



این فایل، کتابچه آموزشی «آموزش گیت Git، گیت هاب و گیت لب (رایگان) + گواهینامه» است و شما در حال مطالعه مرور و خلاصه کل آموزش هستید. این نسخه با هدف جمع‌بندی سریع کلیه دروس، مفاهیم کلیدی، نکات مهم، و مسیرهای پیشنهادی ادامه یادگیری تهیه شده است. توجه داشته باشید که برخی امکانات تعاملی نسخه کامل آموزش، مانند ویدئوها، تمرينها و آزمونها، پاسخهای تشریحی، فایل‌های ضمیمه و بهروزرسانی‌های دوره، در این کتابچه در دسترس نیست. برای مشاهده نسخه کامل آموزش و بهره‌مندی از همه امکانات، به صفحه دوره در این [لينك](#) مراجعه کنید.

## خلاصه و مرور آموزش

# آموزش گیت Git، گیت هاب و گیت لب (رایگان) + گواهینامه

## آموزش گیت Git، گیت هاب و گیت لب (رایگان)

### مقدمات گیت و نصب آن

#### مقدمه‌ای بر گیت و اهمیت یادگیری آن

گیت (Git) یک ابزار بنیادی در دنیای توسعه نرم‌افزار و مدیریت پروژه محسوب می‌شود. در بسیاری از شرکت‌ها و تیم‌های توسعه، تسلط بر گیت نه تنها یک مزیت، بلکه یک ضرورت بهشمار می‌رود. حتی برنامه‌نویسانی که به تنها‌یی کار می‌کنند، برای مدیریت بهتر پروژه‌های شخصی خود به گیت نیاز دارند. امروزه داشتن پروژه‌ها و فعالیت‌های ثبت‌شده در سامانه‌هایی مانند گیت‌هاب (GitHub) بخشی از رزومه شغلی برنامه‌نویسان است و مورد توجه کارفرمایان قرار می‌گیرد.

#### تاریخچه و کاربردهای گیت

گیت توسط لینوس توروالدز، خالق هسته لینوکس، طراحی و توسعه بافته است. هدف اولیه آن مدیریت نسخه‌های هسته لینوکس بود، اما به سرعت به ابزاری جهانی و پرکاربرد تبدیل شد. گیت به عنوان یک سیستم کنترل نسخه توزیع شده (Distributed Version Control System) نه تنها در پروژه‌های نرم‌افزاری، بلکه در نگارش و مدیریت کتاب، پایان‌نامه، مقالات و سایر پروژه‌های مبتنی بر فایل نیز کاربرد دارد.

#### مفهوم کنترل نسخه و مشکلات رایج بدون گیت

بدون استفاده از سیستم‌های کنترل نسخه مانند گیت، مدیریت پروژه‌های چندفایلی یا چندنفره می‌تواند به آشفته‌یی منتهی شود. نمونه‌هایی از این مشکلات عبارتند از:

- ایجاد پوشه‌های متعدد با نام‌های مانند "New Folder"، "Final"، یا نام‌گذاری بر اساس تاریخ برای نگهداری نسخه‌های مختلف پروژه.
- دشواری در بازگشت به نسخه قبلی پس از ایجاد اشتباه یا بروز مشکل در نسخه فعلی.
- ناتوانی در تشخیص اینکه آخرین نسخه‌ی سالم کدام است.
- تداخل و همپوشانی کار اعضاي تیم در صورت همکاری چند نفر بر روی یک پروژه.

گیت با ارائه راهکاری ساخت‌یافته، این مشکلات را برطرف می‌کند و امکان ثبت دقیق تغییرات، همکاری همزمان چند کاربر و بازگشت سریع به نسخه‌های قبلی را فراهم می‌آورد.

## نصب و راهاندازی گیت

برای استفاده از گیت، ابتدا لازم است این ابزار را بر روی سیستم‌عامل مورد نظر نصب کنید. بسته به سیستم‌عامل مراحل متفاوتی وجود دارد:

- **ویندوز:** مراجعه به سایت رسمی Git SCM و دریافت نسخه مخصوص ویندوز.
- **مک:** دریافت نسخه مخصوص مک از سایت Git SCM.
- **لینوکس:** استفاده از دستوراتی مانند `apt-get install git` در توزیع‌های مبتنی بر دبیان، یا `yum install git` و `dnf install git` در سایر توزیع‌ها.

پس از نصب، می‌توان با اجرای دستور `git` در محیط خط فرمان یا ترمینال از نصب صحیح اطمینان حاصل کرد. در صورت نصب موفق، دستورات و راهنمایی مربوط به گیت نمایش داده می‌شوند و پیام خطا مشاهده نخواهد شد.

## عملکرد، مزایا و قابلیت‌های گیت

گیت پروژه‌ها را به صورت سلسله‌ای از نسخه‌ها مدیریت می‌کند. کاربر می‌تواند پس از تکمیل هر مرحله‌ی مهم در توسعه پروژه، یک نسخه جدید ثبت کند. در صورت بروز اشکال یا نیاز به بازگشت، امکان رجوع به نسخه قبلی وجود دارد. برخی از نکات کلیدی عملکرد گیت عبارتند از:

- نگهداری تاریخچه تغییرات برای هر فایل و کل پروژه.
- امکان ثبت شاخه‌های مختلف (Branches) برای کار بر روی ویژگی جدید یا رفع باگ، بدون آسیب به نسخه اصلی پروژه.
- ترکیب تغییرات از شاخه‌های مختلف با استفاده از فرآیند Merge.
- قابلیت بازگشت به هر نقطه از تاریخچه و بازیابی وضعیت قبلی پروژه.
- تسهیل همکاری همزمان چند نفر بر روی بخش‌های مختلف یک پروژه و ادغام تغییرات آن‌ها.
- شناسایی دقیق اینکه در چه زمانی، چه کسی و چه تغییری در کجا اعمال کرده است.

این ویژگی‌ها باعث شده گیت ابزاری قدرتمند برای کنترل پروژه‌های گوناگون در مقیاس‌های مختلف باشد.

## جمع‌بندی و توصیه عملی

یادگیری و بهکارگیری گیت برای تمام برنامه‌نویسان و مدیران پروژه امری حیاتی است. توصیه می‌شود کاربران پیش از ادامه آموزش، نسخه مناسب گیت را متناسب با سیستم‌عامل خود نصب نموده و از عملکرد صحیح آن اطمینان حاصل کنند. وجود گیت بر روی هر

رايانه‌اي که فعالیت حرفه‌اي در آن انجام می‌شود، ضروري است و مسیر حرفه‌اي کاربران را به صورت قابل توجهی بهبود خواهد بخشید.

## اولین اينيت (init) و اولين کاميit

### مقدمه‌اي بر استفاده از Git و مدیريت مخزن

در آغاز کار با Git، فرض بر اين است که نرم‌افزار Git روی سیستم نصب شده باشد. Git ابزاری قدرتمند برای مدیریت نسخه و کنترل تغیيرات فایل‌ها در پروژه‌های نرم‌افزاری است. هر بار که Git اجرا می‌شود، وضعیت فایل‌های موجود در مسیر جاري را بررسی کرده و تشخیص می‌دهد که آیا فایل‌ها قبلًا دیده شده‌اند، تغیير کرده‌اند یا فایل جدیدی ایجاد شده است.

### آغاز پروژه با Git

برای شروع یک پروژه جدید با Git، نخست باید یک دایرکتوری جدید ساخته و وارد آن شد. سپس با اجرای دستور زیر، مخزن Git در آن دایرکتوری ایجاد می‌شود:

```
bash
```

```
git init
```

این دستور یک مخزن جدید Git در دایرکتوری فعلی راه‌اندازی می‌کند و پوشه‌ای مخفی به نام `git` ایجاد می‌نماید که اطلاعات داخلی خود را در آن ذخیره می‌کند. فایل‌ها و پوشه‌هایی که نامشان با نقطه (.) آغاز می‌شود، در محیط‌هایی مانند لینوکس و مک به طور پیش‌فرض پنهان هستند.

### بررسی وضعیت فایل‌ها

برای مشاهده وضعیت فعلی مخزن و فایل‌ها، می‌توان از دستور زیر استفاده کرد:

```
bash
```

```
git status
```

این دستور نشان می‌دهد که در حال حاضر روی کدام شاخه (branch) هستید، وضعیت تغیيرات به چه صورت است و چه فایل‌هایی به Git اضافه نشده‌اند یا تغیير یافته‌اند.

### مفاهیم کلیدی: Stage و Commit

بر اساس دو مفهوم اصلی `stage` و `commit` کار می‌کند:

▪ مرحله‌ای است که تغیيرات فایل‌ها برای ذخیره‌سازی آماده می‌شوند.

▪ عملی است که تغیيرات آماده‌شده (staged) را ثبت و در مخزن ذخیره می‌کند.

فرآيند معمول کار با فایل‌ها چنین است:

۱. ایجاد یا ویرایش یک فایل در دایرکتوری پروژه.

۲. افزودن فایل تغیير یافته به مرحله `stage`، مثلًا با دستور:

۳. ثبت تغیيرات مرحله `stage` با دادن یک پیام توضیحی مناسب:

### مثال عملی

فرض کنید در حال ساخت یک سایت ساده هستید:

۱. یک فایل جدید به نام `index.html` ایجاد می‌شود و با یک ویرایشگر متنی، محتوا به آن افزوده و ذخیره می‌گردد.
۲. پس از ذخیره فایل، با اجرای `git status` مشاهده می‌شود که فایل یادشده "untracked" است، یعنی Git تاکنون آن را ردگیری نکرده است.
۳. با استفاده از دستور `git add index.html` این فایل به فضای `stage` اضافه می‌شود.
۴. مجدداً با `git status` می‌توان دید که فایل آماده `commit` شده است.
۵. سرانجام با دستور `git commit -m "index file created"` فایل به صورت رسمی در مخزن ثبت می‌شود.

### بهروزرسانی و ثبت تغییرات فایل‌ها

اگر در آینده فایل `index.html` ویرایش شود، مراحل زیر تکرار می‌گردد:

- ابتدا با `git status` مشاهده می‌شود که فایل ویرایش شده و در وضعیت `unstaged` قرار دارد.
- افزودن تغییرات به `stage` با دستور: `git add -A` یا برای افزودن تمامی تغییرات:
- انجام `commit` با پیام مناسب:

پس از انجام `commit`، اجرای مجدد `git status` نشان می‌دهد که هیچ تغییری برای `commit` باقی نمانده است و مخزن در حالت پایدار قرار دارد.

### خلاصه

فرآیند معمول کار با Git هنگام انجام تغییرات در پروژه به شرح زیر است:

۱. ایجاد یا ویرایش فایل‌های پروژه.
۲. افزودن فایل‌ها به مرحله `stage` با `git add`.
۳. ثبت تغییرات با `git commit -m "پیام توضیحی"`.

تکرار و تمرین این مراحل به درک دقیق‌تر نحوه عملکرد Git و مدیریت مؤثر نسخه‌ها در پروژه‌های نرم‌افزاری کمک خواهد کرد.

## بررسی تاریخچه کارها

### مروری بر عملیات‌های پایه‌ای Git

در این بخش، به مرور مباحث مطرح شده در مرحله دوم و بررسی عملیات پایه‌ای در Git پرداخته می‌شود. ابتدا یک مخزن جدید با استفاده از دستور `git init` در شاخه مورد نظر ایجاد می‌گردد. سپس یک فایل جدید با نام `index` ساخته و به مخزن افزوده می‌شود. فایل مذکور حداقل دو مرتبه مورد `commit` قرار می‌گیرد. برای مشاهده سوابق `commit`‌ها می‌توان از دستور `git log` بهره برد که فهرستی از تمامی `commit`‌ها به همراه یک شناسه (کد یونیک) برای هر کدام ارائه می‌کند. به عنوان مثال، یک `commit` با پیام "index file created" قابل مشاهده است.

## ایجاد و مدیریت فایل‌های جدید در Git

در ادامه، مراحل توسعه‌ی پروژه با ایجاد سه فایل جدید با نامهای `page3.html` ، `page2.html` ، `page1.html` به روش زیر پیش می‌رود: - ایجاد فایلهای جدید توسط دستور `touch` . - باز کردن فایلهای در یک ویرایشگر متنی و افزودن محتوای مناسب (مانند عبارت "you are on page2" در فایل دوم و "you are on page3" در فایل سوم). - ذخیره نمودن تغییرات در هر فایل.

پس از ایجاد این سه فایل، با اجرای دستور `git status` ، مشاهده می‌شود که این فایلهای به صورت untracked ظاهر می‌شوند، زیرا هنوز در Git ردیابی نمی‌شوند. به طور کلی، فایلهای تازه کاربر به محیط staging افزوده نشوند، غیرقابل پیگیری محسوب می‌شوند.

## مراحل Git در Staging و Add

پس از ویرایش یا اضافه کردن فایلهای، باید آنها را با دستور `git add` به محیط staging منتقل نمود. در این مرحله، فایلهای از حالت tracked به untracked تغییر وضعیت می‌دهند و آماده commit شدن می‌شوند. در صورتی که فایلی (مانند `index`) ویرایش شود اما به staging اضافه نگردد، Git تغییرات جدید را نیز با اجرای مجدد `git status` به صورت تغییرات انجام شده اما نشده نمایش می‌دهد.

می‌توان عملیات افزودن (`add`) و commit را به صورت جداگانه برای هر فایل انجام داد. مثلاً امکان افزودن فقط برخی از فایلهای وجود دارد و commit مربوطه تنها همان فایلهای شامل می‌شود. همچنین می‌توان با استفاده از گزینه‌های ارائه شده، تمام تغییرات موجود را به صورت یکجا یا تفکیکی به staging و سپس commit منتقل کرد.

## بررسی وضعیت مخزن و سوابق تغییرات

همواره می‌توان با اجرای دستور `git status` ، وضعیت فعلی مخزن را مشاهده کرد. این دستور راهنمای مفیدی برای آگاهی از فایلهای staged و تغییرات آماده برای commit می‌باشد. پس از انجام commit‌ها، با بهره‌گیری از `git log` تمامی commit‌ها قابل مشاهده خواهد بود. علامت `head` در این لیست نشان‌دهنده موقعیت فعلی بر روی جدیدترین commit است. حرکت به میان commit‌های پیشین و مشاهده تاریخچه یکی از ویژگی‌های قدرتمند Git است که در مراحل پیشرفت‌تر نیز بررسی خواهد شد.

## اهمیت یادگیری تدریجی Git

یادگیری Git روندی تدریجی و پویا است. هیچ کس توانایی تسلط کامل بر تمام جنبه‌های این ابزار بزرگ و پیچیده را ندارد؛ حتی سازندگان Git نیز ممکن است در بخش‌هایی نیاز به مراجعه مجدد به مستندات یا جستجو داشته باشند. بهترین روش برای یادگیری Git، شروع از مفاهیم و دستورات پایه است تا فرصت کسب تجربه عملی حاصل گردد. تسلط نسبی و کاربردی بر Git برای شروع کار، به ویژه برای افراد تازه‌کار، کافی است و آشنایی با ویژگی‌های پیشرفت‌هه می‌تواند در آینده صورت گیرد. همکاری و همفکری با دیگر کاربران و جستجوی راه حل در هنگام مواجهه با مشکلات، بخشی جدایی‌ناپذیر از کار با Git و توسعه نرم‌افزار محسوب می‌شود.

## بررسی تغییرات انجام شده

### کار با Git: بررسی تغییرات و مدیریت Stage

در این درس با ابزارها و دستورات کلیدی برای بررسی و مدیریت تغییرات فایلهای در سیستم کنترل نسخه Git آشنا می‌شویم. این دستورات برای مدیریت بهتر پروژه‌ها و پیگیری دقیق تغییرات بسیار ضروری هستند.

## مراحل ابتدایی کار با Git

در مراحل اولیه کار با Git، معمولاً اقدامات زیر انجام می‌شود:

- ایجاد مخزن جدید با `git init`:
- اضافه کردن فایل‌ها به مرحله `staging` با `git add`:
- ثبت تغییرات توسط `git commit` به همراه پیام توضیحی با `-m .option`.

در صورت ننوشتن پیام، Git از کاربر درخواست می‌کند که پیام مربوط به `commit` را وارد کند.

## مفهوم Head و مشاهده تاریخچه تغییرات

یکی از امکانات مهم Git قابلیت بازگشت به عقب یا جلو در تاریخچه commit‌ها است. دستور `git log` برای مشاهده تاریخچه‌ها استفاده می‌شود. در اینجا، `Head` به آخرین `commit` موجود اشاره دارد. این ویژگی به کاربر اجازه می‌دهد وضعیت فعلی پروژه را با آخرین `commit` مقایسه کند.

## مشاهده وضعیت فایل‌ها با `git status`

دستور `git status` اطلاعاتی درباره وضعیت فعلی فایل‌ها ارائه می‌دهد؛ به ویژه تغییراتی که هنوز `commit` نشده‌اند یا فایل‌هایی که به مرحله `staging` افزوده شده‌اند. برای مثال، اگر فایلی تغییر یابد، این دستور آن تغییر را نمایش می‌دهد.

## بررسی تغییرات با `git diff`

برای مشاهده جزئیات تغییرات ایجادشده در فایل‌ها نسبت به آخرین `commit`، از دستور `git diff HEAD` استفاده می‌شود. این دستور بدین صورت عمل می‌کند:

- مقایسه وضعیت فعلی فایل‌ها با آخرین `commit` ثبت‌شده (Head).
- نمایش اختلاف خطوط بین نسخه قبلی و فعلی هر فایل.
- خطوط حذف شده با علامت منفی (-) و خطوط جدید با علامت مثبت (+) مشخص می‌شوند.

خواندن خروجی `git diff` (معمولًا با فرمت `diff -u` یا `unified diff`) مهارتی است که با تمرین قابل تسلط است. این خروجی به کاربر نشان می‌دهد که کدام خطوط در هر فایل تغییر یافته‌اند.

## بررسی تغییرات چند فایل

در صورتی که تغییرات در چند فایل ایجاد شده باشد، `git diff` تمام فایل‌های تغییر یافته به مرحله `staging` منتقل می‌شوند. برای هر بخش، نسخه قبلی و نسخه فعلی خطوط نشان داده می‌شود تا کاربر بتواند تغییرات دقیق را بررسی کند.

## انتقال همه تغییرات به Stage

با استفاده از دستور `-A` از `git add` یا `-a` از `git add`، تمام فایل‌های تغییر یافته به مرحله `staging` منتقل می‌شوند. در این حالت، تغییرات آماده `commit` شدن هستند، اما هنوز `commit` نشده است.

## بررسی تغییرات مرحله staged

برای مشاهده تغییرات فایل‌هایی که به مرحله `staging` اضافه شده‌اند (و نه تغییرات همه فایل‌های پروژه)، می‌توان از دستور `git diff-staged` استفاده کرد. این دستور مشخص می‌کند کدام تغییرات قرار است در `commit` بعدی ثبت شوند.

## خارج کردن فایل از Stage

اگر پس از بررسی متوجه شوید که برخی فایل‌ها نباید stage شوند، می‌توان آنها را با دستور git reset از مرحله staging خارج کرد. پس از اجرا، آن فایل دیگر آماده commit شدن نیست و به حالت unstaged باز می‌گردد.

### بازگردانی تغییرات یک فایل به آخرین وضعیت commit

در شرایطی که تغییرات اعمال شده روی یک فایل ناخواسته باشد و نیاز به بازگشت به آخرین نسخه ثبت شده داشته باشید، از دستور git checkout – استفاده می‌شود. این دستور محتويات فایل مورد نظر را به نسخه موجود در آخرین (Head) commit بازمی‌گرداند و همه تغییرات فعلی آن را حذف می‌کند.

### خلاصه دستورات کلیدی و کاربردی

- .commit: مشاهده تفاوت فایل‌های فعلی با آخرین commit

- .commit: مشاهده تفاوت فایل‌های staged با آخرین commit

- .staging: خارج کردن یک فایل از مرحله staging

- --git checkout: بازگرداندن فایل به آخرین نسخه commit شده.

### تمرین پیشنهادی

برای تقویت مهارت کار با Git و بهتر درک دستورات فوق توصیه می‌شود مراحل زیر را انجام دهید:

- یک پوشه (directory) جدید ایجاد و در آن init git کنید.

- چند فایل ایجاد و چندین commit ثبت کنید.

- در برخی از فایل‌ها تغییرات جزئی اعمال و diff تغییرات را بررسی کنید.

- از دستورات git add, git diff, git diff –staged, git reset – استفاده کنید تا فرآیند افزودن، مشاهده تغییرات، خارج کردن از stage و بازگردانی فایل‌ها را تمرین نمایید.

اجرای این تمرینات سبب تسلط بیشتر بر مفاهیم پایه‌ای Git و مدیریت تغییرات در پروژه‌ها می‌شود.

## آشنایی با شاخه‌ها یا همان برنج‌ها (Branch)

### مفهوم Git در Branch

در سیستم کنترل نسخه Git، یکی از مفاهیم کلیدی شاخه یا Branch است. Branch به معنی شاخه بوده و اهمیت بالایی در مدیریت پروژه‌های نرم‌افزاری دارد. بخشی از قدرت Git در قالب همین شاخه‌ها ظهور می‌یابد و به توسعه دهندگان امکان می‌دهد تا فعالیت‌های موازی و مستقل روی پروژه انجام دهند.

### کاربرد شاخه‌ها در پروژه‌ها

در حالت عادی، توسعه پروژه به شکل خطی و متوالی پیش می‌رود؛ هر تغییر به صورت Commit ذخیره می‌شود و یک توالی خطی از Commit‌ها ایجاد می‌شود. برای مثال، ممکن است پروژه با Commit یک شروع شده و سپس Commit‌های دو و سه به صورت متوالی افزوده شوند. با این حال، همیشه فرآیند توسعه خطی نیست و نیاز به ایجاد شاخه‌های جدید برای اضافه کردن قابلیت‌ها یا اصلاحات وجود دارد.

فرض کنید توسعه پروژه‌ای مانند یک برنامه بانک در حال انجام است و نیاز به افزودن یک قابلیت جدید مانند کپچا (Captcha) مطرح می‌شود. در این حالت، یک شاخه‌ی جدید برای توسعه این قابلیت ایجاد شده و تغییرات مربوط به آن به طور مجزا انجام می‌شود. پس از تکمیل، شاخه جدید با شاخه‌ی اصلی پروژه، که معمولاً "master" نامیده می‌شود، ادغام (merge) می‌شود. این ساختار اجازه می‌دهد بخش‌های مختلف پروژه به صورت مستقل توسعه پیدا کنند و سپس در زمان مناسب با پروژه‌ی اصلی ادغام شوند.

## مزایای استفاده از Branch

- امکان توسعه مازول‌های مختلف پروژه به صورت مستقل.
- تسهیل همکاری گروهی و کار همزمان چند نفر روی یک پروژه بدون تداخل تغییرات.
- ایزوله شدن ویژگی‌ها یا رفع اشکال‌ها تا زمان آماده‌سازی کامل و ادغام با شاخه‌ی اصلی.
- انعطاف‌پذیری در بازگشت به وضعیت‌های قبلی پروژه یا حذف تغییرات معین در هر شاخه.

## عملیات اصلی با Branch‌ها در Git

مدیریت شاخه‌ها در Git از طریق دستورات متنوع ساده و قدرتمند انجام می‌شود:

- نمایش شاخه‌های موجود با دستور: `git branch` این دستور لیستی از تمام شاخه‌ها را نمایش داده و شاخه‌ی فعلی را با علامت ستاره مشخص می‌کند.
- ایجاد شاخه‌ی جدید و نام‌گذاری آن: `git branch fix-pages` به عنوان مثال: دستور شاخه‌ای به نام `fix-pages` می‌سازد.
- جابجایی به یک شاخه‌ی دیگر: `git checkout fix-pages` `git checkout <branch-name>` مانند: شاخه‌ی مورد نظر فعال می‌شود؛ تمامی تغییرات جدید مربوط به این شاخه خواهند بود.

شاخه‌های جدید معمولاً بر اساس وضعیت جاری شاخه‌ی اصلی (مثلاً master) ایجاد می‌شوند و تا زمان ادغام، تأثیری بر شاخه‌ی اصلی نخواهند داشت. این امکان وجود دارد که هر فرد یا تیم، شاخه مخصوص به خود را برای انجام تغییرات و توسعه ویژگی‌ها داشته و پس از تکمیل کار، شاخه را با بخش اصلی پروژه ادغام کند.

## نمونه گردش‌کار با Branch در Git

در یک پروژه، ممکن است برای اصلاح صفحات وب، شاخه‌ای به نام `fix-pages` ساخته شود. مراحل زیر نمونه‌ای از گردش‌کار استاندارد با شاخه‌هاست:

1. ساخت شاخه جدید: `git branch fix-pages`
2. جابجایی به شاخه‌ی مورد نظر: `git checkout fix-pages`
3. اعمال تغییرات مورد نیاز بر فایل‌ها و ذخیره آنها.
4. افزودن تغییرات به Staging Area: `git add`
5. ثبت تغییرات (Commit) با پیامی مناسب: `git commit -m "added HTML to pages"` تکرار این مراحل برای هر دسته از تغییرات توصیه می‌شود؛ به طوری که هر `commit` شامل یک وظیفه یا مفهوم مجزا باشد.

۶. برگشت به شاخه‌ی master: `git checkout master`

۷. ادغام شاخه‌ی فرعی با شاخه‌ی اصلی: `git merge fix-pages`

در این حالت، تغییرات از شاخه‌ی fix-pages به شاخه‌ی master منتقل خواهد شد. Git تغییرات فایل‌ها را شناسایی کرده و آنها را با وضعیت جدید جایگزین می‌کند.

## نمایش وضعیت پروژه و شاخه‌ها

- برای بررسی وضعیت فعلی شاخه و فایل‌ها: `git status` این دستور شاخه فعلی و فایل‌های تغییر یافته را نمایش می‌دهد.

- مشاهده تاریخچه Commit‌ها: `git log` این دستور لیست Commit‌های اخیر و پیام‌های مرتبط را ارائه می‌دهد.

## نکات تکمیلی و توصیه‌ها

- در صورت نیاز به جابجایی بخش‌هایی از تغییرات بین شاخه‌ها یا تصحیح commit‌ها (مانند ریست کردن تغییرات)، از دستوراتی مانند `git reset` استفاده می‌شود.

- توصیه می‌شود commit‌ها به صورت منظم و با پیام‌های گویا انجام شوند. هر commit بهتر است شامل یک تغییر مشخص و مستقل باشد.

- استفاده موثر از شاخه‌ها به ساختاردهی بهتر پروژه و کاهش ریسک تداخل یا خرابکاری منجر می‌شود.

در مجموعه، استفاده اصولی از Branch‌ها در Git مدیریت مؤثر، توسعه منعطف و همکاری بهینه بر روی پروژه‌های نرم‌افزاری را فراهم می‌کند. توصیه می‌شود این روند را در پروژه‌های عملی خود پیاده‌سازی کرده و تجربه عملی کسب کنید. در مباحث پیشرفته‌تر، امکانات بیشتری برای مدیریت شاخه‌ها و حل تعارضات وجود دارد که در درس‌های آینده بررسی خواهند شد.

## کمی بیشتر در مورد برنجها

در پروژه‌نویسی با گیت، می‌توان با سرعت و کارآمدی، امور مختلف مربوط به مدیریت کد و همکاری تیمی را پیش برد. مراحل پایه شامل ایجاد یک دایرکتوری جدید، مقداردهی مقدماتی با دستور `git init`، افزودن فایل‌ها به مخزن با `git add`، و ثبت تغییرات با `git commit` همراه با توضیحات مناسب برای هر کامیت می‌باشد. مشاهده تغییرات انجام‌شده با دستور `git diff`، درک مفهوم ناحیه استیجینگ و کامیت، و همچنین استفاده از قابلیت `branch` و `checkout` برای توسعه خطوط موازی از پروژه، از اصول کلیدی کار با گیت به شمار می‌رود.

## مراحل ایجاد و مدیریت شاخه‌ها

برای توسعه ویژگی‌های جدید یا رفع ایرادات، توصیه می‌شود به جای کار روی شاخه اصلی (master)، یک شاخه جدید ایجاد شود. این رویکرد امکان توسعه موازی را فراهم می‌کند و از بهم ریختگی کد اصلی جلوگیری می‌نماید. برای نمونه، به منظور اضافه کردن لینک صفحات از فایل `index.html` به صفحات دیگر، یک شاخه جدید به نام `linking pages` ساخته می‌شود. با استفاده از دستور `git checkout` می‌توان به این شاخه منتقل شد و تغییرات لازم را اعمال کرد.

## حذف و ویرایش فایل‌ها

در صورت وجود فایل‌هایی که دیگر مورد نیاز نیستند، دستور `git rm` برای حذف آن‌ها هم از گیت و هم از فایل سیستم استفاده می‌شود. پس از اعمال تغییرات لازم در فایل‌ها با استفاده از یک ویرایشگر مناسب، وضعیت پروژه با `git status` بررسی می‌گردد تا اطمینان حاصل شود تغییرات به درستی ثبت شده‌اند.

## ثبت و جداسازی کامیت‌ها

بهتر است هر کامیت تنها شامل یک اقدام یا تغییر باشد. این رویکرد به خوانایی و پیگیری تاریخچه تغییرات کمک می‌کند. پس از افزودن فایل‌های تغییریافته با `git add -a`، ثبت تغییرات با دستور `git commit -m "پیام مناسب انجام می‌شود. با جابجایی بین شاخه‌ها، تغییرات هر شاخه مجزا باقی می‌ماند تا زمانی که ادغام (merge)` صورت گیرد.

## ادغام شاخه‌ها و مدیریت آن‌ها

ادغام شاخه‌های توسعه‌ای به شاخه اصلی پروژه با دستور `git merge` انجام می‌شود. پس از موفقیت‌آمیز بودن ادغام و انتقال تغییرات به شاخه اصلی، شاخه توسعه‌ای که دیگر نیاز به آن نیست با دستور `git branch -d` حذف می‌گردد. این کار باعث نگهداری تمیزی و نظم مخزن می‌شود و از افزوده شدن تعداد زیادی شاخه بلااستفاده جلوگیری می‌کند.

## بررسی تاریخچه و وضعیت پروژه

برای مشاهده تاریخچه کامیت‌ها و اتفاقات رخداده در پروژه می‌توان از `git log` بهره برد. همچنین با اجرای `git status` وضعیت فعلی و میزان پاکیزگی مخزن قابل بررسی است. این کنترل منظم کمک می‌کند تا پروژه همواره در وضعیت مطلوب نگهداری شود.

در نهایت، با بهره‌گیری حرفه‌ای از ویژگی‌های شاخه‌بندی گیت، توسعه چندجانبه، تشکیل تیم‌های توسعه موازی، و نگهداری شاخه اصلی در وضعیت پایدار می‌سر می‌شود. پس از هر ادغام لازم است شاخه‌های مازاد حذف شوند تا نظم پروژه همچنان حفظ گردد. این مبانی پایه کار عملی با گیت و شاخه‌ها را تشکیل می‌دهد و زمینه را برای مباحث پیشرفته‌تر فراهم خواهد نمود.

## آشنایی و استفاده از گیت هاب (GitHub)

### مرج کردن و کار با گیت ریموت

کار با سیستم کنترل نسخه گیت (Git) یکی از مهارت‌های اساسی برای توسعه‌دهندگان در محیط‌های حرفه‌ای است. تاکنون مفاهیم بنیادی گیت از جمله ایجاد شاخه (Branch)، انجام کامیت (Commit)، بازگشت به حالت‌های قبلی و مرج کردن (Merge) بررسی شده است. همچنین فرآیند اطلاع‌رسانی به سایر توسعه‌دهندگان درباره اعمال تغییرات روی شاخه‌های مختلف و رفع باگ‌ها نیز تشریح گردیده است.

### آشنایی با ریموت‌ها در گیت

یکی از مهم‌ترین قابلیت‌های گیت نسبت به سایر سیستم‌های کنترل نسخه، ساختار توزیع‌شده (Distributed) آن است. این قابلیت به مشترکان اجازه می‌دهد تا بتوانند از نقاط مختلف و به صورت مستقل کار کنند. مفهوم "ریموت" در گیت به مخزن‌هایی (Repository) اشاره دارد که خارج از دایرکتوری فعلی کاربر قرار گرفته‌اند. افزودن ریموت، به کاربر این قابلیت را می‌دهد که فعالیت‌های خود را به یک مخزن دیگر در شبکه (یا حتی به‌طور محلی) متصل کند.

در گیت، با استفاده از دستورات خاص، کاربر می‌تواند: - یک ریموت را به پروژه خود اضافه کند. - تغییرات را به ریموت ارسال کند (Push). - تغییرات را از ریموت دریافت کند (Pull).

این امکانات بستر هماهنگی تیمی و اشتراک‌گذاری کد را فراهم می‌آورد، به ویژه در شرایطی که از سرویس‌هایی نظیر GitHub، GitLab یا مخزن داخلی شرکت استفاده می‌شود. برای مثال، یک مخزن ممکن است در آدرسی نظیر `git.company.com` قرار داشته باشد.

## کلون کردن یک مخزن

برای بهره‌برداری از یک پروژه اینترنتی، کافی است آدرس مخزن مورد نظر را دریافت و دستور `git clone` را اجرا نمود. با این کار، یک دایرکتوری جدید شامل تمامی فایل‌ها و تاریخچه پروژه ساخته می‌شود. به عنوان نمونه، کلون کردن یک پروژه آموزشی لینوکسی به سادگی با دریافت آدرس پروژه و اجرای دستور کلون انجام خواهد شد. پس از کلون کردن، پروژه بر روی سیستم کاربر در دسترس بوده و روند توسعه قابل پیگیری است.

## ویرایش، کامیت و پوش تغییرات

پس از اعمال هرگونه تغییر روی فایل‌های پروژه مانند ویرایش فایل README.md، می‌توان وضعیت فایل‌ها را با دستور `git status` بررسی نمود. مراحل بعدی شامل افزودن فایل‌های تغییر یافته (`git add`)، ثبت تغییرات با توضیح مناسب (`git commit`)، و در نهایت ارسال تغییرات به مخزن ریموت با دستور `git push origin master` است. در صورت نیاز به شناسایی هویت کاربر برای ارسال تغییرات، گیت درخواست نام کاربری و گذرواژه را مطرح می‌کند. پس از تایید اعتبار، تغییرات به مخزن ریموت افزوده شده و قابل مشاهده برای سایر اعضا خواهد بود.

## بروزرسانی پروژه با تغییرات جدید

در بسیاری از موارد، دیگر توسعه‌دهندگان نیز ممکن است هم‌زمان بر روی پروژه کار کنند و تغییراتی را مستقیماً روی مخزن ریموت ثبت نمایند. برای دریافت این تغییرات، کافی است دستور `git pull origin master` اجرا شود تا آخرین بهروزرسانی‌ها به شاخه فعلی افزوده گردد. برای سهولت بیشتر، می‌توان با پارامتر `-u` در اولین `push` یا `pull` مبدأ و مقصد پیش‌فرض را تعیین نمود تا در دفعه‌های بعدی فقط از دستور ساده‌تر استفاده شود.

## مدیریت و سطوح دسترسی به ریپوزیتوری‌ها

هر کاربر می‌تواند چندین ریپوزیتوری مختلف داشته باشد که برخی از آن‌ها عمومی و برخی نیازمند سطحی از مجوز برای `push` کردن هستند. در محیط‌های شرکتی، فرآیند اعطای مجوزهای لازم می‌تواند با مکانیزم‌هایی مانند نام کاربری و رمزعبور یا استفاده از کلیدهای امن (SSH Key) انجام شود. تولید و ثبت کلید SSH اولین گام برای تسهیل ارسال امن تغییرات به مخزن، بدون نیاز به وارد کردن رمز عبور در هر مرتبه است.

## جمع بندی

با آشنایی با مفاهیم کلیدی گیت شامل شاخه‌ها، کامیت، مرج، مدیریت ریموت‌ها، کلون کردن پروژه، پوش و پول تغییرات و مدیریت مجوزها، کاربر قادر به مشارکت مؤثر در پروژه‌های تیمی و حرفه‌ای خواهد بود. مباحث پیشرفته‌تر از جمله کار با اجزای داخلی گیت و مدیریت کلیدهای امنیتی می‌تواند گام بعدی در مسیر تسلط بر گیت باشد.

## بررسی و حل کانفلیکت‌های (Conflict) ریموت

### اضافه کردن Remote به پروژه‌های گیت

در فرایند کار با Git، مدیریت ریموت‌ها بخش کلیدی و مهمی از همکاری تیمی و هماهنگی پروژه‌هاست. معمولاً پس از کلون کردن یک مخزن Git از سرویس‌هایی مانند GitHub، یک ریموت با نام پیش‌فرض Origin به پروژه اختصاص می‌یابد. این ریموت بیانگر آدرس اصلی مخزن بر روی سرور است و به عنوان نقطه مرجع برای انجام عملیات `pull` و `push` مورد استفاده قرار می‌گیرد.

در شرایطی ممکن است بخواهید پس از شروع پروژه و بدون کلون اولیه، یک ریموت به پروژه خود اضافه کنید. برای این کار، می‌توان با اجرای دستور `git remote add` و مشخص کردن یک نام برای ریموت (اغلب Origin) و آدرس ریموت، ارتباط لازم را برقرار کرد.

این آدرس می‌تواند از نوع HTTPS، SSH یا هر شیوه دیگری باشد و در محیط‌های شرکتی معمولاً توسط مدیر سیستم در اختیار کاربر قرار می‌گیرد.

پس از اضافه کردن ریموت، با دستور `git remote add [نام_برخوردار] [آدرس]` و افزودن گزینه `-v` می‌توان لیست کامل ریموت‌ها و آدرس‌های مرتبط با آنها را مشاهده کرد. این اطلاعات بیانگر امکان `push` یا `pull` به ریموت‌های مختلف پروژه است. Git امکان مدیریت و ثبت چندین ریموت برای یک پروژه را فراهم می‌کند. این قابلیت مخصوصاً در پروژه‌هایی که نسخه‌های پشتیبان روی سرویس‌های متعددی مثل GitHub و GitLab یا محیط‌های مختلف شرکت نگهداری می‌شوند، اهمیت می‌یابد.

## مدیریت تغییرات همزمان و حل Conflicts

در پروژه‌هایی که چندین نفر به طور همزمان روی یک فایل کار می‌کنند، احتمال بروز تداخل یا `conflict` وجود دارد. این اتفاق زمانی رخ می‌دهد که دو نفر همزمان بخش مشابهی از یک فایل را تغییر داده باشند و تغییرات هر دو کاربر روی شاخه اصلی (برای مثال `master`) اعمال شود.

در این شرایط، اگر کاربر اول بدون `pull` گرفتن از ریموت اقدام به `push` کند، در صورت وجود تغییر همزمان از سوی کاربر دوم، Git با خطا مواجه می‌شود و پیام رد شدن (`reject`) را اعلام می‌کند. Git پیشنهاد می‌دهد که ابتدا تغییرات جدید از ریموت دریافت شود (`git pull`) و سپس عملیات `push` انجام بگیرد.

در زمان اجرای `git pull`، Git تلاش می‌کند تغییرات حاصل از هر دو طرف را به صورت خودکار `merge` نماید. اگر تغییرات در بخش‌های مختلف فایل باشند، این ادغام به طور اتوماتیک انجام می‌گیرد. اما اگر هر دو نفر دقیقاً یک بخش یکسان از فایل را تغییر داده باشند، Git قادر به ادغام خودکار نخواهد بود و `conflict` رخ می‌دهد.

در این حالت، وضعیت فایل مورد نظر به حالت `conflict` در می‌آید. می‌توان با استفاده از دستور `git status` وضعیت شاخه و فایل‌های دچار تداخل را مشاهده کرد. برای رفع `conflict`، لازم است فایل مربوطه را در ویرایشگر متن باز کرده و بخش‌هایی که توسط Git با علائم خاص (مانند `>>>>>`، `<<<<<`، `=====`، `>>>>>`، `<<<<<`) تشانه‌گذاری شده‌اند را به شکل صحیح و مورد نیاز اصلاح نمود.

پس از اصلاح فایل، باید آن را `stage` کرده (`git add`) و سپس با یک پیام مناسب `commit` نمود. در نهایت تغییرات جدید به ریموت `push` می‌شود و وضعیت پروژه به حالت پایدار بازمی‌گردد. این فرآیند متداول‌ترین حالت بروز پیچیدگی در کار با Git است و پیشگیری از بروز آن با `commit` و `merge` زودهنگام تغییرات توصیه می‌شود.

## اهمیت ادغام به موقع تغییرات

ادغام یا `merge` به موقع تغییرات به شاخه اصلی پروژه، از بروز تداخل‌های پیچیده جلوگیری می‌کند. به تعویق انداختن عملیات `merge` برای مدت طولانی، باعث افزایش احتمال بروز `conflict` شده و مدیریت پروژه را دشوار می‌سازد. Git و راهبری پروژه نمی‌توانند به صورت خودکار منظور توسعه‌دهنده را حدس بزنند و لازم است اعضای تیم با `push` و `merge` مرتب و منظم، ریسک `conflict` را به حداقل برسانند.

## جمع‌بندی

آشنایی با مفاهیم مدیریت ریموت، انجام عملیات `push` و `pull`، و برطرف کردن `conflict`، مهارت‌های پایه‌ای برای کار با Git در محیط‌های شرکتی و تیمی به شمار می‌آیند. با تسلط به این موضوعات و تمرین عملی، می‌توان به راحتی در پروژه‌های سازمانی مشارکت مؤثر داشت و دانش خود را با گسترش امکانات پیشرفته‌تر Git افزایش داد. مروری مستمر بر این مراحل و استفاده از منابع آموزشی تکمیلی، زمینه‌ساز متخصص شدن در کار با Git خواهد بود.

## تگ زدن برای شناسایی نسخه‌ها

## مدیریت نسخه‌ها با استفاده از دستور Tag در Git

سیستم مدیریت نسخه Git ابزار قدرتمندی برای سازماندهی و مدیریت پروژه‌های نرم‌افزاری چندنفره و در مقیاس‌های مختلف فراهم می‌کند. یکی از قابلیت‌های کلیدی Git، امکان برچسب‌گذاری یا Tagging بر روی نسخه‌های خاصی از کد منبع است که برای کنترل نسخه‌ها و انتشار نرم‌افزار بسیار مفید می‌باشد.

### مفهوم Tag و کاربرد آن در پروژه‌های نرم‌افزاری

در جریان توسعه یک پروژه نرم‌افزاری، معمولاً توسعه‌دهندگان بر روی شعبه‌هایی (Branch) مانند master یا development کار می‌کنند. با پیشرفت پروژه، گاهی لازم است که وضعیت کد در یک نقطه معین، مانند آماده‌شدن برای ارائه یا انتشار (Release)، ثبت و علامت‌گذاری شود تا بعداً بتوان به آسانی به آن حالت مشخص بازگشت یا از آن استفاده کرد. این فرایند با استفاده از قابلیت Tag در Git انجام می‌شود.

همانند یک برچسب یا نشانه عمل می‌کند که به یک commit مشخص مرتبط می‌شود و معمولاً برای تعیین نسخه‌های خاص نرم‌افزار مانند v1.0، v2.0 و غیره استفاده می‌شود. این امر باعث می‌شود در هر زمان بتوان نسخه‌های مختلف یک پروژه را مشاهده، مقایسه یا بازیابی کرد، بی‌آنکه مجبور به یادآوری دقیق شاخه یا commit موردنظر باشید.

### نحوه ایجاد و مشاهده Tag‌ها در Git

برای مدیریت Tag‌ها در Git، دستورات زیر رایج است:

- مشاهده آخرین commit‌ها:

```
git log
```

- مشاهده Tag‌های موجود:

`git tag` با نام و توضیح (message) برای یک نسخه مشخص، به عنوان مثال برای نسخه ۰.۰.۱:

`v2.0 -m "اولین نسخه قابل ارائه. این همان چیزی است که اجرا می‌شود."`

در این دستور، گزینه `-a` به معنای "annotate" است که امکان افزودن پیام توضیحی به Tag را می‌دهد. استفاده از پیام توضیحی کمک می‌کند تا بعداً علت یا ویژگی‌های این نسخه به راحتی قابل درک باشد.

### تخصیص commit به Tag‌های قبلی

در عمل می‌توان Tag را به commit را به commit نیز اختصاص داد. برای این منظور، ابتدا شناسه commit مورد نظر را (که معمولاً می‌توانید با چند حرف اول آن از طریق خروجی `git log` بیابید) یادداشت کرده و سپس Tag را بر روی آن قرار می‌دهید:

`"پس از اصلاح فایل html، آماده انتشار" git tag -a v1.8 <commit-hash> -m`

### جستجو و نمایش Tag‌های خاص

در پروژه‌هایی با تعداد زیاد Tag، می‌توان از فیلترهای جستجو استفاده کرد، مثلاً:

`*git tag -l 'v`

این دستور تمامی Tag‌هایی که با حرف "v" آغاز می‌شوند را نمایش می‌دهد.

همچنین، برای مشاهده اطلاعات یک Tag خاص، مانند تغییرات و پیام توضیحی آن، می‌توان استفاده کرد از:

`git show v1.8`

## انتشار Remote Repositoryها به Tag (Push)

به طور پیشفرض، زمانی که دستور `git push` اجرا می‌شود، Tag‌های جدید به مخزن راه دور (remote) ارسال نمی‌شوند. برای انتشار یک Tag خاص باید به طور صریح آن را `push` کرد:

```
git push origin v1.8
```

همچنین می‌توان تمام Tag‌ها را با دستور زیر منتشر نمود:

```
git push --tags
```

پس از انتشار، سایر هم‌تیمی‌ها می‌توانند با استفاده از دستور `git pull`، Tag‌های جدید را دریافت کنند تا در مخزن محلی خود آن‌ها را داشته باشند.

## استفاده از Tag برای بازگشت (Checkout) به نسخه‌های قبلی

یکی دیگر از قابلیت‌های مهم Tag، امکان بازگشت به نسخه‌های خاصی از پروژه است. برای این کار کافیست با دستور زیر به Tag موردنظر سوییچ کنید:

```
git checkout v1.8
```

در این وضعیت، HEAD دقیقاً بر روی آن نسخه قرار می‌گیرد، اما توجه داشته باشید که این حالت مشابه استفاده از branch نیست. در چنین شرایطی نمی‌توان commit جدیدی را مستقیماً بر روی این Tag ایجاد کرد، اما امکان ساخت یک branch جدید از همان نقطه وجود دارد تا توسعه جدید را بر اساس آن ادامه داد.

## جمع‌بندی

Tag‌ها ابزار مهمی در Git برای مدیریت و علامت‌گذاری نسخه‌های مهم یک پروژه هستند و معمولاً توسط توسعه‌دهندگان یا مدیران پروژه در زمان انتشار نسخه‌های جدید یا milestone‌های مهم استفاده می‌شوند. با استفاده از Tag‌ها می‌توان به سادگی بین نسخه‌های مختلف جابجا شد، انتشارها را مدیریت کرد و برای اهداف مختلف مانند رفع اشکال یا ارائه نسخه تاریخی پروژه، به وضعیت‌های قبلی بازگشت.

## امضا کردن تگ‌ها و کامیت‌ها

### مفهوم هش و امضا در Git

در Git، هر کامیت دارای یک هش منحصر به فرد است. این هش امکان تغییر کامیت را از بین می‌برد، زیرا هرگونه تغییر در محتوای کامیت موجب تغییر هش خواهد شد. علاوه بر این، هش‌ها تنها هویت نویسندۀ را همراه دارند و به طور پیشفرض فاقد امضای دیجیتال هستند، بنابراین امکان جعل آن‌ها وجود دارد.

در دنیای کامپیوتر، امضا کردن به معنای امضای دیجیتال است. امضای دیجیتال سطح بالاتری از امنیت و اصالت را ارائه می‌دهد، به‌طوری که اگر یک کامیت با امضای دیجیتال همراه باشد، این اطمینان حاصل می‌شود که فقط صاحب کلید خصوصی مربوط به آن امضا می‌تواند آن را تولید کند و هیچ شخص دیگری قادر به جعل این امضا نیست، مگر اینکه کلیدهای مرتبط سرقت شده باشند.

### الگوریتم‌های رمزنگاری و معرفی GPG

رمزنگاری کلید عمومی یکی از روش‌های رایج در امنیت کامپیوتوری است. در این روش، دو کلید وجود دارد: یک کلید عمومی (public key) که در اختیار عموم قرار می‌گیرد و یک کلید خصوصی (private key) که محرومانه نزد صاحب کلید باقی می‌ماند. هر پیامی که با

کلید خصوصی رمزگذاری یا امضای شود فقط با کلید عمومی مربوطه قابل باز شدن یا تأیید خواهد بود.

یکی از ابزارهای قدرتمند و پرکاربرد برای این منظور GPG (GNU Privacy Guard) است که مشابه الگوریتم Pretty Good Privacy (GPG) عمل می‌کند. یک استاندارد رمزگاری متن‌باز و رایگان محسوب می‌شود و توسط جامعه نرم‌افزار آزاد توسعه یافته است. از طریق GPG می‌توان انواع فایل‌ها یا پیغام‌ها را رمزگذاری و امضای کرد.

## ایجاد و مدیریت کلیدهای GPG

برای استفاده از امضای دیجیتال در Git ابتدا باید کلید شخصی GPG ایجاد شود. روند ایجاد کلید به این صورت است:

۱. نصب GPG بر روی سیستم عامل.
۲. اجرای دستور `gpg --gen-key` جهت ایجاد جفت کلید جدید. طی این فرآیند، اطلاعات هویتی مانند نام و ایمیل کاربر اخذ شده و کاربر موظف به انتخاب یک گذرواژه مطمئن برای محافظت از کلید خصوصی است.
۳. پس از ساخت کلید، با دستور `gpg --list-keys` می‌توان کلیدهای عمومی موجود را مشاهده کرد. همچنین برای مشاهده کلیدهای خصوصی از دستور `gpg --list-secret-keys` استفاده می‌شود.
۴. بخشی از خروجی این دستورات، شناسه منحصریه‌فرد (key ID) کلید است که در ادامه مورد استفاده قرار می‌گیرد.

## پیکربندی Git برای استفاده از کلید GPG

پس از ایجاد کلید، لازم است Git را طوری پیکربندی کرد که از این کلید برای امضای کامیت‌ها یا تگ‌ها استفاده کند. برای این منظور باید شناسه کلید (signing key) را در تنظیمات Git وارد کرد:

- وارد کردن شناسه کلید با دستور: `git config --global user.signingkey [کلید]`
- این پارامتر به Git اعلام می‌کند که هر زمان نیاز به امضای کامیت یا تگ بود، از این کلید GPG معین استفاده شود.

## امضای تگ‌ها و کامیت‌ها در Git

به کمک GPG قابلیت امضای تگ‌ها و کامیت‌ها در Git فراهم می‌شود. روش انجام این کار به شرح زیر است:

- برای ایجاد تگ امسا شده: `git tag -s [tagname] -m "[message]"` (حرف بزرگ S- گزینه) به معنای sign است و ذخیره تگ همراه با امضای دیجیتال را ممکن می‌سازد. سیستم رمزگاری از کاربر رمز عبور کلید خصوصی را درخواست می‌کند.
- مشاهده تگ و امضای آن: `git show [tagname]` این دستور علاوه بر پیام تگ، بخش امضای دیجیتال را نیز نمایش می‌دهد.
- برای تأیید امضای تگ: `git tag -v [tagname]` این دستور صحت امضای مطابقت آن با کلید عمومی را بررسی کرده و نتیجه را اعلام می‌کند.
- برای امضای کامیت‌ها: `git commit -S -m "[message]"` استفاده از گزینه S- (حرف بزرگ) نشان‌دهنده امضای کامیت است. هنگام اجرای این دستور، همچون امضای تگ، رمز عبور کلید خصوصی درخواست خواهد شد و امضای کامیت اضافه می‌شود.

## نکات عملی و مدیریتی پیرامون امضاهای دیجیتال

اجرای سیاست امضای اجباری در پروژه‌ها معمولاً بر عهده مدیر تیم است. در بسیاری از سازمان‌ها، همه اعضا ملزم به امضای کامیت‌ها یا تگ‌ها هستند و این تنظیمات به صورت مرکزی در همه مخازن اعمال می‌شود. گرچه راه اندازی و استفاده از امضاهای دیجیتال موجب افزایش امنیت و اصالت کدها می‌گردد، اما نیازمند دانش عمومی از اصول رمزگاری، مدیریت کلیدها و مقداری تلاش اولیه برای پیکربندی تیم است.

در نهایت، هدف اصلی آشنایی با فرایند امضای دیجیتال و اهمیت آن در تضمین صحت و اصالت کدهاست. هر چند برخی کاربران تازه‌وارد ممکن است نیاز چندانی به این قابلیت نداشته باشند، اما آگاهی از کاربرد آن در محیط‌های حرفه‌ای نرم‌افزار امری ضروری محسوب می‌شود.

## دیباگ کردن با کمک گیت

### مرور ابزارهای Git برای بررسی تغییرات و یافتن خطاهای

ابزار Git مجموعه‌ای از دستورات و امکانات را در اختیار کاربران می‌گذارد که به تحلیل تاریخچه تغییرات کد، شناسایی منبع خطاهای و بررسی مسئولیت تغییرات در پروژه‌های نرم‌افزاری کمک می‌کنند. این درس به معرفی دو ابزار کلیدی در Git، یعنی git blame و git bisect، با تأکید بر کاربردهای عملی روزمره آن‌ها می‌پردازد.

### استفاده از Help در Git

یکی از ویژگی‌های مفید Git، امکان دریافت راهنمایی دقیق درباره هر دستور است. با استفاده از دستور git help [دستور مورد نظر]، می‌توان توضیح کاملی درباره عملکرد دستور، سوابیج‌ها و مثال‌های کاربردی دریافت کرد. این امکان به طور قابل ملاحظه‌ای در یادگیری و بهره‌گیری مؤثر از دستورات مختلف Git، به ویژه در بررسی عملکردهای ناشناخته مواجه می‌شود، مفید است.

### بررسی مسئولیت تغییرات با git blame

دستور git blame به عنوان ابزاری برای پیدا کردن منشأ هر خط از کد در یک فایل مورد استفاده قرار می‌گیرد. هدف اصلی این ابزار، شناسایی افرادی است که خطوط خاصی از کد را در بازه‌های زمانی مختلف ایجاد یا تغییر داده‌اند. عملکرد اصلی این دستور به شکل زیر است:

- با اجرای git blame [نام فایل]، تاریخچه و نویسنده هر خط از فایل نمایش داده می‌شود.
- می‌توان با تعیین بازه خطوط (Line Range)، تنها اطلاعات مربوط به خطوط خاصی را بررسی کرد. به عنوان مثال، با استفاده از سوئیچ -L x,y می‌توان خطوط x تا y را مشاهده نمود.
- اطلاعات ارائه شده شامل شناسه کامیت، نام و ایمیل نویسنده، تاریخ و پیام کامیت مربوط به هر خط است.

این ابزار در بازبینی کد (Code Review)، رفع باگ‌ها و شناسایی مسئولیت تغییرات نقش مهمی ایفا می‌کند. به عنوان مثال، هنگامی که خطایی در یک بخش خاص از برنامه کشف شود، git blame می‌تواند مشخص کند چه کسی آخرین بار آن بخش را تغییر داده است.

### شناسایی منبع باگ‌ها با git bisect

دستور git bisect به عنوان روشی سیستماتیک برای شناسایی کامیت خاصی است که موجب ایجاد خطای باگ در پروژه شده است. این ابزار با کمک الگوریتم جستجوی دودویی (Binary Search) بین کامیت‌های مختلف پروژه عمل می‌کند. مکانیزم عملکرد git bisect به شرح زیر است:

۱. آغاز فرآیند با دستور `git bisect start` و معرفی دو نقطه مرجع:

۲. یک کامیت که در آن مشکل وجود دارد (bad).

۳. یک کامیت قدیمی‌تر که در آن مشکل نداشته (good).

۴. Git به طور خودکار بین این دو نقطه، کامیت میانی را انتخاب می‌کند و پروژه را به آن وضعیت منتقل می‌کند.

۵. کاربر باید پس از هر بار بررسی وضعیت فعلی پروژه (مثلًاً اجرای برنامه و مشاهده وجود یا عدم وجود خطا)، اعلام کند که این مرحله good (خوب) یا bad (بد) است.

۶. Git این روند را تا زمانی ادامه می‌دهد که منبع خطا (کامیت مشکل‌ساز) مشخص شود.

این روش خصوصاً در پروژه‌های بزرگ و با تاریخچه طولانی ارزشمند است و به صورت چشمگیری سرعت یافتن اولین ایجادکننده یک باگ را افزایش می‌دهد.

## جمع‌بندی

ابزارهای git blame و git bisect از مهم‌ترین دستورات Git برای تحلیل تاریخچه کد، شناسایی تغییرات کلیدی، اختصاص مسئولیت خطوط کد به افراد مختلف و شناسایی کامیت‌های خطا‌ساز هستند. تسلط بر این ابزارها، نقش مهمی در ساده‌سازی روند دیباگینگ و توسعه گروهی نرم‌افزار دارد و موجب افزایش شفافیت و کارآمدی پروژه می‌شود.

## آشنایی با گیت لب و مشارکت در پروژه‌ها

### بررسی اجمالی Git و کاربردهای آن

یک سیستم کنترل نسخه توزیع شده است که به کاربران اجازه می‌دهد تاریخچه پروژه‌های نرم‌افزاری خود را مدیریت کرده و به صورت موثر با دیگران همکاری کنند. از طریق آموزش‌های گذشته، مهارت‌های ابتدایی همچون نصب Git، ایجاد ریپازیتوری (init)، ثبت تغییرات (commit) و همکاری با مخازن راه دور (remote) به تفصیل بررسی شده‌اند. کاربران با استفاده از این مفاهیم می‌توانند پروژه‌های مختلف را کلون (clone) کنند، تغییرات ایجاد شده را به سرورهای اشتراکی پوش (push) نمایند و با دیگر اعضای تیم تعامل داشته باشند.

### قابلیت‌های پیشرفته Git

در کلاس‌های اخیر، برخی از قابلیت‌های پیشرفته Git نظیر دیباگ (debug)، تنظیم هویت کاربری (config)، و بازیابی نسخه‌های پیشین مورد بررسی قرار گرفته‌اند. اهمیت امضای دیجیتال تغییرات و سایر امکانات استاندارد Git نیز تشریح شده است. با این حال، یادآوری می‌شود که آنچه تاکنون آموزش داده شده، تنها بخش کوچکی از امکانات بسیار گستردگی Git است. کاربران برای مواجهه با مسائل جدید می‌توانند به جستجوی راهکارها و مستندسازی مراجعه کنند.

### معرفی GitHub و تفاوت آن با GitLab

علاوه بر GitHub، ابزارهایی نظیر GitLab نیز وجود دارند که خدمات مشابهی ارائه می‌دهند. GitLab یک سایت مدیریت مخزن کد است که امکان تعریف پروژه‌ها به صورت خصوصی (private) یا عمومی را برای کاربران فراهم می‌کند. برخلاف GitHub که برخی امکانات خصوصی‌سازی پروژه‌ها نیازمند پرداخت هزینه‌اند، در GitLab می‌توان پروژه‌های خصوصی نامحدودی تعریف کرد. کاربران می‌توانند نسخه‌های تحت وب، ابری، و یا مجازی (VPS/VM) را نصب و استفاده کنند. Bitnami از جمله سایت‌هایی است که ماشین‌های مجازی آماده GitLab را ارائه می‌دهند. شرکتها، سازمان‌ها و افراد زیادی بسته به نیازهای خود ممکن است GitLab یا GitHub را به عنوان بستر اصلی ذخیره و توسعه پروژه‌ها انتخاب کنند.

## شیوه مشارکت در پروژه‌ها با استفاده از Git

فرآیند همکاری در پروژه‌های متن‌باز یا سایر پروژه‌های توسعه نرم‌افزاری در GitHub و GitLab با مدل مشارکت در شرکت‌های خصوصی تفاوت دارد. در شرکت‌ها، اعضای تیم دسترسی مستقیم به مخزن پروژه دارند. اما در پروژه‌های اوپن سورس، امکان ارسال تغییرات بدون دسترسی مستقیم به پروژه اصلی از طریق مفاهیم فورک (fork) و پول ریکوئست (pull request) یا مرج ریکوئست (merge request) فراهم می‌گردد.

مراحل مشارکت در پروژه‌های عمومی به شرح زیر است:

۱. یافتن ریپازیتوری مورد نظر در GitHub یا GitLab.
۲. انجام عملیات فورک (fork)، که یک نسخه مستقل از پروژه را در حساب کاربری خودتان ایجاد می‌کند.
۳. کلون کردن پروژه فورک شده در رایانه شخصی، ایجاد تغییرات دلخواه، و ثبت تغییرات با commit.
۴. پوش (push) کردن تغییرات به مخزن شخصی.
۵. ارسال پول ریکوئست (در GitHub) یا مرج ریکوئست (در GitLab) به مخزن اصلی پروژه.
۶. در صورت پذیرش درخواست، تغییرات ارسال شده با پروژه اصلی ادغام خواهند شد.

این فرآیند به افراد اجازه می‌دهد بدون نیاز به کسب دسترسی مستقیم از مسئول پروژه، در توسعه آن مشارکت داشته باشند. مدیر پروژه می‌تواند در صورت تأیید تغییرات، آن‌ها را مرج کند.

## نکات تکمیلی درباره Git و رفع مشکلات

یادگیری دستورات تکمیلی Git و حل مشکلات جدید مستلزم جستجو و ارجاع به مستندات است. Git ابزار قدرتمندی است که به راحتی نمی‌توان موجب اختلال جبران ناپذیر در آن شد، مگر اینکه به طور مستقیم فایل‌های اصلی را حذف یا ویرایش کنید. تقریباً تمام اقدامات قبل بازگشت‌اند و با کمی تجربه، کاربران می‌توانند مشکلات احتمالی را مدیریت کنند. توصیه می‌شود تمرکز اصلی کاربران، یادگیری روش حل مسئله در Git باشد، نه صرفاً حفظ دستورات.

## پیشنهادهایی برای تمرین و توسعه مهارت

برای تسلط بیشتر بر مباحث ارائه شده، توصیه می‌شود:

- یک مخزن جدید در حساب کاربری خود در GitHub یا GitLab ایجاد کرده و با آن تمرین کنید.
- پروژه‌های آموزشی، نظیر مخزنی با نام "git-tutorial"، مناسبی برای آزمودن تغییرات و ارسال pull request هستند.
- از پروژه‌های سندباکس (sandbox) جهت آزمون عملی دستورات و فرآیندهای مختلف استفاده نمایید.

این شیوه موجب یادگیری عمیق‌تر کار با Git می‌شود و امکان دریافت بازخورد از جامعه کاربران فراهم می‌گردد.

## جمع‌بندی

در این مجموعه آموزشی، اصول و امکانات مهم Git بررسی و تمرین شده‌اند. با وجود وسعت بسیار زیاد Git، مهارت‌های کسب شده مبنای مناسبی برای توسعه بخشی تخصصی‌تر خواهند بود. پیشنهاد می‌شود از هر فرصتی برای تمرین عملی، رفع اشکالات احتمالی و جستجوی راهکارهای جدید بهره‌مند شوید. Git ابزاری ایمن و قدرتمند است که مدیریت پروژه‌های نرم‌افزاری را به صورت مؤثری بهبود می‌بخشد.

# مراجع یادگیری دروس دانشگاهی (ویژه دانشجویان)

## آزمون جامع و گواهینامه