüştürecek ayrı bir adım gerekir. İmaj oluşturmak (build), Dockerfile tarifinden çalıştırılabilir imaj üretmektir. Bu işlem sırasında Docker gerekli dosyaları bağlamdan alır, Dockerfile adımlarını uygular ve sonuçta etiket verilmiş bir imaj oluşturur.

Basit komut şöyledir:

```
$ docker build -t stock-api:dev .
```

Burada `-t stock-api:dev` imaja ad ve etiket verir, sondaki `.` ise bağlam olarak mevcut klasörü gösterir. Bu küçük nokta önemlidir; Docker hangi dosyaları görebileceğini buradan anlar.

Build süreci ekip açısından "bu uygulama tarif edilen ortamdan yeniden üretilebiliyor mu?" sorusunun yanıtıdır.

7.8.1 Bağlam (build context) nedir?

`docker build` komutunun sonundaki nokta ilk bakışta gözden kaçabilir; oysa Docker'a hangi dosyaları görebileceğini bu işaret bildirir. Bağlam (build context), Docker'ın imaj üretirken erişebileceği dosya alanıdır. Komutta sondaki `.` kullanıldığında mevcut klasör bağlam olur.

```
project/
├─ Dockerfile
├─ package.json
├─ src/
└─ node_modules/
```

Eğer bağlama gereksiz büyük klasörler girerse build yavaşlar. `node_modules`, log dosyaları veya gizli dosyalar bağlama taşınmamalıdır. Bu bizi `.dockerignore` ihtiyacına götürür.

7.8.2 `.dockerignore` neden gerekir?

`.dockerignore`, Docker build sırasında hangi dosyaların bağlam dışında kalacağını belirtir. Bir ders projesinde `node_modules` klasörü veya `.env` dosyası yanlışlıkla bağlama girerse imaj büyüyebilir veya gizli bilgi riski oluşabilir. Teslim haftasında build'in dakikalarca sürmesi bazen koddan değil, gereksiz dosyaların bağlama taşınmasından kaynaklanır.

Kısa bir örnek:

```
node_modules
.env
.git
dist
```

Bu dosya hız, güvenlik ve tekrarlanabilirlik açısından önemlidir.

Etiket verme

Etiket (tag), imajın hangi sürüm veya amaç için üretildiğini gösterir. `stock-api:dev`, geliştirme için; `stock-api:1.0.0`, belirli sürüm için kullanılabilir. Etiket verilmezse hangi imajın neye karşılık geldiği karışır. Etiket, teknik çıktının izlenebilirliğini artırır; hangi sürümün hangi ortamda çalıştığı daha kolay anlaşılır.

7.8.3 Oluşturma hatalarını okuma

Oluşturma hatası çıktığında bütün Dockerfile'ı değiştirmek yerine hangi adımın kırıldığına bakmak gerekir (Tablo 7.4).

Tablo 7.4: İmaj oluşturma hataları

Hata	Problem	Çözüm
`COPY package.json` başarısız	Dosya bağlam içinde yok	Klasör konumunu ve oluşturma bağlamını kontrol et
`npm ci` başarısız	Kilit dosyası veya paket sorunu	`package-lock.json` ve Node sürümünü kontrol et
Paket indirilemiyor	Ağ veya kayıt deposu problemi	Bağlantı ve depo erişimini kontrol et
`failed to solve`	Alt adımlardan biri başarısız oldu, bu yalnızca genel bir sarmalayıcı mesajdır	Hatanın asıl nedenini gösteren `ERROR` satırını oku
Bağlam çok büyük	Oluşturma dakikalar sürer	`.dockerignore` dosyasını kontrol et
Gizli dosya kopyalanmış	İmaj risk taşır	`.env` ve anahtarları bağlamdan çıkar
Yanlış temel imaj	Sürüm veya mimari uyuşmaz	`FROM` satırındaki etiketi doğrula

Oluşturma hatasını doğru okumak, yanlış müdahaleyi ve zaman kaybını azaltır. Örneğin bağlamın gereğinden büyük olması yüzünden dakikalarca süren bir oluşturmayı tekrar tekrar denemek yerine `.dockerignore` dosyasını kontrol etmek, aynı sorunu birkaç saniyede çözebilir.

7.9 Volume Kavramı

Konteyneri silip yeniden oluşturduğumuzda içindeki bazı verilerin kaybolduğunu görmek başlangıçta şaşırtıcı olabilir. Özellikle veritabanı konteyneri kullanan okuyucular "dün eklediğim kayıtlar nereye gitti?" sorusuyla karşılaşır. Bu soru bizi volume kavramına götürür.

Volume, konteyner yaşam döngüsünden bağımsız biçimde veri saklamaya yarayan mekanizmadır. Türkçe olarak kalıcı veri alanı diyebiliriz. Konteyner silinse bile volume duruyorsa veri korunabilir. Bu özellikle veritabanı servisleri için önemlidir.

Verinin kalıcı olması, her zaman güncel ve doğru olduğu anlamına gelmez. Yeni bir konteyner başlatıldığında eski volume içindeki veri ve veritabanı şeması kullanılmaya devam eder; uygulama yeni bir şema bekliyorsa sürüm uyumsuzluğu oluşabilir. Bu nedenle hangi verinin kalıcı olacağı açıkça belirlenmeli, şema değişiklikleri göç ile yönetilmeli ve kritik veriler yedeklenmelidir. Volume silmek ise veri kaybına yol açabileceği için yalnızca adı ve içeriği doğrulandıktan sonra bilinçli biçimde yapılmalıdır.

Konteyner içindeki veri neden kalıcı değildir?

Konteyner, imajdan üretilmiş çalışan örnektir. İçinde dosya değişebilir; fakat konteyner silindiğinde bu değişiklikler de kaybolabilir. Bunu otel odası gibi düşünebiliriz. Odada

kaldığınız süre boyunca eşyalarınızı yerleştirirsiniz; fakat çıkış yaptığınızda oda temizlenir. Kalıcı saklama istiyorsanız ayrı bir depoya ihtiyaç vardır. Docker'da bu ihtiyaç çoğu zaman volume ile karşılanır.

7.9.1 Volume ve bind mount farkı

Volume ve bind mount benzer görünür; ikisi de konteyner dışındaki veriyi konteyner ile ilişkilendirir. Ancak kullanım amacı farklıdır (Tablo 7.5).

Tablo 7.5: Volume, bind mount ve tmpfs farkı

Yaklaşım	Ne zaman kullanılır?	Dikkat
Volume	Veritabanı verisi gibi Docker tarafından yönetilen kalıcı veri	Nerede durduğunu Docker yönetir
Bind mount	Kod klasörünü konteyner içine bağlamak	Host dosya yapısına bağlıdır
tmpfs / bellek bağlama	Geçici önbellek veya deneme verisi	Konteyner kapanınca kaybolur
Paylaşılan named volume	Birden fazla konteyner aynı veriyi kullanır	Yetki ve yedek planı gerekir

Geliştirme ortamında bind mount hot reload için, volume ise veritabanı kalıcılığı için sık kullanılır.

Veritabanı verisini saklamak

Veritabanı konteyneri volume olmadan çalıştırılırsa, konteyner silindiğinde kayıtlar da kaybolabilir. Bu yüzden PostgreSQL veya MySQL servislerinde veri dizini çoğu zaman volume'a bağlanır. Bu yaklaşım geliştirme ortamında rahatlık sağlar. Ancak testleri temiz veriyle çalıştırmak istediğinizde eski volume sorun çıkarabilir. Kalıcılık ve temiz başlangıç ihtiyacı ayrı düşünülmelidir.

Geliştirme ortamında volume kullanımı

Geliştirme ortamında volume iki amaçla karşımıza çıkar: veriyi korumak ve kod değişikliklerini konteynere yansıtmak. Kod klasörünü bind mount ile bağladığınızda dosyayı host üzerinde değiştirir, konteyner içinde sonucu görebilirsiniz. Bu hot reload için kullanışlıdır. Ancak host ve konteyner dosya izinleri, işletim sistemi farkları ve eski volume verisi bazen beklenmeyen hatalar üretebilir.

7.10 Network Kavramı

Bir uygulama tek başına çalıştığında ağ konusu çok görünür olmayabilir. Ancak web uygulaması veritabanına bağlanacak, cache servisine gidecek veya dış dünyaya port açacaksa network ayarları belirleyici hale gelir. Docker'da network, konteynerlerin birbirini nasıl gördüğünü ve host makineyle nasıl konuştuğunu düzenler.

Geliştirme ortamında ağ ilişkisini okumak için üç soru çoğu zaman yeterlidir: Konteyner dış dünyaya hangi porttan açılıyor? Konteyner başka konteynere hangi adla ulaşıyor? `localhost` kimin localhost'u?

Bu sorular yanlış anlaşıldığında uygulama çalışıyor gibi görünür ama veritabanına bağlanamaz. Dolayısıyla Docker network bilgisi, kurulum hatalarını okumak için gereklidir.

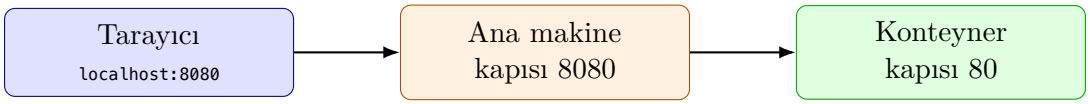
Konteyner network nedir?

Konteyner network, konteynerlerin birbirleriyle ve dış dünyayla nasıl iletişim kuracağını belirleyen sanal ağ düzenidir. Bir ofisteki dahili telefon sistemi gibi düşünülebilir. Dışarıdan herkes her dahili numarayı doğrudan arayamaz; bazı hatlar dışarı açılır, bazıları içeride kalır. Docker'da da bazı portlar host'a publish edilir, bazı servisler yalnızca aynı network içinden görünür.

Port açmak ve eşleştirmek

Port açmak, konteyner içindeki bir bağlantı noktasını ana makineden erişilebilir hale getirmektir. `-p 8080:80` yazıldığında sıra sabittir: solda ana makine, sağda konteyner. Tarayıcıdan `http://localhost:8080` açarsınız; istek konteynerin 80 numaralı kapısına düşer. Tersi yazılırsa "konteyner çalışıyor ama sayfa açılmıyor" hatası çoğu zaman buradadır.

Dışarı açılmayan bağlantı noktası içeride dinleyebilir. Veritabanı 5432'de çalışıyor olabilir; fakat `-p` yoksa ana makineden görünmez. Konteynerler birbirine, yayımlanan bağlantı noktasından değil, aynı ağdaki servis adından ulaşır: `db:5432`. Ana makine 8080 doluysa eşleme yine başarısız olur; bu kod hatası değil, ortam çakışmasıdır.



Şekil 7.3: Ana makine kapısının konteyner kapısıyla eşlenmesi

Şekil 7.3 bu üç adımı ayırır. Hangi kapının dışarı açıldığı README veya Compose dosyasında görünür olmalıdır.

7.10.1 `localhost` yanılgıları

`localhost` her zaman "benim bilgisayarım" anlamına gelmez; içinde bulunduğunuz ortama göre değişir. Konteyner içinden `localhost` dediğinizde çoğu zaman konteynerin kendisini işaret edersiniz, host makineyi değil. Bu yüzden backend konteyneri veritabanına `localhost` ile bağlanmaya çalışırsa yanlış yere bakabilir. Compose içinde servis adı kullanmak bu yanılgıyı azaltır.

Docker Compose içinde servisler genellikle birbirine servis adıyla ulaşabilir. Örneğin backend konteyneri veritabanına `db:5432` adresiyle bağlanabilir. Buradaki `db`, Compose dosyasındaki servis adıdır. Bu, ekip için okunabilir ve tekrarlanabilir bir yapı sağlar. Ancak servis adı yalnızca aynı Docker network içindeki konteynerler için anlamlıdır; host makineden aynı ad doğrudan çalışmayabilir.

7.11 Ortam Değişkenleri ve Konteyner Yapılandırma

Aynı uygulama geliştirme, test ve üretim ortamında farklı ayarlara ihtiyaç duyabilir. Veritabanı adresi, port, log seviyesi, API anahtarı veya çalışma modu değişebilir. Bu değerleri koda yazmak, ortamları birbirine karıştırır ve güvenlik riski üretir.

Ortam değişkenleri (environment variables), uygulamanın dışarıdan konfigüre edilmesini sağlar. Docker ile çalışırken bu değerler konteyner başlatılırken verilebilir veya Compose dosyasında tarif edilebilir. Böylece aynı imaj farklı ortamlarda farklı ayarlarla çalışabilir.

Ancak burada kritik ayrım vardır: yapılandırma ile gizli bilgiler aynı şey değildir. Port numarası yapılandırma içerisine dahil edilebilir ancak veritabanı parolası gizli bir bilgi olmalıdır. Yapılandırma ve gizli bilgiler bu yüzden birbirinden ayrıştırılmalıdır: yapılandırma paylaşılabilir, gizli bilgiler ise dikkatle korunmalıdır.

Konteyner içinde ortam değişkeni

Konteyner içinde ortam değişkeni, uygulamanın çalışma sırasında okuduğu dış ayardır. Örneğin `NODE_ENV=development` uygulamanın geliştirme modunda çalışacağını gösterebilir. Bu bilgi imajın içine sabitlenmek zorunda değildir; konteyner başlatılırken verilebilir. Böylece aynı imaj farklı ortam ihtiyaçlarına uyarlanabilir.

7.11.1 `.env` dosyası

`.env` dosyası ortam değişkeni değerlerini yerel geliştirme ortamında toplamak için kullanılır. Örnek biçim paylaşılabilir, gerçek değerler paylaşılmamalıdır.

```
DATABASE_URL=postgresql://app:password@db:5432/appdb
NODE_ENV=development
```

Bu örnek, beklenen formatı gösterir. Gerçek parola içeren `.env` dosyası repoya eklenmemeli; bunun yerine `.env.example` paylaşılmalıdır.

7.11.2 Gizli bilgi ve yapılandırma farkı

Yapılandırma, paylaşılması genellikle sorun olmayan ayardır; gizli bilgi ise erişim yetkisi veren değerdir. Bir geliştirici `APP_PORT=3000` bilgisini repoda tutabilir; fakat `DATABASE_PASSWORD=real-password` değerini tutmamalıdır (Tablo 7.6).

Tablo 7.6: Gizli bilgi ve yapılandırma farkı

Değer	Tür	Davranış
`APP_PORT=3000`	Yapılandırma	Örnek olarak paylaşılabilir
`NODE_ENV=development`	Yapılandırma	Ortama göre değişebilir
`DATABASE_PASSWORD`	Gizli bilgi	Repoya ve imaja gömülmez
`API_TOKEN`	Gizli bilgi	Güvenli alanda saklanır

Bu ayrım yapılmazsa teknik kolaylık güvenlik riskine dönüşür: `DATABASE_PASSWORD` değerini `APP_PORT` kadar sıradan görüp repoya eklemek, küçük bir rahatlık uğruna erişim ve maliyet riski taşıyan bir sızıntı yaratabilir.

Gizli bilgiyi imaj içine eklemek, gizli değeri imajı alan herkesin erişebileceği hale getirebilir. Örneğin Dockerfile içinde `ENV API_KEY=...` yazmak veya `.env` dosyasını imaja kopyalamak risklidir. İmaj kayıt deposuna push edildiğinde bu bilgi daha geniş bir alana yayılabilir.

Doğru yaklaşım, gizli değerleri runtime sırasında güvenli biçimde vermektir. Geliştirme ortamında `.env` kullanılabilir; üretim ortamında platformun gizli bilgi yönetimi tercih edilmelidir. İş etkisi açıktır: sızan anahtar maliyet, veri erişimi ve güven kaybı doğurabilir.

7.12 Docker Compose

Tek bir konteyner çalıştırmak başlangıç için yeterli olabilir. Ancak gerçekçi bir geliştirme ortamında çoğu zaman web uygulaması, veritabanı ve cache birlikte çalışır. Bunların her birini ayrı komutlarla başlatmak, portları ve ortam değişkeni değerlerini elle hatırlamak ekip için yorucudur.

Docker Compose, birden fazla servisi tek dosyada tarif edip birlikte çalıştırmamızı sağlar. `compose.yaml` dosyası hangi servislerin olduğunu, hangi portların açıldığını, hangi volume'ların bağlandığını ve hangi ortam değişkeni değerlerinin verildiğini görünür hale getirir.

Küçük bir örnek:

```yaml
services:
  web:
    build: .
    ports:
      - "3000:3000"
    environment:
      DATABASE_URL: postgresql://app:app@db:5432/appdb
    depends_on:
      - db
      - redis

  db:
    image: postgres:16
    environment:
      POSTGRES_USER: app
      POSTGRES_PASSWORD: app
      POSTGRES_DB: appdb
    volumes:
      - db_data:/var/lib/postgresql/data

  redis:
    image: redis:7
```

```
volumes:
  db_data:
```

Servis haritası şöyle okunabilir:

```
web  -> db
web  -> redis
db   -> db_data volume
web  -> host 3000 portu
```

Docker Compose neden var?

Bir web uygulaması yalnızca statik sayfa gösteriyorsa tek konteyner yeterli olabilir. Ancak uygulama kullanıcı girişi yapıyor, veritabanına yazıyor ve oturum bilgisini cache üzerinde tutuyorsa artık birden fazla servis vardır. Compose bu servisleri aynı geliştirme senaryosunda birlikte tarif eder. Böylece ekip "önce veritabanını başlat, sonra Redis'i aç, sonra web uygulamasını çalıştır" bilgisini komut hafızasına değil dosyaya koyar.

7.12.1 `compose.yaml` dosyasının genel yapısı

`compose.yaml` genellikle proje kökünde yer alır ve versiyonlanır. İçinde `services`, gerektiğinde `volumes` ve `networks` bölümleri bulunur. Bu dosya ekip için paylaşılabilir kurulum bilgisidir. Ancak üretim ortamının gerçek gizli değerleri bu dosyaya yazılmamalıdır. Geliştirme için örnek değerler kullanılabilir; gerçek değerler ortam değişkeni veya gizli bilgi yönetimiyle verilmelidir.

Service, port, volume, network ve environment alanları

Compose içindeki servis, çalışacak her ana parçayı ifade eder: web, db, redis gibi. Port dış dünyaya açılan bağlantı noktasıdır. Volume kalıcı veri alanıdır. Network servislerin birbirini görmesini sağlar. Environment ise servislerin ayarlarını taşır. Bir ekip bu alanları anladığında Compose dosyasını ezberlemez; hangi alanın hangi geliştirme problemini çözdüğünü okuyabilir.

Çok konteynerli uygulama (Multi-container)

Çok konteynerli uygulama, her işi tek büyük konteyner içine koymak yerine servisleri ayrı konteynerlerde çalıştırma yaklaşımıdır. Web uygulaması bir konteyner, veritabanı başka bir konteyner, cache başka bir konteyner olabilir. Bu ayrım sorumlulukları netleştirir. Ancak geliştirme ortamında bile gereksiz servis çoğaltmak karmaşa yaratır; her konteynerin ne işe yaradığı açık olmalıdır.

7.13 Uygulama, Veritabanı ve Önbellek (Cache)

Modern web uygulamalarında tek süreç çoğu zaman yeterli değildir. Uygulama kodu çalışır, veritabanı veri saklar, cache servisi geçici veriyi hızlı tutar. Bu servisleri birlikte çalıştırmak, sistem davranışını daha gerçekçi biçimde görmemizi sağlar.

Geliştirme ortamında basit akış şöyle olabilir:

1. `compose.yaml` servisleri tarif eder.
2. `docker compose up` servisleri başlatır.
3. Web uygulaması `db` servis adına bağlanır.
4. Redis gibi cache servisi gerekiyorsa ayrı servis olarak çalışır.
5. Loglar ve health durumu kontrol edilir.

Bu adımların amacı yalnızca konteyner başlatmak değildir. Amaç, ekip üyelerinin aynı yerel sistemi benzer biçimde kurabilmesini sağlamaktır.

Web uygulaması

Web uygulaması çoğu zaman kullanıcının etkileşime geçtiği servistir. Compose içinde build edilen veya hazır imajdan çalışan bir servis olarak tarif edilir. Port publish edilirse tarayıcıdan erişilebilir hale gelir. Ancak web konteynerin ayakta olması veritabanına bağlanabildiği anlamına gelmez; loglar ve hata mesajları ayrıca kontrol edilmelidir.

PostgreSQL veya MySQL

PostgreSQL veya MySQL gibi ilişkisel veritabanları Docker Compose ile yerelde kolayca çalıştırılabilir. Bu, ekip üyelerinin veritabanı kurulumunu tek tek yapmasını azaltır. Ancak veritabanı için volume kullanılıyorsa eski verinin kalabileceği unutulmamalıdır. Göç, başlangıç verisi ve temiz başlangıç ihtiyacı proje dokümantasyonunda açık yazılmalıdır.

Redis

Redis, cache, oturum veya kuyruk benzeri geçici veri ihtiyaçlarında kullanılabilir. Ders projelerinde her zaman gerekmez. Eğer kullanılacaksa Compose içinde ayrı servis olarak tarif edilmesi, uygulamanın gerçekçi bağımlılıklarını görünür kılar. Ancak cache servisi yokken de çalışması gereken yerler varsa bu davranış ayrıca test edilmelidir.

7.13.1 Servis bağımlılıkları ve ilk çalıştırma sırası

Servis bağımlılığı, bir servisin başka bir servis hazır olmadan tam çalışamamasıdır. Web uygulaması veritabanına ihtiyaç duyuyorsa db servisi başlamadan bağlantı hatası alabilir.

```
$ docker compose up
web-1  | Error: connect ECONNREFUSED db:5432
db-1   | database system is ready to accept connections
```

Bu çıktı bize zamanlama sorununu gösterir. `depends_on` başlatma sırasını düzenleyebilir; fakat servisin gerçekten hazır olup olmadığını her zaman garanti etmeyebilir. Bu yüzden uygulama tarafında tekrar deneme veya health check gibi mekanizmalar gerekebilir.

7.14 Geliştirme ve üretim Docker farkları

Geliştirme ortamı ile üretim ortamı aynı hedefe sahip değildir. Geliştirme ortamında hızlı değişiklik yapmak, log görmek ve anında yenileme (hot reload) kullanmak isteriz. Üretim ortamında ise güvenlik, kararlılık, küçük imaj ve tahmin edilebilir davranış önceliklidir.

Bu farkı görmezsek geliştirme kolaylığı üretim riskine dönüşebilir. Örneğin geliştirme imajı içinde gereksiz paketler, hata ayıklama araçları veya kaynak kodun tamamı bulunabilir. Üretim ortamında bunlar saldırı yüzeyini ve imaj boyutunu artırabilir.

Tablo 7.7: Geliştirme ve üretim Docker farkları

Ölçüt	Geliştirme	Üretim
Öncelik	Hız ve gözlemleme	Kararlılık ve güvenlik
Kod değişimi	Anında yenileme istenir	Sabit derleme çıktısı tercih edilir
Bağımlılıklar	Geliştirme araçları olabilir	Gereksiz bağımlılıklar azaltılır
Günlük (log) / hata ayıklama	Daha ayrıntılı olabilir	Kontrollü ve güvenli olmalıdır

Tablo 7.7 ileri seviye optimizasyon reçetesi değildir. Yalnızca yanlış ortam varsayımının iş riskine dönüşebileceğini gösterir.

Geliştirme imajı ve anında yenileme

Geliştirme imajı, geliştiricinin hızlı deneme yapmasını kolaylaştırabilir. Anında yenileme, kod değiştiğinde uygulamanın otomatik yenilenmesini sağlar. Bu, ders projesinde çok faydalıdır; çünkü her küçük değişiklik için konteyneri yeniden build etmek zaman kaybettirir. Ancak anında yenileme ve bağlama mount ayarları üretim yaklaşımı değildir; geliştirme kolaylığı için vardır.

Üretim imajı

Üretim imajı, güvenli, küçük ve tahmin edilebilir çalışma çıktısı olmalıdır. Debug kolaylığı için kullanılan araçların üretim imajında kalması gerekmeyebilir. Bu bir eksiklik değil, bilinçli tercihtir. Üretim imajının amacı geliştiricinin rahatça deneme yapması değil, uygulamanın kararlı biçimde çalışmasıdır. Bu yüzden gereksiz dosya, geliştirme bağımlılığı ve gizli bilgiler imaj dışında tutulmalıdır.

Gereksiz geliştirme bağımlılıkları

Geliştirme bağımlılıkları test, statik denetim, anında yenileme veya derleme süreci için gerekli olabilir. Ancak üretim çalışması sırasında bu paketlerin çoğuna ihtiyaç yoktur. Gereksiz bağımlılık imajı büyütür, güvenlik taramasını zorlaştırır ve bakım maliyeti üretir. Bu yüzden hangi bağımlılığın hangi ortamda gerektiği açık olmalıdır.

7.14.1 Küçük imaj ve çok aşamalı oluşturma

Derleme araçları ve kaynak kodun tamamı çoğu zaman yalnızca imaj üretilirken gereklidir; uygulama çalışırken bunlara ihtiyaç kalmaz. Bunları final imajda taşımaya devam etmek gereksiz boyut ve saldırı yüzeyi ekler. Çok aşamalı oluşturma, uygulamayı bir aşamada derleyip yalnızca gerekli çıktıyı daha küçük bir final imajına taşımamızı ve üretim imajını sadeleştirmemizi sağlar.

```
FROM node:20 AS build
WORKDIR /app
COPY . .
RUN npm ci && npm run build

FROM nginx:alpine
COPY --from=build /app/dist /usr/share/nginx/html
```

Bu örnekte build araçları final imaja taşınmaz; yalnızca üretilen statik çıktı taşınır. Böylece imaj daha küçük ve daha tahmin edilebilir hale gelir.

7.15 Docker ile Sık Yapılan Hatalar

Docker kullanırken hataların çoğu aracın karmaşıklığından değil, yanlış varsayımdan doğar. Bir ekip "konteyner çalışıyor, o halde uygulama hazır" diye düşünebilir. Oysa konteyner ayakta olsa bile port yanlış olabilir, veritabanı hazır olmayabilir, gizli bilgi imaja gömülmüş olabilir veya volume eski veriyi taşıyor olabilir.

Küçük bir vaka düşünelim. Ekip Docker Compose ile uygulamayı çalıştırır. Web servisi açılır, fakat login çalışmaz. İlk tahmin kod hatasıdır. Loglara bakınca veritabanının eski volume içindeki eski şemayla çalıştığı görülür. Sorun uygulama kodunda değil, kalıcı veri alanındadır.

Sık karşılaşılan hatalar Tablo 7.8 içinde toplanmıştır.

Tablo 7.8: Docker ile sık yapılan hatalar

Hata	Etkisi	Düzeltme
Gizli değer imaja gömülür	Erişim ve maliyet riski	Değer çalışma anında verilir
İmaj gereksiz büyür	Oluşturma ve dağıtım yavaşlar	`.dockerignore` ve çok aşamalı oluşturma
Bağlantı noktası çakışır	Uygulama dışarıdan açılmaz	Ana makine bağlantı noktasını ve çalışan süreçleri kontrol et
Eski kalıcı veri kalır	Yanlış şema veya kayıt kullanılır	Hangi verinin duracağını belgele
Konteyner sanal makine gibi kullanılır	Bakım karmaşası artar	Tek süreç, tek amaç
`localhost` konteyner içinden kullanılır	Servis birbirini bulamaz	Compose servis adını kullan
Sağlık denetimi yok	"Ayakta" sanılır, henüz hazır değildir	`healthcheck` tanımla; bağımlı servis bunu bekleyerek başlasın
Geliştirme imajı üretime gider	Saldırı yüzeyi büyür	Üretim imajını sade tut

Tablo 7.8 Docker'ın yalnızca teknik araç olmadığını gösterir. Yanlış kullanım ekip zamanını, güvenliği ve teslim güvenilirliğini etkiler.

İmaj içine gizli bilgi koymak

İmaj içine gizli bilgi koymak, gizli değeri imajın parçası haline getirir. Örneğin Dockerfile içinde gerçek API anahtarı yazılırsa, imaj kayıt deposuna gönderildiğinde bu değer de taşınır. Bu durum yetkisiz erişim ve maliyet riski doğurur. Doğru yaklaşım, imajı genel tutmak ve gizli değeri çalışma anında güvenli ortamdan vermektir.

Gereksiz büyük imaj üretmek

Gereksiz büyük imaj, taşınması ve build edilmesi yavaş olan imajdır. Bir çantaya seyahat için gerekli birkaç eşya yerine bütün dolabı koymak gibi düşünebilirsiniz. `node_modules`, test çıktıları, kaynak kontrol klasörleri veya geliştirme araçları gereksiz yere imaja girerse boyut artar. Bu durum CI süresini, kayıt deposu maliyetini ve dağıtım hızını etkiler.

7.15.1 `.dockerignore` kullanmamak

`.dockerignore` kullanmamak, gereksiz veya hassas dosyaların bağlama girmesine neden olabilir. Bu hem build süresini uzatır hem de `.env` gibi dosyaların yanlışlıkla imaja taşınma riskini artırır. Küçük projelerde bile `.dockerignore` dosyası, tekrarlanabilir ve güvenli build alışkanlığının parçası olmalıdır.

7.15.2 Port çakışması

Port çakışması, host üzerinde aynı portu iki servisin kullanmaya çalışmasıdır. Küçük bir kontrol akışı yeterlidir:

1. Hata mesajında hangi portun dolu olduğunu bulun.
2. `docker ps` ile çalışan konteynerleri kontrol edin.
3. Gerekirse eski konteyneri durdurun.
4. Compose dosyasında host portunu değiştirin.
5. README'deki erişim adresini güncelleyin.

Bu akış ekip düzeni için önemlidir; çünkü port değiştiyse herkes aynı yeni adresi bilmelidir.

Volume yüzünden eski verinin kalması

Volume yüzünden eski verinin kalması, özellikle veritabanı şeması değiştiğinde karşımıza çıkar. Kod yeni tablo bekler, fakat volume eski veritabanını taşır. Bu durumda konteyneri silmek yetmeyebilir; volume hâlâ duruyordur. Eski veriyi korumak bazen istenir, bazen temiz başlangıç gerekir. Bu karar bilinçli verilmelidir; aksi halde hata ayıklama yanlış yönde ilerler.

Konteyneri sanal makine gibi kullanmaya çalışmak

Konteyneri sanal makine gibi kullanmaya çalışmak, içine girip elle paket kurmak, dosya düzenlemek ve bu değişikliklerin kalıcı olmasını beklemek anlamına gelir. Bu yaklaşım

Docker'ın tekrarlanabilirlik fikrini zayıflatır. Doğru davranış, gerekli değişikliği Dockerfile veya Compose dosyasına yazmaktır. Böylece ortam kişisel müdahalelerle değil, versiyonlanan tarifle yeniden üretilebilir.

7.16 Atölye

Konteyner anlatısı ancak çalışan bir uygulama serisi ile tam olarak kavranabilir. Bu atölyede imaj çekilir, konteyner çalıştırılır, kapı eşlenir ve kalıcı veri denenir.

Görev 1: Docker motorunu ve imajı doğrula

Komut

```
$ docker version
$ docker pull nginx:alpine
$ docker image ls nginx:alpine
```

Beklenen çıktı

```
Client: Docker Engine ...
Server: Docker Engine ...
nginx   alpine   <kimlik>   ...
```

Görev 2: İlk konteyneri başlat ve listele

Komut

```
$ docker run -d --name atolye-web nginx:alpine
$ docker ps --filter name=atolye-web
```

Beklenen çıktı

```
<konteyner-kimligi>
CONTAINER ID   IMAGE          STATUS        NAMES
<kimlik>       nginx:alpine   Up ...        atolye-web
```

Görev 3: Log, süreç ve durumu incele

Komut

```
$ docker logs --tail 3 atolye-web
$ docker exec atolye-web nginx -v 2>&1
$ docker inspect --format '{{.State.Status}}' atolye-web
```

Beklenen çıktı

```
... start worker process ...
nginx version: nginx/...
running
```

Görev 4: Konteyneri durdur ve kaldır

Komut

```
$ docker stop atolye-web
$ docker rm atolye-web
$ docker ps -a --filter name=atolye-web
```

Beklenen çıktı

```
atolye-web
atolye-web
CONTAINER ID   IMAGE   STATUS   NAMES
```

Görev 5: Adlandırılmış volume ile veriyi koru

Komut

```
$ docker volume create atolye-data
$ docker run --rm -v atolye-data:/data nginx:alpine \
  sh -c "printf 'kalici\n' > /data/not.txt"
$ docker run --rm -v atolye-data:/data nginx:alpine \
  cat /data/not.txt
```

Beklenen çıktı

```
atolye-data
kalici
```

İkinci geçici konteyner, ilk konteynerin volume'a yazdığı veriyi okur.

Görev 6: Kullanıcı tanımlı ağda ad çözümle

Komut

```
$ docker network create atolye-net
$ docker run -d --name atolye-nginx \
  --network atolye-net nginx:alpine
$ docker run --rm --network atolye-net nginx:alpine \
  sh -c 'wget -qO- http://atolye-nginx | head -n 1'
$ docker stop atolye-nginx
$ docker rm atolye-nginx
$ docker network rm atolye-net
```

Beklenen çıktı

```
<ag-kimligi>
<konteyner-kimligi>
<!DOCTYPE html>
atolye-nginx
atolye-nginx
atolye-net
```

Görev 7: Kendi imajını üret ve porta aç

Komut

```
$ mkdir -p ~/gelistirme-atolyesi/bolum-7/web
$ cd ~/gelistirme-atolyesi/bolum-7/web
$ printf 'Atolye calisiyor\n' > index.html
$ printf '%s\n' 'FROM nginx:alpine' \
  'COPY index.html /usr/share/nginx/html/index.html' \
  > Dockerfile
$ docker build -t atolye-web:1.0 .
$ docker run -d --name atolye-app -p 8080:80 atolye-web:1.0
$ curl -fsS http://localhost:8080
```

Beklenen çıktı

```
... naming to docker.io/library/atolye-web:1.0 ...
<konteyner-kimligi>
Atolye calisiyor
```

Görev 8: Compose yapılandırmasını doğrula ve çalıştır

Komut

```
$ printf '%s\n' 'services:' '  web:' \
  '    image: atolye-web:1.0' '    ports:' \
  '      - "8081:80"' > compose.yaml
$ docker compose config --quiet
$ docker compose up -d
$ docker compose ps
$ curl -fsS http://localhost:8081
$ docker compose logs --tail 3 web
$ docker compose down
```

Beklenen çıktı

```
[+] Running ...
NAME    IMAGE            STATUS   PORTS
...     atolye-web:1.0   Up ...   0.0.0.0:8081->80/tcp
Atolye calisiyor
... GET / HTTP/1.1" 200 ...
[+] Removed ...
```

`docker compose config --quiet` geçerli yapılandırmada çıktı üretmez.

Görev 9: Ürettiğin nesneleri denetle ve temizle

Komut

```
$ docker image inspect atolye-web:1.0 --format '{{.RepoTags}}'
$ docker stop atolye-app
$ docker rm atolye-app
$ docker volume rm atolye-data
$ docker image rm atolye-web:1.0
```

Beklenen çıktı

```
[atolye-web:1.0]
atolye-app
atolye-app
atolye-data
Untagged: atolye-web:1.0
```

Bu dokuz görev boyunca aynı döngü tekrar etti: bir imaj edinilir veya üretilir, konteyner olarak çalıştırılır, kapısı ve kalıcı verisi doğrulanır, sonunda hepsi temizlenir. Docker Compose bu adımları tek dosyada topladığında aynı döngü tek komuta indirgenir. Bu atölyeyi elle tekrar edebilen bir okuyucu, artık `docker compose up` çıktısını ezbere değil, ne çalıştığını, hangi kapıdan açıldığını ve verinin nerede durduğunu bilerek okuyacaktır.

Bölüm 8

CI/CD: Sürekli Entegrasyon ve Sürekli Dağıtım

Bir uygulamanın yerelde çalışması önemli bir adımdır; fakat son adım değildir. Kullanıcıların erişebileceği bir ortama çıkmak istediğimizde artık başka sorular devreye girer. Kod hangi kontrolden geçecek? Testler nerede çalışacak? Hangi sürüm yayına alınacak? Gizli değerler nerede tutulacak? Hatalı bir dağıtım olursa nasıl geri dönülecek?

Bu bölümde yayına alma sürecini DevOps[1] uzmanlığı derinliğinde değil, Yönetim Bilişim Sistemleri ilgililerinin okuyabileceği bir operasyonel kontrol problemi olarak ele alacağız. GitHub Actions, VPS, Docker imajı, Coolify ve sürüm geri alma gibi kavramlar burada tek başına teknik araçlar değildir. Kalite, risk, sorumluluk ve iş sürekliliğiyle birlikte düşünülmelidir.

Yerelde çalışan uygulama, kontrollü ve izlenebilir bir akışla yayına çıkmalıdır. Bu akış otomasyon içerir; fakat otomasyon denetimin yerini almaz. Tam tersine, denetimi daha görünür hale getirmelidir.

8.1 CI/CD Nedir?

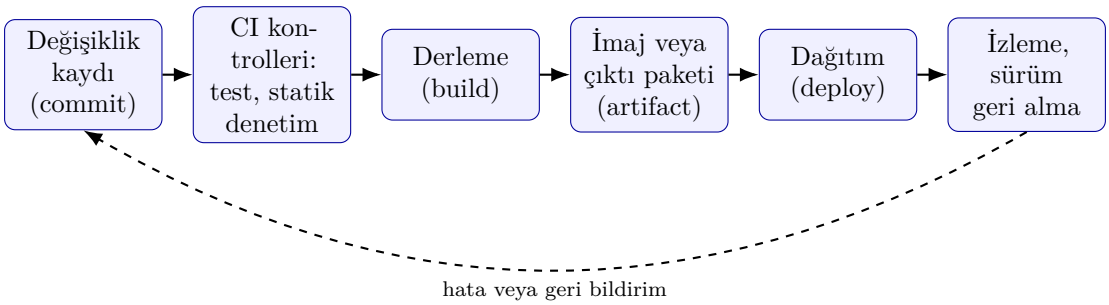
Bir ekipte herkes kendi bilgisayarında kod yazıp en sonunda dosyaları sunucuya elle kopyalıyorsa, süreç başlangıçta basit görünebilir. Ancak ekip büyüdükçe veya teslim yaklaşınca bu yöntem risk üretir. Hangi kod son sürüm? Testler çalıştı mı? Yanlış dosya sunucuya gitti mi? Eski sürüme nasıl dönülecek? Bu soruların yanıtı belirsizse, yayına alma süreci kişisel hafızaya bağlı hale gelir.

Sürekli entegrasyon ve sürekli teslimat/dağıtım (CI/CD), bu belirsizliği azaltmak için geliştirme ve yayına alma adımlarını otomasyonla kontrol altına alan yaklaşımdır. Sürekli entegrasyon (Continuous Integration, CI), kod değişikliklerinin otomatik olarak test edilmesi ve projeye entegre edilebilir olup olmadığının kontrol edilmesidir. Sürekli

[1] Lwakatare, L. E., Kilamo, T., Karvonen, T., Sauvola, T., Heikkilä, V., Itkonen, J., Kuvaja, P., Mikkonen, T., Oivo, M., & Lassenius, C. (2019). DevOps in practice: A multiple case study of five companies. *Information and Software Technology*, *114*, 217–230. https://doi.org/10.1016/j.infsof.2019.06.010

teslimat veya dağıtım (Continuous Delivery/Deployment) ise testlerden geçen çıktının kontrollü biçimde yayına hazırlanması veya otomatik yayına alınmasıdır.

Basit akış Şekil 8.1 içindeki gibi düşünülebilir.



Şekil 8.1: Değişiklik kaydından yayına uzanan CI/CD akışı

Ancak CI/CD yalnızca hızlı yayın yapmak değildir. Gizli bilgi yönetimi, dal koruması, test sonucu, dağıtım günlüğü ve sürüm geri alma planı da bu işin parçasıdır. CI/CD, teknik otomasyondan çok operasyonel güvenilirlik aracıdır.

Manuel yayına alma çoğu zaman şu şekilde ilerler: biri dosyaları derler, sunucuya bağlanır, eski dosyaların üstüne yenilerini kopyalar, servisi yeniden başlatır ve çalışıp çalışmadığına bakar. Bu akışta ekip düzeni kişiye bağlıdır, geri dönebilme belirsizdir, kalite kontrol ise çoğu zaman "bende çalıştı" seviyesinde kalır. Küçük projede bile bu yöntem, hangi adımın ne zaman ve kim tarafından yapıldığını izlemeyi zorlaştırır.

Sürekli entegrasyon, her değişikliğin projeye dahil edilmeden önce otomatik kontrollerden geçmesini sağlar.[2] Değişiklik isteği (pull request) açıldığında testler, statik denetim (lint) ve derleme (build) çalışabilir. Bu, ekibin "bu kod ana dala girebilir mi?" sorusunu kişisel tahmin yerine otomatik sinyallerle değerlendirmesine yardımcı olur. Ancak CI geçmesi tek başına iş kuralının doğru olduğu anlamına gelmez; inceleme ve kabul kriterleri yine gereklidir.

CI/CD'nin geliştirme ortamıyla ilişkisi

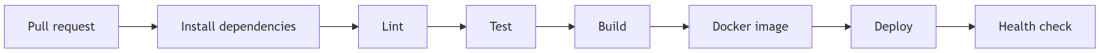
Geliştirme ortamında ne kadar tekrarlanabilirlik kurarsak CI/CD ortamında o kadar az sürpriz yaşarız. Yerelde Node.js 20 kullanıp CI'da Node.js 18 ile derleme almak farklı sonuç üretebilir. Dockerfile, kilit dosyası, çalışma zamanı versiyonu ve test komutları bu yüzden önemlidir. CI/CD, geliştirme ortamındaki disiplini sunucuya taşımazsa otomasyon yalnızca hatayı daha hızlı görünür hale getirir.

[2] Shahin, M., Babar, M. A., & Zhu, L. (2017). Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, *5*, 3909–3943. https://doi.org/10.1109/ACCESS.2017.2685629 Laukkanen, E., Itkonen, J., & Lassenius, C. (2017). Problems, causes and solutions when adopting continuous delivery—A systematic literature review. *Information and Software Technology*, *82*, 55–79. https://doi.org/10.1016/j.infsof.2016.10.001

8.2 İş Hattı (Pipeline) Kavramı

Statik denetim, test ve derleme adımlarını her seferinde elle ve aynı sırayla çalıştırmak, ekip büyüdükçe unutulan veya atlanan bir alışkanlığa dönüşür. İş hattı (pipeline), bir değişikliğin belirli adımlardan geçerek kontrol edilmesi ve gerekiyorsa yayına hazırlanmasıdır. Bu adımlar elle de yapılabilir; fakat iş hattı onları yazılı ve tekrarlanabilir hale getirir. Böylece ekip "hangi kontrollerden geçtik?" sorusuna log ve sonuç üzerinden cevap verebilir.

Şekil 8.2 basit bir iş hattını gösterir.



Şekil 8.2: CI/CD iş hattındaki temel kontrol ve yayın adımları

Her adımın ayrı bir işlevi vardır. Bağımlılık kurulumunda paketler doğrulanır. Statik denetim kod standardını kontrol eder. Test davranışı sınar. Derleme, projenin üretilebilir olduğunu gösterir. Dağıtım ise çıktıyı hedef ortama taşır. Sağlık denetimi ve günlüğü okuma, yayına alınan sistemin gerçekten çalışıp çalışmadığını gösterir.

İş hattının değeri yalnızca otomasyonda değil, izlenebilirliktedir. Hangi adım başarısız olduysa, ekip tartışmaya oradan başlayabilir.

İş hattı, tetikleyici, iş ve adım

İş hattı büyük akıştır. Tetikleyici (trigger), bu akışın ne zaman başlayacağını belirler; örneğin değişiklik isteği açılınca. İş (job), iş hattı içindeki adım grubudur; test işi veya derleme işi gibi. Adım (step) ise iş içindeki tek basamaktır; bağımlılık kurmak veya test komutu çalıştırmak gibi. Bu ayrım, hata çıktığında "iş hattı bozuldu" demek yerine hangi işin ve hangi adımın sorunlu olduğunu görmeyi sağlar.

Çalıştırıcı, çıktı paketi ve önbellek

Çalıştırıcı (runner), iş hattı adımlarını çalıştıran makinedir. Çıktı paketi (artifact), derleme sonucunda saklanan çıktı olabilir; örneğin `dist/` klasörü veya test raporu. Önbellek (cache) ise tekrar eden bağımlılık kurulumlarını hızlandırmak için kullanılır. Bu kavramlar teknik görünür; fakat ekip açısından zaman, maliyet ve izlenebilirlik üretir. Yanlış önbellek eski bağımlılığı taşıyabilir, eksik çıktı paketi ise yayına alınacak çıktının kaybolmasına neden olabilir.

İş hattı çıktısını okuma

İş hattı çıktısını okurken küçük bir sıra izlemek faydalıdır: önce hangi işin başarısız olduğuna bakın, sonra hangi adımın kırıldığını bulun, ardından logdaki ilk anlamlı hatayı okuyun, son olarak bunun kod, ortam, gizli değer veya dış servis problemi olup olmadığını ayırın. Bu disiplin, ekip içinde rastgele deneme yapmak yerine kontrollü hata ayıklama sağlar.

8.3 GitHub Actions'a Giriş

GitHub üzerinde kod tutuyorsanız, belirli olaylar olduğunda otomatik komutlar çalıştırmak isteyebilirsiniz. Değişiklik isteği açılınca test çalışsın, ana dala birleştirilince derleme alınsın, belirli etiket geldiğinde Docker imajı yayınlansın. GitHub Actions bu tür iş akışlarını repository içinde tanımlamayı sağlar.

GitHub Actions'ı bir otomasyon motoru gibi düşünebiliriz. Repository içindeki YAML dosyaları, hangi olayda hangi komutların çalışacağını tarif eder. Bu dosyalar kodla birlikte versiyonlanır. Böylece iş hattı ayarları kişisel bilgisayarda değil, projenin içinde yaşar.

Ancak otomasyon dosyasının repoda olması, gizli değerlerin de repoda olacağı anlamına gelmez. Komutlar ve akış paylaşılır; gizli değerler GitHub Secrets veya hedef platformun gizli bilgi alanında tutulur.

8.3.1 `.github/workflows` klasörü

GitHub Actions iş akışı (workflow) dosyaları genellikle `.github/workflows/` klasöründe bulunur. Örneğin `.github/workflows/ci.yml` dosyası değişiklik isteği kontrollerini tanımlayabilir. Bu klasörün repoda durması ekip düzeni sağlar; çünkü CI davranışı da kod gibi incelenebilir, geçmişi izlenebilir ve gerektiğinde geri alınabilir.

8.3.2 İş akışı dosyası ve YAML

İş akışı dosyası YAML formatındadır. Girintiler önemlidir; yanlış girinti iş hattının hiç çalışmamasına neden olabilir. Küçük bir CI örneği şöyle görünebilir:

```yaml
name: ci

on:
  pull_request:
  push:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 20
      - run: npm ci
      - run: npm test
      - run: npm run build
```

Bu dosya, kod değiştiğinde hangi kontrollerin çalışacağını açıkça gösterir.

Tetikleyici, iş ve adım alanları

Tetikleyici, iş akışının ne zaman başlayacağını söyler. İş (job), çalışacak adım grubudur. Adım (step), bu iş içindeki komuttur veya hazır action kullanımıdır. Bunu okulda yapılan kontrol listesi gibi düşünebilirsiniz: teslim geldiğinde kontrol başlar, önce biçim kontrolü yapılır, sonra içerik incelenir, sonra sonuç kaydedilir. GitHub Actions'ta da benzer bir sıra vardır.

8.3.3 Hazır action kullanmak ve komut çalıştırmak

Hazır action, sık yapılan bir işi paketlenmiş adım olarak kullanmamızı sağlar. Örneğin `actions/checkout` kodu runner'a indirir, `actions/setup-node` Node.js ortamını hazırlar. Bunun yanında doğrudan komut da çalıştırabiliriz:

```
- run: npm test
```

Bu satır basit görünür; fakat iş hattının kalite kontrol noktasını oluşturur. Test komutu başarısız olursa PR'ın birleştirilmemesi sağlanabilir.

8.4 Değişiklik İsteği Üzerinde CI Çalıştırmak

Değişiklik isteği üzerinde CI çalıştırmak, ana dala girmeden önce değişikliği kontrol etmek demektir. Bu yaklaşım özellikle ekip çalışmalarında önemlidir. Çünkü ana dal herkesin dayandığı ortak haktır; buraya kırık kod girmesi ekibin tamamını etkiler.

PR CI akışı şöyle kurulabilir:

1. Geliştirici özellik dalı üzerinde değişiklik yapar.
2. PR açılır.
3. CI otomatik olarak test, statik denetim ve derleme çalıştırır.
4. Sonuç PR üzerinde görünür.
5. Başarılı sonuç ve inceleme sonrası birleştirme yapılır.

Bu akış, kalite kontrolü kişisel dikkat düzeyinden çıkarıp ekip standardına dönüştürür.

PR açıldığında iş akışını tetiklemek

PR açıldığında iş akışını tetiklemek için YAML içinde `pull_request` olayı tanımlanır. Böylece değişiklik ana dala girmeden önce kontroller çalışır. Ekip açısından bu, geri dönebilme ve kalite kontrol sağlar; çünkü sorun birleştirme sonrasında değil, PR aşamasında görünür olur.

Dal bazlı kontrol

Dal bazlı kontrol, hangi dallarda hangi iş hattının çalışacağını belirler. Örneğin her PR için test çalışabilir, fakat üretim dağıtımı yalnızca ana dala birleştirme sonrasında tetiklenebilir. Bu ayrım önemlidir; çünkü her dalın sunucuya yayın yapması operasyonel risk üretir.

8.4.1 Test, statik denetim ve derleme komutları

Test, statik denetim ve derleme komutları farklı sorulara cevap verir. Test davranışı kontrol eder, statik denetim kod standardını denetler, derleme ise projenin üretilebilir olduğunu gösterir.

```
$ npm run lint
[OK] no lint errors
$ npm test
[OK] 42 tests passed
$ npm run build
[OK] build completed
```

Bu çıktı bize üç ayrı sinyal verir. Biri geçip diğeri kalabilir. Örneğin testler geçerken derleme başarısızsa, yayınlanabilir çıktı üretilemiyor demektir.

8.4.2 PR birleştirme iznini CI sonucuna bağlamak

PR birleştirme iznini CI sonucuna bağlamak, başarısız kontrol varken ana dala geçişi engeller. GitHub arayüzünde bu çoğu zaman zorunlu kontroller ayarıyla yapılır. PR üzerinde şöyle bir durum görülebilir:

```
Checks
[OK] lint
[OK] unit tests
[X] build
```

Bu durumda birleştirmek, bilinen bir hatayı ana dala taşımaktır. Yönetim Bilişim Sistemleri açısından bu karar kalite ve iş sürekliliği riskidir.

8.5 Derleme İş Hattı

Çıktı yalnızca bir geliştiricinin bilgisayarında üretilebiliyorsa, o kişi izinliyken veya bilgisayarı kapalıyken yayın da fiilen durur. Derleme iş hattı, koddan çalıştırılabilir veya yayınlanabilir çıktı üretme sürecidir. Web uygulamasında bu çıktı `dist/` klasörü, backend için Docker imajı olabilir. Sürecin kişisel bilgisayara bağlı kalmaması gerekir.

Şekil 8.3 bağımlılıktan çıktıya giden yolu gösterir.

Şekil 8.3: Derleme iş hattında bağımlılıktan çıktı paketine ilerleyiş

Bu akışta her adım ekip için bir kontrol noktasıdır. Çalışma zamanı doğru mu, bağımlılıklar kuruluyor mu, derleme komutu geçiyor mu, çıktı saklanıyor mu? Bu soruların yanıtı iş hattı loglarında görünmelidir.

Bağımlılıkları kurmak

Bağımlılıkları kurmak, projenin ihtiyaç duyduğu paketleri CI ortamına yüklemektir.

`npm ci` gibi komutlar kilit dosyasına bağlı kalarak daha tekrarlanabilir kurulum sağlar. Bu adım başarısızsa sorun çoğu zaman kodun iş mantığında değil, paket sürümlerinde, kayıt deposu erişiminde veya çalışma zamanı uyumsuzluğundadır.

8.5.1 Derleme komutunu çalıştırmak

Derleme komutu, projenin yayınlanabilir çıktıya dönüşüp dönüşmediğini gösterir.

```
$ npm run build
[OK] built in 4.2s
```

Bu çıktı olumlu bir sinyaldir; fakat tek başına uygulamanın iş kurallarını doğru çalıştırdığını göstermez. Derlemenin geçmesi, yayın için gerekli ama yeterli olmayan bir koşuldur.

Derleme çıktısını saklamak

Derleme çıktısını saklamak, sonraki adımın aynı çıktıyı kullanmasını sağlar. Örneğin frontend derlemesi sonucunda oluşan `dist/` klasörü çıktı paketi olarak saklanabilir. Böylece test edilen ve derlenen çıktı ile dağıtılan çıktı arasındaki ilişki güçlenir. Çıktı saklanmıyorsa her aşamada yeniden üretim yapılır ve fark oluşma ihtimali artar.

Çalışma zamanı sürümünü sabitlemek

Çalışma zamanı sürümünü sabitlemek, CI ortamının hangi Node.js, Python veya Java sürümüyle çalışacağını açık yazmaktır. Yerelde Node.js 20 ile çalışan proje CI'da Node.js 18 ile hata verebilir. Bu, lokantada aynı tarifi farklı fırın sıcaklığıyla denemeye benzer. Tarif aynı olsa bile sonuç değişebilir. Bu yüzden iş akışı içinde çalışma zamanı sürümü açıkça belirtilmelidir.

8.6 Test İş Hattı

Test iş hattı, değişikliğin beklenen davranışı bozup bozmadığını otomatik olarak kontrol eder. Bu süreç yalnızca geliştiriciyi korumaz; ekibi ve kullanıcıyı da korur. Çünkü testten geçmeyen kodun ana dala girmesi, ileride daha pahalı hatalara neden olabilir.

Basit test iş hattı akışı:

```
Kurulum -> Birim testleri -> Entegrasyon testleri -> E2E duman testi -> Rapor
```

Her projede bütün test türleri aynı kapsamda olmak zorunda değildir. Küçük bir ders projesinde birkaç birim testi ve duman kontrolü yeterli olabilir. Ancak hangi testin neyi güvence altına aldığı yazılı olmalıdır.

8.6.1 Testleri otomatik çalıştırmak

Testleri otomatik çalıştırmak, "test etmeyi unuttuk" riskini azaltır. İş hattı içinde şu satır yeterli bir başlangıç olabilir:

```
- run: npm test
```

Bu satır her PR'da çalışıyorsa, testler ekip alışkanlığına değil süreç kuralına bağlanmış olur. Ancak testlerin anlamlı olması yine geliştiricinin sorumluluğundadır.

8.6.2 Test çıktısını okumak

Test çıktısını okumak, yalnızca "kırmızı mı yeşil mi?" diye bakmak değildir.

```
FAIL tests/stock.test.ts
  expected "critical" but received "normal"
```

Hata mesajı, testin hangi iş kuralında başarısız olduğunu gösterir. Sorun test ortamında mı, iş kuralında mı, beklenen değer yanlış mı? İş hattı çıktısı tartışmayı bu sorulara taşır.

E2E test senaryolarını iş hattına bağlamak

E2E testleri kullanıcı akışını baştan sona dener. İş hattına bağlanacaksa önce uygulama başlatılır, test verisi hazırlanır, kritik kullanıcı akışı çalıştırılır, sonuç raporlanır ve gerekirse ekran görüntüsü çıktı paketi olarak saklanır. Bu akış ekip için değerlidir; çünkü yalnızca fonksiyonların değil, kullanıcının gerçekten yaptığı yolculuğun da kırılmadığını gösterir.

8.6.3 Başarısız testte birleştirmeyi durdurmak

Başarısız test varken birleştirmeyi durdurmak, ekibin fark ettiği bir sorunu bile bile ana dala geçirmemesi demektir.

```
Required check "test" failed.
Merge blocked.
```

Çıktıdaki `Merge blocked` satırı teknik bir kuraldan fazlasını anlatır: ekip, kalite kapısını otomatik hale getirmiştir. Elbette bazı durumlarda testin kendisi hatalı olabilir; o zaman test düzeltilir, kapı kaldırılmaz.

8.7 Docker imajı oluşturmak ve yayımlamak

Docker imajı oluşturmak ve yayımlamak, uygulamayı çalıştırılabilir bir paket haline getirip kayıt deposuna (registry) göndermek anlamına gelir. Bu yaklaşım özellikle aynı imajın CI, hazırlama veya üretim ortamlarında kullanılmasını kolaylaştırır. Hazırlama ortamı (staging), test ile üretim arasında yer alan ve üretim verisi taşımadan gerçek dağıtımın bir provasını yapmaya yarayan ortamdır. Ancak imaj üretimi de kontrol gerektirir: etiket doğru mu, gizli bilgi imaja girdi mi, derleme gerçekten geçti mi?

Basit akış:

1. Dockerfile üzerinden imaj oluşturulur.
2. İmaj uygun etiketle işaretlenir.

3. Kayıt deposuna giriş yapılır.
4. İmaj kayıt deposuna gönderilir (push).
5. Dağıtım sistemi bu imajı kullanır.

Bu adımların her biri izlenebilir olmalıdır. Aksi halde hangi kodun hangi imaj olarak yayına çıktığı belirsizleşir.

8.7.1 CI içinde Docker build

CI içinde Docker build almak, imajın yalnızca geliştiricinin bilgisayarında değil, temiz bir otomasyon ortamında da üretilebildiğini gösterir.

```
$ docker build -t ghcr.io/example/stock-api:sha-abc123 .
[OK] image built
```

Çıktıdaki `[OK] image built` satırı, Dockerfile'ın CI ortamında da çalıştığını gösterir. Eğer burada hata varsa, sunucuya geçmeden önce yakalanır.

8.7.2 İmaj Etiketi Stratejisi

İmaj etiketi stratejisi, hangi imajın hangi sürüme karşılık geldiğini belirler. Tablo 8.1 üç yaygın etiketleme yaklaşımını karşılaştırır.

Tablo 8.1: İmaj Etiketi Stratejisi

Tag	Anlamı	Dikkat
`latest`	Son imajı işaret eder	Sürüm geri alma için belirsiz olabilir
Git SHA	Belirli commit'e bağlar	İzlenebilirlik güçlüdür
Semantik sürüm	`1.2.0` gibi sürüm bilgisi verir	Sürümleme disiplini gerekir

Ders projelerinde Git SHA veya kısa commit etiketi, hangi kodun yayına çıktığını izlemek için oldukça kullanışlıdır.

8.7.3 Kayıt Deposuna Göndermek

Kayıt deposuna göndermek (push), üretilen imajı dağıtım sisteminin erişebileceği yere taşımaktır.

```
$ docker push ghcr.io/example/stock-api:sha-abc123
[OK] pushed
```

Bu çıktı, imajın yerel CI ortamından çıkıp paylaşılan kayıt deposuna taşındığını gösterir. Kayıt deposu erişimi için gereken token gizli bilgi olarak tutulmalı, loglarda görünmemelidir.

8.7.4 İmaj Oluşturma Hatalarını Okumak

İmaj oluşturma hataları çoğu zaman yanlış bağlam, eksik dosya veya paket sorunu yüzünden çıkar; Tablo 8.2 bu hataları dört örnek üzerinden sınıflandırır.

Tablo 8.2: İmaj Oluşturma Hatalarını Okumak

Bulgu	Risk	Aksiyon
Dockerfile dosya bulamıyor	İmaj üretilemez	Build context ve `COPY` yolları kontrol edilir
Paket kurulumu başarısız	CI durur	Kilit dosyası ve kayıt deposu erişimi incelenir
Gizli değer gerekiyor	Derleme gizli bilgiye bağımlı hale gelir	Gizli değerin derleme anında mı çalışma zamanında mı gerektiği ayrılır
Hedef mimari uyuşmuyor	Sunucuda konteyner başlamaz	`--platform` bayrağıyla hedef mimari belirtilir

Bu hataları doğru okumak, dağıtım aşamasındaki daha pahalı kesintileri azaltır.

8.8 Gizli Bilgi ve Ortam Değişkenleri

Yayına alma sürecinde bazı bilgiler herkese açık olmamalıdır. Kayıt deposu (registry) token'ı, veritabanı parolası, SSH anahtarı, API anahtarı ve üretim bağlantı adresleri yanlış yerde tutulursa sistem riske girer. Bu bilgiler bazen "sadece iş hattı çalışsın diye" hızlıca YAML dosyasına yazılmak istenir. İşte risk burada başlar.

Gizli bilgi (secret), erişim yetkisi veren değerdir. Ortam değişkeni (environment variable) ise uygulamanın çalışma sırasında dışarıdan aldığı ayardır. Her ortam değişkeni gizli bilgi değildir; fakat gizli bilgiler çoğu zaman ortam değişkeni olarak verilir (Tablo 8.3).

Tablo 8.3: Gizli değer ve ortam değişkeni örnekleri

Değer	Hassas mı?	Nerede tutulmalı?
`NODE_ENV=production`	Hayır	İş akışı veya platform ayarı
`APP_PORT=3000`	Hayır	Config olarak
`DATABASE_PASSWORD`	Evet	Gizli bilgi yönetimi alanında
`REGISTRY_TOKEN`	Evet	GitHub Actions'ın gizli değer alanı veya platformun gizli bilgi yönetimi
`SSH_PRIVATE_KEY`	Evet	Sıkı erişimli gizli bilgi alanında

Buradaki hedef, gizli bilgiyi otomasyona vermek ama repoda, logda veya imaj içinde tutmamaktır.

Gizli bilgi nedir?

Gizli bilgi, ele geçirildiğinde sisteme erişim sağlayabilecek değerdir. Örneğin bir kayıt deposu token'ı, saldırganın imaj çekmesine veya bazen imaj göndermesine izin verebilir. Bu yüzden gizli bilgi sıradan bir ayar gibi davranılamaz. Bir geliştirici token'ı YAML dosyasına yazıp commit ederse, teknik kolaylık güvenlik sorununa dönüşür.

GitHub Actions'ın gizli değer alanı

GitHub Actions'ın gizli değer alanı, iş akışı (workflow) içinde kullanılacak gizli değerleri repository dışında saklamaya yarar. Örneğin kayıt deposu token'ı `secrets.REGISTRY_TOKEN` olarak okunabilir. Ancak bu alanda durması tek başına yeterli değildir; iş akışı içinde bu değeri loga yazdırmamak, gereksiz işlere (job) vermemek ve yetkisini sınırlamak gerekir.

Ortama göre gizli bilgi yönetimi

Geliştirme, hazırlama ve üretim ortamlarının gizli değerleri farklı olmalıdır. Aynı veritabanı parolasını her ortamda kullanmak bir ortamdan diğerine risk taşır. Ortam bazlı gizli bilgi yönetimi, hangi iş hattının hangi gizli değere erişeceğini ayırır. Böylece PR kontrolü üretim ortamının gizli değerlerine gereksiz erişim kazanmaz.

Gizli bilgiyi log'a yazdırmamak

Gizli bilgiyi log'a yazdırmak, onu gizli alandan çıkarıp daha geniş bir izleme alanına taşır. Örneğin hata ayıklamak için `echo $DATABASE_URL` yazmak masum görünebilir; fakat loglar ekip üyeleri veya dış servisler tarafından görülebilir. Doğru yaklaşım, gizli bilginin varlığını kontrol etmek ama değerini yazdırmamaktır.

Üretim ortamı gizli değerlerinin repo dışında tutulması

Üretim ortamı gizli değerleri repoda tutulmamalıdır. `.env.production` dosyasını repoya eklemek kısa vadede dağıtımı kolaylaştırır; fakat uzun vadede sızıntı riski doğurur. Üretim değerleri GitHub Secrets, Coolify environment alanı veya kullanılan platformun secret yönetiminde tutulmalıdır. Repo, gizli bilginin kendisini değil, hangi gizli değerlere ihtiyaç duyulduğunu açıklayan örnek dosyayı taşımalıdır.

8.9 VPS Kavramı

Bir uygulamayı yayına almak için bir yere ihtiyaç duyarız. Bu yer bazen yönetilen bir platform, bazen paylaşımlı hosting, bazen de kendi yönettiğimiz bir sunucu olabilir. VPS, bu seçeneklerden biridir ve ders projelerinde sık karşılaşılır.

Sanal özel sunucu (Virtual Private Server, VPS), fiziksel bir sunucunun sanallaştırılmış ve kullanıcıya ayrılmış parçasıdır. Size belirli CPU, RAM, disk ve ağ kaynağı verilir. Üzerine Docker, Coolify, veritabanı veya başka servisler kurabilirsiniz. Bu esneklik güçlüdür; fakat sorumluluk da getirir.

VPS kullanmak, yalnızca "uygulamayı internete açmak" değildir. SSH erişimi, firewall, alan adı (domain), DNS, yedekleme, güncelleme ve güvenlik gibi konular da devreye girer. Yönetim Bilişim Sistemleri açısından VPS, teknik özgürlük ile operasyonel sorumluluk arasındaki dengeyi gösterir.

VPS nedir?

VPS'i apartmandaki bağımsız daire gibi düşünebiliriz. Bina fiziksel sunucudur, daire ise size ayrılan sanal alandır. Kendi dairenizde ne kuracağınızı büyük ölçüde siz belirlersiniz. Ancak kapıyı açık bırakırsanız sorumluluk da sizdedir. VPS, uygulama yayınlamak için esnek bir alan sunar; fakat yönetim ve güvenlik bilgisi gerektirir.

8.9.1 Paylaşımlı hosting, VPS ve bulut sunucu

Paylaşımlı hosting daha sınırlı ve hazır bir ortam sunar. VPS daha fazla kontrol verir. Bulut sunucu (cloud server) ise sağlayıcıya göre daha geniş ölçekleme ve yönetim seçenekleri sunabilir. Tablo 8.4 bu üç seçeneği güçlü yön ve sınır açısından karşılaştırır.

Tablo 8.4: Paylaşımlı hosting, VPS ve bulut sunucu farkı

Seçenek	Güçlü taraf	Sınır
Paylaşımlı hosting	Kolay başlangıç	Kontrol sınırlı
VPS	Esneklik ve maliyet dengesi	Yönetim sorumluluğu kullanıcıda
Cloud server	Ölçek ve servis ekosistemi	Maliyet ve karmaşıklık artabilir

Hangi seçeneğin doğru olduğu proje ihtiyacına ve ekip kapasitesine bağlıdır.

VPS seçerken temel kriterler

VPS seçerken CPU, RAM, disk, lokasyon, trafik limiti, yedekleme, fiyat ve sağlayıcı güvenilirliği birlikte değerlendirilmelidir. Küçük bir ders projesi için çok güçlü sunucu gerekmeyebilir; fakat veritabanı, Docker ve dağıtım paneli aynı VPS üzerinde çalışacaksa RAM yetersiz kalabilir. Kapasite, fiyat sıralamasına göre değil projenin gerçek ihtiyacına göre belirlenmelidir.

Domain, DNS, firewall ve SSH ilişkisi

Alan adı kullanıcıların yazdığı addır, DNS bu adı sunucu IP'sine yönlendirir, firewall hangi kapıların açık olduğunu belirler, SSH ise sunucuya güvenli yönetim erişimi sağlar. Bunu bir iş yerine benzetebiliriz: tabela alan adı, adres defteri DNS, güvenlik kapısı firewall, yönetici anahtarı SSH gibidir. Bu parçalar doğru kurulmazsa uygulama çalışsa bile erişim veya güvenlik sorunu yaşanır.

8.9.2 Sunucuda Docker çalıştırmak

Sunucuda Docker çalıştırmak, uygulamayı sunucuya kurulu bağımlılıklardan daha bağımsız hale getirebilir.

```
$ docker ps
CONTAINER ID   IMAGE              STATUS
abc123         stock-api:latest   Up 2 minutes
```

Çıktıdaki `Up 2 minutes` durumu, konteynerin sunucuda çalıştığını gösterir. Ancak konteynerin ayakta olması uygulamanın dış dünyadan erişilebilir olduğu anlamına gelmez; port, alan adı, firewall ve loglar ayrıca kontrol edilmelidir.

8.10 Coolify ile dağıtım

VPS üzerinde uygulama yayınlamak SSH ile bağlanıp komutları elle sırayla yazmaktan ibaret kalırsa, hangi adımın ne zaman ve nasıl çalıştığı yalnızca o komutları yazan kişinin hafızasında kalır. Coolify, kendi VPS'iniz üzerinde uygulama yayınlamayı kolaylaştıran kendi sunucusunda çalışan bir dağıtım platformu olarak düşünülebilir. GitHub repository bağlama, Dockerfile veya Compose üzerinden uygulama çalıştırma, ortam değişkeni tanımlama, alan adı ve SSL ayarı yapma gibi işleri daha görünür hale getirir.

Coolify gibi araçların değeri, sunucuda her şeyi elle komutlarla yönetme ihtiyacını azaltmasıdır. Ancak bu, operasyonel sorumluluğun bittiği anlamına gelmez. Yanlış ortam değişkeni, eksik gizli bilgi, hatalı Dockerfile veya bozuk derleme (build) yine dağıtım sorununa yol açabilir.

Şekil 8.4 kaynak koddan çalışan servise giden yolu gösterir.

Şekil 8.4: Coolify ile kaynak koddan çalışan servise yayın akışı

Bu akışta her adımın bir kontrol karşılığı vardır. Repo doğru mu, derleme geçti mi, konteyner ayakta mı, alan adı çalışıyor mu, loglar temiz mi?

Coolify nedir?

Coolify'ı kendi sunucunuzdaki küçük bir dağıtım paneli gibi düşünebilirsiniz. Uygulama kaynağını bağlar, derleme ve dağıtım sürecini yönetir, ortam değişkeni değerlerini tanımlar, alan adı ve SSL ayarlarını kolaylaştırır. Pratik sınır şudur: Coolify süreci görünür hale getirir, fakat kötü Dockerfile'ı, yanlış gizli değeri veya hatalı uygulama kodunu kendiliğinden doğru yapmaz.

VPS üzerinde Coolify çalıştırmak

VPS üzerinde Coolify kullanmak, sunucuyu yalnızca ham bir Linux makinesi olarak değil, uygulamaları yönetebileceğiniz bir platform gibi kullanmak anlamına gelir. Bu yaklaşım ders projelerinde faydalı olabilir; çünkü dağıtım adımlarını daha görsel ve izlenebilir hale getirir. Ancak VPS kaynakları sınırlıysa Coolify, uygulama ve veritabanı aynı makinede yarışabilir. Kaynak planı yapılmalıdır.

8.10.1 GitHub repository bağlamak

GitHub repository bağlamak, Coolify'ın hangi kodu derleyeceğini bilmesini sağlar. Bu adımda doğru repo, doğru dal ve doğru derleme yöntemi seçilmelidir.

```
Repository: example/stock-app
Branch: main
Build: Dockerfile
```

Bu küçük bilgi seti dağıtımın kaynağını belirler. Yanlış dal seçilirse test edilmemiş kod yayına çıkabilir.

Dockerfile veya Docker Compose ile dağıtmak

Tek servisli uygulamalarda Dockerfile yeterli olabilir. Web uygulaması, veritabanı ve Redis gibi birden fazla servis gerekiyorsa Docker Compose daha anlamlı hale gelir. Seçim şu soruya bağlıdır: Yayına alınacak şey tek konteyner mi, yoksa ilişkili servislerden oluşan bir sistem mi? Coolify bu tarifleri çalıştırabilir; fakat tariflerin doğru olması yine proje ekibinin sorumluluğundadır.

Ortam değişkeni, alan adı ve SSL ayarları

Ortam değişkeni uygulamanın ayarlarını, alan adı kullanıcıların erişeceği adresi, SSL ise güvenli HTTPS bağlantısını belirler. Bu üçlü doğru kurulmadığında uygulama çalışsa bile kullanıcı güvenli erişim sağlayamayabilir. Gizli değerler Coolify'ın ortam değişkeni veya gizli bilgi alanlarında tutulmalı, repo içinde yer almamalıdır. Alan adı ve SSL ayarı ise kullanıcı güveni açısından yalnızca teknik değil, operasyonel bir gerekliliktir.

8.10.2 Log okumak ve dağıtım hatalarını çözmek

Dağıtım başarısız olduğunda ilk yapılacak iş log okumaktır. Bir ekip uygulamanın açılmadığını görür ve sunucuyu yeniden başlatır. Oysa logda eksik ortam değişkeni açıkça yazıyor olabilir.

Üç yaygın belirti farklı yerlere işaret eder. Derleme geçtiği halde uygulama açılmıyorsa neden genellikle eksik bir çalışma zamanı ayarıdır ve ilk bakılacak yer uygulama loglarıdır. Alan adı çalışmıyorsa sorun çoğunlukla DNS veya SSL yapılandırmasıdır; alan adı yönlendirmesi ve sertifika durumu kontrol edilmelidir. Veritabanı bağlantısı kurulamıyorsa neden genellikle eksik bir gizli değer veya kapalı bir ağ erişimidir; bu durumda `DATABASE_URL` değeri ve servis erişimi denetlenir.

Log okumadan yapılan müdahale, sorunu çözmek yerine büyütebilir.

8.11 GitHub Actions ve Coolify Birlikte Kullanım Senaryosu

GitHub Actions ve Coolify birlikte düşünüldüğünde iyi bir görev ayrımı kurulabilir. GitHub Actions kodun test, statik denetim, derleme ve imaj üretim kontrollerini yapar. Coolify ise hedef VPS üzerinde uygulamayı çalıştırır ve alan adı/SSL gibi dağıtım tarafını yönetir.

Bir senaryo şöyle olabilir: PR açılınca CI çalışır. CI geçmeden PR birleştirilemez. Ana dala birleştirilince dağıtım tetiklenir. Docker imajı oluşturulur veya Coolify repo üzerinden derleme alır. Uygulama yeniden dağıtılır. Sonra log ve erişim kontrol edilir.

Bu akışın amacı her şeyi otomatikleştirmek değil, her adımı görünür hale getirmektir. Otomasyon ne kadar artarsa, kontrol noktalarının da o kadar açık olması gerekir.

PR açılınca CI çalışır

PR açılınca CI çalışması, değişikliğin daha üretime yaklaşmadan kontrol edilmesini sağlar. Test, statik denetim ve derleme sonucu PR üzerinde görünür. Bu aşamada Coolify'a dağıtım yapmak gerekmez; önce kodun ana dala girmeye uygun olup olmadığı anlaşılır. Bu ayrım kalite kapısını erken konuma yerleştirir.

Birleştirme sonrası otomatik dağıtım

Ana dala birleştirme sonrasında dağıtım tetikleniyorsa süreç şöyle okunabilir: PR incelemesi tamamlanır, CI başarılı olur, birleştirme yapılır, main güncellenir, dağıtım sistemi yeni sürümü alır. Bu sıra ekip düzeni açısından önemlidir; çünkü üretime giden yol ana dal üzerinden izlenebilir hale gelir.

Docker imaj oluşturulur

Docker imajı oluşturulması, dağıtım için sabit bir çalışma paketi üretir.

```
$ docker build -t ghcr.io/example/stock-app:sha-abc123 .
```

Bu etiket, imajı belirli commit'e bağlar. Böylece sorun çıktığında hangi kodun hangi imaj olarak yayına alındığı daha kolay bulunur.

Coolify uygulamayı yeniden dağıtır

Coolify uygulamayı yeniden dağıttığında eski konteyner durdurulabilir, yeni derleme çıktısı veya imaj üzerinden yeni konteyner başlatılabilir. Burada kritik olan dağıtım sonucunun kontrol edilmesidir. "Dağıtım tetiklendi" ile "uygulama sağlıklı çalışıyor" aynı şey değildir. Log, health check ve kullanıcı akışı ayrı ayrı gözden geçirilmelidir.

Dağıtım sonucu kontrol edilir

Dağıtım sonucu kontrol edilirken küçük bir sıra izlenebilir: derleme sonucu geçildi mi, konteyner ayakta mı, alan adı HTTPS ile açılıyor mu, temel kullanıcı akışı çalışıyor mu, loglarda tekrar eden hata var mı? Bu kontrol ekip düzeni için gereklidir; çünkü dağıtım tamamlandı mesajı her zaman işlevsel başarı anlamına gelmez.

8.12 Dal Koruması ve Zorunlu Kontroller

Ana dal, projenin ortak dayanağıdır. Herkesin doğrudan ana dala push edebildiği bir projede kalite kapısı kişisel dikkate kalır. Bu küçük ekiplerde bile sorun çıkarabilir. Bir kişi yanlışlıkla kırık kod gönderdiğinde herkes etkilenir.

Dal koruması, ana dal gibi kritik dalları korumak için kullanılan kurallardır. Zorunlu kontroller ise belirli CI kontrolleri geçmeden birleştirme yapılmasını engeller. Bu ayarlar, ekip çalışma sözleşmesini teknik kurala dönüştürür.

Bu yaklaşım güveni kişilere karşı değil, sürece karşı kurar. İyi ekiplerde bile hata olur; dal koruması bu hatanın ana dala kolayca taşınmasını zorlaştırır.

Ana dalı korumak

Ana dalı korumak, üretime veya ortak geliştirme hattına giden yolu kontrol etmek demektir. Doğrudan push yerine PR zorunlu tutulabilir. Bu, tek kişinin hızlı kararını yavaşlatıyor gibi görünse de ekip güvenilirliğini artırır. Yönetim Bilişim Sistemleri açısından main, yalnızca Git dalı değil, ortak iş sürekliliği hattıdır.

PR ve inceleme zorunluluğu

PR ve inceleme zorunluluğu, değişikliğin başka bir göz tarafından okunmasını sağlar. Bu, yalnızca hata bulmak için değil, kararın ve etkisinin ekipçe anlaşılması için önemlidir. YZ ile üretilen kodlarda inceleme daha da değerlidir; çünkü akıcı görünen kodun proje bağlamına uyup uymadığı insan tarafından denetlenmelidir.

8.12.1 CI başarılı olmadan birleştirmemek

CI başarılı olmadan birleştirmemek, otomatik kalite sinyalini zorunlu hale getirir.

```
Required checks:
[OK] lint
[OK] test
[OK] build
```

Bu liste geçmeden birleştirme yapılamıyorsa, ekip kırık kodu ana dala taşıma riskini azaltır. Elbette testlerin kendisi de düzenli bakım ister; sahte güven üretmemelidir.

8.12.2 Üretim dağıtımını kontrol altına almak

Üretim dağıtımı kontrolsüzse, bir birleştirme doğrudan kullanıcıları etkileyebilir. Bir proje ekibi ana dala deneysel bir değişikliği birleştirir ve uygulama yayında açılmaz. Sorun teknik görünür; fakat asıl risk kontrolsüz yayın sürecidir.

Üç eksiklik birbirini besler. Her birleştirme doğrudan üretime çıkıyorsa hatalı bir değişiklik kullanıcıyı hemen etkiler; çözüm manuel onay veya hazırlama ortamı eklemektir. CI zorunlu değilse test edilmemiş kod yayına çıkabilir; bu durumda zorunlu kontroller tanımlanmalıdır. Sürüm geri alma planı yoksa kesinti uzar; önceki imaj etiketi ve geri dönüş adımları önceden yazılmalıdır. Bu üç eksiklik birlikte ele alınmazsa, dağıtımın “tamamlandı” demesi ile “güvenli” olması arasındaki fark kaybolur.

8.13 Sürekli Teslimat ve Sürekli Dağıtım

Sürekli teslimat ve sürekli dağıtım ayrımı, otomasyonun nerede duracağını belirler.[3] İki yaklaşım da iş hattı ve test disiplininden faydalanır. Fark, üretime geçişte insan onayının olup olmadığıdır.

Tablo 8.5: Sürekli Teslimat ve Sürekli Dağıtım

Ölçüt	Sürekli teslimat	Sürekli dağıtım
Üretim geçişi	İnsan onayıyla	Otomatik
Risk kontrolü	Daha görünür manuel kapı	Test ve izlemeye daha çok güvenir
Ders projeleri	Genellikle daha güvenli	Fazla otomasyon riski taşıyabilir
Operasyon olgunluğu	Orta düzey yeterli olabilir	Güçlü test, izleme ve sürüm geri alma ister

Tablo 8.5 birini iyi diğerini kötü ilan etmez. Sadece hangi durumda hangi kontrol seviyesinin anlamlı olduğunu gösterir. Yönetim Bilişim Sistemleri açısından otomasyonun amacı insanı süreçten silmek değil, insanın doğru yerde karar vermesini sağlamaktır.

Manuel onaylı yayın

Manuel onaylı yayın, iş hattının çıktıyı hazırladığı ama üretime geçiş için insan onayı beklediği yaklaşımdır. Ders projelerinde bu çoğu zaman güvenlidir; ekip son kez loglara, testlere ve değişiklik özetine bakabilir. Bu onay bürokrasi değil, iş riski kontrolüdür.

Tam otomatik yayın

Tam otomatik yayın, kontroller geçerse değişikliğin üretime kendiliğinden çıkmasıdır. Bu hızlıdır; fakat güçlü test, izleme ve sürüm geri alma düzeneği gerektirir. Aksi halde küçük bir hata doğrudan kullanıcıya ulaşır. Bu yüzden tam otomasyon, süreç olgunluğu olmadan hedeflenmemelidir.

Ders için güvenli öneri

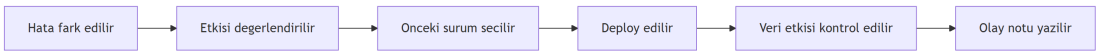
Ders projeleri için güvenli öneri, CI kontrollerini otomatik çalıştırmak ama üretim dağıtımı için bilinçli bir onay veya en azından görünür kontrol noktası bırakmaktır. Böylece okuyucu otomasyonu öğrenir, fakat kontrol sorumluluğunu tamamen süreç dışına atmaz. Bu denge öğrenme açısından da daha sağlıklıdır.

[3] Rodriguez, P., Haghighatkhah, A., Lwakatare, L. E., Teppola, S., Suomalainen, T., Eskeli, J., Karvonen, T., Kuvaja, P., Verner, J. M., & Oivo, M. (2017). Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, *123*, 263–291. https://doi.org/10.1016/j.jss.2015.12.015

8.14 Sürüm geri alma

Yayına alma sürecinde çoğu zaman "nasıl çıkarız?" sorusu sorulur; fakat "hata olursa nasıl döneriz?" sorusu ihmal edilir. Oysa operasyonel güvenilirlik için ikinci soru en az ilki kadar önemlidir. Sürüm geri alma (rollback), hatalı bir yayından sonra önceki güvenli duruma dönme işlemidir.

Şekil 8.5 hatalı yayından önceki sürüme dönüşü gösterir.



Şekil 8.5: Hatalı yayından önceki sürüme dönüş akışı

Sürüm geri alma yalnızca eski kodu çalıştırmak değildir. Veritabanı göçü, kullanıcı verisi, cache ve dış servis etkileri de düşünülmelidir. Bu yüzden sürüm geri alma planı dağıtım sürecinin baştan yazılmış parçası olmalıdır.

8.14.1 Hatalı dağıtım

Bir ekip yeni sürümü yayına alır ve login ekranı çalışmamaya başlar. CI geçmiş, derleme başarılı olmuş, dağıtım tamamlanmıştır. Ancak üretim ortamı içinde eksik bir gizli değer vardır. Kullanıcılar giriş yapamaz. Buradaki sorun yalnızca teknik değil, iş sürekliliği problemidir (Tablo 8.6).

Tablo 8.6: Hatalı dağıtım

Bulgu	Risk	Eylem
Login çalışmıyor	Kullanıcı erişimi kesilir	Hızlı sürüm geri alma veya gizli değer düzeltme kararı alınır
Logda eksik gizli değer var	Konfigürasyon hatası	Ortam değişkenleri kontrol edilir
Önceki sürüm bilinmiyor	Geri dönüş uzar	İmaj etiketi stratejisi kurulur

Üç bulgu da aynı noktaya çıkar: sorunu üretimde canlı canlı araştırmak yerine, dönülecek sürümün önceden belli olması gerekir.

Önceki sürüme dönmek

Önceki sürüme dönmek, bilinen iyi çalışan kod veya imajı tekrar yayına almaktır. Bu işlem kolay görünür; fakat hangi sürümün iyi olduğu bilinmiyorsa zorlaşır. Bu yüzden sürüm notu, imaj etiketi ve dağıtım geçmişi tutulmalıdır. Sürüm geri alma kararı panik anında değil, önceden tanımlı süreçle verilmelidir.

İmaj etiketi ile sürüm geri alma

İmaj etiketi ile sürüm geri alma küçük bir akış olarak düşünülebilir: sorunlu etiket belirlenir, önceki çalışan etiket seçilir, dağıtım sistemi bu etikete döndürülür, uygulama

sağlık denetimi ile kontrol edilir, olay notu yazılır. Bu akış geri dönebilme sağlar; çünkü "son çalışan sürüm hangisiydi?" sorusu etiket üzerinden cevaplanabilir.

8.14.2 Veritabanı göçü riski

Sürüm geri alma her zaman kolay değildir; özellikle veritabanı göçü (migration) varsa. Yeni sürüm tablo yapısını değiştirmişse eski uygulama bu yeni yapıyla çalışamayabilir. Bu yüzden göç planı sürüm geri alma düşüncesiyle birlikte hazırlanmalıdır; Tablo 8.7 üç tipik riski özetler.

Tablo 8.7: Veritabanı göçü riski

Bulgu	Risk	Aksiyon
Migration geri alınamaz	Eski sürüm çalışmayabilir	Migration öncesi yedek ve plan hazırla
Veri dönüştürüldü	Veri kaybı riski	Geri dönüş senaryosunu test et
Kod ve veri uyumsuz	Kesinti uzar	Küçük ve geriye uyumlu değişiklik tercih et

Bu üç risk de aynı ilkeye bağlanır: kod geri alınabilse bile veri her zaman geri alınamayabilir; bu yüzden göç adımları küçük ve geriye uyumlu tutulmalıdır.

Sürüm geri alma planını belgelemek

Sürüm geri alma planı belgelenirken şu kısa adımlar yazılabilir: son çalışan sürüm nereden bulunur, dağıtım nasıl geri alınır, veritabanı etkisi nasıl kontrol edilir, kim karar verir, olay sonrası hangi not yazılır? Bu plan ekip düzeni sağlar; çünkü kesinti anında herkesin aynı soruları yeniden tartışması gerekmez.

8.15 İş Hattı Hatalarını Okuma

İş hattı hatalarını okumak, otomasyonu yönetilebilir kılan beceridir. İş hattı kırıldığında "GitHub bozuldu" veya "sunucu sorunlu" demek kolaydır; fakat çoğu zaman hata belirli bir adım içinde açıkça görünür. Yanlış çalışma zamanı ayarı, eksik gizli değer, YAML girintisi, başarısız test veya bağlantı problemi birbirinden farklı çözümler gerektirir.

Küçük bir vaka düşünelim. PR açılır, CI çalışmaz. Ekip kodda hata arar; fakat sorun YAML dosyasındaki girintidir. Başka bir gün derleme geçer, dağıtım başarısız olur; bu kez eksik `DATABASE_URL` vardır. Hata okumak bu yüzden teknik sabır ister.

Tablo 8.8: İş Hattı Hataları

Hata türü	Muhtemel etki	İlk bakılacak yer
YAML syntax	İş akışı başlamaz	İş akışı dosyası ve girinti
Eksik gizli değer	Derleme/dağıtım kırılır	Repository veya ortam bazlı gizli değer alanı

Hata türü	Muhtemel etki	İlk bakılacak yer
Çalışma zamanı uyumsuzluğu	Paket veya derleme hatası	Kurulum adımı
Test başarısız	Birleştirme durur	Test çıktısı
Dağıtım bağlantısı	Yayın başarısız	Platform logları, SSH, token

Tablo 8.8 iş hattı hatasının iş etkisine dönüşmeden önce sınıflandırılmasını sağlar.

YAML sözdizimi hataları

YAML sözdizimi hataları çoğu zaman girinti veya yanlış anahtar kullanımından çıkar. Bir boşluk farkı iş akışının okunmasını engelleyebilir. Bu durumda kod değil, iş hattı tanımı hatalıdır. Çözüm, iş akışı dosyasını küçük değişikliklerle düzeltmek ve mümkünse PR içinde incelemektir.

Eksik gizli değer

Eksik gizli değer, iş hattının belirli bir adımda kimlik doğrulaması yapamamasına neden olur. Örneğin kayıt deposuna push aşaması token bulamazsa imaj yayınlanamaz. Bu durumda değeri YAML içine yazmak çözüm değildir; doğru gizli değer alanına eklemek ve erişim kapsamını kontrol etmek gerekir.

Yanlış çalışma zamanı sürümü

Yanlış çalışma zamanı sürümü, yerelde çalışan kodun CI'da kırılmasına neden olabilir. Örneğin proje Node.js 20 özelliği kullanırken iş akışı Node.js 18 kuruyorsa derleme hata verebilir. Bu, kodun kötü olduğu anlamına gelmeyebilir; ortam uyuşmuyordur. Çalışma zamanı sürümü README, Dockerfile ve iş akışı içinde tutarlı olmalıdır.

8.15.1 Başarısız test veya derleme

Başarısız test veya derleme, iş hattının en değerli uyarılarındandır.

```
Run npm test
FAIL tests/login.test.ts
Expected status 200, received 401
```

Bu çıktı bize login testinde yetkilendirme sorunu olduğunu gösterir. Birleştirmeyi zorlamak yerine testin neden kırıldığını anlamak gerekir. Belki kod hatalıdır, belki test yeni iş kuralına göre güncellenmelidir.

Dağıtım bağlantı hatası

Dağıtım bağlantı hatası, iş hattının hedef sunucuya veya platforma ulaşamaması olabilir. SSH anahtarı yanlış, token süresi bitmiş, firewall kapalı veya alan adı yönlendirmesi eksik olabilir. Bu hatayı kod hatası sanmak zaman kaybettirir. İlk ayrım şudur: derleme çıktı üretti mi, hata hedef ortama bağlanırken mi oluştu?

8.15.2 Log okuma disiplini

Log okuma disiplini, hatayı ilk anlamlı satırdan başlayarak yorumlamaktır.

```
Error: DATABASE_URL is not defined
```

Bu satır bize uygulamanın veritabanı ayarını bulamadığını söyler. Çözüm Dockerfile'ı baştan yazmak değil, ortam değişkeni ve gizli değer ayarlarını kontrol etmektir. İyi ekipler logu yalnızca kriz anında değil, dağıtım sonrası doğrulama sırasında da okur.

8.16 Atölye

Yerelde geçen testin iş hattında da geçmesini görmek, CI'ı soyut bir kısaltma olmaktan çıkarır. Atölyede küçük bir kalite komutu yazılır, iş akışına bağlanır ve bir dağıtım adayı commit kimliğiyle işaretlenir. Kayıt deposuna gönderme veya gerçek sunucuya dağıtım yoktur. Kontrolün nerede durduğunu görmek yeterlidir.

Görev 1: Küçük uygulama ve test iskeletini kur

Komut

```
$ mkdir -p ~/gelistirme-atolyesi/bolum-8/ci-app
$ cd ~/gelistirme-atolyesi/bolum-8/ci-app
$ git init -b main
$ git config user.name "Atolye Kullanici"
$ git config user.email "atolye@example.com"
$ mkdir -p scripts .github/workflows src tests logs
$ printf '%s\n' \
  "const http = require('node:http');" \
  "const server = http.createServer((req, res) => {" \
  "  if (req.url === '/health') {" \
  "    res.setHeader('Content-Type', 'application/json');" \
  "    res.end('{\"status\":\"ok\"}');" \
  "    return;" \
  "  }" \
  "  res.statusCode = 404; res.end('not found');" \
  "});" \
  "server.listen(3000, '0.0.0.0', () => console.log('ready'));" \
  > src/server.js
$ printf '%s\n' \
  "const test = require('node:test');" \
  "const assert = require('node:assert/strict');" \
  "test('quality gate', () => assert.equal(2 + 2, 4));" \
  > tests/app.test.js
$ printf 'logs/\n' > .gitignore
```

Beklenen çıktı

```
Initialized empty Git repository in <yol>/ci-app/.git/
```

Görev 2: Yerel CI betiğini oluştur

Komut

```
$ printf '%s\n' '#!/usr/bin/env bash' \
  'set -euo pipefail' \
  'node --check src/server.js' \
  'node --test --test-reporter=tap' > scripts/ci.sh
$ chmod +x scripts/ci.sh
$ bash -n scripts/ci.sh
$ ./scripts/ci.sh
```

Beklenen çıktı

```
ok 1 - quality gate
# pass 1
# fail 0
```

`bash -n` sözdizimi doğruysa çıktı üretmez; betik sonraki komutta testi çalıştırır.

Görev 3: Yerel kalite kapısını kaydet

Komut

```
$ git add .gitignore src tests scripts
$ git diff --cached --check
$ git status --short
$ git commit -m "Yerel CI kalite kapisini ekle"
```

Beklenen çıktı

```
A  .gitignore
A  scripts/ci.sh
A  src/server.js
A  tests/app.test.js
[main <kimlik>] Yerel CI kalite kapisini ekle
```

Görev 4: GitHub Actions iş akışını ve imaj tarifini yaz

Komut

```
$ printf '%s\n' 'name: ci' '' 'on:' \
  '  pull_request:' '  push:' \
  '    branches: [main]' '' 'jobs:' '  test:' \
  '    runs-on: ubuntu-latest' '    steps:' \
  '      - uses: actions/checkout@v4' \
  '      - uses: actions/setup-node@v4' \
  '        with:' '          node-version: 20' \
  '      - run: ./scripts/ci.sh' \
  > .github/workflows/ci.yml
$ printf '%s\n' 'FROM node:20-alpine' 'WORKDIR /app' \
  'COPY src ./src' 'EXPOSE 3000' \
  'CMD ["node", "src/server.js"]' > Dockerfile
```

Beklenen çıktı

Komutlar çıktı üretmez; iş akışı, yerelde kullanılan `scripts/ci.sh` betiğini çağırır.

Görev 5: İş akışı yapısını terminalde denetle

Komut

```
$ sed -n '1,40p' .github/workflows/ci.yml
$ rg -n 'pull_request|ubuntu-latest|ci.sh' .github/workflows/ci.yml
$ git diff --check
$ git diff --stat
$ git status --short
$ git add .github/workflows/ci.yml Dockerfile
$ git commit -m "CI workflow ve imaj tarifini ekle"
```

Beklenen çıktı

```
4:  pull_request:
10:    runs-on: ubuntu-latest
16:      - run: ./scripts/ci.sh
 .github/workflows/ci.yml | ...
 Dockerfile               | ...
[main <kimlik>] CI workflow ve imaj tarifini ekle
```

Görev 6: Dağıtım adayını commit kimliğiyle adlandır

Komut

```
$ git log --oneline -2
$ candidate_id=$(git rev-parse --short HEAD)
$ printf '%s\n' "$candidate_id"
$ git tag "candidate-$candidate_id"
```

Beklenen çıktı

```
<kimlik> CI workflow ve imaj tarifini ekle
<kimlik> Yerel CI kalite kapisini ekle
<kisa-kimlik>
```

Görev 7: Aday imajı üret ve metadata'yı denetle

Komut

```
$ docker build -t "ci-atolye:$candidate_id" .
$ docker image inspect "ci-atolye:$candidate_id" \
  --format '{{.Id}} {{.Config.ExposedPorts}}'
```

Beklenen çıktı

```
... naming to docker.io/library/ci-atolye:<kisa-kimlik> ...
sha256:<imaj-kimligi> map[3000/tcp:{}]
```

Görev 8: Sağlık kontrolü ve log doğrulaması yap

Komut

```
$ docker run -d --name ci-atolye-app \
  -p 18080:3000 "ci-atolye:$candidate_id"
$ curl -fsS http://localhost:18080/health
$ docker logs --tail 5 ci-atolye-app
$ ssh -V 2>&1
```

Beklenen çıktı

```
<konteyner-kimligi>
{"status":"ok"}
ready
OpenSSH_9.x ...
```

SSH sürüm denetimi yalnızca istemcinin hazır olduğunu gösterir; uzak sunucuya bağlantı kurulmaz.

Görev 9: Geri dönüş kaydını gör ve yerel nesneleri temizle

Komut

```
$ git show --stat --oneline "candidate-$candidate_id"
$ git status --short
$ docker stop ci-atolye-app
$ docker rm ci-atolye-app
$ docker image rm "ci-atolye:$candidate_id"
```

Beklenen çıktı

```
<kimlik> CI workflow ve imaj tarifini ekle
ci-atolye-app
ci-atolye-app
Untagged: ci-atolye:<kisa-kimlik>
```

`git status --short` çıktı üretmiyorsa etiket temiz bir commit'i gösterir. Bu commit kimliği, gerçek dağıtım yapılmadan önce geri dönüş adayı olarak kaydedilmiştir.

Bu dokuz görev boyunca kod hiçbir sunucuya gönderilmedi; yine de kalite kapısından geçti, bir imaj olarak paketlendi, sağlık kontrolünden geçirildi ve geri dönüş için etiketlendi. CI/CD'nin özü tam olarak budur: otomasyon süreci hızlandırır, ama her adımın izlenebilir ve geri alınabilir kalmasını sağlayan disiplin insana aittir. Bir sonraki bölümde bu disiplinin başka bir yüzüne, geliştirme ortamında biriken güvenlik ve gizlilik risklerine bakacağız.

Bölüm 9

Güvenlik, Gizlilik ve Geliştirme Ortamı Denetimi

Güvenlik çoğu zaman üretim ortamı, saldırılar ve büyük kurumlar üzerinden düşünülür. Bu bakış anlaşılırdır; çünkü güvenlik ihlalleri genellikle kullanıcı verisi, para kaybı veya itibar sorunu olduğunda görünür hale gelir. Ancak sorunların önemli bir kısmı çok daha önce, geliştirme ortamında başlar.[1]

Bir `.env` dosyasının repoya gönderilmesi, CI logunda token görünmesi, SSH private key'in paylaşılması, YZ aracına müşteri verisi yapıştırılması veya VPS üzerinde gereğinden fazla yetkili kullanıcıyla işlem yapılması ilk bakışta küçük teknik hatalar gibi durabilir. Ancak bu hatalar gizlilik, iş sürekliliği, maliyet ve güven kaybına dönüşebilir.

Bu bölümde güvenliği geliştirme ortamında oluşan gerçek vakalar üzerinden ele alacağız; soyut korku dili yerine somut bulgulara bakacağız. Her vakada aynı soruları soracağız: bulgu nedir, risk nedir, hangi aksiyon alınmalıdır? Audit yaklaşımı da tam olarak bu düzenli bakma alışkanlığıdır.

9.1 Geliştirme Ortamında Güvenlik Neden Önemlidir?

Bir proje ekibi uygulamayı yerelde geliştirir. Veritabanı bağlantısı için `.env` dosyası kullanılır. Teslim yaklaştığında biri "her şey repoda dursun, herkes kolay çalıştırsın" diyerek `.env` dosyasını da commit eder. Başlangıçta bu pratik görünür. Herkes aynı ayarlarla projeyi çalıştırabilecektir.

Ancak bu varsayım bozulur. `.env` içinde gerçek veritabanı şifresi, e-posta servis token'ı veya dağıtım anahtarı varsa artık kurulum kolaylığının ötesinde bir erişim riski oluşmuştur. Repo özel bile olsa ekip üyeleri, entegrasyonlar ve ileride değişen erişimler düşünülmelidir. Repo public olursa risk daha da büyür.

[1] Rajapakse, R. N., Zahedi, M., Babar, M. A., & Shen, H. (2022). Challenges and solutions when adopting DevSecOps: A systematic review. *Information and Software Technology*, *141*, Article 106700. https://doi.org/10.1016/j.infsof.2021.106700

Geliştirme ortamı bu yüzden güvenlik zincirinin dışında değildir. Yerel makine, Git repository, CI/CD iş hattı, konteyner kayıt deposu ve VPS birbirine bağlıdır. Tablo 9.1, bir noktadaki gevşekliğin nasıl başka bir noktaya taşınabildiğini beş örnekle gösterir.

Tablo 9.1: Geliştirme Ortamında Güvenlik Neden Önemlidir?

Bulgu	Risk	Aksiyon
`.env` depoya girdi	Gizli bilgi sızar	Değer iptal edilir, yenisi üretilir
CI logunda token göründü	Yetkisiz erişim	Log incelenir, token yenilenir
VPS root erişimi herkeste	Hasar alanı büyür	Yetki sınırlandırılır
YZ'ye gerçek log yapıştırıldı	Kişisel veri dışarı çıkar	Örnek veri ve politika kullanılır
İmaj içine parola gömüldü	Kayıt deposu sızıntısı	Değer çalışma anında verilir

Tablodaki beş satır aslında tek bir noktayı vurgular: güvenlik burada teknik ustalıktan çok, düzenli kontrol ve sorumluluk alışkanlığı olarak işler.

Geliştirme ortamı da saldırı yüzeyidir

Geliştirme ortamı da saldırı yüzeyidir; çünkü burada kod, gizli bilgi, test verisi, repository erişimi ve dağıtım yetkisi bulunabilir. Geliştirici bilgisayarı üretim sunucusu değildir; fakat üretime giden anahtarları taşıyorsa risk zincirinin parçası haline gelir. Yönetim Bilişim Sistemleri açısından önemli olan, güvenliği yalnızca son kullanıcı ekranında değil, geliştirme sürecinin tamamında okumaktır.

9.1.1 Yerel makineden üretime sızıntı

Yerel makineden üretime sızıntı, geliştirme bilgisayarındaki değerin yanlışlıkla üretim ortamına etki etmesidir. Örneğin geliştirici yerel test için üretim veritabanı şifresini `.env` dosyasına koyar. Daha sonra bu dosya commit edilir veya YZ aracına hata ayıklama için gönderilir. Teknik hata küçük görünür; fakat üretim verisine erişim riski doğar (Tablo 9.2).

Tablo 9.2: Yerel makineden üretime sızıntı

Bulgu	Risk	Aksiyon
Yerel `.env` üretim gizli değeri içeriyor	Üretim sistemine yetkisiz erişim	Geliştirme ve üretim gizli değerleri ayrılır
Dosya repoya gitti	Gizli bilgi geçmişte kalır	Gizli bilgi döndürülür, geçmiş incelenir
YZ'ye gönderildi	Dış paylaşım riski	Değer iptal edilir, politika gözden geçirilir
Üretim adresi yerel `.env` içinde	Test komutu üretim verisine dokunabilir	Ortam dosyaları ayrılır

Bulgu	Risk	Aksiyon
`.gitignore` sonra eklendi	Dosya geçmişte kalır	Geçmiş incelenir, değer döndürülür

Tablodaki beş bulgunun ortak noktası, hiçbirinin tek başına büyük görünmemesidir; asıl risk, küçük kolaylık kararlarının üst üste binmesinden doğar.

Repo, CI/CD ve VPS zinciri

Repo, CI/CD ve VPS zincirini bir teslimat hattı gibi düşünebiliriz. Kod repoya girer, iş hattı çalışır, imaj üretilir, sunucuya dağıtılır. Bu zincirin herhangi bir halkasında yanlış yetki veya gizli bilgi sızıntısı varsa risk ilerleyebilir. Repo yalnızca kod deposu değildir; dağıtım sürecinin başlangıç noktasıdır. Bu yüzden erişim, gizli bilgi ve log kontrolleri zincir halinde değerlendirilmelidir.

9.1.2 Teknik riskin iş riskine dönüşmesi

Teknik risk iş riskine dönüşebilir. Örneğin API token'ı sızarsa yalnızca bir karakter dizisi kaybolmaz; servis maliyeti artabilir, müşteri verisi açığa çıkabilir, uygulama durabilir veya kurum güven kaybedebilir. Tablo 9.3 bu zinciri beş örnekle somutlaştırır.

Tablo 9.3: Teknik riskin iş riskine dönüşmesi

Teknik bulgu	İş etkisi	Aksiyon
Açık token	Yetkisiz kullanım ve maliyet	Token iptali ve erişim analizi
Eski bağımlılık	Güvenlik açığı	Güncelleme ve test planı
Backup yok	Veri kaybı	Yedekleme ve geri dönüş testi
Açık VPS portu	Hizmet kesintisi veya veri sızıntısı	Firewall ve erişim denetimi
YZ'ye gerçek log gönderildi	Gizlilik ve uyum riski	Veri paylaşım politikası uygulanır

Yönetim Bilişim Sistemleri bakış açısı bu dönüşümü görmeyi gerektirir: teknik ayrıntı, karar ve sorumluluk zincirine bağlanmalıdır.

9.2 Denetim

Denetim (audit), bir sistemi suçlamaktan çok, görünmeyen riskleri düzenli biçimde bulmak için yapılır. Geliştirme ortamı denetimi ise repository, gizli bilgi, erişim, CI/CD, Docker, VPS ve YZ veri paylaşımı gibi alanlara belirli aralıklarla bakma alışkanlığıdır.

Küçük bir başlangıç listesi şöyle olabilir:

```
## Geliştirme Ortamı Audit Kontrolü
- [ ] Repoda `.env` veya gizli bilgi var mı?
- [ ] `.gitignore` ve `.dockerignore` güncel mi?
- [ ] GitHub Actions gizli bilgileri gereğinden geniş yetkili mi?
- [ ] CI loglarında gizli değer görünüyor mu?
- [ ] Docker image içine gizli bilgi gömülmüş mü?
- [ ] VPS üzerinde root erişimi sınırlı mı?
- [ ] YZ araçlarına gerçek veri gönderiliyor mu?
- [ ] Güvenlik dokümantasyonu güncel mi?
```

Bu liste, dönem projesinde PR öncesi, dağıtım öncesi veya teslim öncesi birkaç dakikada gözden geçirilebilecek somut bir kontrol alışkanlığı sunar.

9.2.1 Denetim nedir?

Audit, belirli bir sistemin kayıtlarını, ayarlarını ve davranışlarını kontrol etmektir. Küçük bir proje için şu maddeler yeterli bir başlangıçtır:

- Repoda gizli dosya veya gerçek gizli bilgi var mı?
- Kimlerin repository erişimi var?
- CI/CD hangi gizli bilgilere erişebiliyor?
- Son dağıtım hangi commit'ten yapılmış?
- Loglarda hassas veri görünüyor mu?

Bu maddelerin her biri doğrudan bakılabilir, somut bir kontrol noktasıdır.

9.2.2 Geliştirme ortamı denetimi nedir?

Geliştirme ortamı denetimi, yerel makineden üretime giden hazırlık hattını kontrol etmektir. Şu sorularla başlanabilir:

- `.env.example` gerçek gizli bilgi içermeden gerekli değişkenleri gösteriyor mu?
- Docker imaj oluşturma bağlamı (build context) içinde gereksiz veya gizli dosya var mı?
- VPS üzerinde yalnızca gereken portlar açık mı?
- SSH anahtarları kişisel ve izlenebilir mi?
- YZ'ye gönderilen örnekler maskelenmiş mi?

Bu kontroller küçük görünür; fakat sızıntıların çoğu küçük ihmalden başlar.

9.2.3 Varlık, erişim, gizli bilgi, log ve değişiklik geçmişi

Denetim sırasında varlık, erişim, gizli bilgi, log ve değişiklik geçmişi birlikte düşünülmelidir. Örneğin bir dağıtım token'ı vardır, bu token'a CI erişir, başarısız dağıtım logunda token'ın bir kısmı görünür ve bu durum commit geçmişindeki bir iş akışı değişikliğinden sonra başlamıştır (Tablo 9.4).

Tablo 9.4: Varlık, erişim, gizli bilgi, log ve değişiklik geçmişi

Alan	Soru	Aksiyon
Varlık	Hangi sistem veya değer korunuyor?	Envantere ekle
Erişim	Kim kullanabiliyor?	Yetkiyi sınırla
Gizli bilgi	Nerede saklanıyor?	Repo dışına taşı
Log	Ne görünür oldu?	Maskeleme ve log kontrolü yap
Geçmiş	Ne zaman değişti?	Commit ve PR'ı incele

Bu beş satırı yan yana okuduğunuzda dikkat çeken şey şudur: sorun tek bir anda ortaya çıkmaz, varlıktan geçmişe uzanan zincirin herhangi bir halkasında birikir.

9.2.4 Bulguyu risk ve eylem planı olarak yazmak

Audit bulgusu yalnızca "sorun var" diye yazılırsa aksiyona dönüşmez. Daha iyi yapı bulgu, risk ve aksiyon şeklindedir (Tablo 9.5).

Tablo 9.5: Risk ve Eylem Planı

Durum	Risk	Eylem
`.env` dosyası repoda	Gizli bilgi sızıntısı	Gizli bilgi iptal edilir, `.env` repodan çıkarılır
Root SSH açık	Sunucuda geniş hasar riski	Dağıtım kullanıcısı oluşturulur
CI logunda token var	Yetkisiz erişim	Token yenilenir, log adımı düzeltilir
YZ'ye gerçek token gitti	Dış ortamda yetki sızıntısı	Token iptal edilir, politika yazılır
İmaj içinde parola var	Kayıt deposu erişimi sızıntıya dönüşür	İmaj yeniden üretilir

Bu format, teknik gözlemi yönetilebilir işe çevirir.

9.3 Repository ve Gizli Bilgi Sızıntısı Vakası

Bir ekip projeyi kolay kurulsun diye `.env` dosyasını repoya ekler. Dosyada veritabanı bağlantısı, e-posta servisi token'ı ve JWT gizli anahtarı vardır. Repo başlangıçta private olduğu için ekip bunu büyük sorun olarak görmez. Ancak daha sonra repo public yapılır veya erişim verilen kişi sayısı artar.

Varsayım burada bozulur: "Private repo güvenlidir" varsayımı yeterli değildir. Repository erişimi zamanla değişebilir. Ayrıca gizli bilgi bir kez commit geçmişine girdiyse dosyayı son commit'ten silmek her zaman yeterli olmaz (Tablo 9.6).[2]

[2] Rahman, M. R., Imtiaz, N., Storey, M.-A., & Williams, L. (2022). Why secret detection tools are not enough: It's not just about false positives—An industrial case study. *Empirical Software Engineering*, *27*(3). https://doi.org/10.1007/s10664-021-10109-y

Tablo 9.6: Repository ve Gizli Bilgi Sızıntısı Vakası

Bulgu	Risk	Aksiyon
`.env` repoda	Gizli değerler erişilebilir	Gizli değerler döndürülür
Gizli bilgi commit geçmişinde	Eski commit'ten okunabilir	Geçmiş ve erişim analizi yapılır
`.env.example` yok	Ekip gerçek değer paylaşmaya yönelir	Güvenli şablon oluşturulur
Repo sonradan public yapıldı	Eski gizli bilgi açığa çıkar	Tüm gizli değerler döndürülür
Erişim listesi şişmiş	İzlenemeyen kopyalama riski	Yetkiler gözden geçirilir

Bu vaka bize şunu gösterir: gizli bilgi yönetimi, proje başında kurulması gereken bir alışkanlıktır.

9.3.1 `.env` dosyasının GitHub'a gönderilmesi

`.env` dosyası gerçek değerleri taşıyorsa GitHub'a gönderilmemelidir. Repoda tutulması gereken dosya çoğu zaman `.env.example` olmalıdır.

```
# .env.example
DATABASE_URL=postgresql://user:password@localhost:5432/appdb
JWT_SECRET=change-me
```

Bu örnek yalnızca beklenen formatı gösterir; içinde gerçek bir gizli bilgi yoktur. Böylece ekip kurulumu anlar, gizli değerler ise paylaşılmamış olur.

9.3.2 API key, token ve veritabanı şifresi

API key, token ve veritabanı şifresi erişim yetkisi verir. Bir token sızarsa saldırgan servis maliyeti oluşturabilir, veri çekebilir veya dağıtım sürecini etkileyebilir; Tablo 9.7 bu beş değeri risk ve aksiyonlarıyla sıralar.

Tablo 9.7: API key, token ve veritabanı şifresi

Değer	Risk	Aksiyon
API key	Servis kötüye kullanımı	Key iptal edilir, yenisi üretilir
Veritabanı şifresi	Veri erişimi	Şifre döndürülür, erişim logu incelenir
Dağıtım token'ı	Yayın hattı riski	Token yetkisi azaltılır
JWT gizli anahtarı	Oturum sahteciliği	Gizli anahtar yenilenir, oturumlar düşürülür
E-posta servisi token'ı	İzinsiz gönderim ve maliyet	Token iptal edilir

Bu beş değerin ortak özelliği, hiçbirinin "sadece bir metin parçası" olmamasıdır; her biri doğrudan bir sisteme erişim anlamına gelir.

9.3.3 `.gitignore` neden tek başına yeterli olmayabilir?

`.gitignore`, henüz Git tarafından takip edilmeyen dosyaların eklenmesini engeller. Ancak dosya daha önce commit edildiyse `.gitignore` tek başına onu geçmişten çıkarmaz. Bir geliştirici `.env` dosyasını commit edip sonra `.gitignore` eklerse, gizli bilgi hâlâ geçmişte duruyor olabilir. Bu yüzden `.gitignore` koruyucu alışkanlıktır; sızıntı sonrası temizlik aracı değildir.

9.3.4 Gizli bilgi commit geçmişine girdiyse ne yapılır?

Gizli bilgi commit geçmişine girdiyse ilk yapılacak iş değeri iptal etmek veya döndürmektir. Geçmişi temizlemek gerekebilir; fakat değer zaten görülmüş kabul edilmelidir. Bu yüzden "dosyayı sildim, sorun çözüldü" düşüncesi eksiktir (Tablo 9.8).

Tablo 9.8: Gizli bilgi commit geçmişine girdiyse ne yapılır?

Adım	Neden
Gizli bilgiyi iptal et	Eski değer artık güvenilir değildir
Yeni gizli bilgi üret	Sistem çalışmaya devam etsin
Erişim loglarını kontrol et	Kötüye kullanım var mı görülsün
Repo geçmişini değerlendir	Gerekiyorsa geçmiş temizliği yapılsın
Ekip notu yaz	Aynı hata tekrar etmesin

Bu beş adım tek bir mantığı izler: önce görünür hâle gelen değeri geçersiz kılmak, sonra nedenini anlamak. Sıradaki vaka bu mantığın CI/CD tarafında da neden geçerli olduğunu gösterir.

9.4 CI/CD Loglarında Gizli Bilgi Vakası

Bir iş hattı başarısız olur. Geliştirici hata ayıklamak için environment variable değerlerini loga yazdırır. Amaç sorunu anlamaktır. Ancak log içinde registry token'ı veya veritabanı bağlantısı görünür hale gelir. İş hattı logları ekip üyeleri, entegrasyonlar veya platform tarafından görülebilir.

İş hattı logu o gün için sorunsuz kapanmış gibi görünür; kimse geri dönüp bakmaz. Oysa log dosyası orada durmaya devam eder ve ekip üyeleri, entegrasyonlar veya platform tarafından haftalar sonra da açılabilir. Tablo 9.9 bu kalıcılığın nasıl sızıntıya dönüştüğünü beş örnekle gösterir.

Tablo 9.9: CI/CD Loglarında Gizli Bilgi Vakası

Bulgu	Risk	Aksiyon
Gizli bilgi logda göründü	Yetkisiz erişim	Gizli bilgi döndürülür

Bulgu	Risk	Aksiyon
Debug komutu gizli bilgi yazdırıyor	Tekrar sızıntı	Komut kaldırılır
Log erişimi geniş	Gizlilik riski	Erişimler sınırlandırılır
Maskeleme, değer parçalara bölününce çalışmadı	Token logda kısmen görünür kaldı	Değer iptal edilir; loglama ayarı gözden geçirilir
Çıktı paketine `.env` kopyalandı	İş hattı çıktı paketi sızdırır	Çıktı paketi içeriği denetlenir

Tablodaki dördüncü satır özellikle öğreticidir: ekip maskeleme kullandığı için kendini güvende sanmış, fakat değer parçalandığında bu güven yanılgıya dönüşmüştür.

Token'ın log'a yazılması

Token'ın loga yazılması, gizli bilginin gizli saklandığı alandan çıkması demektir. Örneğin `echo $REGISTRY_TOKEN` komutu iş hattı logunda değeri gösterebilir. Bazı platformlar maskeleme yapsa bile buna güvenerek gizli bilgi yazdırmak doğru değildir. Token görünür olduysa iptal edilmeli ve yenilenmelidir.

Gizli bilgi maskeleme (secret masking) sınırları

Gizli bilgi maskeleme (secret masking), platformun bilinen gizli değerini logda gizlemeye çalışmasıdır. Ancak maskeleme her durumda kusursuz sayılamaz: değer parçalanırsa, encode edilirse veya başka biçimde yazılırsa yine görünür olabilir. Bu yüzden doğru güvenlik davranışı, platformun maskelemesine güvenmek değil, gizli bilgiyi loga hiç yazdırmamaktır.

Token yetkisini ve süresini sınırlamak

Token yetkisi ve süresi sınırlıysa sızıntı etkisi de sınırlanabilir. Sadece registry'ye image push etmesi gereken token'a tüm repository yönetim yetkisi verilmemelidir. Süresiz token yerine gerektiğinde yenilenebilen ve kapsamı dar token tercih edilmelidir. Bu, en az yetki ilkesinin CI/CD tarafındaki karşılığıdır.

9.5 SSH Private Key ve VPS Erişimi Vakası

VPS'e bağlanmak için SSH anahtarı kullanılır. Ekipte biri bağlantı kolay olsun diye private key dosyasını mesajla paylaşır. Başlangıçta bu pratik görünür; herkes sunucuya hızlıca girebilir. Ancak private key, sunucu kapısının anahtarıdır. Paylaşıldığı anda erişim kontrolü zayıflar.

Bu vakada asıl sorun teknik dosyanın paylaşılması değil, yetki izlenebilirliğinin kaybolmasıdır. Kim bağlandı, hangi işlem yapıldı, anahtar kime verildi, biri ekipten ayrıldığında erişim nasıl iptal edilecek? Bu sorular yanıtsız kalır (Tablo 9.10).

Tablo 9.10: SSH Private Key ve VPS Erişimi Vakası

Bulgu	Risk	Aksiyon
Private key paylaşıldı	Yetkisiz erişim	Key iptal edilir, kişisel key kullanılır
Passphrase yok	Key ele geçerse doğrudan kullanılır	Passphrase ve erişim politikası eklenir
Erişim sahipleri belirsiz	Denetim yapılamaz	Kullanıcı ve key envanteri tutulur
Key repoya commit edildi	Depo erişimi sunucu erişimine dönüşür	Key iptal edilir, `.gitignore` güncellenir
Ayrılan kişinin key'i duruyor	Yetki geri alınamaz	Sunucudan public key silinir

Tablonun son iki satırı birbirini tamamlar: anahtar repoya girdiyse sorun kod tarafındadır, kullanıcı ayrılmasına rağmen anahtar duruyorsa sorun süreç tarafındadır.

Public key ve private key farkı

Public key paylaşılabilir; private key paylaşılmamalıdır. Public key kilide takılan kimlik gibi, private key ise kapıyı açan anahtar gibidir. VPS'e erişim için sunucuya public key eklenir, kullanıcı kendi private key'iyle bağlanır. Private key paylaşılırsa kişisel erişim ile ortak erişim birbirine karışır.

9.5.1 `~/.ssh` klasörü ve dosya izinleri

`~/.ssh` klasörü kullanıcının SSH anahtarlarını ve bağlantı ayarlarını tutar. Bu klasör proje repository'sinin parçası değildir ve versiyonlanmamalıdır. Örneğin `~/.ssh/id_ed25519` private key olabilir; bu dosya gizli kalmalıdır. Repoda en fazla "VPS'e kişisel SSH key ile bağlanılır; private key paylaşılmaz" gibi bir dokümantasyon notu bulunmalıdır.

Passphrase kullanımı

Passphrase, private key dosyası ele geçse bile kullanımını zorlaştıran ek korumadır. Tek başına bütün riski çözmez; fakat anahtarın doğrudan kullanılmasını engelleyebilir. Ekip çalışmasında passphrase kullanımı, cihaz kaybı veya dosya kopyalanması gibi durumlara karşı ek güvenlik katmanı sağlar.

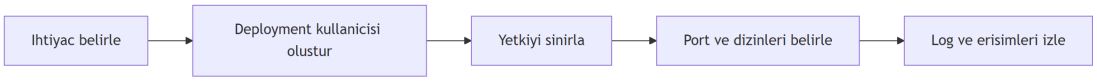
Kaybolan veya sızan key'i iptal etmek

Kaybolan veya sızan key güvenilir kabul edilmemelidir. Sunucudaki `authorized_keys` içinden ilgili public key kaldırılmalı, gerekiyorsa yeni key oluşturulmalı ve erişim logları incelenmelidir. "Bir daha kullanmayız" demek yeterli değildir; sunucu tarafında erişim gerçekten iptal edilmelidir.

9.6 Gereğinden fazla yetkili dağıtım kullanıcısı vakası

Deployment için sunucuya bağlanan kullanıcının gereğinden fazla yetkili olması sık görülen bir risktir. Her işi kök kullanıcı (root) ile yapmak hızlı görünür. Fakat yanlış komut, hatalı script veya ele geçirilmiş key daha geniş hasar üretebilir.

Şekil 9.1 sınırlı yetkili bir dağıtım erişimini gösterir.



Şekil 9.1: Sınırlı yetkili kullanıcıyla güvenli dağıtım erişimi

Bu yaklaşım hızdan biraz feragat ettirebilir; fakat hata alanını daraltır. Yönetim Bilişim Sistemleri açısından bu, iş sürekliliği ve sorumluluk ayrımıdır.

En az yetki ilkesi (least privilege)

En az yetki ilkesi (least privilege), bir kullanıcıya veya token'a yalnızca görevini yapması için gereken yetkinin verilmesidir. Dağıtım kullanıcısının bütün sunucuyu yönetmesi gerekmiyorsa bu yetki verilmemelidir. Aksi halde dağıtım sürecindeki küçük hata tüm sistemi etkileyebilir.

9.6.1 Root kullanıcıyla dağıtım riski

Root kullanıcıyla dağıtım yapmak, her komuta en geniş yetkiyi vermektir. Bir script yanlış dizini silerse veya kötü niyetli biri erişim kazanırsa hasar alanı büyür. Tablo 9.11 bu büyümenin beş farklı görünümünü listeler.

Tablo 9.11: Root kullanıcıyla dağıtım riski

Bulgu	Risk	Aksiyon
Dağıtım root ile yapılıyor	Tüm sistem etkilenebilir	Sınırlı kullanıcı oluştur
SSH root açık	Brute force ve yetki riski	Root SSH kapatılır veya sınırlandırılır
Komutlar denetlenmiyor	Hata izlenemez	Dağıtım logu tutulur
Tek paylaşılan root parolası	Sorumluluk izlenemez	Kişisel anahtar ve kullanıcı kullanılır
Sudo sınırsız	Tek komut tüm sistemi etkileyebilir	Komut yetkisi daraltılır

Tablodaki son satır özellikle can alıcıdır: sudo yetkisi sınırsız bırakıldığında, dağıtım kullanıcısı ayrı olsa bile pratikte root ile aynı hasar alanına sahip olunur.

Dağıtım kullanıcısı oluşturmak

Dağıtım kullanıcısı oluşturmak için akış şu şekilde düşünülebilir: ayrı kullanıcı oluşturulur, yalnızca gerekli dizinlere erişim verilir, Docker veya servis yönetimi için gereken

minimum yetki tanımlanır, SSH key kişisel veya CI özelinde eklenir, erişim ve loglar belgelenir. Bu adımlar geri dönebilme ve denetim açısından önemlidir.

Sadece gereken portları ve dizinleri açmak

Sunucuda yalnızca gereken portlar ve dizinler açılmalıdır. Örneğin web uygulaması için `80` ve `443`, SSH için sınırlı `22` erişimi gerekebilir. Veritabanı portunu internete açmak çoğu projede gereksiz risktir. Bu karar `SECURITY.md` veya dağıtım belgesinde yazılabilir; fakat gerçek gizli bilgi veya private key bu dokümanda yer almaz.

9.7 Docker Image İçinde Gizli Bilgi Bulunması

Bir ekip Dockerfile içine `ENV DATABASE_URL=...` yazar. Amaç uygulamanın kolay çalışmasıdır. "Zaten imajı biz kullanıyoruz, kimse görmeyecek" diyerek imajı oluşturup kayıt deposuna (registry) push ederler. Daha sonra fark edilir ki üretim veritabanı bilgisi imajın içine girmiştir.

Bu karar anında gözden kaçan şey, imaj kayıt deposuna gittiğinde erişim alanının artık ekibin kendi sınırlarında kalmamasıdır. Ayrıca Docker layer geçmişi bazı bilgilerin sonradan silinse bile iz bırakmasına neden olabilir. Tablo 9.12 bu izin nerede kaldığını beş bulguyla gösterir.

Tablo 9.12: Docker imajı içinde gizli bilgi

Durum	Risk	Eylem
Gizli bilgi Dockerfile içinde	İmajdan okunabilir	Gizli bilgi iptal edilir, Dockerfile düzeltilir
Gizli bilgi layer geçmişinde	Eski katmanda kalabilir	İmaj yeniden üretilir, kayıt deposu temizlenir
Kayıt deposu erişimi geniş	Daha çok kişi gizli bilgiye ulaşabilir	Erişim sınırlandırılır
`.dockerignore` eksik	`.env` imaja kopyalanabilir	Dosya dışlama listesi güncellenir
Herkese açık kayıt deposu	İmaj içeriği herkese açık olabilir	Erişim ve etiket politikası yazılır

İlk iki satır aynı kaynaktan besleniyor: gizli bilgi imaja bir kez girdiğinde, onu Dockerfile'dan silmek de katmandan silmek de garanti değildir. Sıradaki ayrım tam olarak bunu önlemeyi hedefler.

Derleme anındaki (build-time) ve çalışma zamanındaki (runtime) gizli değer farkı

Derleme anındaki (build-time) gizli değer, imaj üretilirken gereken gizli değerdir. Çalışma zamanındaki (runtime) gizli değer ise konteyner çalışırken gereken değerdir. Çoğu uygulama gizli değeri runtime sırasında almalıdır. Veritabanı parolasını derleme aşamasında imaja koymak, onu imajın parçası haline getirebilir. Bu ayrım yapılmadığında dağıtım kolaylığı güvenlik riskine dönüşür.

Docker layer geçmişi

Docker image katmanlardan oluşur. Bir katmanda gizli değer kopyalanıp sonraki katmanda silinirse, gizli değerin izi önceki katmanda kalabilir. Bu yüzden "sonradan sildim" her zaman yeterli değildir. Doğru davranış, gizli değeri baştan imaja sokmamaktır.

Kayıt deposu riski

Kayıt deposu (registry), imajların paylaşıldığı yerdir. Erişimi genişse veya imaj herkese açıksa, imaj içindeki yanlış bilgi daha geniş alana yayılır. Bu yüzden kayıt deposu erişimi, imaj etiketleri ve silme politikası denetim kapsamında değerlendirilmelidir. Gizli bilgi sızıntısı denetiminde Dockerfile'ın yanında kayıt deposundaki imajlar da kontrol edilmelidir.

Coolify tarafında gizli bilgi yönetimi

Coolify tarafında gizli değerler uygulamanın environment alanında yönetilerek repodan ve imajdan ayrılabilir. Ancak yanlışlıkla loga yazdırılan veya gereksiz yetkili kullanıcıların görebildiği gizli bilgi yine risk üretir. Platform gizli bilgi alanı kullanmak doğru başlangıçtır; erişim ve log disipliniyle tamamlanmalıdır.

9.8 Eski ve Açık İçeren Bağımlılık Vakası

Bir proje aylar önce kurulmuş paketlerle çalışmaya devam eder. Uygulama sorunsuz açıldığı için ekip bağımlılıklara bakmaz. Ancak kullanılan paketlerden biri bilinen bir güvenlik açığı içeriyor olabilir. "Çalışıyor" olmak, "güvenli" olmak anlamına gelmez.

Bağımlılık güvenliği, projenin kullandığı dış paketlerin izlenmesi ve gerektiğinde güncellenmesidir. Bu konu yalnızca teknik hijyen değildir; zayıf paketler veri güvenliği, servis sürekliliği ve kurum itibarı açısından risk doğurabilir. Tablo 9.13 bu riski beş bulguyla listeler.

Tablo 9.13: Eski ve Açık İçeren Bağımlılık Vakası

Bulgu	Risk	Aksiyon
Eski paket	Bilinen açık olabilir	Güvenlik uyarıları incelenir
Kilit dosyası güncel değil	Farklı ortamda farklı paket kurulabilir	Kilit dosyası kayda alınır
Güncelleme test edilmedi	Uygulama kırılabilir	Güncelleme PR ve testle yapılır
Tarama yok	Açık fark edilmez	Düzenli güvenlik taraması eklenir
Transitif bağımlılık eski	Dolaylı paket açık taşıyabilir	Kilit dosyası ve tarama birlikte okunur

Son satır uyarıcıdır: büyük bir sürüm güncellemesinden çekinip onu tamamen ertelemek, beraberindeki küçük güvenlik yamasını da ertelemek anlamına gelebilir.

Bağımlılık güvenliği

Bağımlılık güvenliği, projenin kendi yazmadığı ama kullandığı paketlerin güvenliğini izlemektir. Bir paket projeye hız kazandırabilir; fakat bakım görmüyorsa veya açık içeriyorsa risk taşır. Yönetim Bilişim Sistemleri açısından bu, dış tedarik bağımlılığına benzer: kullandığınız bileşenin güvenilirliği sizin sisteminizi de etkiler.

9.8.1 Açık içeren paketler

Açık içeren paket, bilinen güvenlik zafiyeti barındıran bağımlılıktır. Bir proje ekibi "paket çalışıyor" diyerek uyarıları görmezden gelebilir. Ancak bu açık kötüye kullanılırsa uygulama veya veri riske girebilir (Tablo 9.14).

Tablo 9.14: Açık içeren paketler

Bulgu	Risk	Aksiyon
Güvenlik uyarısı var	Sömürülebilir açık	Paket güncellemesi planlanır
Paket terk edilmiş	Uzun vadeli bakım riski	Alternatif değerlendirilir
Güncelleme kırıyor	İş sürekliliği riski	Testlerle kontrollü geçiş yapılır
Uyarı kapatılmış	Risk görünmez olur	Uyarı gerekçesi kayda alınır
Büyük sürüm güncellemesi riskli görülüp tümüyle ertelendi	Beraberindeki güvenlik yaması da hiç uygulanmadı	Büyük sürüm beklesin; güvenlik yaması ayrı ve öncelikli uygulanır

Tablonun son satırı ilk dördünden farklıdır: burada sorun bir açığın fark edilmemesi değil, fark edilip bilerek ertelenmesidir.

Kilit dosyası ve güvenlik güncellemeleri

Kilit dosyası, hangi paket sürümlerinin kurulduğunu sabitler. Bu dosya olmadan farklı makinelerde farklı alt sürümler kurulabilir. Güvenlik güncellemeleri yapılırken kilit dosyasının değişimi PR içinde görülmelidir. Böylece ekip yalnızca "paket güncellendi" demekle kalmaz; hangi sürümlerin değiştiğini de izler.

Güncelleme ve kırılma riski

Güvenlik güncellemesi yapmak önemlidir; fakat güncelleme uygulamayı kırabilir. Bu yüzden güncelleme, PR, test ve sürüm geri alma adımlarıyla kontrollü biçimde ilerlemelidir. Eski pakette kalmak riskli olabilir; kontrolsüz güncelleme de risklidir. Doğru davranış, riski görünür kılıp küçük ve test edilebilir değişikliklerle ilerlemektir.

9.9 YZ Aracına Gizli Veri Gönderme Vakası

YZ aracı hata ayıklamada çok yardımcı olabilir. Okuyucu bir hata mesajını, kod parçasını veya API yanıtını paylaşarak açıklama isteyebilir. Ancak bu kolaylık, gizli

veri paylaşımı riskini de beraberinde getirir. Özellikle gizli bilgi, müşteri verisi, kurum içi kod ve gerçek loglar dikkatle ele alınmalıdır.

Küçük bir vaka düşünelim. Okuyucu, üretim hatasını çözmek için log çıktısını YZ'ye gönderir. Log içinde gerçek müşteri e-postaları ve erişim token'ı vardır. Amaç yardım almaktır; fakat veri kontrolsüz şekilde dış ortama taşınmıştır. Tablo 9.15 bu taşınmanın beş farklı biçimini sıralar.

Tablo 9.15: YZ Aracına Gizli Veri Gönderme

Bulgu	Risk	Aksiyon
Gerçek müşteri verisi paylaşıldı	Gizlilik ihlali	Veri maskeleme politikası uygulanır
Token istem metnine yapıştırıldı	İstem üçüncü bir tarafa (YZ sağlayıcısına) gidebilir	Token iptal edilir, yeni değer güvenli alana taşınır
Kurumsal kod paylaşıldı	Fikri mülkiyet riski	Kullanım politikası netleştirilir
Üretim logu olduğu gibi yapıştırıldı	Kişisel veri dışarı çıkar	Log maskelenir veya örnek üretilir
Private key sohbete yazıldı	Sunucu erişimi paylaşılmış olur	Key iptal edilir

İkinci satır özellikle akılda kalıcıdır: istem metnine yapıştırılan bir token, artık yalnızca ekip içinde değil, YZ sağlayıcısının da erişebileceği bir ortamdadır.

YZ'ye hangi veri verilmez?

YZ'ye gerçek gizli bilgi, token, private key, müşteri verisi, kişisel veri, kurum içi gizli kod ve sözleşmeye bağlı bilgiler verilmemelidir. Yardım almak için bağlam gerekiyorsa değerler maskelenmeli veya temsili örnek üretilmelidir. Yönetim Bilişim Sistemleri açısından bu, veri yönetişimi ve bilgi güvenliği davranışıdır.

Gizli bilgi, token ve key paylaşımı

Gizli bilgi, token ve key paylaşımı erişim riskidir. Bir API token'ı YZ istemine yazılırsa artık o değerin gizli kaldığından emin olamayız. Bu durumda yapılacak şey değeri iptal etmek ve yenilemektir. "Sadece bir kere gönderdim" güvenlik açısından yeterli savunma değildir.

Müşteri verisi ve kurumsal kod paylaşımı

Müşteri verisi ve kurumsal kod, teknik yardım almak uğruna kontrolsüz paylaşılmamalıdır. Gerçek ad, e-posta, sipariş, fatura veya kurum içi algoritma bilgisi dış araçlara gönderilmeden önce politika kontrolü gerekir. Bu risk yalnızca teknik değil, hukuki ve kurumsal güven meselesidir.

Maskelenmiş örnek üretmek

Maskelenmiş örnek üretmek, gerçek verinin yapısını koruyup değerlerini değiştirmektir. Örneğin `customer@example.com`, `sk_test_xxx`, `user-123` gibi temsili değerler kullanıla-

bilir. Amaç YZ'ye hatanın biçimini göstermek, gerçek kişiyi veya gerçek erişim değerini paylaşmamaktır.

YZ kullanım politikası

YZ kullanım politikası, hangi verinin paylaşılabileceğini, hangisinin maskelenmesi gerektiğini ve hangisinin kesinlikle gönderilmeyeceğini açıklar. Ders projesinde bile küçük bir politika faydalıdır: gizli bilgi paylaşma, müşteri verisini maskele, kurumsal kod için izin al, YZ çıktısını denetle. Bu politika güvenliği günlük çalışma davranışına bağlar.

9.10 Yanlış Yapılandırılmış VPS Vakası

Bir VPS kiralanır, Docker kurulur ve uygulama yayına alınır. İlk günlerde her şey yolunda görünür: tarayıcıda sayfa açılır, ekip rahatlar. Haftalar geçtikçe bu rahatlık hiç sorgulanmaz hâle gelir; oysa firewall ayarları hâlâ yapılmamıştır, root SSH erişimi hâlâ açıktır, veritabanı portu dış dünyaya hâlâ açıktır ve sistem paketleri hiç güncellenmemiştir. Uygulama çalışıyor olabilir; fakat sunucu yönetimi zayıftır.

"Uygulama açılıyorsa sunucu tamamdır" düşüncesi burada yanıltıcı çıkar. Sunucu güvenliği, uygulamanın erişilebilirliği kadar önemlidir. Tablo 9.16 bu beş eksiği listeler.

Tablo 9.16: Yanlış Yapılandırılmış VPS Vakası

Bulgu	Risk	Aksiyon
Gereksiz port açık	Yetkisiz erişim	Firewall ile sınırlandır
Root SSH açık	Geniş yetki riski	Sınırlı kullanıcı ve key kullan
Güncelleme yok	Bilinen açıklar kalır	Güncelleme planı yap
Backup yok	Veri kaybı	Yedekleme ve geri dönüş testi kur
Veritabanı portu internete açık	Doğrudan veri erişimi	Bağlantı noktası yalnızca iç ağa açık bırakılır, internete kapatılır

İlk dört satır klasik bir ihmal listesi gibi görünse de son satır farklıdır: burada erişim yetkisi değil, hangi ağdan erişilebileceği unutulmuştur.

Açık portlar ve zayıf parola

Açık portlar, dış dünyaya açılan kapılardır. Gerekmediği halde veritabanı portunun internete açık olması ciddi risktir. Zayıf parola da bu riski büyütür. Doğru yaklaşım, yalnızca gerekli portları açmak, mümkünse SSH key kullanmak ve parola tabanlı erişimi sınırlamaktır.

Root SSH erişimi

Root SSH erişimi açıksa, ele geçirilen erişim tüm sunucuyu etkileyebilir. Deneyimsiz bir kullanıcı için kolay görünebilir; fakat risk büyüktür. Daha sağlıklı yaklaşım, sınırlı yetkili kullanıcıyla bağlanmak ve gerektiğinde kontrollü `sudo` kullanmaktır. Böylece hata ve kötüye kullanım alanı daralır.

9.10.1 Güncellenmeyen sistem paketleri

Güncellenmeyen sistem paketleri bilinen güvenlik açıklarını taşıyabilir. Sunucuda düzenli güncelleme kontrolü yapılmalıdır.

```
$ apt list --upgradable
openssl/stable  ...
docker-ce/stable ...
```

Bu çıktı, güncellenebilecek paketleri gösterir. Her güncelleme hemen üretime uygulanmak zorunda değildir; fakat güncelleme ihtiyacı bilinmeli ve planlanmalıdır.

9.10.2 Log takibi ve backup eksikliği

Log takibi olmadan sorunlar geç fark edilir; backup olmadan veri kaybı geri alınamaz hale gelir. Örneğin:

```
$ docker logs app
Error: database connection timeout
```

Bu çıktı tekrar ediyorsa yalnızca anlık hata değil, süreklilik sorunu vardır. Backup ise "hata olursa ne yaparız?" sorusunun veri tarafındaki cevabıdır. Backup alınmıyorsa sürüm geri alma planı eksik kalır.

9.11 Üretim veritabanına yerel bilgisayardan bağlanma

Bir geliştirici yerelde hata ayıklamak için üretim veritabanına bağlanır. "Sadece okuyacağım" diye düşünür. Ancak yanlış script çalışır, test verisi üretim tablosuna yazılır veya gerçek veri silinir. Bu vaka, ortam ayrımının neden önemli olduğunu gösterir.

Üretim veritabanı gerçek sistemin kalbidir. Yerel geliştirme ortamından doğrudan üretime bağlanmak, özellikle yazma yetkisi varsa, büyük risktir. Okuma yetkisi bile kişisel veri ve gizlilik açısından dikkat gerektirir. Tablo 9.17 bu riski beş bulguyla listeler.

Tablo 9.17: Üretim veritabanına Yerel Bilgisayardan Bağlanma

Bulgu	Risk	Aksiyon
Yerel `.env` üretim veritabanını gösteriyor	Yanlış işlem gerçek veriyi etkiler	Geliştirme veritabanı ayrı tutulur
Geniş yetkili kullanıcı	Silme veya değiştirme riski	Yetki sınırlandırılır
Test verisi üretime yazıldı	Rapor ve stok bozulur	Temizlik ve olay kaydı yapılır
Yedek alınmadan deneme yapıldı	Geri dönüş belirsizleşir	Önce yedek, sonra sorgu
Salt okunur sanılan hesap yazma izni de taşıyordu	Kullanıcı verisi istemeden değişebilir	Gerçek salt okunur (read-only) hesap ayrılır

Son satır özellikle öğreticidir: bir hesabın salt okunur sanılması, gerçekten öyle olduğu anlamına gelmez; yetki her zaman doğrulanmalı, varsayılmamalıdır.

Geliştirme ve üretim ayrımı

Geliştirme ortamı deneme yapmak içindir; üretim ortamı gerçek kullanıcı ve gerçek veri içindir. Bu iki ortam aynı bağlantı bilgilerini kullanıyorsa hata alanı büyür. Ayrım yalnızca isimlendirme değil, veri, yetki, erişim ve sorumluluk ayrımıdır.

9.11.1 Yanlış `.env` kullanımı

Yanlış `.env` kullanımı, uygulamanın beklenmeyen ortama bağlanmasına neden olabilir. Yerelde `DATABASE_URL` üretim adresini gösteriyorsa test komutu üretim verisini etkileyebilir. Bu yüzden `.env.example` açık olmalı, gerçek `.env` dosyaları ortam adlarıyla dikkatle ayrılmalı ve üretim değerleri yerel makinede gereksiz tutulmamalıdır.

Üretim verisini bozma riski

Üretim verisini bozmak yalnızca teknik hata değildir; raporları, kullanıcı güvenini ve iş kararlarını etkiler. Yanlışlıkla silinen stok hareketi, hatalı yönetim raporuna dönüşebilir. Bu yüzden üretim verisine erişim sınırlı, kayıtlı ve gerekçeli olmalıdır. Testler için anonim veya sahte veri kullanılmalıdır.

Yetki sınırlandırma

Yetki sınırlandırma, üretim veritabanına erişen kullanıcının yalnızca gereken işlemleri yapabilmesidir. Raporlama için okuma yetkisi gerekiyorsa yazma yetkisi verilmemelidir. Göç için ayrı, uygulama için ayrı, analiz için ayrı kullanıcılar düşünülebilir. Bu ayrım hata etkisini daraltır.

9.12 Geliştirme Ortamı Güvenlik Kontrol Listesi

Şimdi önceki vakaları uygulanabilir bir denetim listesine dönüştürelim. Bu liste, her projede aynı ağırlıkta kullanılmak zorunda değildir. Küçük bir ödev için bazı maddeler fazla olabilir; ekipli ve yayına çıkan bir proje için ise bu maddeler temel kontrol sağlar.

Güvenlik burada da aynı mantıkla işler: düzenli tekrarlanan küçük kontroller, teslim gecesi yaşanacak paniği önler. Tablo 9.18 bu kontrolleri alan, soru ve kanıt üçlüsüyle listeler.

Tablo 9.18: Geliştirme ortamı güvenlik kontrol listesi

Alan	Kontrol sorusu	Kanıt
Repo	Gizli bilgi veya `.env` commit edilmiş mi?	`git status`, dosya listesi
Gizli bilgi	`.env.example` gerçek değer içermiyor mu?	Örnek dosya
SSH	Private key paylaşılmadan mı kullanılıyor?	Kullanıcı/key envanteri

Alan	Kontrol sorusu	Kanıt
CI/CD	Gizli bilgiler loga yazılıyor mu?	İş hattı logları
Docker	Image içine gizli bilgi girmiş mi?	Dockerfile ve oluşturma bağlamı
VPS	Gereksiz port açık mı?	Firewall kuralları
YZ	Veriler maskeleniyor mu?	YZ kullanım politikası
Dokümantasyon	Güvenlik notları güncel mi?	`SECURITY.md` veya README
Bağımlılık	Bilinen açık var mı?	Kilit dosyası ve tarama çıktısı
Yedek	Geri dönüş denendi mi?	Yedek ve sürüm geri alma notu
Erişim	Üretim yetkisi gerekçeli mi?	Kullanıcı ve anahtar envanteri

Bu tablo önceki dokuz vakanın bir özetidir; her satır, bölüm boyunca ayrıntısıyla görülmüş bir bulgunun kısa hâlidir.

9.12.1 Repo ve `.gitignore` kontrolü

Repo içinde gerçek `.env`, private key, üretim veri dökümü, müşteri verisi ve servis token'ı tutulmamalıdır. Kontrol için şu maddeler kullanılabilir:

- `.env` repoda mı?
- `.gitignore` içinde `.env`, gizli bilgi dosyaları ve yerel çıktı klasörleri var mı?
- Daha önce commit edilmiş gizli bilgi var mı?
- Varsa gizli bilgi döndürüldü mü?

Daha önce commit edilmiş gizli bilgi için `.gitignore` tek başına yeterli değildir.

9.12.2 `.env.example` ve gizli bilgi kontrolü

`.env.example`, beklenen değişkenleri güvenli şablon olarak göstermelidir. Kötü örnek gerçek değer taşır; iyi örnek format gösterir.

```
# Kötü
DATABASE_URL=postgresql://prod_user:real_password@prod-db:5432/app

# Daha iyi
DATABASE_URL=postgresql://user:password@localhost:5432/app
```

Okuyucu bu ayrımı gördüğünde doküman paylaşılırken gizli bilgi paylaşılmadığını anlar.

9.12.3 SSH key ve GitHub erişim kontrolü

SSH key ve GitHub erişimi "kim hangi kapıyı açabiliyor?" sorusuyla denetlenmelidir. Kontrol listesi kısa olabilir:

- Her kullanıcı kendi SSH key'ini mi kullanıyor?
- Private key paylaşılmış mı?
- Ekipten ayrılan kişinin erişimi kaldırıldı mı?
- Kaybolan key iptal edildi mi?
- GitHub erişim rolleri gereğinden geniş mi?

Public key paylaşılabilir; private key paylaşılmamalıdır.

9.12.4 GitHub Actions secrets kontrolü

GitHub Actions secrets kontrolü, iş hattının hangi bilgiye hangi yetkiyle eriştiğini görmektir; Tablo 9.19 bunu beş örnekle somutlaştırır.

Tablo 9.19: GitHub Actions secrets kontrolü

Gizli değer	Yetki	Kullanım yeri
`REGISTRY_TOKEN`	İmaj gönderme	İmaj yayın işi
`DEPLOY_KEY`	Dağıtımı tetikleme	Dağıtım işi
`DATABASE_URL`	Uygulama bağlantısı	Üretim ortamı
`SMTP_PASSWORD`	Posta gönderme	Bildirim işi
`SENTRY_DSN`	Hata izleme	İzleme işi

Bu değerler loga yazdırılmamalı ve yalnızca gereken işlerde (job) kullanılmalıdır.

9.12.5 Docker imajı ve kayıt deposu kontrolü

Docker imajı ve kayıt deposu kontrolünde üç soru sorulmalıdır:

- İmaj içine gizli bilgi veya gereksiz dosya girdi mi?
- Kayıt deposundan imajı kim çekebilir?
- Hangi etiket üretimde çalışıyor?

Gizli bilgi imajın katman (layer) geçmişine girdiyse sonradan dosyayı silmek yetmeyebilir. Bu pratik sonuç, gizli bilgiyi imaja baştan koymamayı gerektirir.

9.12.6 VPS kullanıcı, port ve firewall kontrolü

VPS kontrolü kullanıcı yetkisi, açık port ve firewall üçlüsüyle başlamalıdır. Port ve firewall bulgularının somut bir örneği zaten Tablo 9.16 üzerinde görülmüştü; burada onu tekrarlamak yerine, kullanıcı ve erişim tarafını tamamlayan kontrol sorularına bakalım (Tablo 9.20).

Tablo 9.20: VPS kullanıcı, port ve firewall kontrolü

Kontrol	Risk	Aksiyon
Parola ile SSH	Deneme-yanılma	Anahtar zorunlu tutulur
Dağıtım kullanıcısı root'tan ayrılmamış	Root ile karışma riski	Sınırlı kullanıcı kullan

Kontrol	Risk	Aksiyon
SSH ve erişim logları tutulmuyor	İhlal geç fark edilir	Log kaydı ve inceleme takvimi kurulur

Root ile işlem yapmak hızlıdır; fakat hata veya sızıntı durumunda iş sürekliliğini daha fazla etkileyebilir.

9.12.7 YZ veri paylaşımı kontrolü

YZ aracına gönderilecek bağlam üç gruba ayrılabilir; bu ayrım Tablo 9.21 içinde yer alır.

Tablo 9.21: YZ veri paylaşımı kontrolü

Veri türü	Karar
Genel hata mesajı	Paylaşılabilir
Açık kaynak kod parçası	Paylaşılabilir
Proje mimari özeti	Paylaşılabilir
Örnek API yanıtı	Maskelenmeli
Müşteri benzeri sahte veri	Maskelenmeli
Kurumsal olmayan örnek log	Maskelenmeli
Gerçek token	Paylaşılmamalı
Müşteri verisi	Paylaşılmamalı
Gizli kurumsal kod	Paylaşılmamalı

Bu ayrım, YZ kullanımını yasaklamak yerine güvenli hale getirir.

9.12.8 Doküümantasyon güncelliği kontrolü

Güvenlik dokümantasyonu yaşayan bir kayıt olmalıdır. Güncel olmayan güvenlik notu yanlış güven duygusu üretir. Bu durumun nasıl fark edileceği Tablo 9.22 ile küçük bir kontrol listesiyle özetlenmiştir.

Tablo 9.22: Dokümantasyon güncelliği kontrolü

Kontrol	Son güncelleme	Sorumlu	Kanıt
Gizli bilgi rotasyonu	2026-07-01	Ekip lideri	Token yenileme kaydı
Firewall	2026-07-02	DevOps sorumlusu	Kural listesi
YZ veri politikası	2026-07-03	Proje ekibi	`SECURITY.md`
Erişim ve yetki listesi	2026-07-04	Sistem yöneticisi	SSH ve panel kullanıcı listesi
Bağımlılık tarama	2026-07-05	Geliştirici	Son tarama raporu
Sızıntı müdahale notu	2026-07-06	Ekip lideri	Panik planı kaydı

Asıl değer, gerçekten doğrulanabilir kayıt tutmaktan gelir; kanıt sütunu boş kalan bir tablo, kontrolün gerçekten yapılıp yapılmadığını da şüpheli bırakır.

9.13 Güvenlik Kültürü

Güvenlik kültürü, projede güvenli davranışların sonradan eklenen bir kontrol adımı olmaktan çıkıp günlük çalışma biçiminin parçası hâline gelmesidir. Bir ekip düşünelim: PR incelemesi sırasında yalnızca kodun çalışıp çalışmadığına bakılıyor. Kimse gizli bilgi var mı, gereksiz port açıldı mı, YZ'den gelen kod güvenli mi diye sormuyor. Proje çalışıyor olabilir; fakat güvenlik davranışı eksiktir.

Güvenlik kültürü, herkesin güvenlik uzmanı olması anlamına gelmez. Temel soruların ekip alışkanlığına dönüşmesi anlamına gelir: Bu değer gizli mi? Bu yetki fazla mı? Bu log ne gösteriyor? Bu paket güncel mi? Bu veriyi YZ'ye gönderebilir miyiz? Bu değişiklik için sürüm geri alma planımız var mı?

Bu kültürün iş etkisi büyüktür. Hatalar tamamen bitmez; fakat erken fark edilir, küçük kalır ve öğrenmeye dönüşür.

Güvenlik sadece sonda yapılmaz

Güvenliği yalnızca teslimden önce yapmak, yangın alarmını bina dolduktan sonra takmaya benzer. Proje boyunca gizli bilgi yönetimi, erişim, bağımlılık ve log davranışı düşünülmediyse son gün yapılacak kontrol yüzeysel kalır. Güvenlik PR, CI/CD, Dockerfile ve dokümantasyon içinde küçük kontroller olarak yaşamalıdır.

PR incelemesi sırasında güvenlik

PR incelemesi sırasında güvenlik soruları da sorulmalıdır. Örneğin yeni bir environment variable eklendiyse `.env.example` güncellendi mi, gizli bilgi repoya girdi mi, logda hassas veri var mı? Bu sorular küçük görünür; fakat gizli bilgi sızıntısını üretime gitmeden yakalayabilir.

CI/CD içinde güvenlik

CI/CD içinde güvenlik, iş hattının yalnızca test ve derleme çalıştırması değil, gizli bilgi kullanımını, bağımlılık uyarılarını ve dağıtım yetkilerini de kontrollü yönetmesidir. Bir iş akışı gereksiz üretim ortamı gizli değerine erişiyorsa, test geçse bile güvenlik tasarımı zayıftır. CI/CD güvenlik kapısıdır; kontrolsüz gizli bilgi dağıtım noktası olmamalıdır.

9.13.1 Gizli bilgi sızıntısı olursa panik planı

Gizli bilgi sızıntısı olursa panik yerine plan gerekir:

1. Sızan gizli bilgi belirlenir.
2. Gizli bilgi iptal edilir veya döndürülür.
3. Yeni değer güvenli alana eklenir.
4. Erişim logları kontrol edilir.
5. Repo, CI ve dokümantasyon düzeltilir.
6. Olaydan öğrenilen ders yazılır.

Bu akış, krizi yönetilebilir hale getirir. Asıl hedef, etkisini sınırlamak ve tekrarını önlemektir; hatayı saklamak bu hedefe hizmet etmez.

9.13.2 Kurumsal politika ve uyum

Kurumsal politika ve uyum, güvenlik davranışının kişisel tercihe kalmamasını sağlar. Hangi YZ aracı kullanılabilir, hangi veri dışarı çıkarılamaz, üretim erişimi kimde olur, loglar ne kadar tutulur, gizli bilgi sızıntısı kime bildirilir? Bu soruların cevabı yazılı olduğunda ekip daha tutarlı hareket eder. Yönetim Bilişim Sistemleri açısından güvenlik, teknik kontrol kadar organizasyonel düzen meselesidir.

Sonsöz

Bu yolculuğa başlarken ilk sorumuz oldukça basitti: "Benim bilgisayarımda çalışan kod, neden senin bilgisayarında çalışmıyor?" Bu sorunun peşine takılarak işletim sistemlerinin dosya yapısından terminal komutlarına, versiyon kontrol sistemlerinden konteyner teknolojilerine ve nihayetinde otomatik test hatlarından güvenlik kültürüne kadar uzanan bir haritayı adım adım dolaştık.

Birçok araç, komut ve yapılandırma dosyası gördük. Çoğu zaman bir geliştirme ortamı kurmanın, en yeni teknolojileri art arda dizmek olduğu düşünülür. Bu oldukça anlaşılır bir reflekstir; çünkü teknoloji dünyası bize sürekli yeni ve daha hızlı araçlar sunar. Ancak geliştirme ortamını yalnızca "hangi araçları kullanıyoruz?" sorusuyla eşitlemek, konuyu daraltmak anlamına gelir.

Burada kritik ayrım şudur: Git kullanmak takım çalışması demek değildir; Docker kullanmak uygulamanın her yerde sorunsuz çalışacağını garanti etmez; CI/CD hattı kurmak kaliteli yazılım ürettiğiniz anlamına gelmez. Eğer dal (branch) kurallarınız yoksa Git sadece bir yedekleme aracıdır. Eğer konteyner içine gereksiz dosyalar dolduruyorsanız Docker sadece daha ağır bir sanal makinedir. Eğer testleriniz neyi kontrol ettiğini bilmiyorsa CI/CD hattı sadece daha hızlı hata yayımlayan bir otomasyondur.

Bu kitabın ilk bölümünde bir iddiada bulunmuştuk: Geliştirme ortamı, yazılım üreten ekiplerin çalışma düzenini taşıyan bir yapıdır. Yönetim Bilişim Sistemleri (YBS) perspektifinden baktığımızda asıl değer, bu teknik yapıların iş sürecine, ekip iletişimine ve organizasyonel kültüre nasıl bağlandığını okuyabilmektir. Bir problemi çözerken "bu hatayı nasıl düzeltirim?" diye sormak yazılımcının refleksidir; "bu hata neden daha önce yakalanamadı ve sürecimizde hangi kontrol eksik?" diye sormak ise sistem düşüncesinin refleksidir.

İlk bakışta her aracı mükemmel düzeyde öğrenmek iyi bir hedef gibi görünebilir. Ancak teknoloji hızla değişir. Bugün kullandığımız paket yöneticileri, terminal araçları veya bulut servisleri birkaç yıl içinde yerini yenilerine bırakacaktır. Hatta yapay zeka destekli kodlama asistanları, bizim yerimize birçok yapılandırma dosyasını saniyeler içinde üretecektir. Peki bu durum geliştirme ortamlarını anlamayı gereksiz kılar mı?

Bu sorunun yanıtı bizi kitabın temel felsefesine götürür. Araçlar değişse bile kodun başkaları tarafından anlaşılabilir olması, ortamların tekrarlanabilir olması, hataların erken yakalanması ve güvenlik açıklarının öngörülmesi ihtiyacı değişmeyecektir. Yapay zekanın yazdığı bir yapılandırma dosyasını canlı ortama alırken, o dosyanın hangi portları açtığını veya hangi ortam değişkenlerine ihtiyaç duyduğunu okuyamıyorsanız, hızlanmış değil sadece riski büyütmüş olursunuz.

Bu yaklaşım sihirli bir çözüm değildir; fakat düşünmeyi kolaylaştırır. Kariyeriniz boyunca birçok farklı projede, farklı rollerde (yazılımcı, analist, proje yöneticisi veya ürün sahibi) yer alabilirsiniz. Hangi rolde olursanız olun, bu kitapta konuştuğumuz zemin sizinle birlikte gelecektir.

Toparlamak gerekirse, modern geliştirme ortamlarını anlamak bize sadece kod yazmayı değil, sürdürülebilir ve güvenli bir ürün inşa etmeyi öğretir. Araçların adlarına veya komutların sözdizimine takılmayın; onların çözdüğü problemlere odaklanın. Asıl önemli olan, teknik araçlar ile iş değerleri arasındaki dengeyi görebilmektir.

Bu kitaptaki kavramların, kendi projelerinizi inşa ederken size sağlıklı bir düşünme haritası sağlamasını dilerim.

Emre Akadal
2026

Kaynakça

Beaulieu-Jones, B. K., & Greene, C. S. (2017). Reproducibility of computational workflows is automated using continuous analysis. *Nature Biotechnology*, *35*(4), 342. https://doi.org/10.1038/nbt.3780

Buchgeher, G., Schoeberl, S., Geist, V., Dorninger, B., Haindl, P., & Weinreich, R. (2023). Using architecture decision records in open source projects—An MSR study on GitHub. *IEEE Access*, *11*, 63725–63740. https://doi.org/10.1109/ACCESS.2023.3287654

Casalicchio, E., & Iannucci, S. (2020). The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency and Computation: Practice and Experience*, *32*(17). https://doi.org/10.1002/cpe.5668

Dakhel, A. M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., & Jiang, Z. M. (2023). GitHub Copilot AI pair programmer: Asset or liability? *Journal of Systems and Software*, *203*, Article 111734. https://doi.org/10.1016/j.jss.2023.111734

Davila, N., & Nunes, I. (2021). A systematic literature review and taxonomy of modern code review. *Journal of Systems and Software*, *177*, Article 110951. https://doi.org/10.1016/j.jss.2021.110951

Decan, A., Mens, T., & Grosjean, P. (2019). An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, *24*(1), 381–416. https://doi.org/10.1007/s10664-017-9589-y

Enck, W., & Williams, L. (2022). Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Security & Privacy*, *20*(2), 96–100. https://doi.org/10.1109/MSEC.2022.3142338

German, D. M., Adams, B., & Hassan, A. E. (2016). Continuously mining distributed version control systems: An empirical study of how Linux uses Git. *Empirical Software Engineering*, *21*(1), 260–299. https://doi.org/10.1007/s10664-014-9356-2

Han, S., Wang, M., Zhang, J., Li, D., & Duan, J. (2024). A review of large language models: Fundamental architectures, key technological evolutions, interdisciplinary technologies integration, optimization and compression techniques, applications, and challenges. *Electronics*, *13*(24). https://doi.org/10.3390/electronics13245040

Krewinkel, A., & Winkler, R. (2017). Formatting open science: Agilely creating multiple document formats for academic manuscripts with Pandoc Scholar. *PeerJ Computer Science*. https://doi.org/10.7717/peerj-cs.112

Laukkanen, E., Itkonen, J., & Lassenius, C. (2017). Problems, causes and solutions when adopting continuous delivery—A systematic literature review. *Information and Software Technology*, *82*, 55–79. https://doi.org/10.1016/j.infsof.2016.10.001

Lethbridge, T. C., Singer, J., & Forward, A. (2003). How software engineers use documentation: The state of the practice. *IEEE Software*, *20*(6), 35. https://doi.org/10.1109/MS.2003.1241364

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., & Neubig, G. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, *55*(9). https://doi.org/10.1145/3560815

Lwakatare, L. E., Kilamo, T., Karvonen, T., Sauvola, T., Heikkilä, V., Itkonen, J., Kuvaja, P., Mikkonen, T., Oivo, M., & Lassenius, C. (2019). DevOps in practice: A multiple case study of five companies. *Information and Software Technology*, *114*, 217–230. https://doi.org/10.1016/j.infsof.2019.06.010

Mavridis, I., & Karatza, H. (2019). Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing. *Future Generation Computer Systems*, *94*, 674–696. https://doi.org/10.1016/j.future.2018.12.035

McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E. (2016). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, *21*(5), 2146–2189. https://doi.org/10.1007/s10664-015-9381-9

Raemaekers, S., van Deursen, A., & Visser, J. (2017). Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software*, *129*, 140–158. https://doi.org/10.1016/j.jss.2016.04.008

Rahman, M. R., Imtiaz, N., Storey, M.-A., & Williams, L. (2022). Why secret detection tools are not enough: It's not just about false positives—An industrial case study. *Empirical Software Engineering*, *27*(3). https://doi.org/10.1007/s10664-021-10109-y

Rajapakse, R. N., Zahedi, M., Babar, M. A., & Shen, H. (2022). Challenges and solutions when adopting DevSecOps: A systematic review. *Information and Software Technology*, *141*, Article 106700. https://doi.org/10.1016/j.infsof.2021.106700

Ritchie, D. M. (1984). The evolution of the UNIX time-sharing system. *AT&T Bell Laboratories Technical Journal*, *63*(8), 1577–1593. https://doi.org/10.1002/j.1538-7305.1984.tb00054.x

Rodriguez, P., Haghighatkhah, A., Lwakatare, L. E., Teppola, S., Suomalainen, T., Eskeli, J., Karvonen, T., Kuvaja, P., Verner, J. M., & Oivo, M. (2017). Continuous deployment of software intensive products and services: A systematic mapping

study. *Journal of Systems and Software*, *123*, 263–291. https://doi.org/10.1016/j.jss.2015.12.015

Shahin, M., Babar, M. A., & Zhu, L. (2017). Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, *5*, 3909–3943. https://doi.org/10.1109/ACCESS.2017.2685629

Tyree, J., & Akerman, A. (2005). Architecture decisions: Demystifying architecture. *IEEE Software*, *22*(2), 19. https://doi.org/10.1109/MS.2005.27

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, *30*.

Westerman, S. J. (1997). Individual differences in the use of command line and menu computer interfaces. *International Journal of Human-Computer Interaction*, *9*(2), 183–198. https://doi.org/10.1207/s15327590ijhc0902_6

Dizin

İNETD İnternet Teknolojileri Derneği

Yönetim Bilişim Sistemleri için Modern Geliştirme Ortamları

Emre Akadal

Emre Akadal, İstanbul Üniversitesi İktisat Fakültesi Yönetim Bilişim Sistemleri Bölümü öğretim üyesidir. Dr. Akadal, İstanbul Üniversitesi bünyesindeki İstanbul BTC'nin genel koordinatörü olarak görev yapmaktadır. Aynı zamanda blokzincir alanında çeşitli eğitimler ve dersler vermekte, bu konuda projeler gerçekleştirmektedir. Araştırmacı ayrıca optimizasyon, veritabanı, yazılım geliştirme gibi alanlarda çeşitli çalışmalar gerçekleştirmektedir.

Eylül, 2026.

Çevrimiçi ortamda dağıtılmıştır.

www.akadal.tr

emre@akadal.tr

ISBN: **978-625-96238-5-6**

İnternet Teknolojileri Derneği (İNETD)

Yayıncı Kimliği: 76485

www.inetd.org.tr

info@inetd.org.tr

yayinevi@inetd.org.tr