



Tag 2: Vertiefung Git-Workflow, CI/CD & GitLab CI

18.06.2024, Daniel Krämer & Malte Fischer

© Copyright 2024 anderScore GmbH

- **Tag 1 – Einführung in Git und GitLab, Git-Workflow im Team**
 - Einführung & Kursüberblick
 - Grundlagen von Git
 - Git Rebase und Merge-Strategien
 - Git Remote
 - Grundlagen von GitLab
 - Git-Workflow im Team
- **Tag 2 – Vertiefung Git-Workflow, CI/CD & GitLab CI**
 - Gitflow-Workflow
 - Tags, Releases & deren Verwaltung
 - GitLab-Runner
 - Einführung in GitLab CI/CD & gitlab.yml
- **Tag 3 – GitOps, Docker in der Entwicklung und Deployment-Strategien**
 - GitOps Grundlagen
 - Lokale Entwicklung mit Docker
 - Container/Docker-Registry
 - Erstellen von Release- und Tagged-Images
 - Möglichkeiten des Deployments & Verwaltung von Konfiguration
 - Abschlussübung & Diskussion

- **Tag 1 – Einführung in Git und GitLab, Git-Workflow im Team**
 - Einführung & Kursüberblick
 - Grundlagen von Git
 - Git Rebase und Merge-Strategien
 - Git Remote
 - Grundlagen von GitLab
 - Git-Workflow im Team
- **Tag 2 – Vertiefung Git-Workflow, CI/CD & GitLab CI**
 - Gitflow-Workflow
 - Tags, Releases & deren Verwaltung
 - GitLab-Runner
 - Einführung in GitLab CI/CD & gitlab.yml
- **Tag 3 – GitOps, Docker in der Entwicklung und Deployment-Strategien**
 - GitOps Grundlagen
 - Lokale Entwicklung mit Docker
 - Container/Docker-Registry
 - Erstellen von Release- und Tagged-Images
 - Möglichkeiten des Deployments & Verwaltung von Konfiguration
 - Abschlussübung & Diskussion

Git

Gitflow-Workflow

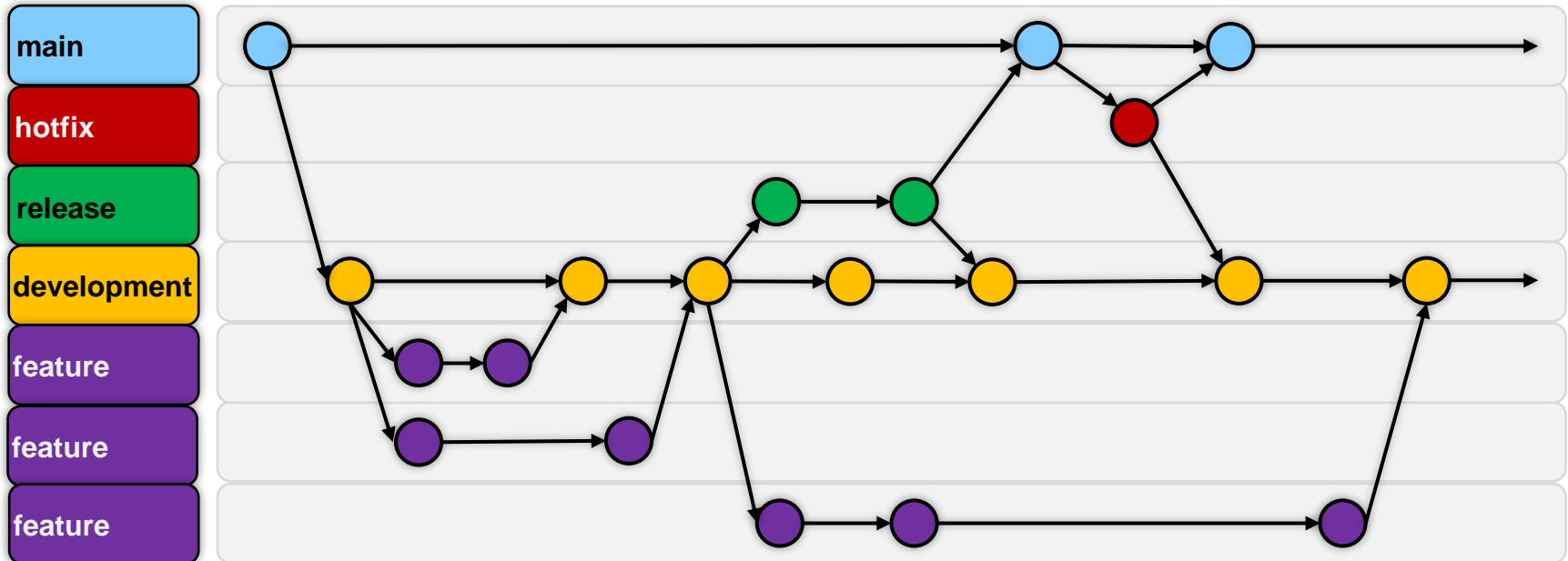
Inhalt

- Was ist der Gitflow-Workflow?
- Aufbau des Gitflow-Workflows
 - Branches und deren Verwendung
- Arbeiten mit Gitflow-Workflow
- Use-Cases und Fazit

Was ist der Gitflow-Workflow?

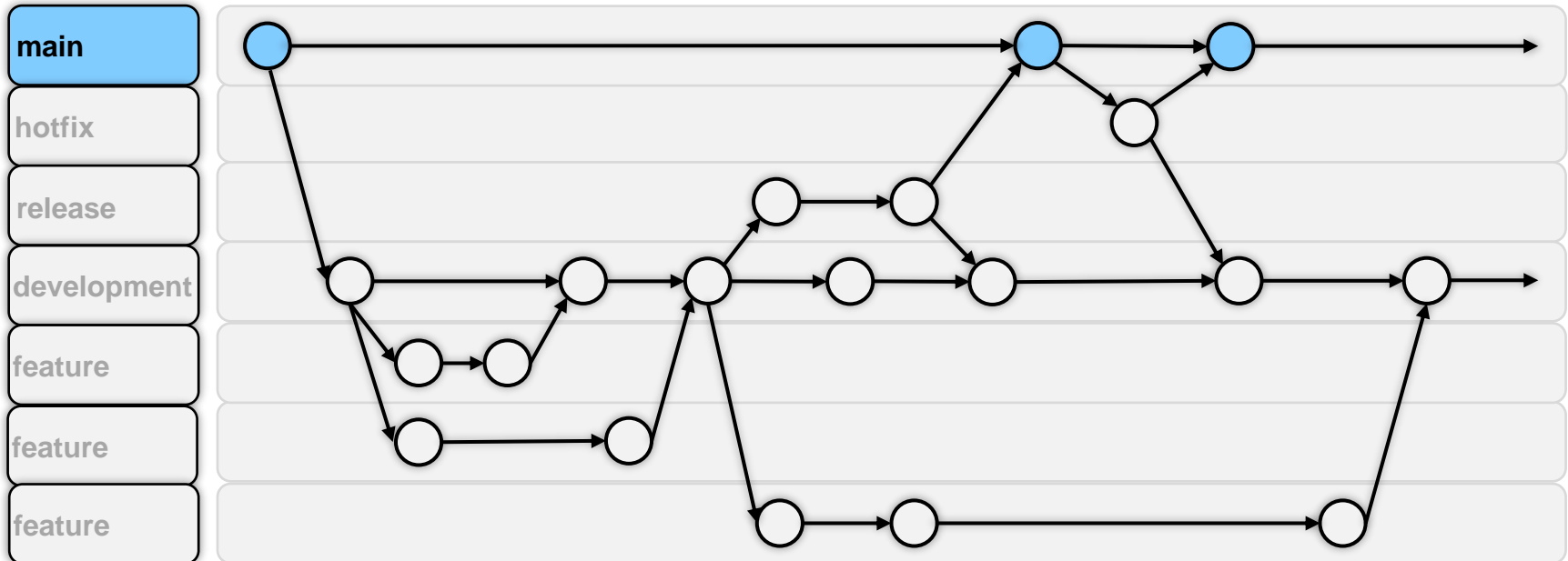
- Workflow mit Feature Branches und mehreren primären Branches
- 2010 von Vincent Dreissen auf nvie veröffentlicht
- Beliebt bei vielen Teams, teils als veraltet angesehen
- Immer noch weit verbreitet

- Besteht aus mehreren Branches mit zugeteilten Rollen
- main und development zu Beginn erstellt und existieren dauerhaft
- Release und Feature Branches erlauben getrenntes Arbeiten und isoliertes Experimentieren



Main Branch

- Enthält ausschließlich offizielle Releases
- Existiert fortlaufenden im Projekt
- Keine direkten Commits, nur Merge-Requests

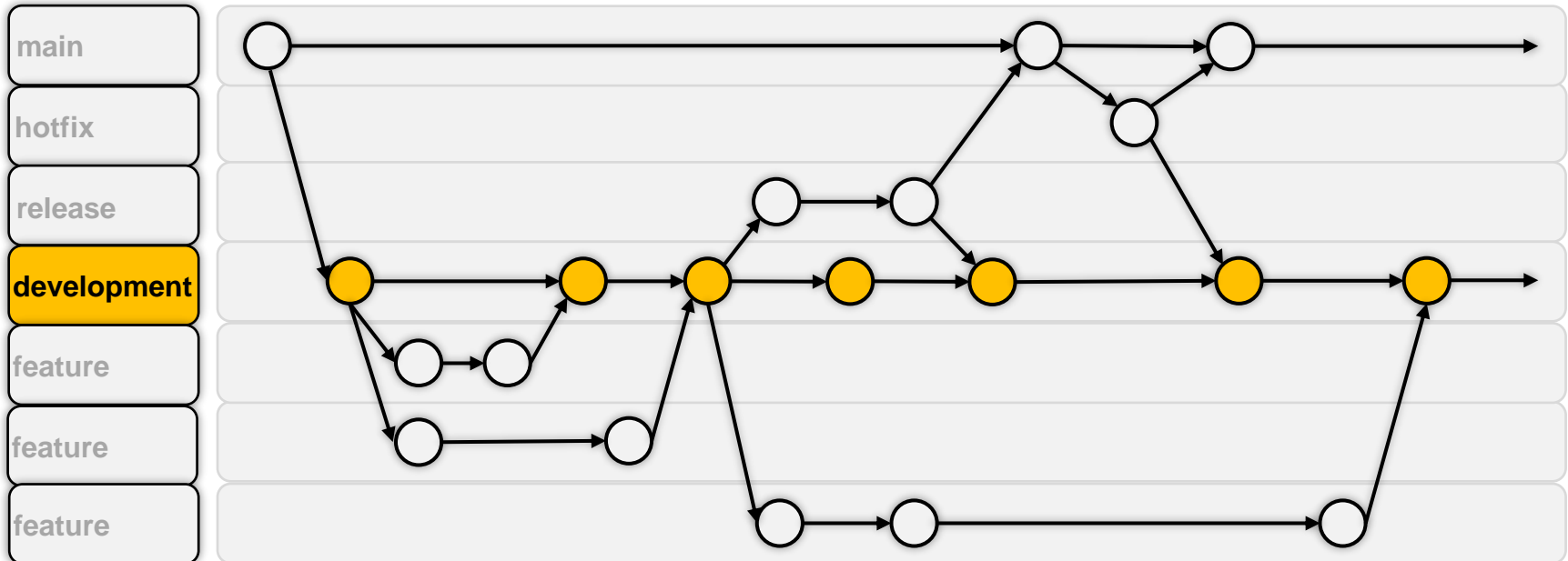


- Commits oft mit Release-Tags versehen
- Bietet Überblick über alle Releases
 - Ermöglicht auschecken älterer Version
 - Fehlerbehebung an alten Versionen möglich



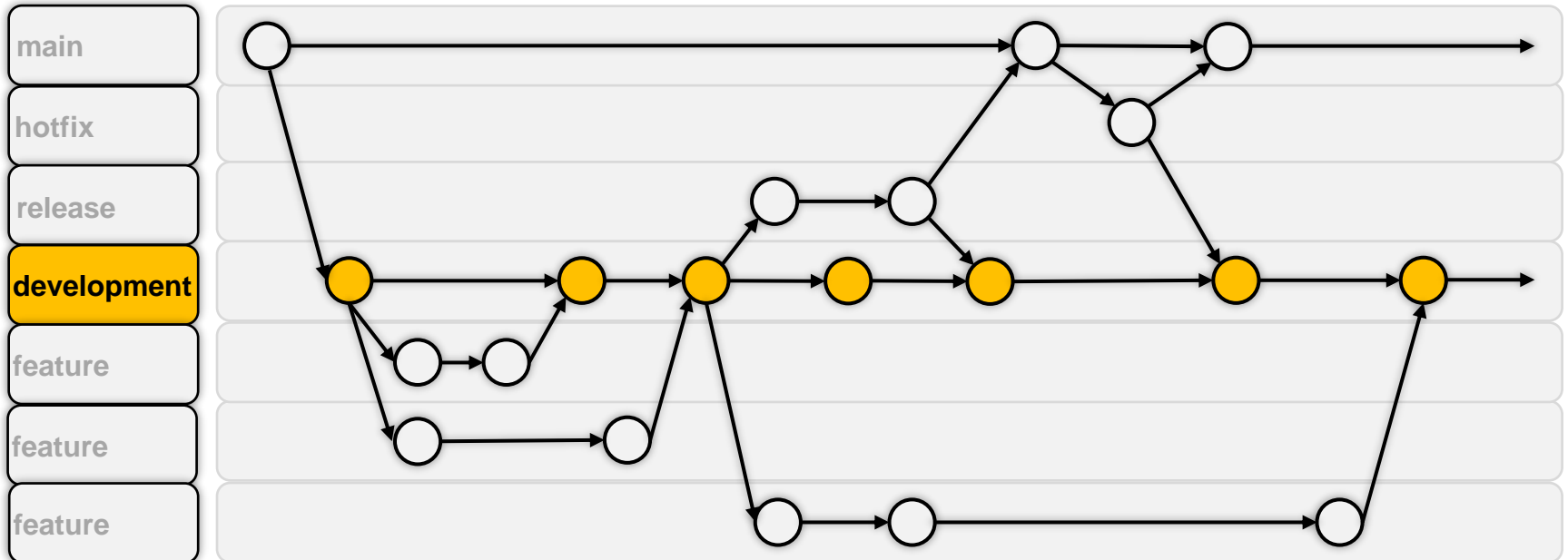
Development Branch

- aka **dev**
- Neben **main** der zweite dauerhaft existierende Branch
- Integration von Features und Bugfixes
- Sollte immer eine vollfunktionsfähige Version enthalten



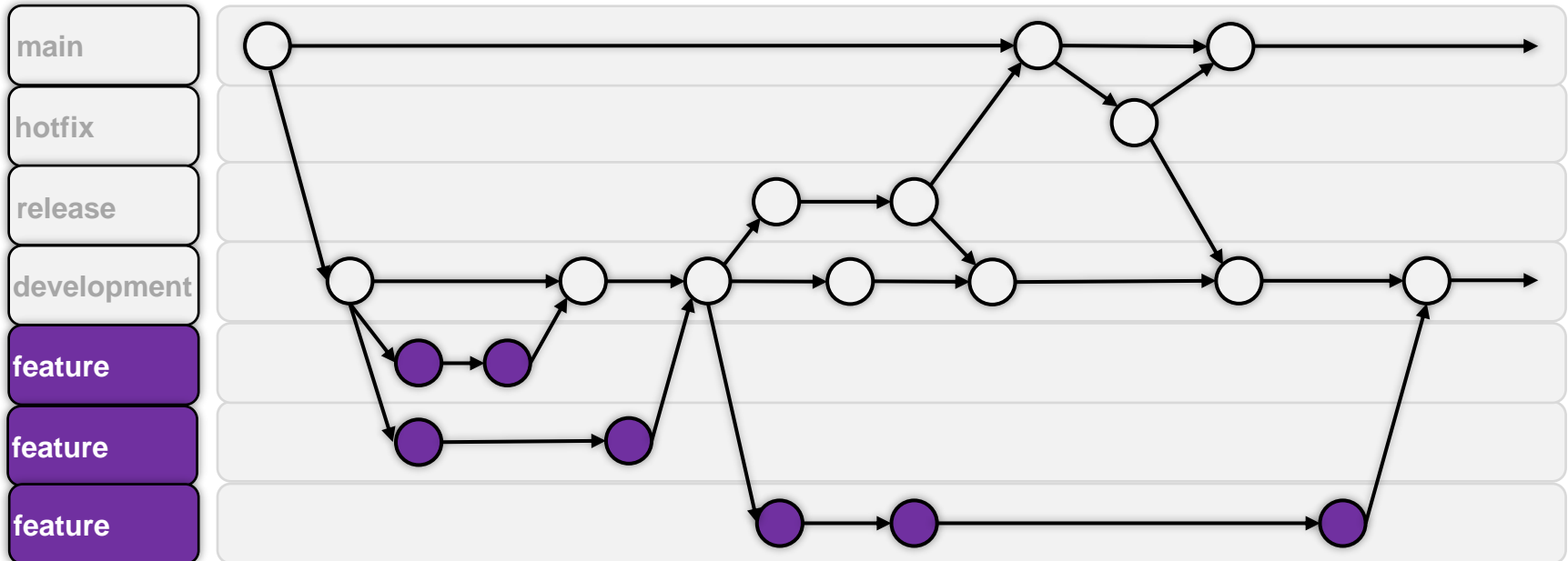
Development Branch

- Direkte Commits unüblich und zu vermeiden
- Enthält die komplette Historie des Projekts



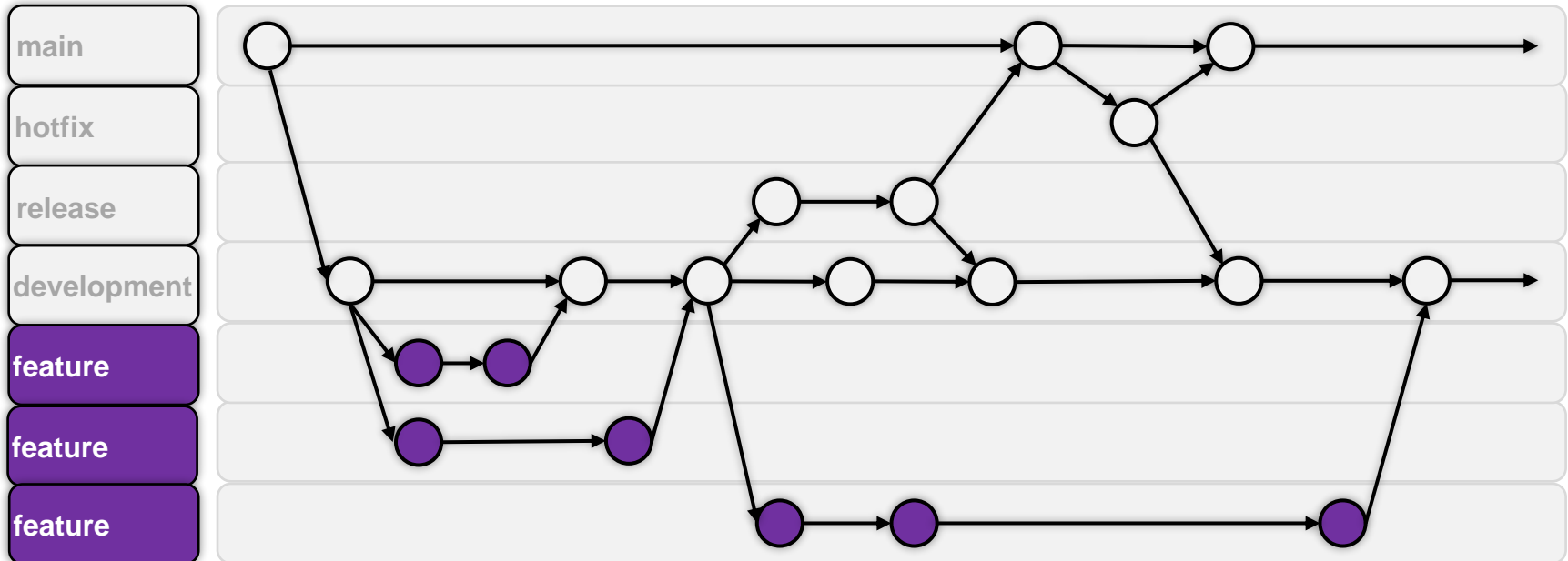
Feature Branches

- Zweigen von **dev** ab
- Entwickler arbeiten isoliert an Features
- Experimente, PoC, usw. möglich



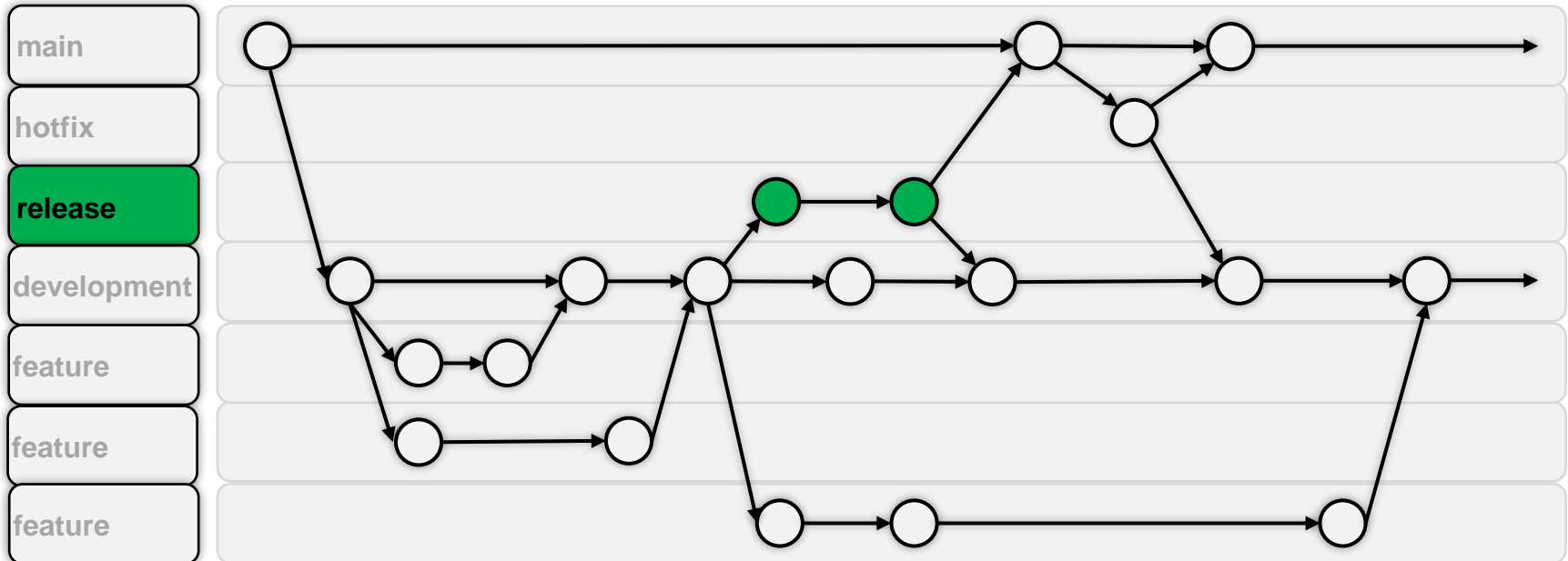
Feature Branches

- Langlebigere Branches als bei anderen Workflows
 - Können Feature-Integration erschweren
 - Entwickler für aktuellen Stand zuständig
 - ➔ Möglichst Feature regelmäßig auf Development rebasen



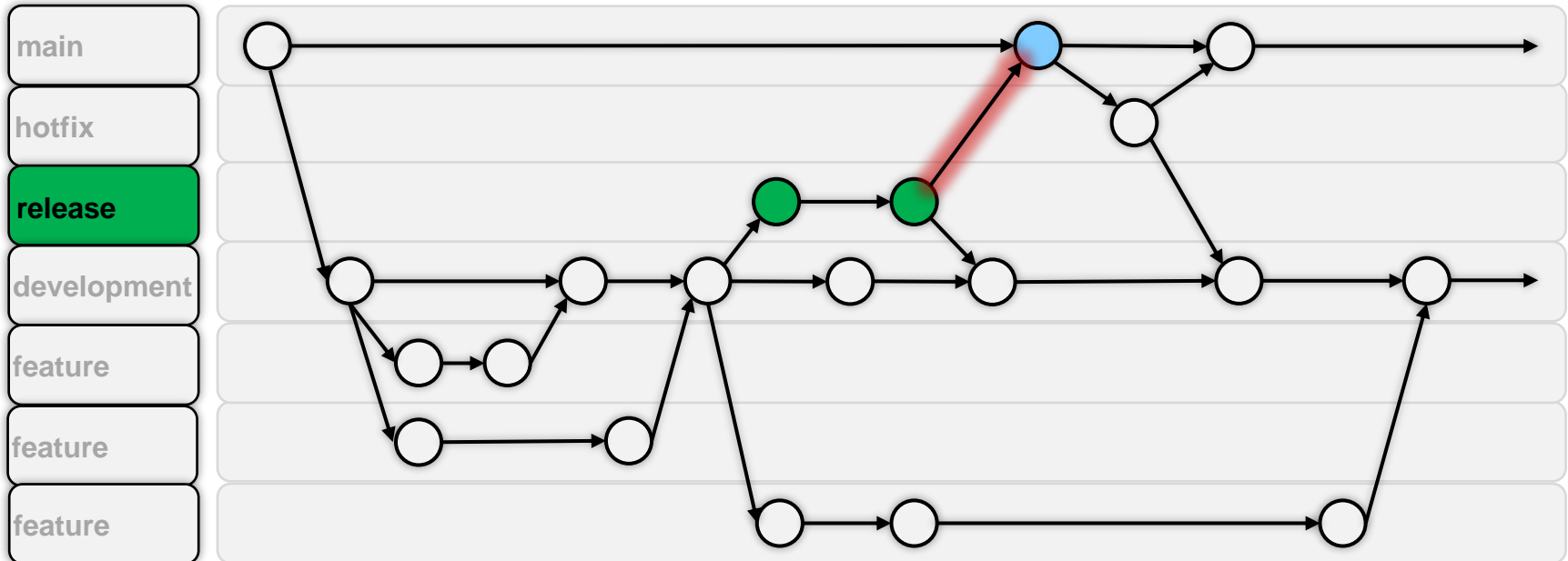
Release Branches

- Werden für die Release Phase verwendet
- Zweigen von **dev** ab, wenn dieser für ein Release bereit ist
- Keine neuen Features!



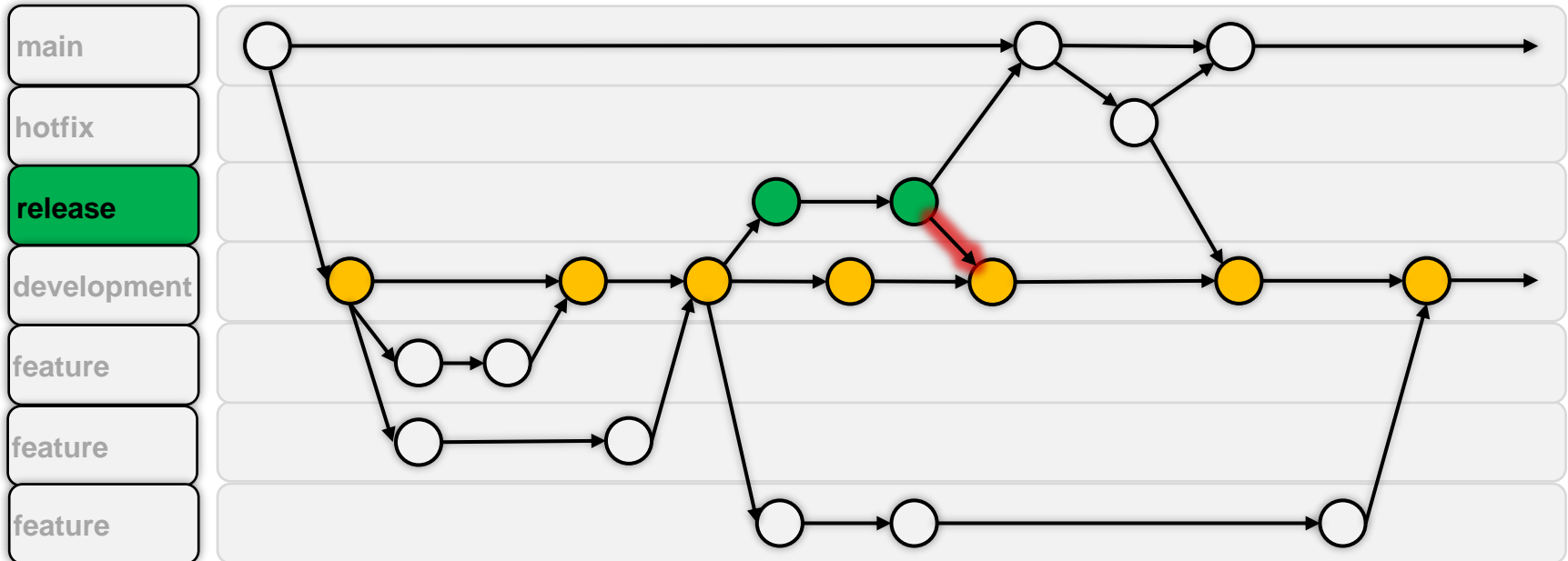
Release Branches

- Nur Bugfixes und Anpassungen für das Release
- Nach Abschluss in **main** mergen
- Neues Release → neuer Branch



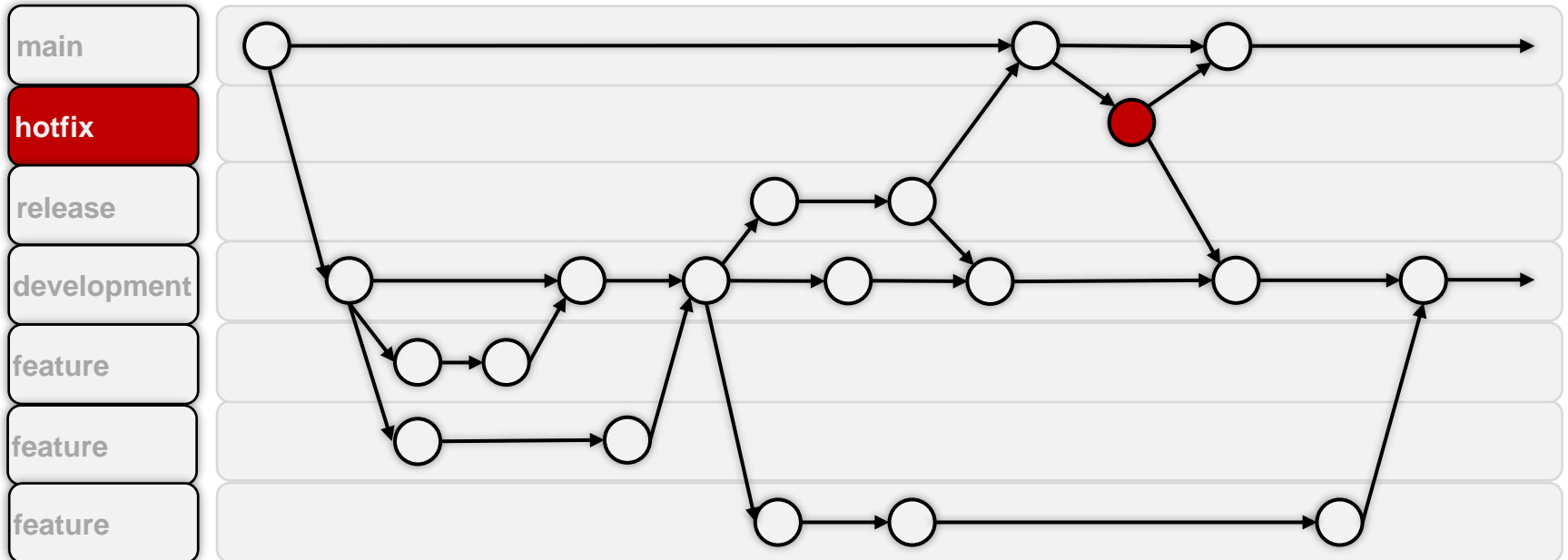
Release Branches

- **Wichtig:** Bugfixes und Anpassungen nach Release in **dev**
 - Sonst Anpassungen nur in **main**
 - Andernfalls gleichen Bugs erneut



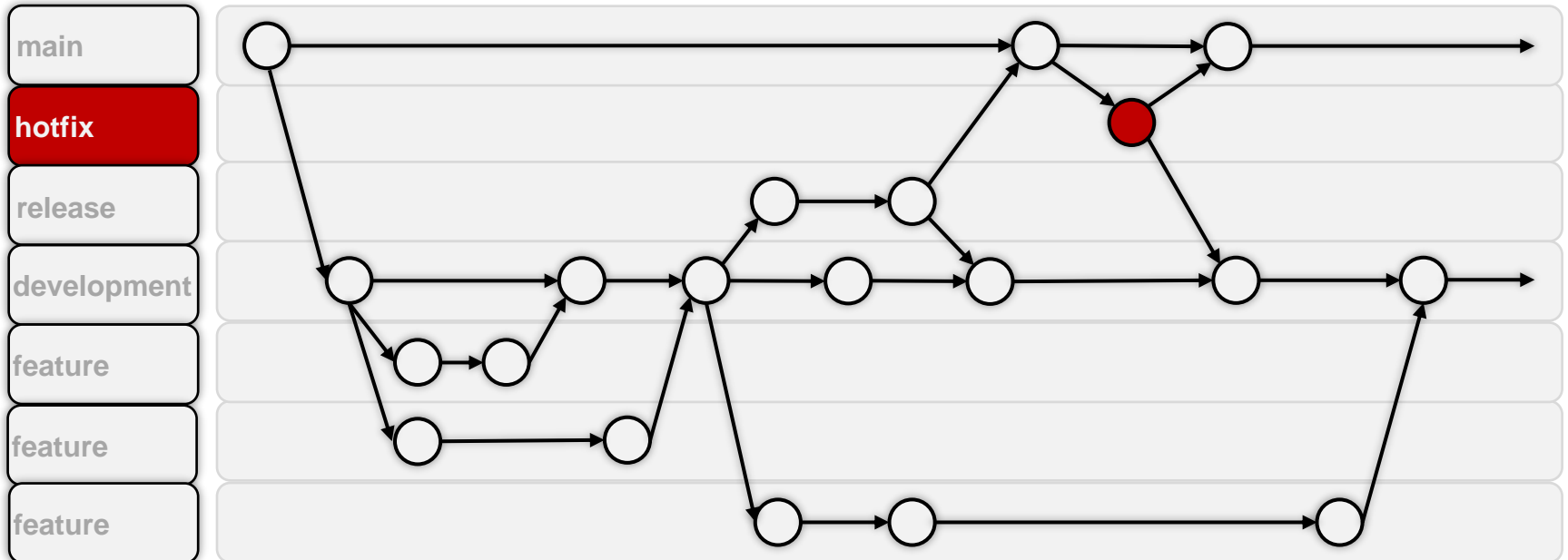
Hotfix Branches

- Hotfixes der Releases
- Zweigen vom **main** ab
- Hotfix fertig → zurück nach **main**
- Sehr dringende Bugfixes



Hotfix Branches

- **Wichtig:** Bugfixes nach **dev** (analog zu **main**)

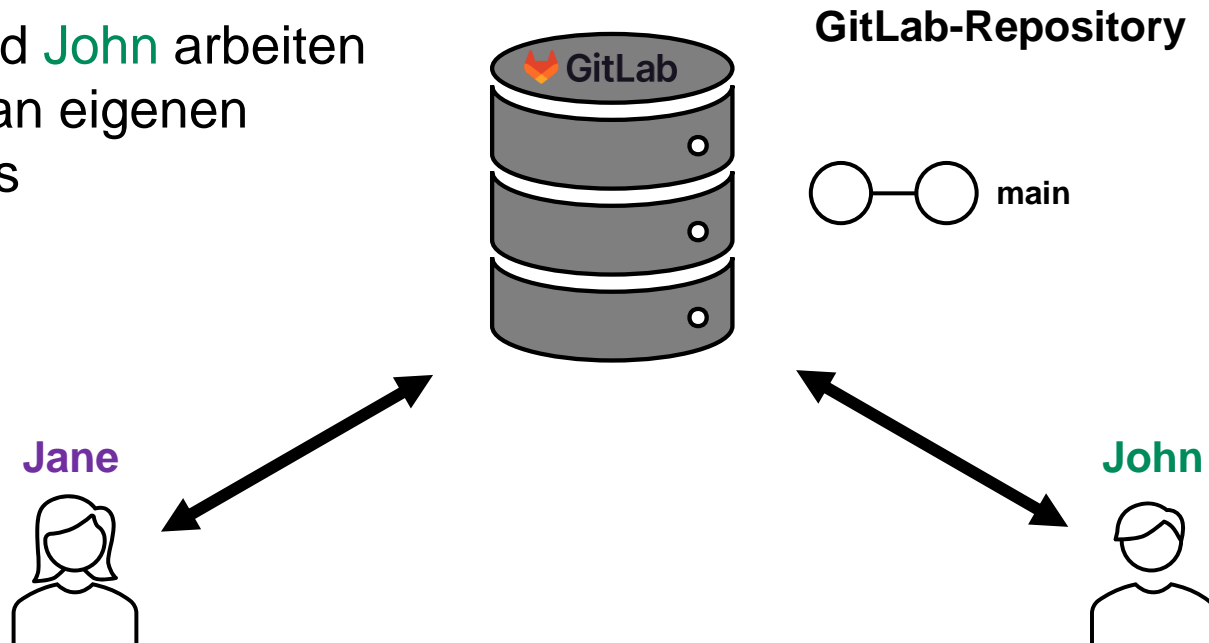


Git

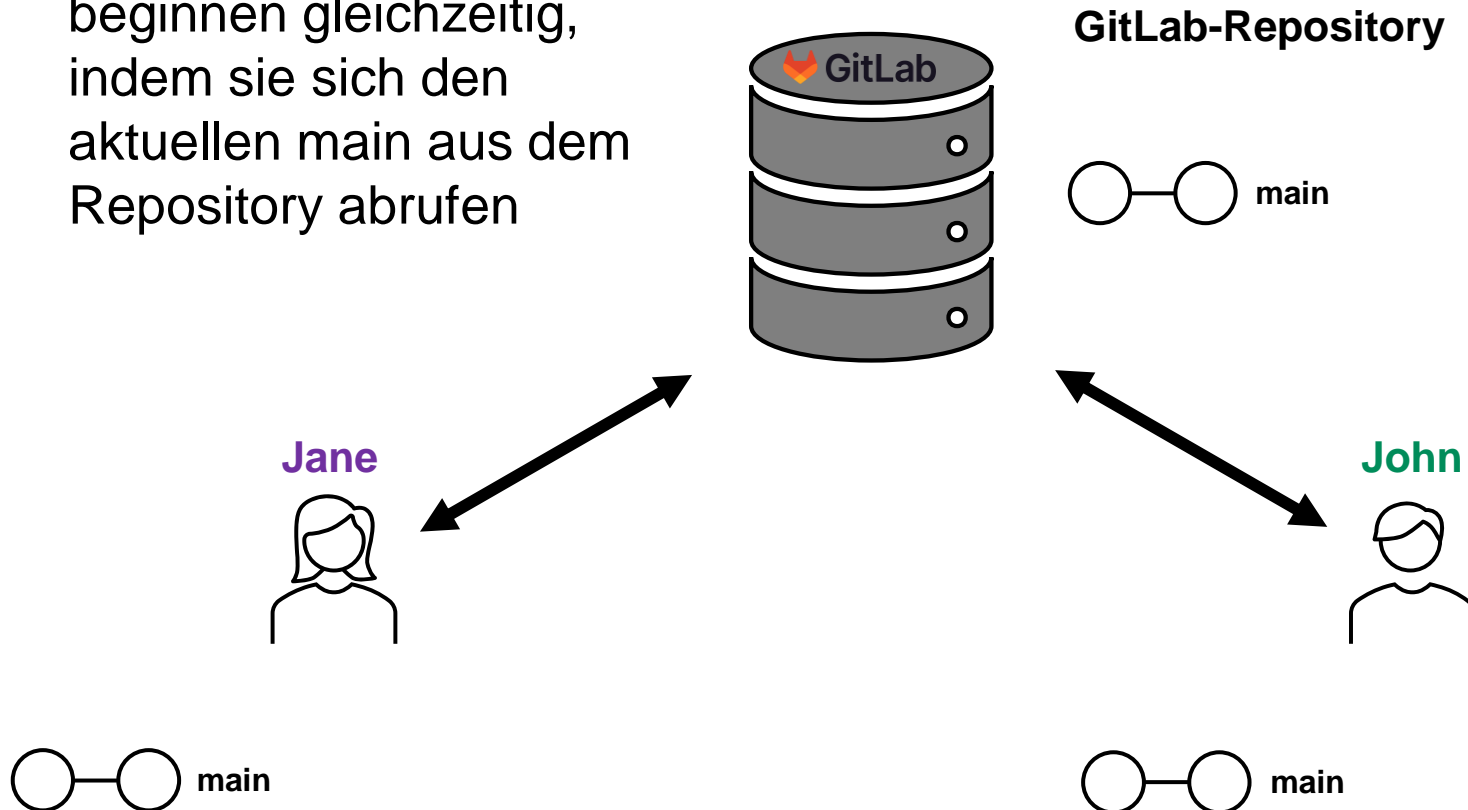
Arbeiten im Gitflow-Workflow

Beispielszenario

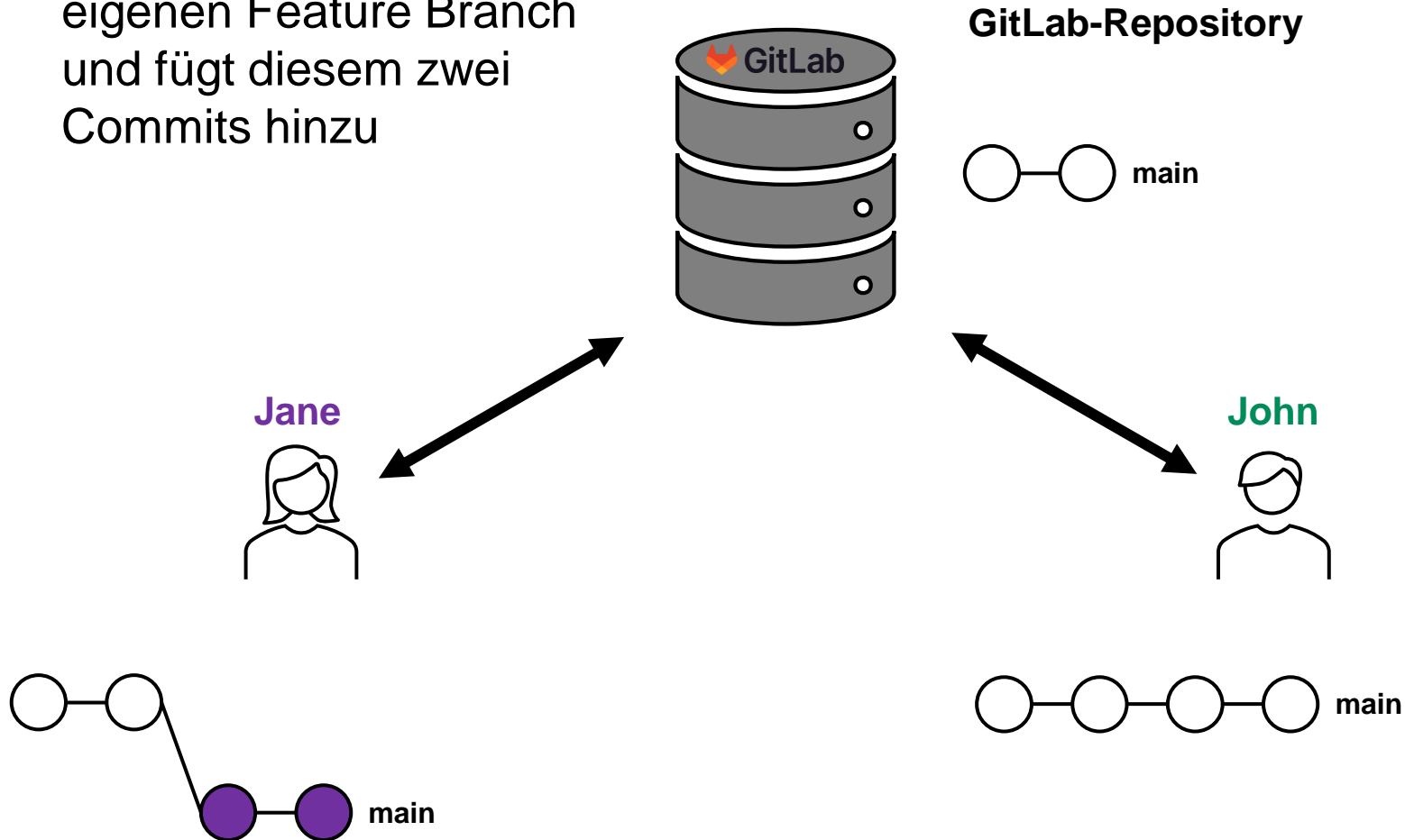
- Jane und John arbeiten jeweils an eigenen Features



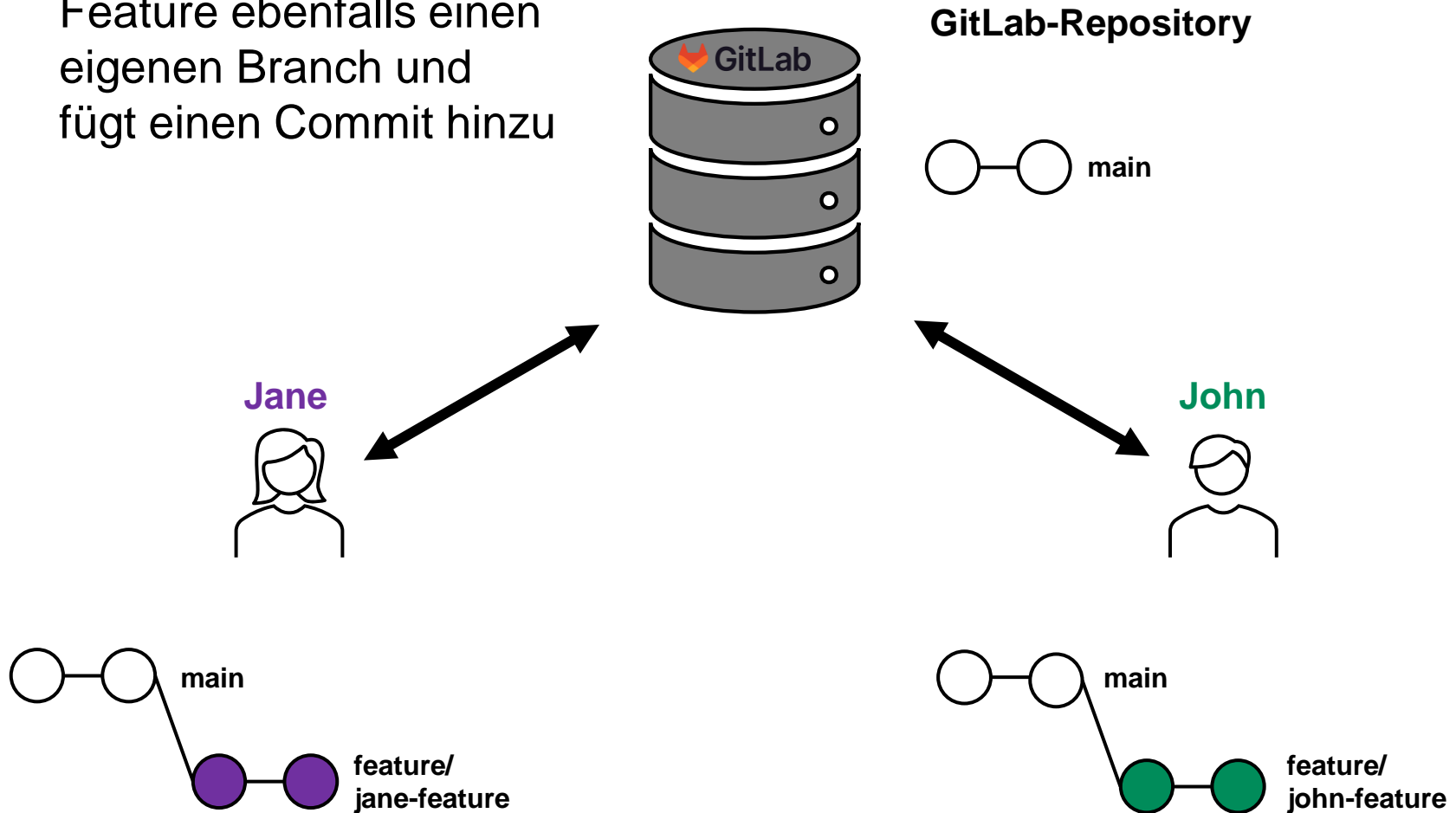
- Jane und John beginnen gleichzeitig, indem sie sich den aktuellen main aus dem Repository abrufen



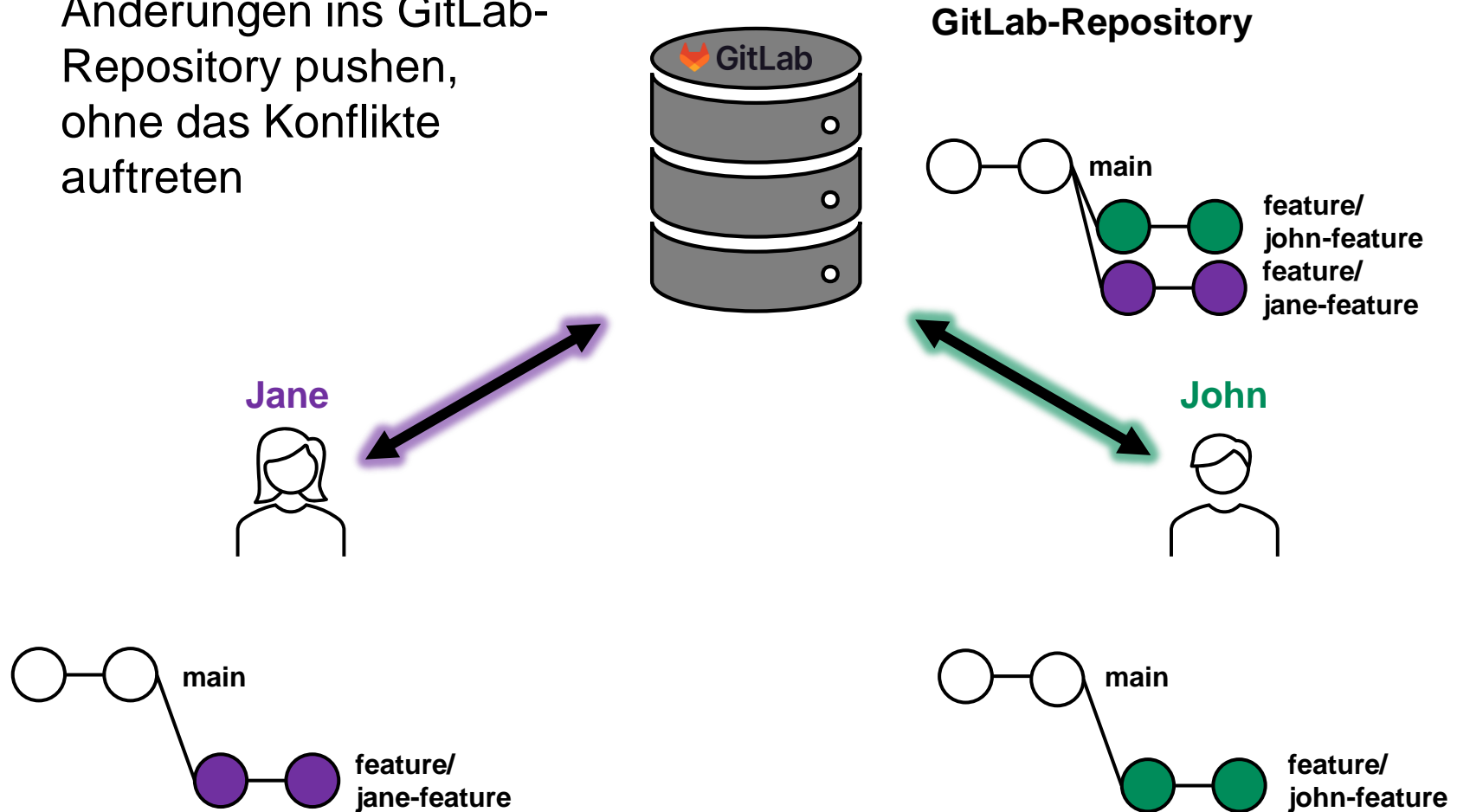
- Jane erstellt lokal einen eigenen Feature Branch und fügt diesem zwei Commits hinzu



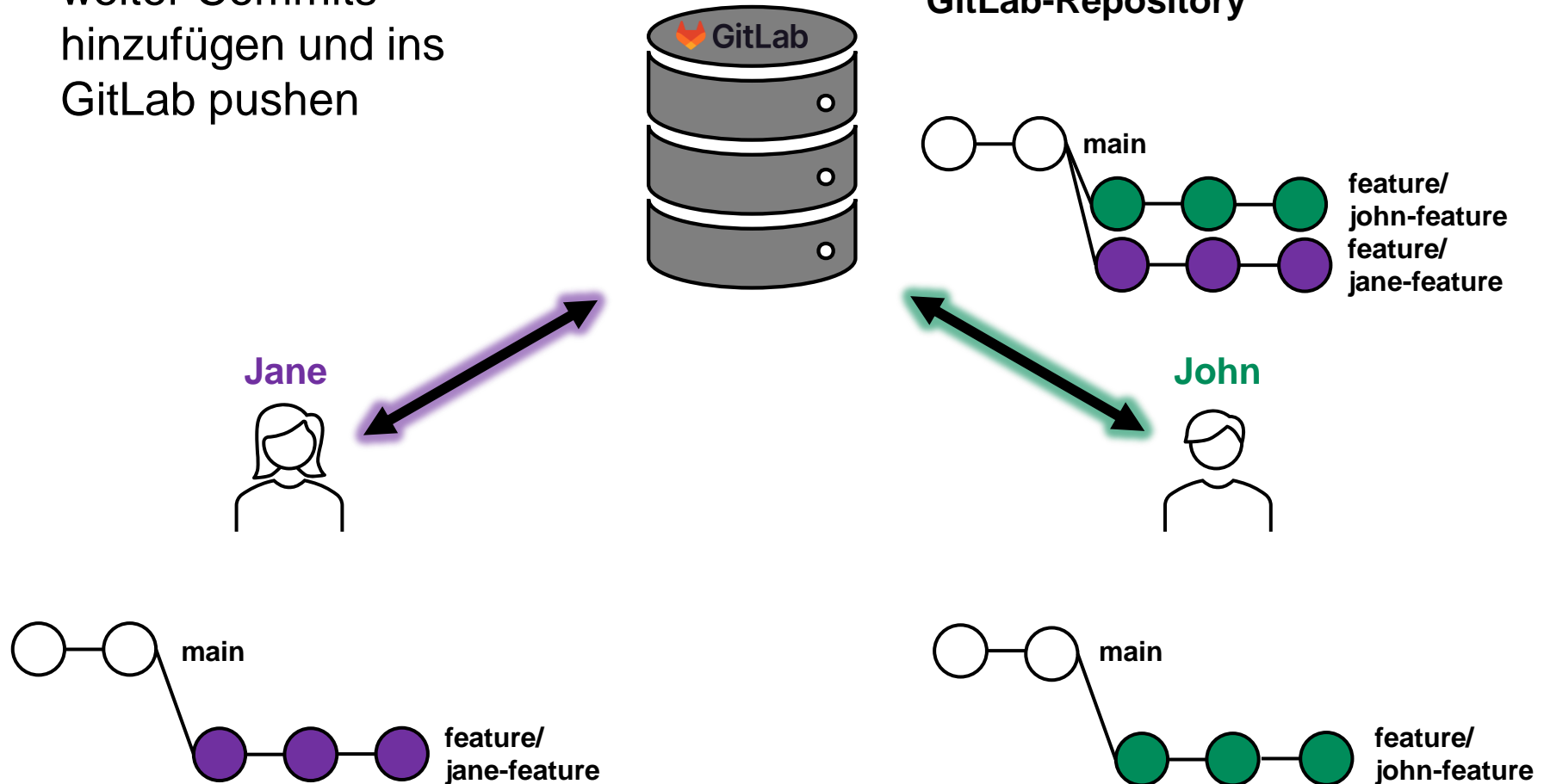
- John erstellt für sein Feature ebenfalls einen eigenen Branch und fügt einen Commit hinzu



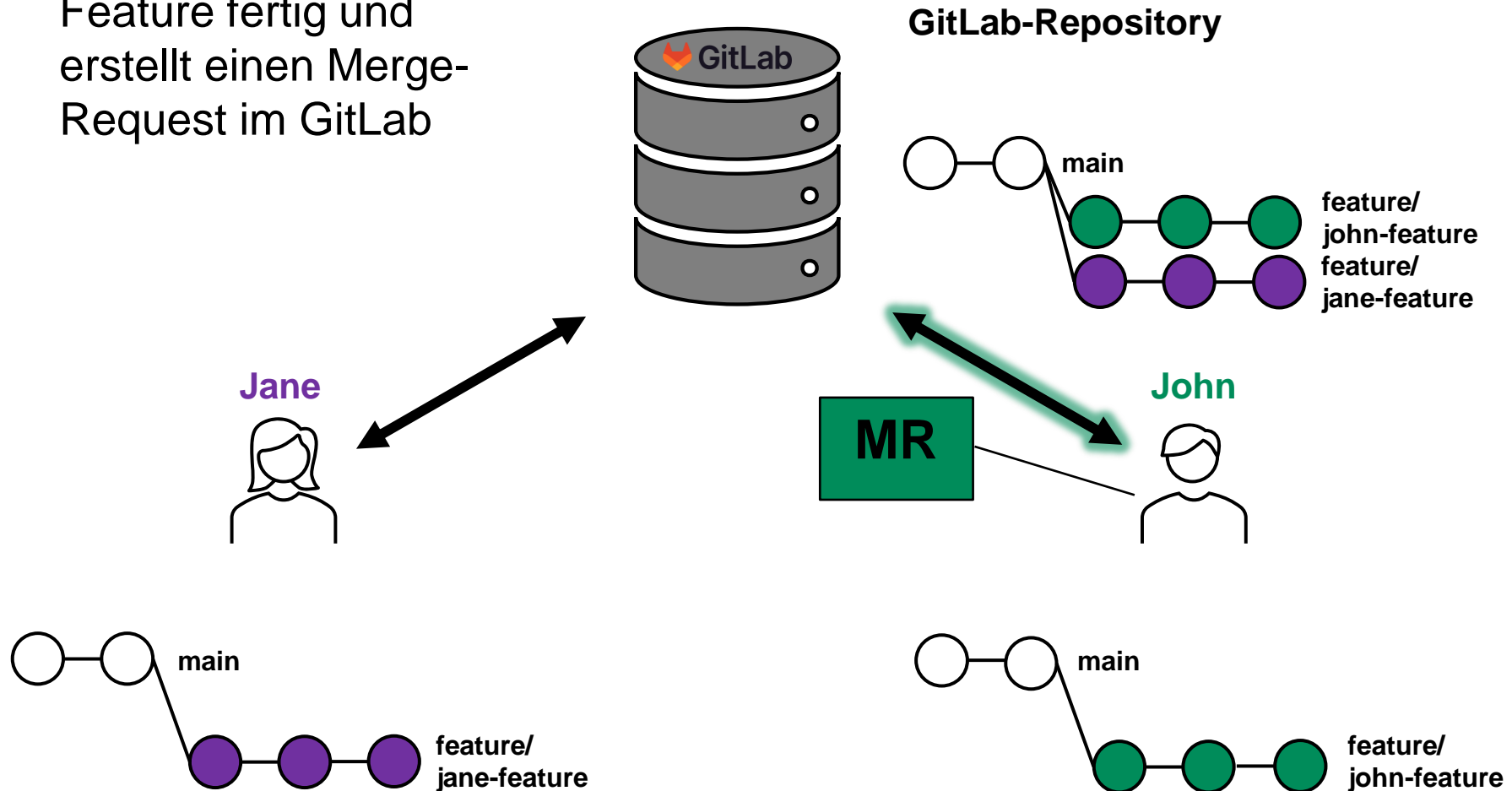
- Beide können ihre Änderungen ins GitLab-Repository pushen, ohne dass Konflikte auftreten



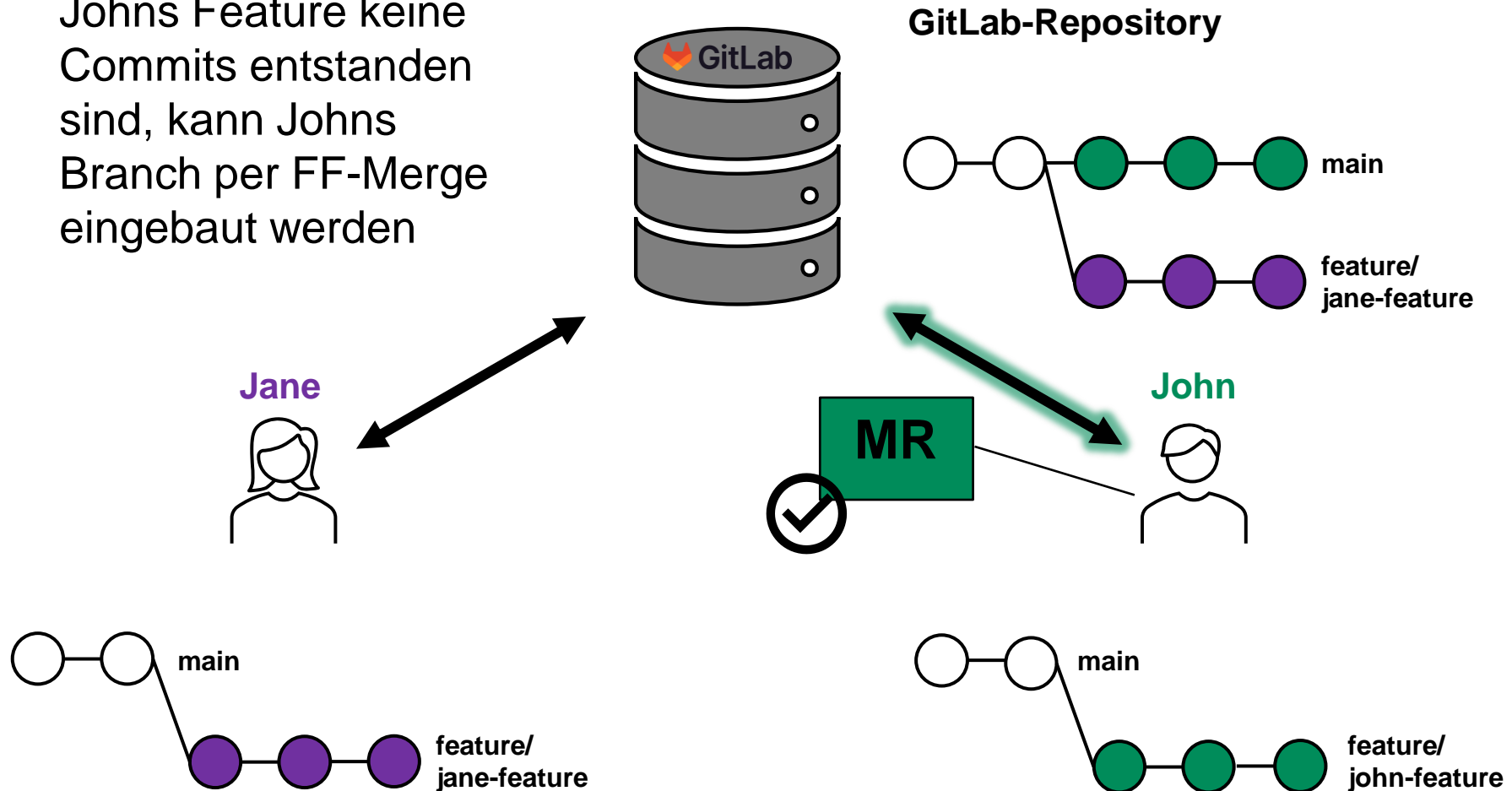
- Beide können beliebig weiter Commits hinzufügen und ins GitLab pushen



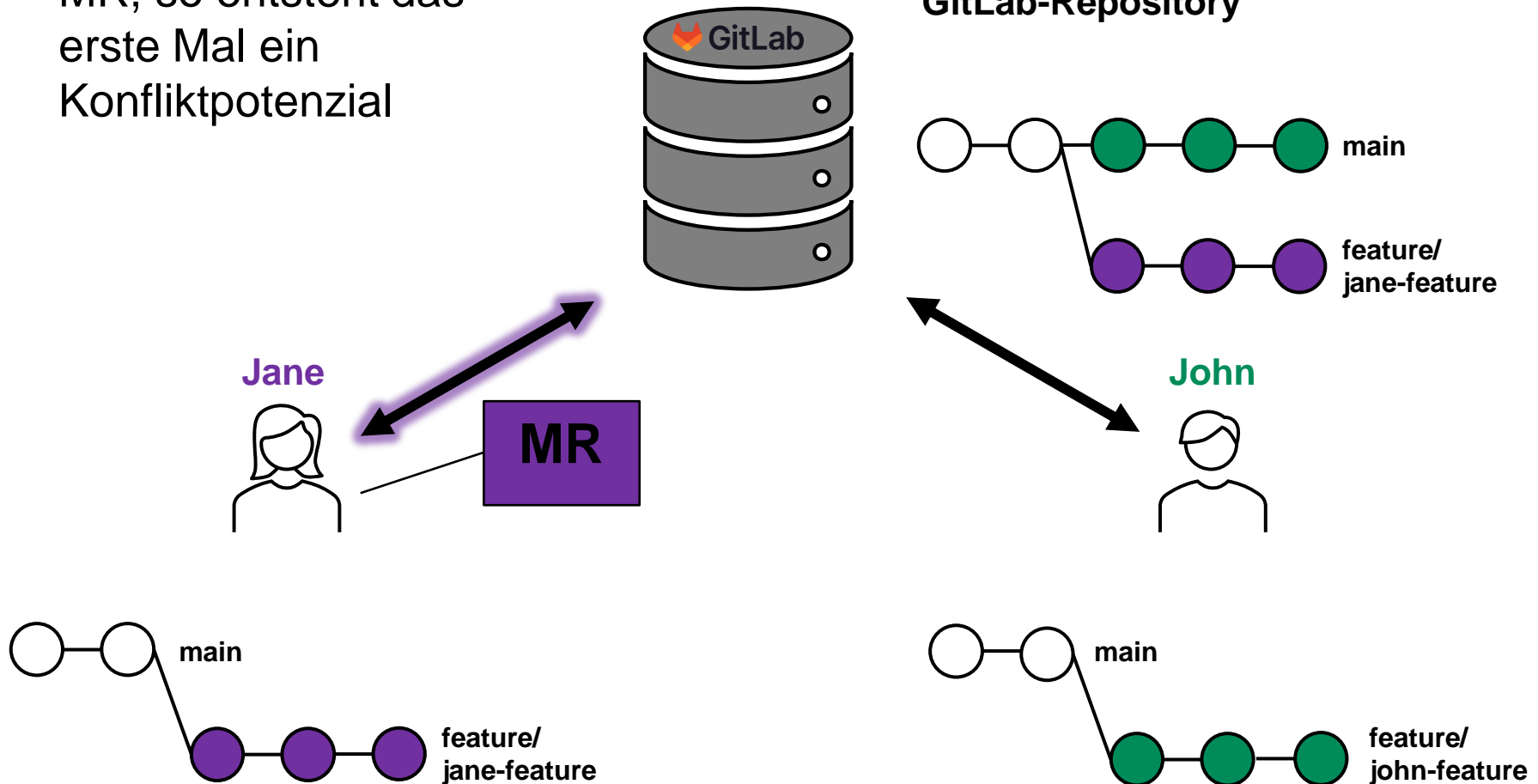
- John ist mit seinem Feature fertig und erstellt einen Merge-Request im GitLab



- Da auf dem main seit Johns Feature keine Commits entstanden sind, kann Johns Branch per FF-Merge eingebaut werden



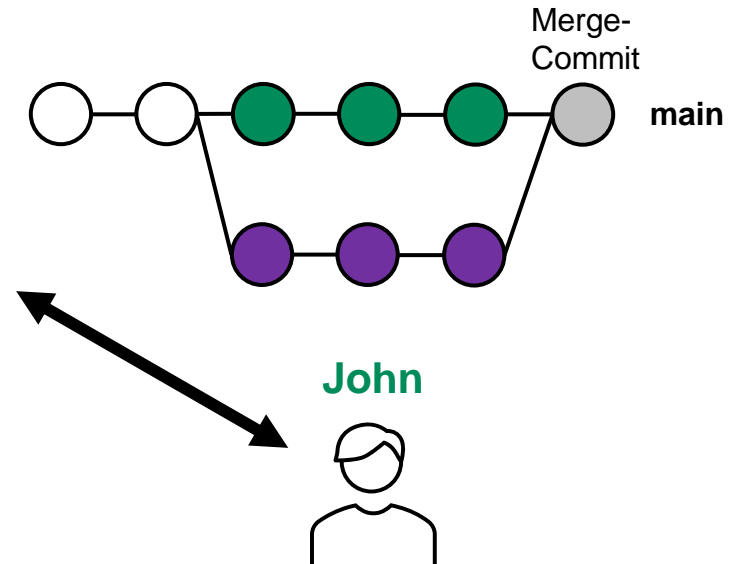
- Stellt nun **Jane** ihren MR, so entsteht das erste Mal ein Konfliktpotenzial



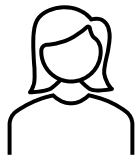
- Stehen **Janes** Änderungen nicht in Konflikt zu **Johns**, dann kann der Merge-Request ausgeführt werden



GitLab-Repository



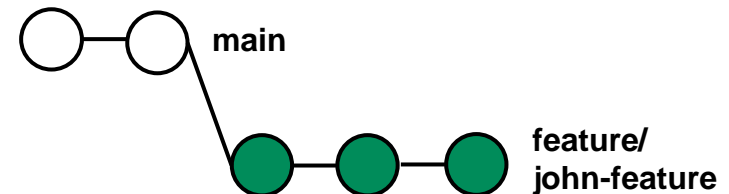
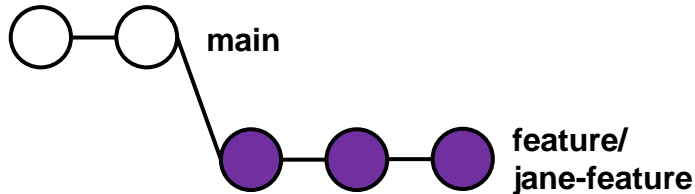
Jane



MR



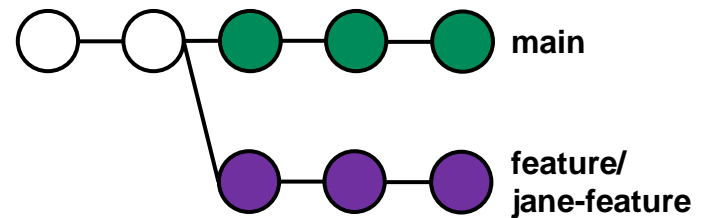
John



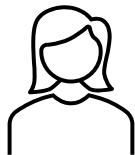
- Stehen **Janes** Änderungen im Konflikt mit **Johns** Änderungen, so kann der MR nicht gemerged werden



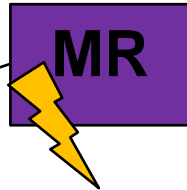
GitLab-Repository



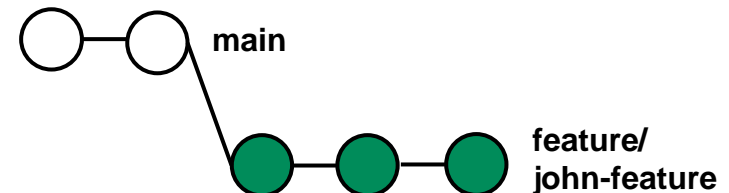
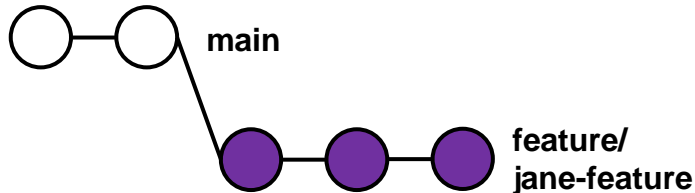
Jane



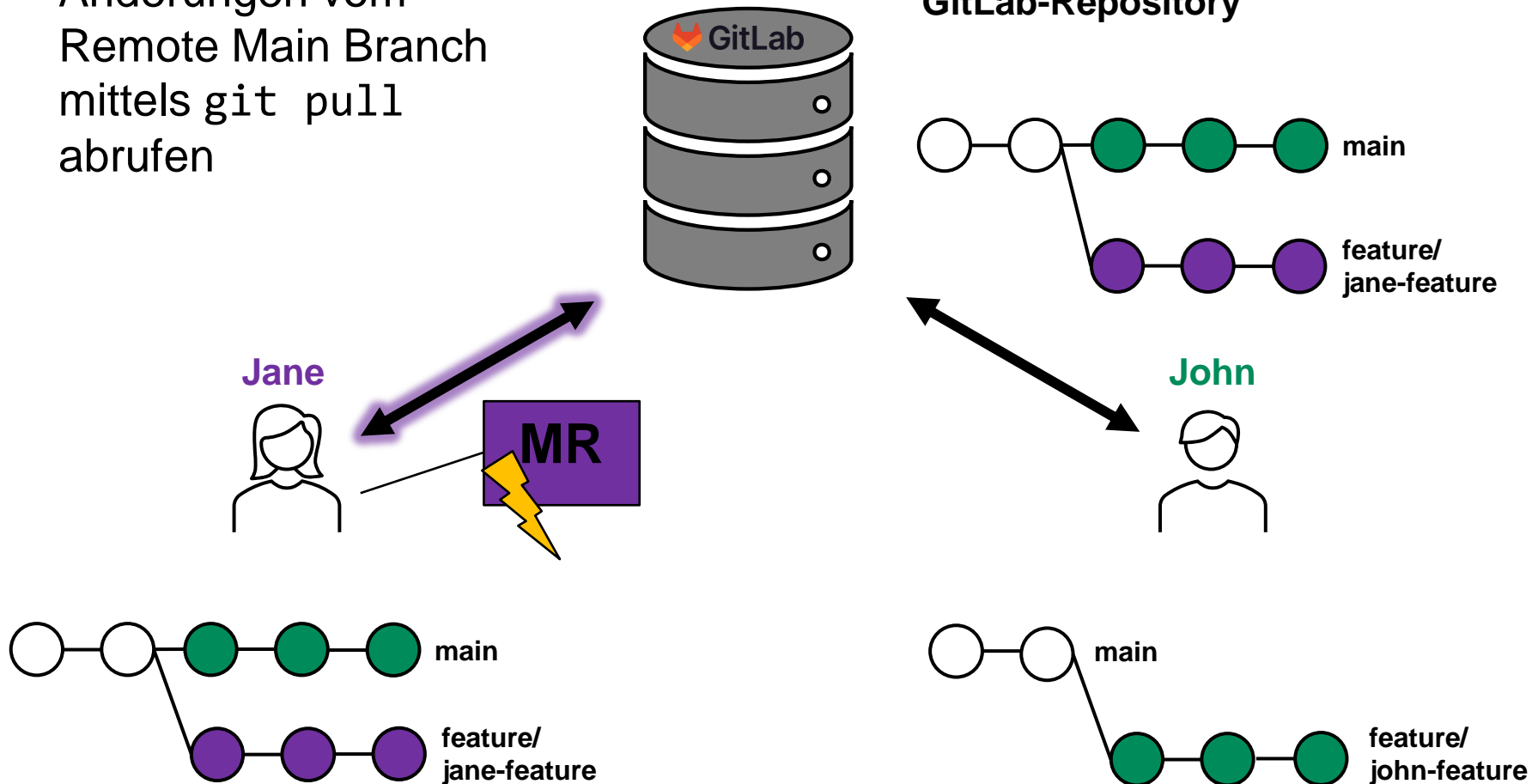
MR



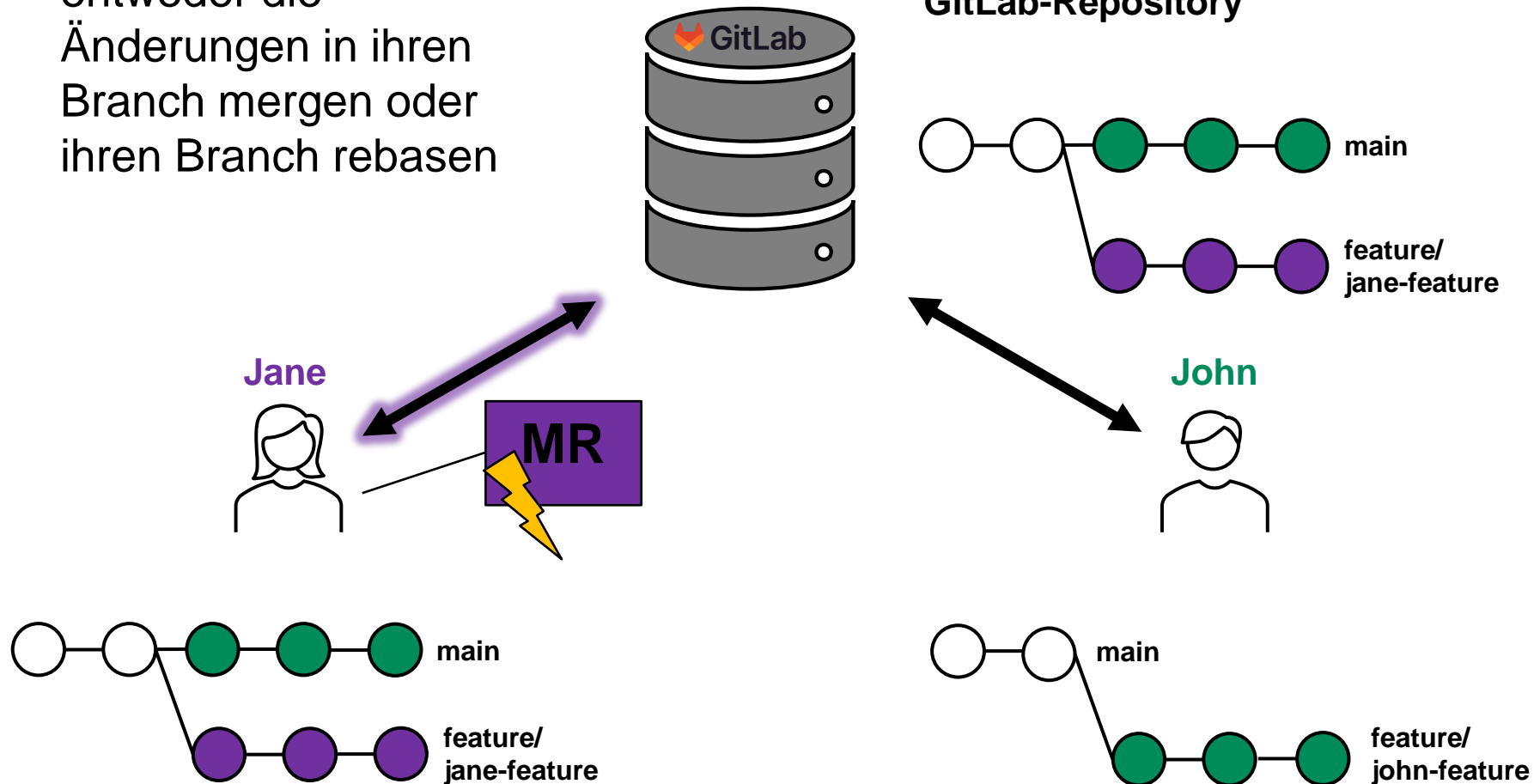
John



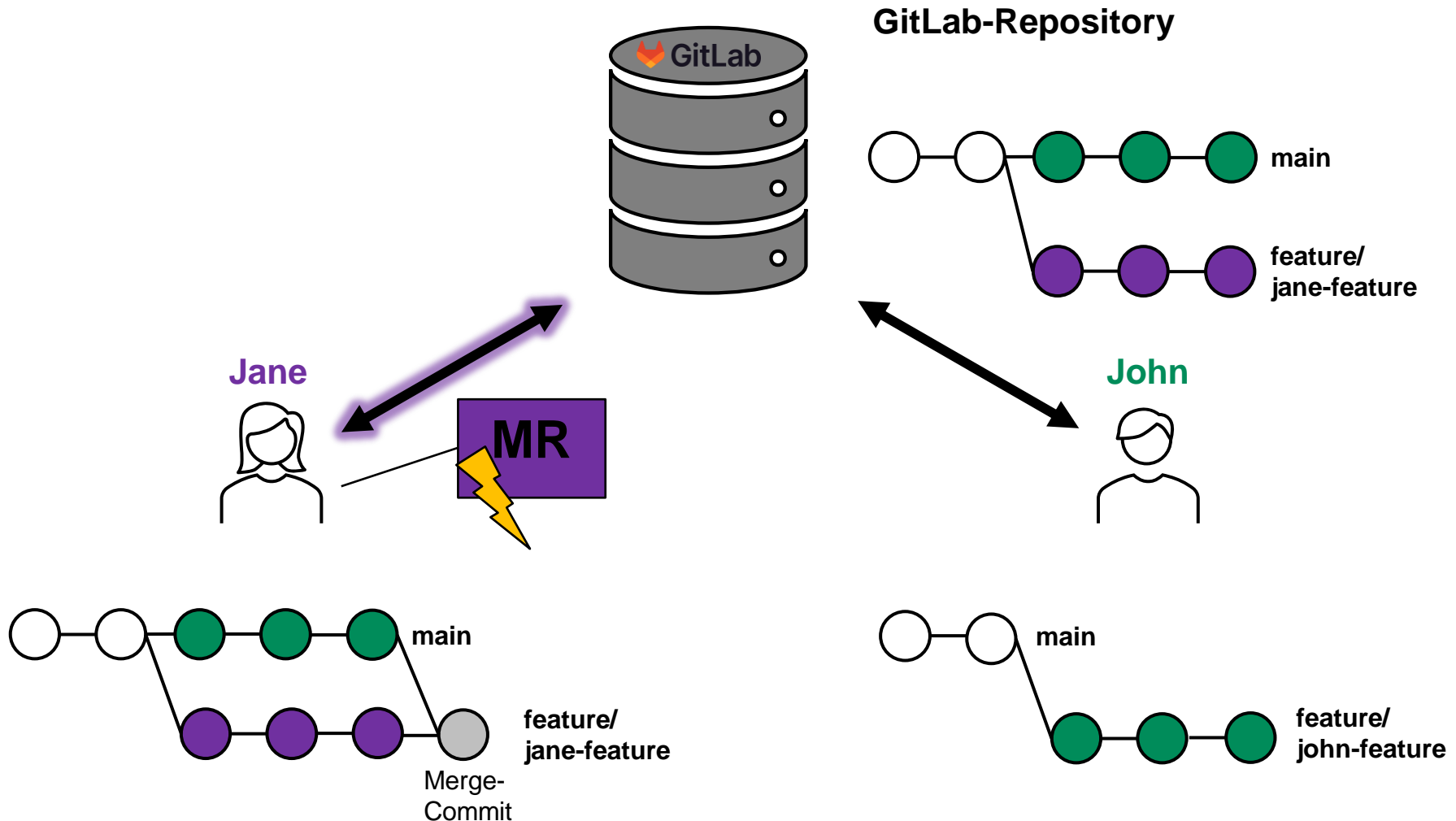
- Jane muss die Änderungen vom Remote Main Branch mittels `git pull` abrufen



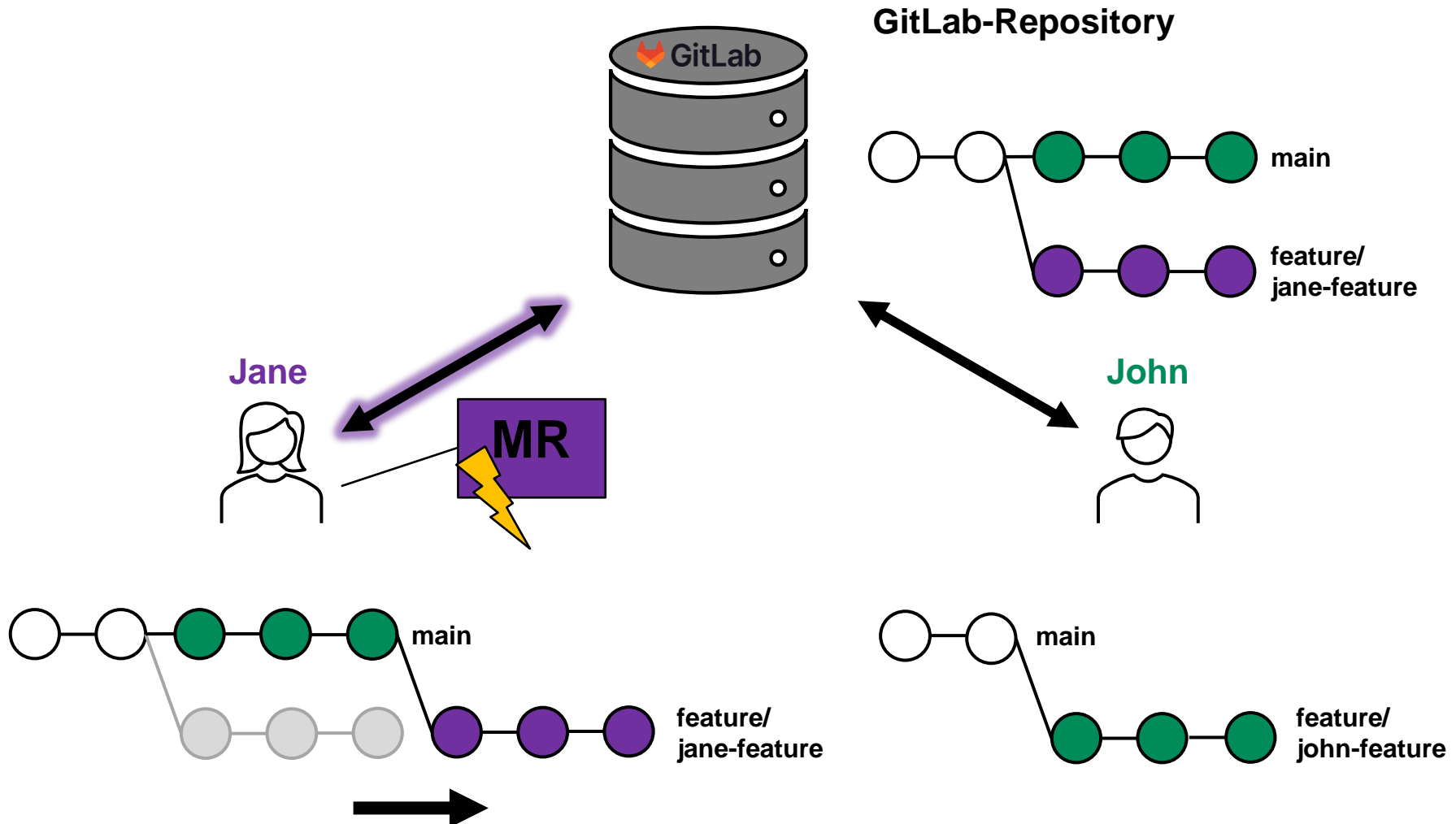
- Jane kann nun entweder die Änderungen in ihren Branch mergen oder ihren Branch rebasen



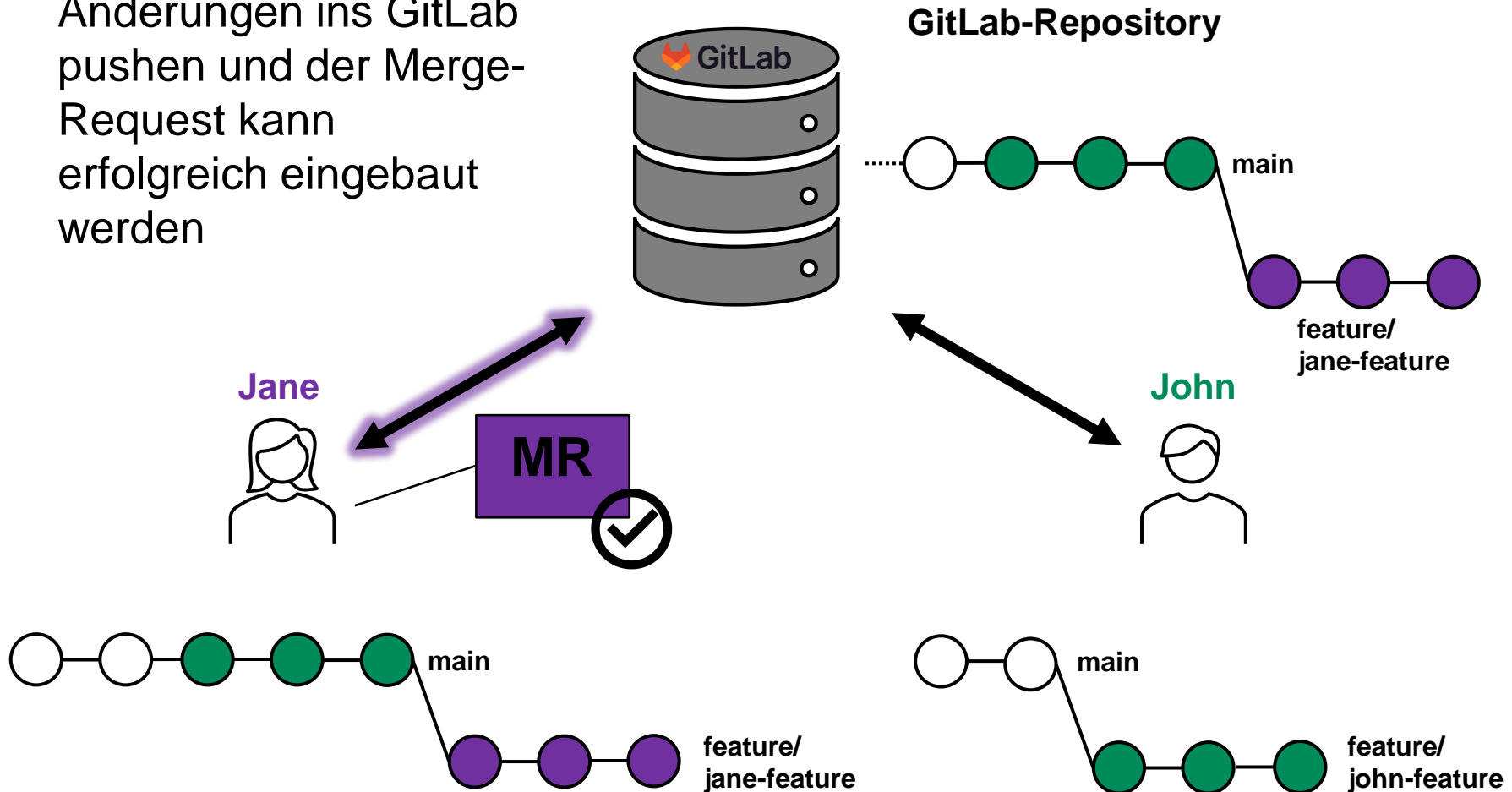
Merge



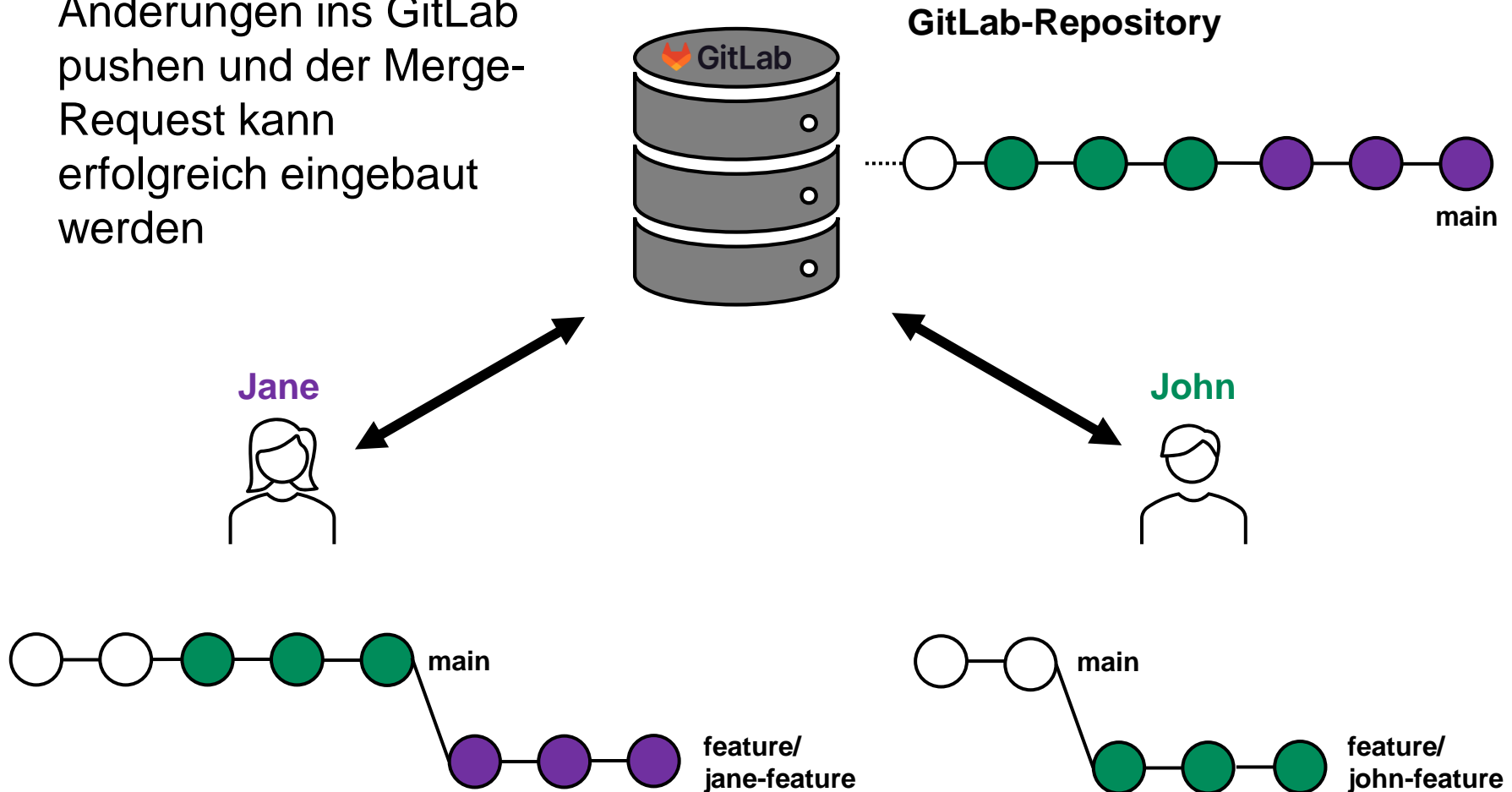
Rebase



- Danach kann Jane ihre Änderungen ins GitLab pushen und der Merge-Request kann erfolgreich eingebaut werden



- Danach kann Jane ihre Änderungen ins GitLab pushen und der Merge-Request kann erfolgreich eingebaut werden



Gitflow-Workflow

Use Cases und Fazit

- Workflow stammt von 2010 und ist teilweise überholt
- Vorteile
 - Klare, strukturierte Aufteilung
 - Parallele Entwicklung vereinfacht
 - Robustere Releases durch dedizierte Branches
 - Bessere Versionshistorie
- Nachteile
 - Langlebige Branches erhöhen Konfliktpotenzial beim Mergen
 - Erschwert Continuous Deployment
 - Continuous Integration möglich, aber weniger effektiv
 - Zusätzlicher Overhead durch Branches
- Geeignet für
 - Größere Teams oder komplexe Projekte
 - Projekte ohne hochfrequente Releases
 - Entwicklung mehrerer zeitgleich betriebener

- Vincent Driessen selbst hat im Jahre 2020 an seinen [Blogpost](#) eine Notiz zur Rekapitulation angefügt

"This model was conceived in 2010, now more than 10 years ago, and not very long after Git itself came into being. In those 10 years, git-flow (the branching model laid out in this article) has become hugely popular in many a software team to the point where people have started treating it like a standard of sorts — but unfortunately also as a dogma or panacea.

During those 10 years, Git itself has taken the world by a storm, and the most popular type of software that is being developed with Git is shifting more towards web apps — at least in my filter bubble. Web apps are typically continuously delivered, not rolled back, and you don't have to support multiple versions of the software running in the wild.

This is not the class of software that I had in mind when I wrote the blog post 10 years ago. If your team is doing continuous delivery of software, I would suggest to adopt a much simpler workflow (like GitHub flow) instead of trying to shoehorn git-flow into your team.

If, however, you are building software that is explicitly versioned, or if you need to support multiple versions of your software in the wild, then git-flow may still be as good of a fit to your team as it has been to people in the last 10 years. In that case, please read on.

To conclude, always remember that panaceas don't exist. Consider your own context. Don't be hating. Decide for yourself."

– Vincent Driessen, <https://nvie.com/posts/a-successful-git-branching-model/>