Module 2 : Commandes Git Essentielles

Cours Git et GitHub pour Ingénieurs en Électronique

Durée : 60 minutes

Slide 1 : Module 2 - Commandes Essentielles 🚀

Objectifs du module

À la fin de ce module, vous saurez :

- ✅ Créer et initialiser un dépôt Git
- ✅ Utiliser les commandes de base (add, commit, status, log)
- ✅ Créer et gérer des branches
- ✅ Fusionner des branches
- ✅ Résoudre des conflits simples

Format : Théorie + Pratique intensive

Slide 2 : Créer un Nouveau Dépôt 📦

Initialisation d'un projet

Méthode 1 : Nouveau projet

```
# Créer un dossier pour le projet
mkdir mon-projet-arduino
cd mon-projet-arduino

# Initialiser Git
git init

# Résultat
# Initialized empty Git repository in /path/to/mon-projet-arduino/.git/
```

Méthode 2 : Projet existant

```
# Aller dans le dossier du projet
cd mon-projet-existant

# Initialiser Git
git init
```

Que se passe-t-il ?

- Création du dossier `.git/` (caché)
- Ce dossier contient toute la base de données Git
- Le projet est maintenant un dépôt Git

Slide 3 : Structure du Dépôt

Contenu du dossier .git/

```
.git/
├── HEAD              # Pointeur vers la branche actuelle
├── config            # Configuration du dépôt
├── description       # Description du projet
├── hooks/            # Scripts automatiques
├── objects/          # Base de données des objets
├── refs/             # Références (branches, tags)
└── index             # Zone de staging
```

⚠️ **Important :** Ne jamais modifier manuellement le contenu de `.git/`

Slide 4 : Vérifier l'État du Dépôt 🔍

git status - Votre meilleur ami

```
git status
```

Informations fournies :

- Branche actuelle
- Fichiers modifiés
- Fichiers en staging
- Fichiers non suivis

Exemple de sortie :

```
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  modified:   src/main.cpp

Untracked files:
  src/sensor.cpp

no changes added to commit
```

💡 **Conseil :** Utilisez `git status` fréquemment !

Slide 5 : Ajouter des Fichiers 📝

git add - Préparer les modifications

Ajouter un fichier spécifique :

```
git add main.cpp
```

Ajouter plusieurs fichiers :

```
git add main.cpp sensor.cpp config.h
```

Ajouter tous les fichiers modifiés :

```
git add .
# ou
git add -A
```

Ajouter par extension :

```
git add *.cpp
git add src/*.h
```

Ajouter de manière interactive :

```
git add -p  # Permet de choisir les modifications à ajouter
```

Slide 6 : Valider les Modifications 💾

git commit - Enregistrer dans l'historique

Commit avec message en ligne :

```
git commit -m "Ajout du support du capteur DHT22"
```

Commit avec éditeur (message détaillé) :

```
git commit
# Ouvre l'éditeur configuré
```

Commit rapide (add + commit) :

```
git commit -am "Correction du bug de lecture I2C"
# Fonctionne uniquement pour les fichiers déjà suivis
```

Modifier le dernier commit :

```
git commit --amend
# Permet de modifier le message ou ajouter des fichiers oubliés
```

Slide 7 : Messages de Commit 📋

Bonnes pratiques

Format recommandé :

```
Type: Résumé court (50 caractères max)

Description détaillée si nécessaire (72 caractères par ligne)
- Point 1
- Point 2

Références: #123
```

Types courants :

- `feat:` Nouvelle fonctionnalité
- `fix:` Correction de bug
- `docs:` Documentation
- `refactor:` Refactorisation
- `test:` Ajout de tests
- `chore:` Tâches de maintenance

Exemples :

```
git commit -m "feat: Ajout du support WiFi pour ESP32"
git commit -m "fix: Correction de la lecture du capteur de température"
git commit -m "docs: Mise à jour du README avec instructions de build"
```

Slide 8 : Visualiser l'Historique 📜

git log - Explorer les commits

Log basique :

```
git log
```

Log compact (une ligne par commit) :

```
git log --oneline
```

Log avec graphe :

```
git log --oneline --graph --all
```

Log avec statistiques :

```
git log --stat
```

Log d'un fichier spécifique :

```
git log main.cpp
```

Log avec recherche :

```
git log --grep="capteur"
git log --author="Jean"
```

Slide 9 : Exemple de Log 📊

Sortie typique

```
$ git log --oneline --graph --all

* a3f5b2c (HEAD -> main) feat: Ajout support DHT22
* 7d8e9f1 fix: Correction lecture I2C
* 2c4d6e8 docs: Mise à jour README
* 9a1b3c5 feat: Implémentation communication série
* 5e7f9a2 Initial commit
```

Informations visibles :

- Hash du commit (identifiant unique)
- Branche actuelle (HEAD)
- Message de commit
- Graphe des branches

Slide 10 : Exercice Pratique 2 🏋️

Créer votre premier dépôt

Objectif : Créer un projet Arduino simple et faire vos premiers commits

```
# 1. Créer le projet
mkdir projet-led-blink
cd projet-led-blink
git init

# 2. Créer un fichier
echo "# Projet LED Blink" > README.md

# 3. Vérifier le statut
git status

# 4. Ajouter le fichier
git add README.md

# 5. Faire le premier commit
git commit -m "Initial commit: Ajout du README"

# 6. Vérifier l'historique
git log
```

Temps alloué : 10 minutes

Slide 11 : Les Branches - Concept 🌳

Pourquoi utiliser des branches ?

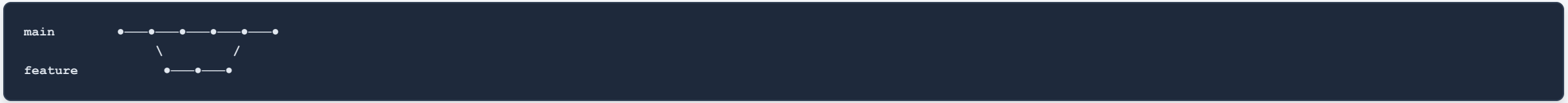
Définition : Une branche est une ligne de développement indépendante

Cas d'usage :

- Développer une nouvelle fonctionnalité
- Corriger un bug sans affecter le code stable
- Expérimenter sans risque
- Travailler en parallèle sur plusieurs tâches

Avantages :

- ✅ Isolation du code
- ✅ Travail en parallèle
- ✅ Facilite la collaboration
- ✅ Retour arrière facile



Slide 12 : Créer une Branche 🌿

git branch et git checkout

Lister les branches :

```
git branch
# * main  (l'astérisque indique la branche actuelle)
```

Créer une nouvelle branche :

```
git branch feature-wifi
```

Changer de branche :

```
git checkout feature-wifi
# Switched to branch 'feature-wifi'
```

Créer et changer de branche (raccourci) :

```
git checkout -b feature-wifi
# Équivalent à :
# git branch feature-wifi
# git checkout feature-wifi
```

Nouvelle syntaxe (Git 2.23+) :

```
git switch feature-wifi          # Changer de branche
git switch -c feature-wifi       # Créer et changer
```

Slide 13 : Travailler avec les Branches

Workflow typique

```
# 1. Créer une branche pour une nouvelle fonctionnalité
git checkout -b feature-capteur-temperature

# 2. Faire des modifications
vim src/temperature.cpp
git add src/temperature.cpp
git commit -m "feat: Ajout du capteur de température"

# 3. Faire d'autres commits si nécessaire
vim src/temperature.cpp
git commit -am "fix: Correction de la calibration"

# 4. Retourner sur main
git checkout main

# 5. Fusionner la branche
git merge feature-capteur-temperature

# 6. Supprimer la branche (optionnel)
git branch -d feature-capteur-temperature
```

Slide 14 : Visualiser les Branches 👁️

Voir l'état des branches

Lister toutes les branches :

```
git branch -a
```

Voir les branches avec leur dernier commit :

```
git branch -v
```

Voir les branches fusionnées :

```
git branch --merged
```

Voir les branches non fusionnées :

```
git branch --no-merged
```

Graphe visuel :

```
git log --oneline --graph --all --decorate
```

Slide 15 : Fusionner des Branches 🔗

git merge - Intégrer les modifications

Types de fusion :

1. Fast-forward (avance rapide)

2. Three-way merge (fusion à trois points)

Commande :

```
git checkout main
git merge feature-capteur
```

Slide 16 : Conflits de Fusion ⚠️

Quand Git ne peut pas fusionner automatiquement

Situation de conflit :

- Deux branches modifient la même ligne
- Git ne sait pas quelle version garder

Exemple de conflit :

```
<<<<<<< HEAD
int temperature = readSensor();
=======
float temperature = getSensorValue();
>>>>>>> feature-capteur
```

Marqueurs de conflit :

- `<<<<<<< HEAD` : Version de la branche actuelle
- `=======` : Séparateur
- `>>>>>>> feature-capteur` : Version de la branche à fusionner

Slide 17 : Résoudre les Conflits 🔧

Processus de résolution

Étapes :

1. **Identifier les fichiers en conflit**

```
git status
# Both modified: src/main.cpp
```

2. **Ouvrir le fichier et choisir la version**

```
// Supprimer les marqueurs et garder le bon code
float temperature = getSensorValue();
```

3. **Marquer comme résolu**

```
git add src/main.cpp
```

4. **Finaliser la fusion**

```
git commit -m "Merge feature-capteur: Résolution des conflits"
```

💡 **Conseil :** Utilisez un outil de merge visuel (VS Code, Meld, KDiff3)

Slide 18 : Annuler une Fusion 

Si la fusion ne se passe pas bien

Avant le commit final :

```
git merge --abort
# Annule la fusion et revient à l'état précédent
```

Après le commit :

```
# Revenir au commit précédent
git reset --hard HEAD~1

# Ou créer un commit qui annule la fusion
git revert -m 1 HEAD
```

⚠️ **Attention :** `git reset --hard` supprime les modifications non commitées

Slide 19 : Supprimer une Branche 🗑️

Nettoyage des branches

Supprimer une branche fusionnée :

```
git branch -d feature-capteur
# Deleted branch feature-capteur
```

Forcer la suppression (branche non fusionnée) :

```
git branch -D feature-experimental
# Deleted branch feature-experimental (was a3f5b2c)
```

Supprimer une branche distante :

```
git push origin --delete feature-capteur
```

💡 **Bonne pratique :** Supprimer les branches après fusion pour garder un dépôt propre

Slide 20 : Exercice Pratique 3 🏋️

Travailler avec les branches

Objectif : Créer des branches et les fusionner

```
# 1. Créer une branche pour une nouvelle fonctionnalité
git checkout -b feature-led-rgb

# 2. Créer un fichier
echo "// Code pour LED RGB" > led_rgb.cpp
git add led_rgb.cpp
git commit -m "feat: Ajout support LED RGB"

# 3. Retourner sur main
git checkout main

# 4. Créer une autre branche
git checkout -b feature-buzzer
echo "// Code pour buzzer" > buzzer.cpp
git add buzzer.cpp
git commit -m "feat: Ajout support buzzer"

# 5. Fusionner les branches
git checkout main
git merge feature-led-rgb
git merge feature-buzzer

# 6. Voir l'historique
git log --oneline --graph --all
```

Temps alloué : 15 minutes

Slide 21 : Commandes de Comparaison 🔍

git diff - Voir les différences

Différences non stagées :

```
git diff
```

Différences stagées :

```
git diff --staged
# ou
git diff --cached
```

Différences entre branches :

```
git diff main feature-capteur
```

Différences d'un fichier spécifique :

```
git diff main.cpp
```

Statistiques des différences :

```
git diff --stat
```

Slide 22 : Ignorer des Fichiers 🚫

Le fichier .gitignore

Créer un .gitignore :

```
# Fichiers compilés
*.o
*.hex
*.bin
*.elf

# Dossiers de build
build/
.pio/

# Fichiers IDE
.vscode/
.idea/

# Fichiers système
.DS_Store
Thumbs.db

# Fichiers temporaires
*.tmp
*.bak
*~

# Secrets
.env
secrets.h
```

Appliquer le .gitignore :

```
git add .gitignore
git commit -m "chore: Ajout du .gitignore"
```

Slide 23 : Templates .gitignore 📋

Pour projets électroniques

Arduino/PlatformIO :

```
# PlatformIO
.pio/
.pioenvs/
.piolibdeps/

# Arduino
*.hex
*.eep
*.elf
*.map

# Build
build/
*.o
*.a
```

KiCad :

```
# KiCad
*.bak
*.kicad_pcb-bak
*-save.kicad_pcb
fp-info-cache
*.net

# Gerber
gerber/
```

Ressource : https://github.com/github/gitignore

Slide 24 : Annuler des Modifications 🔄

Différentes façons de revenir en arrière

Annuler les modifications non stagées :

```
git checkout -- main.cpp
# ou (Git 2.23+)
git restore main.cpp
```

Retirer un fichier du staging :

```
git reset HEAD main.cpp
# ou (Git 2.23+)
git restore --staged main.cpp
```

Annuler le dernier commit (garder les modifications) :

```
git reset --soft HEAD~1
```

Annuler le dernier commit (supprimer les modifications) :

```
git reset --hard HEAD~1
```

⚠️ **Attention :** `--hard` supprime définitivement les modifications

Slide 25 : Voir un Commit Spécifique 👀

git show - Détails d'un commit

Voir le dernier commit :

```
git show
```

Voir un commit spécifique :

```
git show a3f5b2c
```

Voir un fichier à un commit donné :

```
git show a3f5b2c:src/main.cpp
```

Voir les modifications d'un commit :

```
git show --stat a3f5b2c
```

Slide 26 : Commandes Utiles 🛠️

Autres commandes importantes

Renommer un fichier :

```
git mv ancien_nom.cpp nouveau_nom.cpp
```

Supprimer un fichier :

```
git rm fichier.cpp
```

Voir qui a modifié chaque ligne :

```
git blame main.cpp
```

Rechercher dans l'historique :

```
git log -S "fonction_recherchee"
```

Nettoyer les fichiers non suivis :

```
git clean -n  # Voir ce qui serait supprimé
git clean -f  # Supprimer réellement
```

Slide 27 : Alias Git ⚡

Raccourcis personnalisés

Créer des alias :

```
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.lg "log --oneline --graph --all"
```

Utilisation :

```
git st      # au lieu de git status
git co main # au lieu de git checkout main
git lg      # log formaté
```

Alias complexes :

```
git config --global alias.last "log -1 HEAD"
git config --global alias.unstage "reset HEAD --"
```

Slide 28 : Exercice Pratique 4 🏋️

Projet complet

Objectif : Créer un projet Arduino avec branches et .gitignore

```
# 1. Créer le projet
mkdir projet-station-meteo
cd projet-station-meteo
git init

# 2. Créer le .gitignore
cat > .gitignore << EOF
*.hex
*.elf
build/
.pio/
EOF

# 3. Créer le fichier principal
cat > main.ino << EOF
void setup() {
  Serial.begin(9600);
}

void loop() {
  // TODO
}
EOF

# 4. Premier commit
git add .
git commit -m "Initial commit: Structure du projet"

# 5. Créer une branche pour le capteur
git checkout -b feature-dht22
# Ajouter du code...
git commit -am "feat: Ajout capteur DHT22"

# 6. Fusionner
git checkout main
git merge feature-dht22
```

Slide 29 : Bonnes Pratiques 📌

Recommandations

Commits :

- ✅ Faire des commits atomiques (une fonctionnalité = un commit)
- ✅ Écrire des messages descriptifs
- ✅ Commiter régulièrement
- ❌ Ne pas commiter de code non fonctionnel sur main

Branches :

- ✅ Utiliser des noms descriptifs (`feature/wifi`, `bugfix/i2c`)
- ✅ Créer une branche par fonctionnalité
- ✅ Fusionner régulièrement
- ❌ Ne pas garder des branches trop longtemps

Général :

- ✅ Utiliser .gitignore dès le début
- ✅ Vérifier avec `git status` avant de commiter
- ✅ Tester avant de fusionner
- ❌ Ne jamais commiter de secrets (mots de passe, clés API)

Slide 30 : Récapitulatif Module 2 📋

Ce que nous avons appris

✅ Commandes de base

- `git init`, `git add`, `git commit`
- `git status`, `git log`, `git diff`

✅ Gestion des branches

- Créer, changer, fusionner des branches
- Résoudre des conflits
- Supprimer des branches

✅ Outils pratiques

- .gitignore
- Alias
- Annulation de modifications

✅ Bonnes pratiques

- Messages de commit
- Organisation des branches
- Workflow efficace

Slide 31 : Questions ? 🙋

Discussion

Points à clarifier ?

- Commandes pas claires ?
- Problèmes rencontrés dans les exercices ?
- Cas d'usage spécifiques ?

Prochaine étape :

Module 3 - Collaboration avec GitHub

Slide 32 : Pause ☕

Pause de 10 minutes

Avant de continuer :

- Assurez-vous d'avoir un dépôt Git fonctionnel
- Testez les commandes apprises
- Préparez vos questions

Rendez-vous dans 10 minutes pour le Module 3 !

Notes pour le formateur 👨‍🏫

Timing suggéré

- Slides 1-10 : Commandes de base (20 min)
- Slides 11-20 : Branches et fusion (25 min)
- Slides 21-28 : Commandes avancées (15 min)

Points d'attention

- Vérifier que tous suivent les exercices
- Montrer les commandes en live
- Encourager la pratique
- Aider à résoudre les conflits

Exercices supplémentaires

- Créer un conflit volontaire pour pratiquer la résolution
- Utiliser git log avec différentes options
- Expérimenter avec les branches

Ressources

- Cheat sheet Git à distribuer
- Exemples de projets Arduino