Kapitel 9: Praktischer Workflow für Einsteiger

Einleitung: Du kennst jetzt alle wichtigen Git-Befehle und -Konzepte – aber wie setzt du das im Alltag zusammen? Ein guter Workflow gibt dir Struktur und verhindert Chaos. Als Einzelentwickler brauchst du keinen komplexen Prozess, aber ein paar bewährte Praktiken helfen enorm: regelmäßig committen, aussagekräftige Nachrichten schreiben, Features in eigenen Branches entwickeln, und den Main-Branch immer funktionsfähig halten. Dieses Kapitel zeigt dir einen praktikablen Workflow, den du sofort anwenden kannst. Außerdem lernst du GitHub-Features wie Issues kennen, die dir helfen, auch als Solo-Entwickler den Überblick über Aufgaben und Ideen zu behalten.

Ein einfacher Git-Workflow für Einzelentwickler □□

Als Einzelentwickler brauchst du keinen komplizierten Workflow mit Pull Requests, Code Reviews oder Release-Branches. Trotzdem lohnt es sich, ein paar **klare Gewohnheiten** zu etablieren – sie halten dein Projekt übersichtlich, schützen dich vor Datenverlust und machen es leichter, Fehler zu finden oder rückgängig zu machen.

Der empfohlene Workflow im Überblick

```
flowchart TD
    START["□□Arbeitstag beginnen"] --> PULL["git pull\nAktuellen Stand holen"]
    PULL --> BRANCH{"Neues Feature\noder Experiment?"}
    BRANCH -->|Ja| CREATE["Neuen Branch erstellen\nz.B. feature/kontaktformular"]
    BRANCH -->|Nein| WORK["Direkt auf main arbeiten\nbei kleinen Aenderungen"]
    CREATE --> WORK2["Im Feature-Branch arbeiten"]
    WORK --> COMMIT["Regelmaessig committen\nbei logischen Einheiten"]
    WORK2 --> COMMIT
    COMMIT --> MORE{"Weitere\nAenderungen?"}
    MORE -->|Ja| COMMIT
    MORE -->|Nein| MERGE["Feature-Branch mergen\nfalls vorhanden"]
    MERGE --> PUSH["git push\nAendeurngen sichern"]
    PUSH --> END["□□Feierabend"]
```

Dein typischer Arbeitstag mit Git

1. Arbeitsbeginn: Aktuellen Stand holen □□

Bevor du mit der Arbeit beginnst, solltest du sicherstellen, dass dein lokales Repository auf dem neuesten Stand ist – besonders wenn du an mehreren Geräten arbeitest oder gelegentlich direkt auf GitHub kleine Änderungen machst.

```
git pull
```

In **PhpStorm** geht das über **Git → Pull** oder mit dem blauen Pfeil-nach-unten-Button in der Toolbar. Dieser kurze Schritt verhindert, dass du später mit Push-Konflikten kämpfen musst.

2. Entscheiden: Branch oder nicht? □□

Hier kommt die wichtigste Frage des Tages:

Situation	Empfehlung
Kleiner Bugfix, Tippfehler, CSS-Anpassung	Direkt auf `main` arbeiten
Neues Feature, größere Änderung, Experiment	Eigenen Branch erstellen
Du bist unsicher, ob die Änderung funktioniert	Eigenen Branch erstellen

Faustregel: Wenn du dir vorstellst, dass du die Änderung möglicherweise komplett verwerfen willst, gehört sie in einen eigenen Branch.

Einen neuen Branch erstellst du in PhpStorm über die **Branch-Anzeige** unten rechts → **New Branch**. Benenne ihn sprechend, z. B.:

- `feature/newsletter-anmeldung`
- `fix/login-redirect`
- `experiment/neues-layout`

3. Arbeiten und Committen: Die goldene Regel ✍

Committe früh und oft – aber nicht wahllos. Ein Commit sollte eine **logische Einheit** darstellen, also eine abgeschlossene kleine Änderung, die für sich allein Sinn ergibt.

Gute Zeitpunkte für einen Commit:

- Du hast eine neue Funktion fertiggestellt (auch wenn sie klein ist)
- Du hast einen Bug behoben
- Du hast Refactoring abgeschlossen
- Du machst eine Pause oder wechselst das Thema
- Du hast etwas zum Laufen gebracht, das vorher nicht funktionierte

Schlechte Zeitpunkte für einen Commit:

- Der Code kompiliert/funktioniert nicht
- Du bist „mittendrin" in einer Änderung
- Du hast zehn verschiedene Dinge gleichzeitig geändert

> □□ **Tipp:** Lieber fünf kleine Commits mit klaren Nachrichten als ein riesiger Commit mit „diverse Änderungen".

Beispiel für einen typischen Vormittag:

```
09:15  Commit: "Kontaktformular HTML-Struktur erstellt"
09:45  Commit: "Validierung für E-Mail-Feld hinzugefügt"
10:30  Commit: "Formular-Styling angepasst"
11:00  Commit: "E-Mail-Versand implementiert"
```

4. Feature fertig: Mergen □□

Wenn du in einem Feature-Branch gearbeitet hast und zufrieden bist:

1. **Wechsle zurück zu** `main` (in PhpStorm: Branch-Anzeige → `main` → Checkout)
2. **Merge den Feature-Branch** (in PhpStorm: Branch-Anzeige → deinen Feature-Branch → Merge into Current)
3. **Lösche den Feature-Branch** (optional, aber hält die Liste sauber)

Bei einem Einzelentwickler-Projekt entstehen hier selten Konflikte, weil niemand anders parallel an `main` arbeitet.

5. Arbeitsende: Pushen und sichern □□

Am Ende jedes Arbeitstages solltest du pushen – egal ob du „fertig" bist oder nicht. Der Push ist dein Backup in der Cloud. Wenn dein Laptop morgen kaputtgeht, ist deine Arbeit sicher.

```
git push
```

Oder in PhpStorm: **Git → Push** bzw. der grüne Pfeil-nach-oben-Button.

>

> ⚠ **Wichtig:** Pushe nur Branches, mit denen du aktiv arbeitest. Unfertige Feature-Branches kannst du auch pushen – sie liegen dann sicher auf GitHub, ohne `main` zu beeinflussen.

Zusammenfassung: Die wichtigsten Gewohnheiten

Wann	Was	Warum
Arbeitsbeginn	`git pull`	Aktuellen Stand holen, Konflikte vermeiden
Vor größeren Änderungen	Neuen Branch erstellen	Saubere Trennung, einfaches Verwerfen möglich
Nach jeder logischen Einheit	Committen mit guter Nachricht	Nachvollziehbare Historie, einfaches Zurückrollen
Feature abgeschlossen	Branch mergen	Änderungen in `main` integrieren
Arbeitsende	`git push`	Backup in der Cloud, Zugriff von überall

Zusätzliche Tipps für den Alltag □□

Commit-Nachrichten: Schreibe so, dass du in drei Monaten noch verstehst, was du getan hast. „Login gefixt" ist schlecht, „Redirect-Schleife beim Login behoben, wenn Session abgelaufen" ist gut.

Nicht committen: Halte dich an deine `.gitignore` – Dependencies (`vendor/`, `node_modules/`), IDE-Einstellungen und sensible Daten gehören nicht ins Repository.

Regelmäßigkeit schlägt Perfektion: Es ist besser, jeden Tag etwas zu pushen, als alle zwei Wochen einen perfekten Mega-Commit zu machen. Git ist ein Werkzeug für deinen Alltag, nicht für besondere Anlässe.

Der Feature-Branch-Workflow – sauber entwickeln in PhpStorm □□

Direkt auf dem `main`-Branch zu arbeiten ist **keine gute Idee** – selbst wenn du alleine arbeitest. Der sogenannte **Feature-Branch-Workflow** ist eine einfache, aber wirkungsvolle Methode, um dein Projekt sauber und sicher zu halten. Hier erfährst du, warum das so ist und wie du diesen Workflow in PhpStorm praktisch umsetzt.

Warum nicht direkt auf `main` arbeiten?

Der `main`-Branch (früher oft `master` genannt) sollte immer den **stabilen, funktionierenden Zustand** deines Projekts repräsentieren. Wenn du direkt darauf arbeitest, riskierst du mehrere Probleme:

- **Halbfertiger Code im Hauptbranch:** Du fängst ein Feature an, bist mittendrin – und plötzlich musst du einen dringenden Bug fixen. Jetzt ist dein `main` in einem unbrauchbaren Zwischenzustand.
- **Schwierige Fehlersuche:** Wenn alles in einem langen Strang von Commits liegt, ist es schwerer nachzuvollziehen, welche Änderungen zu welchem Feature gehören.
- **Kein einfaches Verwerfen:** Stellst du fest, dass ein Experiment nicht funktioniert, musst du mühsam einzelne Commits rückgängig machen, statt einfach einen Branch zu löschen.

Der Feature-Branch-Workflow löst all diese Probleme elegant.

Das Prinzip des Feature-Branch-Workflows

Die Idee ist simpel:

> **Für jedes neue Feature, jeden Bugfix oder jedes Experiment erstellst du einen eigenen Branch.** Erst wenn die Arbeit fertig und getestet ist, führst du den Branch in `main` zusammen.

```
flowchart TD
    MAIN["main-Branch\nImmer stabil und funktionsfaehig"]
    FEATURE1["feature/login-formular\nNeues Feature entwickeln"]
    FEATURE2["bugfix/navbar-fehler\nBug beheben"]

    MAIN -->|"Branch erstellen"| FEATURE1
    MAIN -->|"Branch erstellen"| FEATURE2
    FEATURE1 -->|"Merge nach Fertigstellung"| MAIN
    FEATURE2 -->|"Merge nach Fertigstellung"| MAIN

    style MAIN fill:#c8e6c9,color:#000000
    style FEATURE1 fill:#bbdefb,color:#000000
    style FEATURE2 fill:#ffe0b2,color:#000000
```

Dadurch bleibt `main` immer in einem Zustand, den du jederzeit deployen oder jemandem zeigen könntest.

Der Workflow Schritt für Schritt

1. Vor dem Start: Aktuellen Stand holen

Bevor du einen neuen Branch erstellst, solltest du sicherstellen, dass dein lokaler `main`-Branch auf dem neuesten Stand ist. In PhpStorm:

1. Wechsle zum `main`-Branch (falls nicht schon dort) – klicke dafür auf den **Branch-Namen unten rechts** und wähle `main`.
2. Führe einen **Pull** durch: **Git → Pull** oder `Ctrl+T` (Windows) / `Cmd+T` (Mac).

2. Neuen Feature-Branch erstellen

Jetzt erstellst du einen Branch für dein Vorhaben:

1. Klicke auf den **Branch-Namen unten rechts** in PhpStorm.
2. Wähle **New Branch** aus dem Popup-Menü.
3. Gib einen **aussagekräftigen Namen** ein, zum Beispiel:
   - `feature/kontaktformular` – für ein neues Feature
   - `bugfix/login-fehler` – für eine Fehlerbehebung
   - `experiment/neues-design` – für etwas, das du ausprobieren möchtest
4. Stelle sicher, dass **Checkout branch** aktiviert ist (damit du direkt auf den neuen Branch wechselst).
5. Klicke auf **Create**.

> 🧠 **Tipp zur Benennung:** Verwende Prefixe wie `feature/`, `bugfix/` oder `experiment/`, gefolgt von einer kurzen Beschreibung. Das macht deine Branch-Liste übersichtlich.

3. Arbeiten und regelmäßig committen

Jetzt arbeitest du ganz normal auf deinem neuen Branch:

- Schreibe Code, teste, verbessere.
- **Committe regelmäßig** kleine, in sich abgeschlossene Schritte – nicht erst am Ende alles auf einmal.
- Schreibe aussagekräftige Commit-Nachrichten, wie du es gelernt hast.

Das Schöne daran: Alles, was du hier commitest, landet **nur in diesem Branch**. Dein `main` bleibt davon unberührt.

4. Feature fertig? Zurück zu `main` wechseln

Wenn du mit deiner Arbeit fertig bist und alles getestet hast:

1. Wechsle zurück zum `main`-Branch – klicke auf den Branch-Namen unten rechts und wähle `main` → **Checkout**.

2. Optional, aber empfohlen: Hole dir den neuesten Stand mit **Pull**, falls sich zwischenzeitlich etwas geändert hat.

5. Feature-Branch in `main` mergen

Jetzt führst du deinen Feature-Branch in `main` zusammen:

1. Klicke wieder auf den **Branch-Namen unten rechts**.
2. Finde deinen Feature-Branch in der Liste (z. B. `feature/kontaktformular`).
3. Klicke darauf und wähle **Merge into Current** (oder „In aktuellen Branch mergen").
4. PhpStorm führt den Merge durch. Wenn es keine Konflikte gibt, ist alles sofort erledigt.
5. Bei einem **Merge-Konflikt** öffnet PhpStorm automatisch das Konflikt-Tool – löse die Konflikte wie gewohnt.

6. Änderungen pushen

Nach dem Merge solltest du deinen aktualisierten `main`-Branch auf GitHub pushen:

- **Git → Push** oder `Ctrl+Shift+K` (Windows) / `Cmd+Shift+K` (Mac).

7. Feature-Branch aufräumen (optional, aber empfohlen)

Der Feature-Branch hat seinen Zweck erfüllt. Du kannst ihn jetzt löschen, um deine Branch-Liste sauber zu halten:

1. Klicke auf den Branch-Namen unten rechts.
2. Finde den gemergten Feature-Branch.
3. Klicke darauf und wähle **Delete** – PhpStorm fragt dich, ob du ihn auch remote löschen möchtest (falls du ihn gepusht hattest).

Der komplette Ablauf visualisiert

```
flowchart TD
    A["1. Auf main wechseln\nund Pull durchfuehren"] --> B["2. Neuen Branch erstellen\nz.B. feature/warenkorb"]
    B --> C["3. Entwickeln und\nregelmaessig committen"]
```

```
    C --> D{"4. Feature fertig\nund getestet?"}
    D -->|"Nein"| C
    D -->|"Ja"| E["5. Zu main wechseln\nund Pull durchfuehren"]
    E --> F["6. Feature-Branch\nin main mergen"]
    F --> G["7. main auf\nGitHub pushen"]
    G --> H["8. Feature-Branch\nloeschen"]
    H --> I["Fertig!"]

    style A fill:#e3f2fd,color:#000000
    style B fill:#e3f2fd,color:#000000
    style C fill:#fff3e0,color:#000000
    style D fill:#fce4ec,color:#000000
    style E fill:#e3f2fd,color:#000000
    style F fill:#c8e6c9,color:#000000
    style G fill:#c8e6c9,color:#000000
    style H fill:#f3e5f5,color:#000000
    style I fill:#c8e6c9,color:#000000
```

Praktisches Beispiel: Ein Kontaktformular entwickeln

Angenommen, du möchtest ein Kontaktformular zu deinem PHP-Projekt hinzufügen:

1. **Pull auf** `main` – sicherstellen, dass du aktuell bist.
2. **Branch erstellen:** `feature/kontaktformular`
3. **Arbeiten:**
   - Commit 1: „HTML-Struktur für Kontaktformular erstellt"
   - Commit 2: „CSS-Styling hinzugefügt"
   - Commit 3: „PHP-Verarbeitung implementiert"
   - Commit 4: „E-Mail-Validierung ergänzt"
4. **Testen** – alles funktioniert wie gewünscht.
5. **Zu** `main` **wechseln** und Pull durchführen.
6. **Merge** von `feature/kontaktformular` in `main`.
7. **Push** des aktualisierten `main` auf GitHub.
8. **Branch löschen:** `feature/kontaktformular` wird nicht mehr benötigt.

Falls du mitten in der Arbeit einen dringenden Bug im Live-System fixen müsstest, wäre das kein Problem: Du könntest einfach zu `main` wechseln, einen neuen `bugfix/`-Branch erstellen, den Fehler

beheben, mergen und dann zu deinem Kontaktformular-Branch zurückkehren.

Zusammenfassung: Die goldenen Regeln

Regel	Warum?
`main` **ist immer stabil**	Du kannst jederzeit deployen oder demonstrieren
Ein Branch pro Aufgabe	Klare Trennung, einfaches Verwerfen
Aussagekräftige Branch-Namen	Du findest dich auch in einer Woche noch zurecht
Regelmäßig committen	Kleine Schritte sind leichter nachzuvollziehen
Nach dem Merge aufräumen	Keine Verwirrung durch alte Branches

Dieser Workflow mag anfangs wie ein Umweg erscheinen, aber nach kurzer Zeit wird er zur Gewohnheit – und du wirst dich fragen, wie du jemals ohne Branches gearbeitet hast. □□

Nützliche GitHub-Features für Einzelentwickler □□

Auch wenn du alleine arbeitest, bietet GitHub eine Reihe von Features, die dir helfen, den Überblick zu behalten, Ideen zu sammeln und deine Projekte professionell zu organisieren. Du musst nicht alles nutzen – aber die folgenden Werkzeuge sind auch für Solo-Projekte überraschend praktisch.

Issues – dein persönliches Aufgaben- und Ideenboard □□

Issues sind weit mehr als nur ein Bug-Tracker für Teams. Als Einzelentwickler kannst du sie nutzen als:

- **To-Do-Liste:** Jedes Feature, das du irgendwann einbauen möchtest, wird ein Issue. So vergisst du nichts und hast immer einen klaren Überblick, was noch ansteht.
- **Bug-Dokumentation:** Wenn dir ein Fehler auffällt, den du gerade nicht beheben kannst, erstellst du ein Issue mit einer kurzen Beschreibung. Später weißt du sofort, worum es ging.
- **Ideensammlung:** Vage Ideen, die du „irgendwann mal" umsetzen willst, parken perfekt in einem Issue – besser als in einem Notizzettel, der verloren geht.

Praktische Tipps für Issues

1. **Labels verwenden**
   GitHub erlaubt dir, Issues mit farbigen Labels zu versehen. Erstelle dir ein paar einfache Kategorien wie:
   - `bug` (rot) – Fehler, die behoben werden müssen
   - `feature` (grün) – Neue Funktionen
   - `idee` (blau) – Vage Konzepte für später
   - `dringend` (orange) – Priorität
2. **Milestones setzen**
   Wenn du auf eine bestimmte Version oder einen Veröffentlichungstermin hinarbeitest, kannst du einen **Milestone** erstellen (z. B. „Version 1.0" oder „Launch März") und Issues diesem zuordnen. So siehst du auf einen Blick, wie viel noch zu tun ist.

3. **Issue-Templates anlegen**
   Wenn du merkst, dass du bestimmte Informationen immer wieder brauchst (z. B. bei Bugs: „Was sollte passieren? Was passiert stattdessen?"), kannst du dir im Repository unter `.github/ISSUE_TEMPLATE/` Vorlagen anlegen.

> □□ **Tipp:** Du kannst Issues direkt in Commit-Nachrichten referenzieren. Schreibst du z. B. `Fix: Login-Fehler behoben, closes #12`, wird Issue #12 automatisch geschlossen, sobald der Commit im `main`-Branch landet.

Projects – visuelles Projektmanagement □□

GitHub Projects ist ein Kanban-ähnliches Board, das dir hilft, deine Issues und Aufgaben visuell zu organisieren. Selbst als Einzelentwickler kann das enorm nützlich sein:

Wie du Projects sinnvoll nutzt

Ein typisches Board für ein Solo-Projekt könnte drei Spalten haben:

```
flowchart LR
    A["□□Backlog\nAlles, was irgendwann gemacht werden soll"] --> B["□□In Arbeit\nWoran du gerade aktiv arbeitest"]
    B --> C["□ Erledigt\nAbgeschlossene Aufgaben"]
```

Der Vorteil gegenüber einer einfachen Issue-Liste:

- Du siehst **auf einen Blick**, wie viel „in der Pipeline" ist
- Du kannst Issues per **Drag & Drop** zwischen Spalten verschieben
- Es zwingt dich, **fokussiert** zu bleiben – wenn „In Arbeit" zu voll wird, merkst du sofort, dass du dich verzettelst

Einrichtung in GitHub

1. Gehe in deinem Repository auf den Reiter **Projects**
2. Klicke auf **New project**
3. Wähle ein Template (z. B. „Board") oder starte leer

4. Füge Spalten hinzu und ziehe bestehende Issues hinein

> □□ **Tipp:** GitHub hat Projects kürzlich überarbeitet (die neue Version heißt „Projects (beta)" oder manchmal einfach „Projects v2"). Diese neue Variante bietet mehr Flexibilität, z. B. eigene Felder, Filter und verschiedene Ansichten.

Releases – Versionen deiner Software veröffentlichen □□

Releases sind ein Weg, um **stabile Versionen** deines Projekts zu markieren und zum Download anzubieten. Das klingt vielleicht nach etwas, das nur große Open-Source-Projekte brauchen – aber auch als Einzelentwickler profitierst du davon:

Warum Releases sinnvoll sind

- **Klare Meilensteine:** Ein Release markiert einen funktionierenden Stand. Wenn du später etwas kaputt machst, weißt du genau, welche Version noch funktioniert hat.
- **Changelog führen:** In der Release-Beschreibung dokumentierst du, was sich seit der letzten Version geändert hat. Das hilft dir (und eventuellen Nutzern), den Überblick zu behalten.
- **Archiv für stabile Versionen:** Du kannst jederzeit auf eine frühere Release-Version zurückgreifen – nicht nur im Git-Log graben, sondern direkt ein fertiges Paket herunterladen.

Einen Release erstellen

1. Gehe im Repository auf **Releases** (rechte Seite oder unter „Code")
2. Klicke auf **Create a new release**
3. **Tag-Version wählen:** Erstelle einen neuen Tag (z. B. `v1.0.0` oder `v0.1.0-beta`) oder wähle einen bestehenden Tag aus. Tags sind im Grunde Lesezeichen auf bestimmte Commits.
4. **Titel und Beschreibung:** Gib dem Release einen aussagekräftigen Namen und beschreibe, was enthalten ist
5. Optional: **Dateien anhängen** – falls du z. B. ein fertiges ZIP mit deiner Anwendung bereitstellen willst

Versionierung verstehen

Eine bewährte Konvention ist **Semantic Versioning** (SemVer):

Format	Bedeutung
`v1.0.0`	Major.Minor.Patch
Major	Große, nicht abwärtskompatible Änderungen
Minor	Neue Features, abwärtskompatibel
Patch	Bugfixes, kleine Korrekturen

Für ein Solo-Projekt musst du das nicht streng befolgen, aber eine konsistente Nummerierung hilft trotzdem beim Überblick.

Weitere nützliche Features im Überblick

Wiki – Dokumentation für dein Projekt □□

Das **Wiki** ist ein einfacher Ort, um längere Dokumentation zu speichern:

- Installationsanleitungen
- Architektur-Entscheidungen
- Notizen zu Konfigurationsoptionen

Für kleine Projekte reicht oft eine gute `README.md`, aber wenn dein Projekt wächst, ist das Wiki ein praktischer Ort für alles, was nicht in die README passt.

GitHub Actions – Automatisierung ⚙

Actions sind GitHubs Werkzeug für Automatisierung (CI/CD). Auch als Einzelentwickler kannst du damit:

- **Automatische Tests** laufen lassen bei jedem Push
- **Code-Qualität** prüfen (z. B. mit PHP_CodeSniffer oder PHPStan)
- **Deployments** automatisieren (z. B. auf einen Webserver)

Das ist etwas fortgeschrittener, aber selbst ein einfacher Workflow, der bei jedem Push deine Tests ausführt, kann viel Ärger sparen.

Discussions – Gedanken sortieren □□

Falls du dein Repository irgendwann öffentlich machst oder einfach einen Ort für „größere Überlegungen" brauchst, die nicht direkt eine Aufgabe sind, bieten **Discussions** einen Rahmen dafür. Für rein private Solo-Projekte ist das meist überflüssig.

Ein einfacher Workflow für Solo-Projekte

Hier ist ein Vorschlag, wie du diese Features kombinieren kannst, ohne dich in Overhead zu verlieren:

```
flowchart TD
    IDEE["□□Neue Idee oder Bug entdeckt"] --> ISSUE["□□Issue erstellen\nmit passendem Label"]
    ISSUE --> BACKLOG["□□Issue ins Project-Board\nSpalte: Backlog"]
    BACKLOG --> START["□□Arbeit beginnen\nIssue in: In Arbeit"]
    START --> BRANCH["□□Feature-Branch erstellen\nz.B. feature/issue-15"]
    BRANCH --> COMMIT["□□Commits machen\nmit Issue-Referenz"]
    COMMIT --> MERGE["□□Branch mergen\nIssue wird geschlossen"]
    MERGE --> DONE["□ Issue in: Erledigt"]
    DONE --> RELEASE{Genug für\nneue Version?}
    RELEASE -->|Ja| TAG["□□Release erstellen\nmit Changelog"]
    RELEASE -->|Nein| IDEE
    TAG --> IDEE
```

Fazit: Weniger ist oft mehr □□

Du musst nicht alle Features nutzen. Für den Anfang empfehle ich:

1. **Issues** für Aufgaben und Bugs – das ist der größte Gewinn für die Organisation
2. **Releases** für wichtige Meilensteine – gibt dir Orientierung und Sicherheit
3. **Projects** optional, wenn du merkst, dass du den Überblick verlierst

Mit diesen Werkzeugen entwickelst du auch alleine **strukturierter und professioneller** – und falls dein Projekt irgendwann wächst oder du es öffentlich machst, hast du bereits eine solide Grundlage.