LearnForgeX

YAPAY ZEKA / AI GELIŞTIRME ARAÇLARI

Claude Code ile AI Destekli Yazılım Geliştirme

Gerçek bir Spring Boot özelliğini, gerçek bir Claude Code terminal oturumunda analiz edip planlayın, uygulayın, test edin ve gözden geçirin: Plan Mode, izin modeli, ve gerçekten yakalanmış hatalarla.

Orta 35 dk

Son güncelleme: 22 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/developing-with-claude-code

---

Claude Code ile AI Destekli Yazılım Geliştirme

Bu derse kadar bu kursta AI'ı hep dışarıdan inceledik: ne olduğunu, nasıl çalıştığını, agent'ların nasıl planlayıp araç kullandığını. Bu ders farklı -- AI'ı bizzat bir geliştirme aracı olarak, kendi terminalinizde, gerçek bir projede kullanacaksınız. Claude Code, terminalde yaşayan, bir proje dizinini gerçekten okuyup analiz edebilen, kod yazıp dosya değiştirebilen, komut çalıştırabilen bir AI kodlama asistanıdır. Bu ders onu soyut olarak anlatmıyor -- gerçek bir Spring Boot özelliğinin (bu platformun kendisine bir Quiz özelliği eklemenin) gerçek bir Claude Code terminal oturumunda, baştan sona nasıl inşa edildiğini, adım adım izliyor.

> Bu derste gösterilen ekran metinleri (Plan Mode soruları, onay ekranları, dosya izin istemleri) gerçek bir Claude Code CLI oturumunda birebir gözlemlendi -- ama CLI'ın kesin arayüz metni ve akışı sürümden sürüme değişebilir. Buradaki amaç bir komut listesini ezberlemek değil, altındaki kalıcı iş akışını (analiz -> plan -> implementasyon -> test -> review -> git) öğrenmektir; kendi ortamınızda ekranlar biraz farklı görünse bile bu akış geçerliliğini korur.

Bu Derste Ne Yapacağız?

1. Claude Code'u bu platformun gerçek kod tabanında (`learning-platform` Spring Boot projesi) terminalden başlatacağız.
2. Ona gerçek bir görev vereceğiz: `enum` konusuna, 5 soruluk, TR+EN, çoktan seçmeli bir Quiz özelliği eklemek.

3. Kod yazmadan önce Plan Mode'un nasıl çalıştığını, netleştirici sorular sorduğunu, ve bir plan dosyası ürettiğini göreceğiz.
4. Planı onaylamadan önce nasıl okunacağını -- ve bunun neden önemli olduğunu, gerçekten yakalanmış bir hatayla -- göreceğiz.
5. İzin modelini (permission model) kullanarak dosyaları tek tek, diff diff onaylayarak implementasyonu izleyeceğiz.
6. Uygulamayı gerçekten çalıştırıp test edeceğiz -- ve burada da gerçek bir hata (kırık bir HTML etiketi) ve gerçek bir içerik kalitesi sorunu (hepsi aynı şıkka konmuş doğru cevaplar) bulup düzelteceğiz.
7. Son olarak, review ve git adımlarını, ve Claude Code'un izin/güvenlik modelini kapsayacağız.

Claude Code Nedir, Neden Farklı?

Muhtemelen bir sohbet penceresinde AI'a kod sorduğunuz olmuştur: bir soru sorarsınız, bir kod parçası alırsınız, onu kendiniz kopyalayıp projenize yapıştırırsınız. Claude Code bunun tersini yapar: proje dizininizde çalışan bir terminal aracıdır, projenizin dosyalarını **gerçekten okuyabilir** (sohbete siz yapıştırmadan), değişiklikleri **gerçekten dosyalara yazabilir**, ve gerekiyorsa **gerçekten komut çalıştırabilir** (`mvn test` , `npm install` , `git diff` gibi). "Ne yapacağını söyleyip kopyala-yapıştır yapmak" ile "bir görevi verip projenizde gerçekten çalışmasına izin vermek" arasındaki bu fark, bu dersin geri kalanının nedenidir -- çünkü bir araç dosyalarınızı gerçekten değiştirebiliyorsa, izin (permission) ve gözden geçirme (review) artık isteğe bağlı bir alışkanlık değil, kullanmanın ayrılmaz bir parçasıdır.

Kurulum ve Proje Dizininde Başlatmak

Kurulum adımları platforma göre değişir ve zamanla güncellenir -- güncel ve doğru talimat için resmi Claude Code kurulum belgelerine bakın. Kurulumdan sonra araç `claude` komutuyla başlatılır. Kritik nokta şu: **bunu proje dizininizin içinde** çalıştırırsınız --

```
cd learning-platform
claude
```

-- çünkü Claude Code, çalıştığı dizini kendi bağlamı olarak kullanır: hangi dosyaların var olduğunu, projenin nasıl yapılandığını, hangi diller/ framework'lerin kullanıldığını oradan çıkarır. Bu, sohbet tabanlı bir AI'a "işte projemin yapısı" diye elle anlatmanız gerekmemesinin sebebidir -- Claude Code kendi başlar ve gerektiğinde dosyaları kendisi okur.

Proje Bağlamı: CLAUDE.md'nin Rolü

Bir proje kökünde `CLAUDE.md` adlı bir dosya varsa, Claude Code'un resmi olarak belgelediği davranış bunu otomatik okuyup projeye özgü kalıcı bağlam olarak kullanmaktır -- mimari

kararlar, "kesinlikle değişmeyecek kurallar", daha önce keşfedilmiş kısıtlar, kod yazım konvansiyonları gibi (bu tür bir dosyanın tam okunma/önceliklendirilme mekanizması sürümden sürüme değişebilir, güncel ayrıntı için resmi belgelere bakmakta fayda var). Bu platformun kendi `CLAUDE.md`'si tam olarak bu işi görüyor: örneğin "Flyway migration numaraları sıralıdır ve geçmişe dönük asla değiştirilmez" ya da "iş mantığı Controller'da değil Service katmanında olmalı" gibi kurallar. Bu dosya olmadan, Claude Code her görevde bu konvansiyonları sıfırdan tahmin etmek zorunda kalırdı -- bazen doğru tahmin eder, bazen (bu dersin ilerleyen bir bölümünde göreceğiniz gibi) etmez. `CLAUDE.md`, bir insan yeni bir ekip arkadaşına "önce şu belgeyi oku" demesinin AI için karşılığıdır.

Adım 1 -- Analiz: Göreve Claude Code'u Sokmak

İş akışının ilk adımı, göreve doğrudan "şunu kodla" demek değil, önce neyin yapılacağını net bir şekilde anlatmaktır. Bu ders için terminale şuna yakın bir istem verildi (kısaltılmış):

```
Java Fundamentals kategorisindeki `enum` konusuna, 5 soruluk, TR+EN, çoktan
seçmeli bir Quiz özelliği eklemek istiyorum. DB şeması, endpoint sözleşmesi
ve mimari kararlar zaten netleşti (aşağıda). Önce projeyi analiz et, sonra
Plan Mode'da bir plan öner -- kod yazmaya HENÜZ başlama.
```

Buradaki iki şey önemli: görev **somut ve sınırlı** (tam olarak hangi konu, kaç soru, hangi diller -- "bir quiz sistemi yap" gibi belirsiz değil), ve istem açıkça **"kod yazmaya henüz başlama"** diyor. İkinci kısım, bir sonraki bölümün konusu olan Plan Mode'u tetikler.

Etkili Bir Görev Nasıl Verilir?

Yukarıdaki istem gelişigüzel yazılmadı -- bir görev tarifini üç seviyede düşünmek faydalı:

- **Kötü:** "Bir quiz sistemi yap." Hangi konu, kaç soru, hangi diller, hangi kısıtlar olduğunu Claude Code'un kendi başına tahmin etmesini gerektirir -- sonuç genellikle kapsamı gereksiz genişletir (istenmemiş attempt geçmişi, leaderboard, zamanlayıcı, admin arayüzü gibi özellikler).
- **Daha iyi:** "`enum` konusuna 5 soruluk, TR+EN, çoktan seçmeli bir quiz ekle. Önce analiz et, plan oluştur, kod yazma." Somut ve sınırlı, ama DB şeması/mimari kararları hâlâ Claude Code'a bırakıyor.
- **En iyi:** yukarıdaki gerçek istem gibi, **mevcut mimari + kapsam + kabul kriterleri (acceptance criteria) + kısıtlar** birlikte verilmiş bir görev. Bu ders için asıl istem, "DB şeması, endpoint sözleşmesi ve mimari kararlar zaten netleşti" satırıyla başlıyordu -- gerçek şema, gerçek endpoint sözleşmesi, gerçek katmanlama kararı, ve kapsam dışı bırakılan özelliklerin (attempt geçmişi, leaderboard, randomizasyon, timer, admin arayüzü) açık listesi önceden verilmişti. Claude Code'dan yalnızca bunu uygulamasını istedik -- DB şeması ya da mimari üzerine kendi başına karar vermesini değil.

Bu üçlü aslında bu dersin -- ve bu uygulamalı görevin planının -- nasıl ortaya çıktığının ta kendisi: kapsam ve kısıtlar önce insan tarafından netleştirildi, Claude Code'a "bir quiz sistemi tasarla" değil "bu tam kapsamı, bu kısıtlarla uygula" verildi.

Adım 2 -- Plan Mode: Kod Yazmadan Önce Planlamak

Plan Mode, Claude Code'un önemli özelliklerinden biridir: dosyaları okuyup analiz eder, ama hiçbir dosyayı **değiştirmeden** önce bir plan üretir -- ve o planı onaylamanızı bekler. Bu, "Bir AI Agent Nedir?"deki gözlemle-karar ver-eyleme geç loop'una bir insan onay noktası eklemenin somut hâlidir.

Plan Mode sırasında Claude Code, plana devam etmeden önce belirsiz kalan noktalarda size **netleştirici sorular** sorabilir. Bu oturumda gerçekten sorulan iki soru şöyleydi:

```
Quiz'in 5 sorusunun (TR+EN) içeriğini kim belirleyecek?
  1. Ben yazayım (Recommended)
  2. Sen (kullanıcı) yaz, ben yalnızca şemaya yerleştireyim
  3. ...
```

```
Kullanıcı quiz'i gönderdikten (submit) sonra tekrar cevap değiştirip
yeniden gönderebilmeli mi?
  1. Hayır, tek seferlik (Recommended)
  2. Evet, sınırsız
  3. ...
```

İki noktaya dikkat edin. Birincisi, sorular gerçekten **belirsiz** noktalar -- yani DB şeması /endpoint gibi zaten netleşmiş kararları tekrar sormuyor, yalnızca plana henüz girmemiş ayrıntıları soruyor. İkincisi, her seçeneğin yanında bir "(Recommended)" etiketi var -- ama bu etiket **"düşünmeden kabul et"** anlamına gelmez, yalnızca Claude Code'un kendi önerisidir. Bu oturumda her iki öneri de bağımsız olarak değerlendirilip (ilkinde "quiz içeriği ben yazarsam bile, yayınlanmadan önce mutlaka gözden geçirilmeli" notuyla, ikincisinde "zaten kalıcı bir attempt-geçmişi tutmuyoruz, o yüzden sayfa yenileme zaten aynı etkiyi veriyor, tek seferlik daha basit" gerekçesiyle) kabul edildi -- kör bir onay değil.

Sorular yanıtlandıktan sonra Claude Code planı günceller ve `/plan` komutuyla önizlemesini isteyebilirsiniz:

```
❯ /plan
#  Current Plan
   /Users/.../.claude/plans/{otomatik-üretilmiş-bir-isim}.md

   Quiz Özelliği -- enum Konusu İçin (Java Kursu, java-basics Kategorisi)
   ...
```

Plan, diskte gerçek bir Markdown dosyası olarak saklanır (`~/.claude/plans/` altında, göreve göre otomatik üretilen bir isimle) -- yani sohbetten kaybolan geçici bir metin değil, `ctrl+g` gibi bir kısayolla editörünüzde açıp inceleyebileceğiniz kalıcı bir belgedir.

İnsan Onayı: Ready to Code Kapısı

Bu oturumda gözlemlediğimiz davranış şuydu: Claude Code, planını yazmayı bitirdiğinde kod yazmaya otomatik geçmedi -- açık bir onay istedi:

```
Claude has written up a plan and is ready to execute.
Would you like to proceed?

  1. Yes, and use auto mode
> 2. Yes, manually approve edits
  3. Tell Claude what to change

ctrl+g to edit in VS Code
```

Bu ekran, "Agent Davranışını Kontrol Etmek"teki "Human-in-the-Loop: Riskli Eylemlerden Önce Onay" kavramının birebir canlı örneğidir -- bir agent'ın (burada Claude Code'un) riskli bir eyleme (dosya yazma/komut çalıştırma) geçmeden önce durup bir insandan onay istemesi. Üç seçenek gerçekten üç farklı otonomi seviyesi sunar: **auto mode**, plan onaylandıktan sonra tüm adımları arka arkaya, sizden tekrar sormadan yürütür; **manually approve edits**, her dosya değişikliğini/komutu tek tek onaylamanızı ister; **tell Claude what to change**, henüz koda hiç geçmeden planı düzeltmenizi sağlar.

Bu ders boyunca kasıtlı olarak **"manually approve edits"** seçildi -- auto mode değil. Sebep, henüz aşina olunmayan bir görevde daha güvenli bir varsayım olmasıdır (bkz. "Güvenlik ve İzinler"). Bu oturumda gerçekten iki sorun çıktı: planın kendisinde bir hata (bir sonraki bölümde), ve implementasyon sırasında bir HTML hatası (birkaç bölüm sonra). İlki -- migration numarası hatası -- hangi mod seçilirse seçilsin, planı onaylamadan ÖNCE okumakla yakalandı, çünkü bu adım mod seçiminden bağımsız gerçekleşir. İkincisi -- HTML hatası -- ise manuel onayla bile dosya diff'inde fark edilmedi, yalnızca uygulama gerçekten test edildiğinde ortaya çıktı. Yine de manuel onay, geri kalan tüm dosyaların mimari kararlara uygunluğunu tek tek doğrulama fırsatı verdi.

Planı Onaylamadan Önce Okumak: Gerçek Bir Hata Örneği

"(Recommended)" etiketine bile kör güvenmemek gerektiğini yukarıda söyledik -- işte tam olarak neden. Claude Code'un ürettiği plan, migration dosyalarını şöyle adlandırıyordu: `enum/V7__enum_quiz_questions.sql` ve `enum/V8__enum_quiz_options.sql`.

Bu, projenin gerçek dosya sistemiyle karşılaştırılınca yanlış çıktı. Flyway migration numaraları bu projede **klasörden bağımsız, tek bir global sıra** izler (`CLAUDE.md`'nin de belirttiği gibi, alt klasörler yalnızca dosya sistemi düzeyinde bir organizasyon) -- ve proje o sırada zaten V258'e ulaşmıştı. `enum/` klasörünün kendisi yalnızca V3-V6'ya sahipti, ama **V7 ve V8 numaraları proje genelinde zaten başka migration'lar tarafından**

kullanılıyordu (`core/V7__category_sort_order.sql` ve `record/V8__records_topic.sql`). Yani plan aynen uygulansaydı, Flyway büyük olasılıkla "birden fazla migration aynı versiyona sahip" hatasıyla başlangıçta çökerdi.

Bu, Claude Code'un "yetersiz" olduğunu göstermiyor -- tam tersine, plan her diğer açıdan (şema, izin modeli, endpoint sözleşmesi, katmanlama) kusursuzdu. Gösterdiği şey şu: bir AI, bir projenin **yerel bir konvansiyonunu** (burada: versiyon numaralarının klasör-bazlı değil proje-geneli olması) yanlış çıkarsayabilir, özellikle konvansiyon `CLAUDE.md` 'de açıkça yazılı olsa bile bağlamdan tam olarak içselleştirilmemişse. Bu yüzden plan, üçüncü seçenek ("Tell Claude what to change") kullanılarak düzeltme isteğiyle geri gönderildi -- kod hiç yazılmadan.

> Bir planı onaylamadan önce, iddia ettiği dosya adlarını/numaralarını gerçek proje durumuyla (`ls` , `find` , `grep`) karşılaştırmak, planın geri kalanını okumaktan çok daha ucuz ve çok daha etkili bir doğrulama adımıdır -- tam olarak bu bölümde olduğu gibi.

İzin Modeli: Dosya Dosya Onaylamak

Düzeltilmiş plan onaylandıktan sonra, bu oturumda "manually approve edits" modunda gözlemlediğimiz davranış şuydu: Claude Code her dosya için ayrı ayrı izin istedi -- dosyanın tam içeriğini gösterip üç seçenek sundu:

```
Do you want to create V259__quiz_schema.sql?
> 1. Yes
  2. Yes, and switch to accept edits (auto-approve file edits and common
     file commands) for this session (shift+tab)
  3. No
```

Bu oturumda her dosya için bilinçli olarak **"1. Yes"** seçildi, hiçbir zaman "2. auto-approve"a geçilmedi -- geri kalan tüm dosyaların (entity'ler, repository'ler, servis, controller, template, JS, migration'lar) her biri tek tek görülüp onaylandı; "Adım 3 -- Implementasyon: Diff'leri Takip Ederek İlerlemek" bölümünde göreceğiniz gibi, her dosyanın mimari kararlara (FK deseni, dil dönüştürücüsü, kolon eşlemesi) gerçekten uyup uymadığı tek tek kontrol edildi.

Ama dikkatli olun: manuel onay her hatayı yakalamaz. "Gerçek Bir Hata Ayıklama: Kırık Bir HTML Etiketi" bölümünde göreceğiniz eksik `>` karakteri, dosya diff'i onaylanırken fark edilmedi -- yalnızca uygulama gerçekten çalıştırılıp test edildiğinde ortaya çıktı. Bu da "Adım 4 -- Test: Çalıştırıp Elle Doğrulamak"ın neden ayrı, atlanamaz bir disiplin olduğunun kanıtı: diff okumak ve uygulamayı gerçekten çalıştırmak birbirinin yerine geçmez, ikisi farklı hata sınıflarını yakalar.

Adım 3 -- Implementasyon: Diff'leri Takip Ederek İlerlemek

Dosya dosya onaylarken, her diff'i bilinen mimari kararlara göre kontrol etmek mantıklıdır -- ezbere "Yes" demek yerine. Örneğin `QuizQuestion.java` onaya geldiğinde, kontrol edilen

noktalar şunlardı: `topic` alanı, projenin `CodeExample` varlığındaki FK deseniyle (`@ManyToOne(fetch = FetchType.LAZY)` + `@JoinColumn(name = "topic_id", nullable = false)`) birebir aynı mıydı; `language` alanına yanlışlıkla `@Enumerated` eklenmiş miydi (bu projede dil, ayrı bir `LanguageAttributeConverter` ile otomatik dönüştürülüyor, `@Enumerated` eklenirse bu mekanizma bozulur); `sort_order` alanı doğru kolon adına eşleniyor muydu. Hepsi doğruydu, dosya onaylandı -- ama bu, "her yeni entity için bu üç noktayı gerçekten kontrol ettikten sonra" onaylandı, dosyanın var olması yeterli görülmedi.

Gerçek Bir Hata Ayıklama: Kırık Bir HTML Etiketi

Migration'lar ve Java tarafı onaylanıp uygulama yeniden başlatıldığında, `/tr/topics/enum` sayfası şu hatayla çöktü:

```
org.attoparser.ParseException: (Line = 208, Column = 24) Malformed markup:
Attribute "class" appears more than once in element
```

Hata mesajının kendisi biraz yanıltıcıydı -- "class iki kez tanımlanmış" gerçek sorunu tam anlatmıyordu. Gerçek kök neden, `topic.html`'in 206. satırındaki `<nav>` açılış etiketinin **kapanış `>` karakterini kaybetmiş** olmasıydı:

```
<nav class="d-flex justify-content-between mt-5 pt-3 border-top"
     aria-label="Konu navigasyonu"
<a th:if="${previousTopic != null}"
   class="btn btn-outline-secondary text-start"
   ...
```

`>` eksik olduğu için parser, `<nav ...`'ın hâlâ açık olduğunu düşünüp sonraki satırlardaki `<a>` etiketinin kendi `class` attribute'unu da aynı (hâlâ kapanmamış) etiketin bir parçası sanmaya devam etti -- iki ayrı `class` attribute'u (biri `nav`'a, biri `a`'ya ait) aynı etikette çakıştı. Düzeltme tek bir karakterdi: 206. satırın sonuna eksik `>`'yi eklemek.

Bu, iki ayrı beceriyi gösteriyor. Birincisi, **gerçek bir stack trace'i okuyup** hatanın (attoparser'ın raporladığı satır/sütun) gerçek kaynağına inmek -- hata mesajının kelimesi kelimesine söylediğine değil, ne anlama geldiğine bakmak. İkincisi, düzeltmeyi Claude Code'a **tam ve spesifik** bir talimatla iletmek ("şurada bir şeyler bozuk, düzelt" değil, "206. satırdaki `<nav>` etiketinin kapanış `>`'si eksik, ekle") -- bu, hem daha hızlı hem daha güvenilir bir düzeltme sağlar.

Adım 4 -- Test: Çalıştırıp Elle Doğrulamak

HTML düzeltildikten sonra uygulama başarıyla çalıştı ve `/tr/topics/enum` sayfasında 5 soruluk quiz gerçekten render oldu: her soru 4 radio-button şıkla, tümü cevaplanana kadar devre dışı kalan bir "Gönder" butonuyla. Tüm sorular cevaplanıp gönderildiğinde skor, doğru/yanlış işaretleme, ve her sorunun açıklaması gerçekten göründü.

Bir özelliğin "çalışıyor görünmesi" ile "doğru çalışması" arasındaki fark, tam olarak bir sonraki bölümün konusu.

Adım 5 -- Review: Çalışıyor ile Doğru Aynı Şey Değildir

İlk testte quiz teknik olarak kusursuz çalıştı -- skor doğru hesaplandı, partial unique index bozulmadı, her soruda tam bir doğru şık vardı. Ama içeriğe daha yakından bakıldığında gerçek bir sorun ortaya çıktı: **5 sorunun 5'inde de doğru şık, ekranda hep ilk sırada (A şıkkı)** yer alıyordu. Teknik olarak hatasız, ama pedagojik olarak anlamsız -- bir öğrenci hiç okumadan her soruda "A" işaretleyip 5/5 alabilirdi.

Bu, "Planı Onaylamadan Önce Okumak: Gerçek Bir Hata Örneği" bölümündeki migration numarası hatasıyla aynı derste öğretilen dersi tekrarlıyor: bir AI'ın önerdiği içerik/plan **çalışıyor olması**, onun **doğru olduğu** anlamına gelmez -- özellikle içerik yazarlığı ("Ben yazayım (Recommended)") bilinçli olarak AI'a bırakıldığında bu ayrım daha da kritik hâle gelir.

Düzeltme, dikkatli bir ayrım gerektiriyordu: **hangi şıkkın doğru olduğu** değişmemeliydi (her şıkkın metni belirli bir doğruluk iddiası taşıyor, "doğru"yu yanlış bir ifadeye taşımak dersi yanlış hâle getirirdi) -- yalnızca **doğru şıkkın ekrandaki konumu** değişmeliydi.

Düzeltme: Yeni Bir Migration mı, Var Olanı mı Değiştirmeli?

Sorunlu migration'lar (`V260`, `V261`) uygulama zaten çalışıp quiz'i render ettiğine göre, Flyway tarafından uygulanmış ve `flyway_schema_history` tablosuna kaydedilmiş durumdaydı. `CLAUDE.md`'nin "Flyway migration numaraları sıralı ve geçmişe dönük asla değiştirilmez" kuralı tam olarak bu duruma cevap veriyor: uygulanmış bir migration'ı yerinde düzenlemek yerine, düzeltme **yeni bir migration** (`V262`) olarak eklendi -- her sorunun 4 şıkkının `sort_order` değerlerini güncelleyen, `is_correct`'e hiç dokunmayan bir dizi `UPDATE`.

Uygulama yeniden başlatılıp quiz tekrar denendiğinde, doğru şıklar artık farklı konumlardaydı (örneğin bir soruda B, başka birinde C) -- ve skor hâlâ doğru şekilde 5/5 hesaplanıyordu, çünkü değişen yalnızca görüntüleme sırasıydı, puanlama mantığı (`is_correct` karşılaştırması) hiç etkilenmemişti.

Adım 6 -- Git: Değişiklikleri Gözden Geçirip Commit Etmek

İş akışının son adımı, dosyaları "çalışıyor" bulduktan sonra bile hemen commit etmek değildir. `git status` ve `git diff` ile hangi dosyaların değiştiğini/eklendiğini görmek, commit'ten önceki son gözden geçirme fırsatıdır -- IDE'nizin (örn. IntelliJ) "Changes" görünümü de aynı işi görsel olarak yapar. Claude Code'a "değişiklikleri commit et" demeden önce bu listeyi kendiniz taramak, buraya kadarki tüm dosya-dosya onay disiplininin doğal bir devamıdır: bir agent'ın önerdiği bir commit mesajı bile, tıpkı önerdiği bir plan gibi, gözden geçirilecek bir taslaktır. İyi bir commit mesajı ne yaptığınızı değil (diff zaten bunu gösterir) **neden** yaptığınızı özetler; ve büyük bir özelliği tek dev bir commit yerine mantıklı parçalara

(örn. şema + seed verisi, entity/repository/servis, controller + template + JS) bölmek, ileride bir hatayı `git bisect` ile izlemeyi kolaylaştırır.

Güvenlik ve İzinler

Claude Code dosya yazabildiği ve komut çalıştırabildiği için, izin modelini bilinçli kullanmak bu aracın güvenli kullanımının merkezindedir. "Agent Davranışını Kontrol Etmek"teki "Tool Erişimini Kapsamlandırmak: En Az Yetki İlkesi" burada da geçerli: bir dizini yeni tanıyorsanız, ya da görev riskli (veritabanı şeması, silme işlemleri, dış servislere istek) içeriyorsa, **"manually approve edits"** ile başlamak, "auto mode"a güvenmekten daha güvenlidir. Auto mode, defalarca kullandığınız, sonuçlarına aşina olduğunuz, düşük riskli görevler için zaman kazandırır -- ama bu dersin kendisinin gösterdiği gibi, yeni bir görevde plan hataları ve implementasyon hataları gerçekten olur, ve manuel onay tam olarak bunları yakalayan mekanizmadır. Kabaca bir kural: bir komutun ya da dosya değişikliğinin sonucunu geri alması zor/pahalıysa (bir migration'ı production'a uygulamak, bir dosyayı silmek, bir `git push --force`), o adımı asla otomatik onaya bırakmayın.

Diğer Claude Code Özellikleri: Kısaca

Bu ders boyunca kullanılan analiz-plan-onay-implementasyon-test-review-git akışı, Claude Code'un çekirdeğidir -- ama araç birkaç ek yetenek de sunar, aynı temel fikrin (bağlamı genişletmek, riskli adımlarda insanı devrede tutmak) uzantıları olarak: proje köküne konan `CLAUDE.md` gibi dosyalar kalıcı bağlam sağlar ("Proje Bağlamı: CLAUDE.md'nin Rolü"nde görüldüğü gibi); **Subagent**'lar, büyük bir görevi bağımsız alt-görevlere bölüp paralel çalıştırmayı sağlar; **MCP (Model Context Protocol)**, "TypeScript ile MCP Sunucusu Oluşturma"da görülen aynı protokolle Claude Code'un dış araçlara/servislere bağlanmasını sağlar; **Skills**, tekrar eden görev kalıplarını paketleyip yeniden kullanılabilir hâle getirir. Bu dersin kapsamı, bunların her birine derinlemesine girmek değil -- hepsinin, bu derste elle yürüttüğünüz aynı temel disiplini (bağlamı ver, planı gözden geçir, riskli adımı onayla) farklı ölçeklerde tekrarladığını görmektir.

Best Practices

- Bir göreve başlamadan önce Claude Code'a "önce analiz et ve Plan Mode'da bir plan öner, kod yazmaya henüz başlama" demek -- bkz. "Adım 1 -- Analiz: Göreve Claude Code'u Sokmak".
- Görevi mevcut mimari + kapsam + kabul kriterleri + kısıtlarla birlikte vermek, yalnızca "şunu yap" demek yerine -- bkz. "Etkili Bir Görev Nasıl Verilir?".
- Bir plan onaylanmadan önce, iddia ettiği dosya adlarını/varsayımlarını gerçek proje durumuyla karşılaştırmak -- bkz. "Planı Onaylamadan Önce Okumak: Gerçek Bir Hata Örneği".
- Yeni/riskli bir görevde "manually approve edits" ile başlamak, yalnızca aşina olunan düşük riskli görevlerde auto mode'a geçmek -- bkz. "Güvenlik ve İzinler".

- Bir hatayı düzeltmesini istemeden önce Claude Code'a tam ve spesifik bir teşhis vermek (hangi dosya, hangi satır, kök neden) -- bkz. "Gerçek Bir Hata Ayıklama: Kırık Bir HTML Etiketi".
- "Çalışıyor" testinden sonra ayrı bir "doğru mu" gözden geçirmesi yapmak, özellikle içerik /mantık bir AI tarafından üretildiyse -- bkz. "Adım 5 -- Review: Çalışıyor ile Doğru Aynı Şey Değildir".
- Uygulanmış bir migration'ı asla yerinde düzenlememek, düzeltmeyi her zaman yeni bir migration olarak eklemek -- bkz. "Düzeltme: Yeni Bir Migration mı, Var Olanı mı Değiştirmeli?".

Yaygın Hatalar

- **"(Recommended)" etiketine ya da bir planın "makul göründüğüne" bakarak kör onay vermek.** "Planı Onaylamadan Önce Okumak: Gerçek Bir Hata Örneği" bölümündeki migration numarası çakışması, tam olarak bu şekilde geçilseydi uygulamanın başlangıçta çökmesine yol açardı.
- **Bir özelliği yalnızca "hatasız çalıştı" diye tamamlanmış saymak.** "Adım 5 -- Review: Çalışıyor ile Doğru Aynı Şey Değildir" bölümünde görüldüğü gibi, quiz teknik olarak kusursuz çalışıyordu ama içerik pedagojik olarak anlamsızdı -- ayrı bir "doğru mu" gözden geçirmesi olmasaydı bu hiç fark edilmezdi.
- **Dosya diff'ini onaylamayı, uygulamayı gerçekten çalıştırıp test etmenin yerine koymak.** "İzin Modeli: Dosya Dosya Onaylamak" bölümünde görüldüğü gibi, `topic.html` 'deki eksik `>` karakteri dosya onaylanırken fark edilmedi -- yalnızca "Adım 4 -- Test: Çalıştırıp Elle Doğrulamak" adımında, uygulama gerçekten çalıştırılınca ortaya çıktı. Diff okumak ve gerçekten çalıştırıp test etmek, birbirinin yerine geçmeyen iki ayrı disiplindir.
- **Uygulanmış bir migration'ı bulunca hemen yerinde düzenlemek.** Bu, Flyway'in checksum doğrulamasını bozar ve `CLAUDE.md`'nin "geçmişe dönük asla değiştirilmez" kuralını ihlal eder -- doğru yol her zaman yeni bir migration eklemektir.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Claude Code, proje dizininizde çalışan, dosyaları gerçekten okuyup yazabilen, komut çalıştırabilen bir terminal AI kodlama asistanıdır -- bir sohbet penceresine kopyala-yapıştır yapmaktan temel olarak farklıdır.
- Kalıcı iş akışı altı adımdır: analiz et, Plan Mode'da planla, insan onayından geçir, implementasyonu dosya dosya onaylayarak izle, test et, review edip commit et.
- İyi bir görev tarifi mevcut mimari + kapsam + kabul kriterleri + kısıtları birlikte verir -- "bir quiz sistemi yap" gibi belirsiz bir görev, Claude Code'un kapsamı kendi başına (ve genellikle gereğinden geniş) tahmin etmesine yol açar.

- Plan Mode, kod yazmadan önce bir plan üretir ve belirsiz noktalarda netleştirici sorular sorar -- "(Recommended)" etiketi kör onay için değil, değerlendirilecek bir öneri içindir.
- "Ready to code?" onay kapısı üç otonomi seviyesi sunar (auto mode / manuel onay / planı değiştir) -- bu, "Agent Davranışını Kontrol Etmek"teki human-in-the-loop kavramının canlı bir örneğidir.
- Bu derste gerçekten yakalanan üç sorun -- migration numarası çakışması (plan hatası), kırık bir HTML etiketi (implementasyon hatası), tüm doğru cevapların aynı şıkka konması (içerik kalitesi hatası) -- her biri farklı bir gözden geçirme disiplinini gösteriyor: planı okumak, hata çıktısını okumak, ve "çalışıyor"u "doğru"dan ayırmak.
- Uygulanmış bir migration asla yerinde düzenlenmez -- düzeltme her zaman yeni bir migration'dır.

Cheat Sheet

- Başlatma: proje dizininde `claude`.
- Bağlam: proje kökündeki `CLAUDE.md`, otomatik okunan kalıcı bağlam.
- İş akışı: analiz -> Plan Mode -> "Ready to code?" onayı -> dosya-dosya implementasyon onayı -> test -> review -> git.
- Onay seviyeleri: "Yes" (bu dosya/adım) / "Yes, auto-approve" (bu oturum için tümü) / "No" (reddet) / "Tell Claude what to change" (planı düzelt).
- Güvenli varsayılan: yeni/riskli görevde manuel onay; yalnızca aşina olunan düşük riskli görevde auto mode.
- Migration kuralı: sıralı, global, geçmişe dönük asla değiştirilmez -- düzeltme her zaman yeni bir dosya.

Terimler Sözlüğü

- **Claude Code:** terminalde çalışan, bir proje dizinini bağlam olarak kullanan, dosya okuyup/yazabilen ve komut çalıştırabilen AI kodlama asistanı.
- **Plan Mode:** Claude Code'un, herhangi bir dosyayı değiştirmeden önce bir plan ürettiği ve onay beklediği çalışma modu.
- **CLAUDE.md:** bir proje kökünde bulunduğunda Claude Code tarafından otomatik okunan, projeye özgü kalıcı bağlam/kural dosyası.
- **İzin modeli (permission model):** Claude Code'un dosya değişikliği/ komut çalıştırma öncesinde sunduğu onay seçenekleri (tek adım, oturum boyunca otomatik, reddet, planı değiştir).
- **Auto mode:** onaylanan plandaki tüm adımların, her adımda tekrar sormadan arka arkaya yürütüldüğü otonomi seviyesi.
- **Review (gözden geçirme):** bir özelliğin yalnızca "çalıştığını" değil, "doğru" olduğunu (içerik, mantık, güvenlik açısından) ayrıca doğrulama adımı -- test etmekten farklı ve ondan sonra gelir.