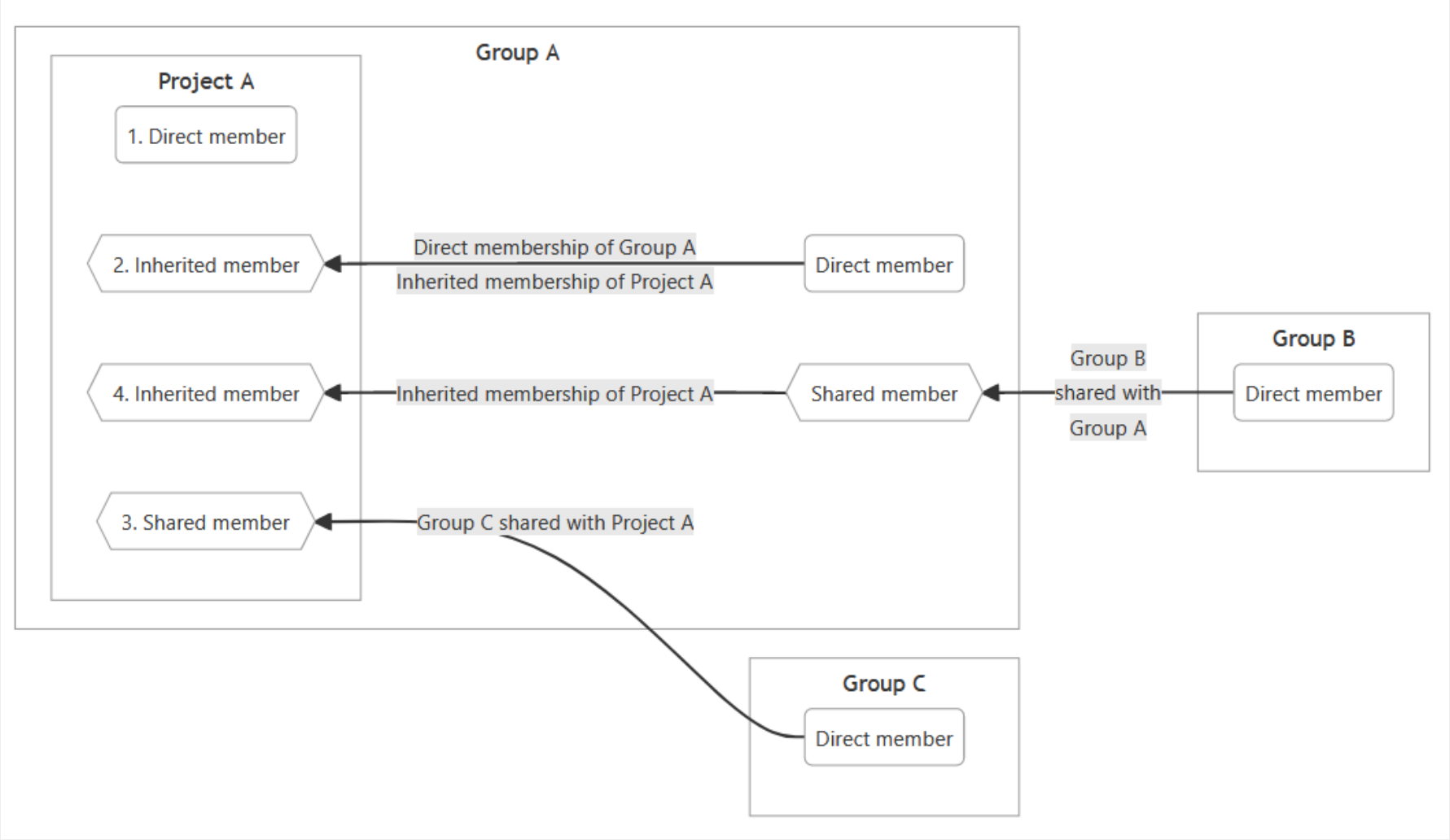
GITLAB 14

ORGANISATION

- Concepts organisationnels dans gitlab
  - **Utilisateurs** : peuvent appartenir à un projet ou un groupe
  - **Projets** : gèrent un dépôt git : peuvent appartenir à un groupe
  - **Groupes** : groupes de projets. Certaines fonctionnalités des projets sont généralisées ici

- Espaces de noms
  - Les utilisateurs et les groupes définissent des namespaces pour dédoublonner les noms de projets
    - un projet dans le namespace utilisateur n'est connu que de cet utilisateur : **projet personnel**
    - deux groupes peuvent définir un projet de même nom : **backend/monitoring et frontend/monitoring**

ORGANISATION



ORGANISATION

- Droits d’accès
  - **Owner** : créateur d’un projet / groupe : tous les droits
  - **Maintainer** : Administrateur : tout sauf suppression, accès aux « **Settings** » projet / groupe
  - **Developper** : lecture / écriture sur le **code, merge requests, issues, CI / CD**
  - Reporter : droits de **commenter** : issues, reviews, …, pas d’accès en écriture au code
  - Guest : droit de **lecture**

voir ici

ACCES

- Login / mot de passe root
  - Url : https://gitlab.myusine.fr
  - Login : root / Mdp : R00tt00R

ACCES

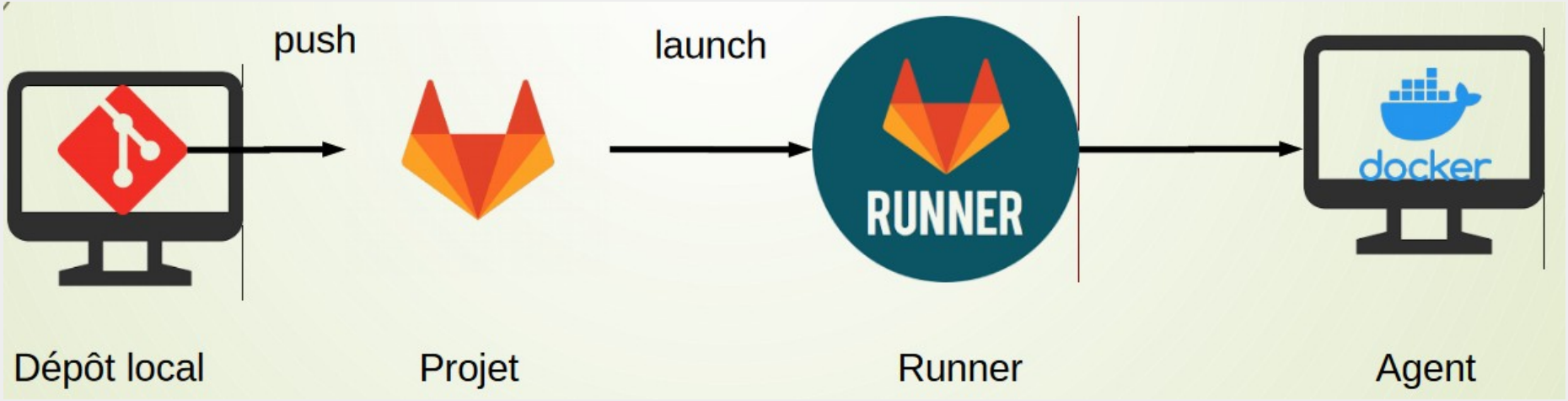
- Jeton d'accès utilisateur : « Personal Access Token »
  - Espace utilisateur → Preferences → Acess Token
  - Donne accès à l'api Gitlab en Lecture / Ecriture

```
# création automatisée sur le serveur en ruby
user = User.find_by_username('automation-bot')
token = user.personal_access_tokens.create(scopes: [:api], name: 'Automation token')
token.set_token('name-your-own-token')
token.save!

# in-line command
sudo gitlab-rails runner "token = User.find_by_username('automation-bot')...; token.save!"

# ex. de client python
# pip install --upgrade python-gitlab
import gitlab
gl = Gitlab(url="https://<host>", token="name-your-own-token")
...
```

GITLAB RUNNER

Présentation

- gitlab-runner est un composant indépendant de Gitlab chargé de **l'intégration continue**
- Il installe des processus appelés **runners** contrôlant l'exécution de scripts ou **jobs** utilisateurs
- Ces jobs sont décrits dans un fichier « **.gitlab-ci.yml** » au format **YAML**, ajouté au projet gitlab
- Chaque job est exécuté, via un runner, dans un **agent** pouvant être :
  - une machine physisque
  - une **VM** (Virtualbox, VMWare, Hyper-V ...)
  - un **conteneur** (Docker, LXC, Kubernetes, ...)



GITLAB RUNNER

- Configuration
  - Un runner peut être associé à un **projet** précis ou à un **groupe** précis d'une instance gitlab
  - Un runner **partagé** peut accepter des scripts de jobs provenant de **plusieurs instances** de gitlab
  - Voir la section « Runners » de l'interface « Settings / CICD » d'un projet ou d'un groupe gitlab
  - La configuration des runners lancés par gitlab-runner se trouve dans **/etc/gitlab-runner/config.toml**

```
# nb de jobs exécutables en parallèle
concurrent = 4
…
[runners.docker]
…
# dns interne au réseau docker pour un gitlab en local
extra_hosts = ["gitlab.myusine.fr:172.17.0.1"]
```

- YAML : introduction
  - format de représentation de données par **sérialisation**, conçu pour être aisément modifiable et lisible
  - Dérivé de la représentation d'un objet **JSON déplié**

```
{                                                   ---
   "key" : "value",                                 key: value
   "other key" : "other value",          =>         other key: other value
   "num" : "3.14"                                   num: 3.14
   "bool" : "false",                                bool: false
   "none" : "null"                                  none: null
}                                                   "with quotes": "possible"
```

- YAML : imbrications
  - Par rapport à JSON :

    - les objets { ... } sont remplacés par une **indentation** de 2 ou 4 caractères

    - les listes [...] sont remplacés par une indentation de 2 ou 4 caractères préfixée par « **- [espace]** »

```
{                                       ---
   "object" : {                         object:
      "key" : "value",          =>          key: value
      "items" : [                          items:
         "item1",                              - item1
         { "k1": "v1", "k2": ... },            - k1: v1
         …                                       k2: v2
         "item3"                               - item3
      ]
}
```

CICD

- .gitlab-ci.yml : écrire un job
  - Un job est un objet yaml :

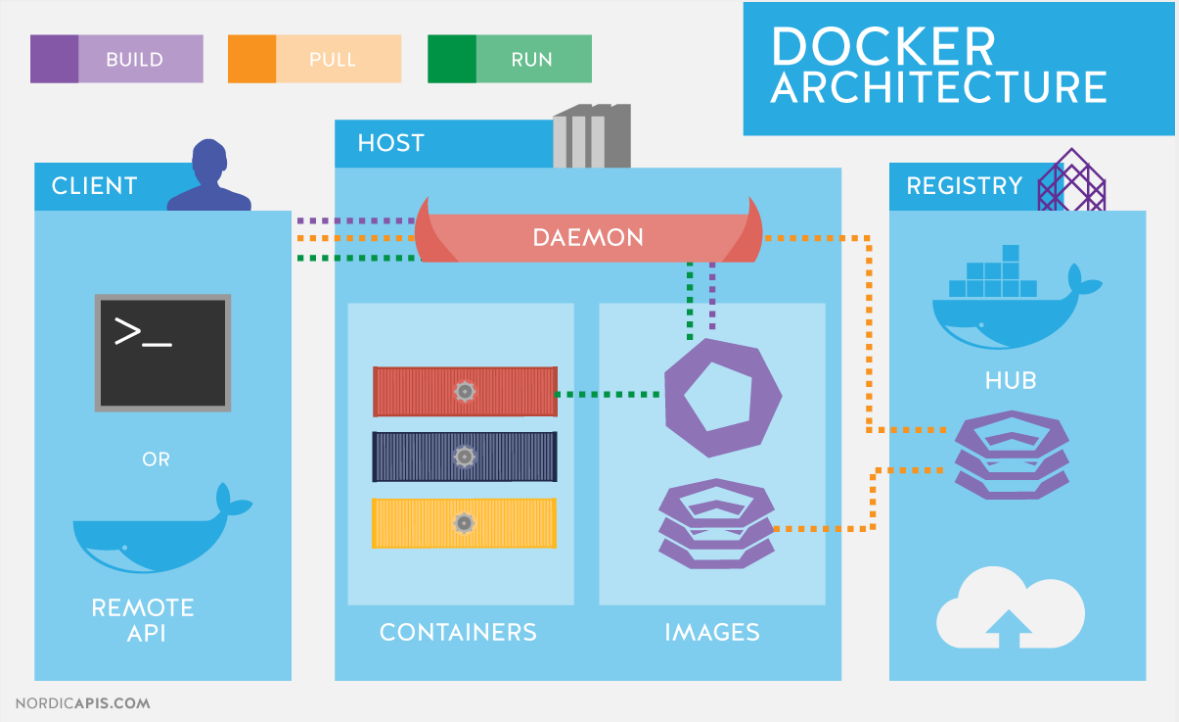
    - ont le nom est **arbitraire**
    - qui contient une clé **script** décrivant une liste de lignes de commandes
    - on ajoute des **tags** correspondants aux runners susceptibles d'exécuter le job

```
unit tests:
    script:
        - echo "launch tests here !"
        - ...
    tags:
        - myusine
        - ...
```

  - Chaque fois que le fichier est **poussé sur gitlab**, on peut observer l'exécution du job dans la section **CICD → pipelines** du projet

CICD

- .gitlab-ci.yml : choisir une image (agent docker)
  - À l'instar d'une image disque, état d'un système de fichier (snapshot) avec librairies et applications déjà installées
  - Vouée à la création de conteneur : environnement isolé du reste de la machine



- **hub.docker.com**

ou registre privé gitlab

- **packages & registries → container**

(plus rapide & images custom)

CICD

- .gitlab-ci.yml : registre d'images privé
  - Ajout d'une image au registre accessible à l'adresse gitlab.myusine.fr:5050
    - Téléchargement plus rapide
    - Ajout d'image customisée

```
# télécharger depuis hub.docker.com
docker pull <image>:<tag>

# copier l'image en préfixant un nouveau nom avec l'URL du projet
docker tag <image>:<tag> gitlab.myusine.fr:5050/myusine/dev/<image>:<tag>

# s'authentifier sur le registre avec les identifiants
docker login gitlab.myusine.fr:5050 -u root -p R00tt00R

# pousser l'image sur le registre
docker push gitlab.myusine.fr:5050/myusine/dev/<image>:<tag>
```

- .gitlab-ci.yml : image – mise en œuvre
  - On utilise la clé **image** :

    - nomenclature : **nom_image:nom_tag** (nom : technologie, tag : version ou environnement)

    - l'image peut être choisie pour tous les jobs (le pipeline) ou pour un job en particulier (subsidiarité)

```
#image pour le pipeline
image: ubuntu:focal

unit tests:
    # image pour le job issue du registre privé gitlab
    image: gitlab.myusine.fr:5050/myusine/python:rc-slim-buster
    script:
        - echo "launch tests here !"
```

- .gitlab-ci.yml : référence et variables prédéfinies
  - « vocable des clés utiles dans gitlab CI : https://docs.gitlab.com/ee/ci/yaml/
  - variables d'environnement prédéfinies : https://docs.gitlab.com/ee/ci/variables/predefined_variables.html

    - pour accéder à la valeur de ces variables, on préfixe toujours par « **$** »

```
unit tests:
    image: $CI_REGISTRY_IMAGE/python:rc-slim-buster
    script:
        - echo "launch tests here !"
```

CICD

- .gitlab-ci.yml : séquencement des jobs
  - **Par défaut**, les jobs sont exécutés **en parallèle**, en fonction
    - du nombre de cpus de la machine qui exécute les jobs
    - du paramètre concurrent dans /etc/gitlab-runner/config.toml
  - On fixe un ordre d'exécution du pipelines par étapes ou « **stages** » grâce à une clé de même nom
    - un job est rattaché à un stage par la clé **stage**
    - deux jobs dans un même stage sont exécutés en parallèle

```
stages:
    - testing
    - building

unit tests:
    stage: testing
    image: $CI_REGISTRY_IMAGE/python:rc-slim-buster
    script:
        - echo "launch tests here !"
```

CICD

- .gitlab-ci.yml : conditionnalité des jobs
  - La clé **rules** permet d'écrire des conditions complexes d'exécution au moyens de clauses
    - **if** : déclenche si l'équation logique en paramètre est vraie. On peut renseigner plusieurs clauses if
    - **changes** : déclenche si au moins un des chemins en paramètre a subi une modification
    - **exists** : déclenche si au moins un des chemins en paramètre existe
    - **when** : conditions spécifiques de déclenchement (**on_success, on_failure, always, never, manual**)

```
# déclenchement du pipeline
workflow:
    rules:
        - if: $CI_PIPELINE_SOURCE == "push"
        - if: $CI_PIPELINE_SOURCE == "api"

unit tests:
    rules:
        - if: $CI_COMMIT_BRANCH =~ /feature[0-9]+/
          changes:
              - README.md
```

un job s'exécute si ou moins une règle est vraie (OU)

une règle est vraie si toutes ses clauses sont vraies (ET)

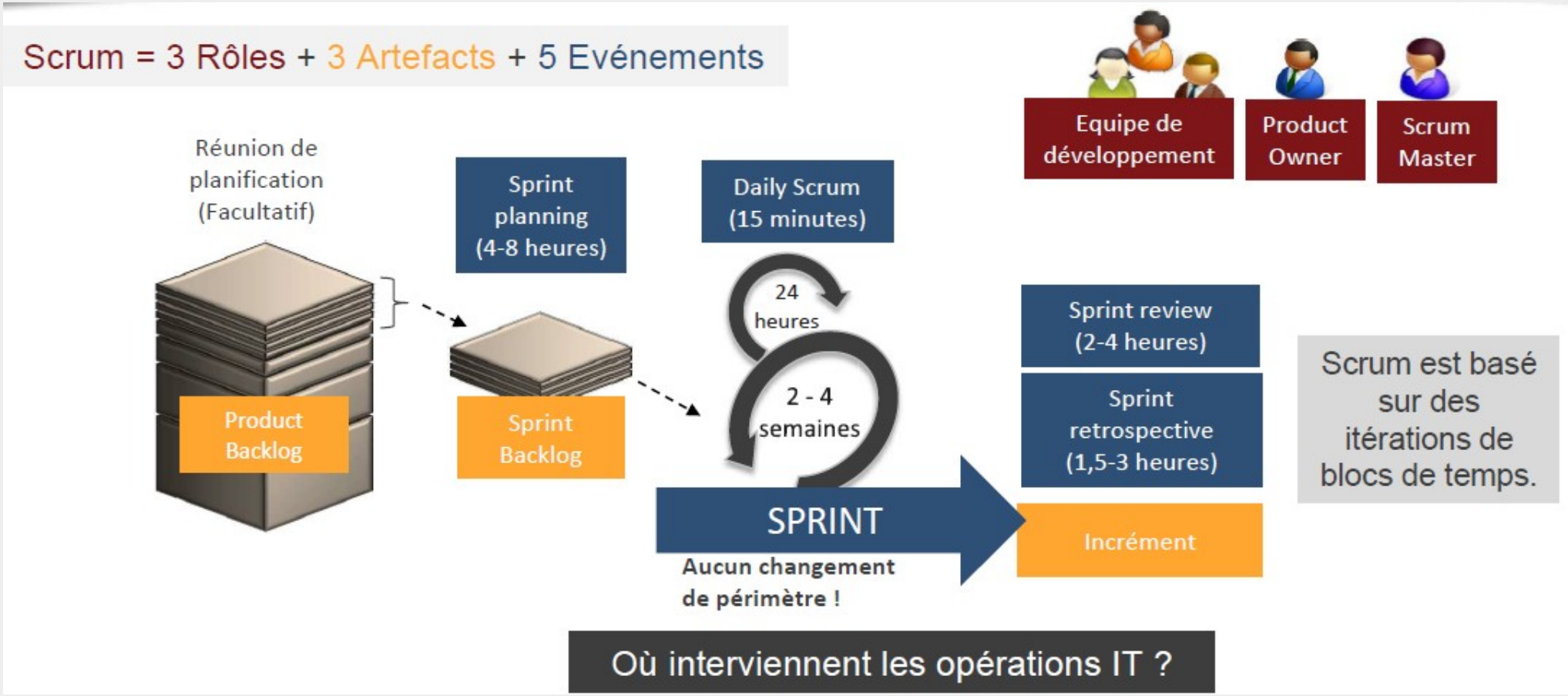
- .gitlab-ci.yml : prélever des artefacts
  - Avec la clé « **artifacts** », toute ressource crée à l'exécution d'un job peut être téléversée
    - dans Gitlab Runner pour être **réutilisable** dans les jobs suivants,
    - dans Gitlab pour y être **téléchargée** depuis l'interface
  - La clé « **paths** » renseigne les chemins de prélèvement des artefacts
  - La clé « **expire_in** » indique le temps de disponibilité d'un artefact avant sa suppression

```
unit tests:
    script:
        # production d'un artefact
        - echo "content" > product.txt
    artifacts:
        expire_in: 1 hour
        paths:
            - product.txt
```

- .gitlab-ci.yml : créer / utiliser le cache
  - Avec la clé « **cache** », toute ressource crée à l'exécution d'un job peut être mise en cache
    - le cache est utilisé typiquement pour éviter de retélécharger des dépendances du projet pour chaque job
  - « **cache:key** » permet de tagger les objets mis en cache pour réutilisation
  - « **cache:paths** » permet de renseigner les chemins à mettre en cache
  - « **cache:untracked** » permet d'ajouter au cache des fichiers non suivis dans le dépôt gitlab
  - « **cache:policy** » permet d'indiquer si l'on veut installer « pull », uploader « push » le cache ou les 2 « pull-push »

```
unit tests:
    script:
        # production d'un cache
        - echo "content" > cache.txt
    cache:
        key: my cache
        paths:
            - cache.txt
        policy: push
```

SCRUM


Scrum = 3 Rôles + 3 Artefacts + 5 Evénements
Equipe de développement
Product Owner
Scrum Master
Réunion de planification (Facultatif)
Sprint planning (4-8 heures)
Daily Scrum (15 minutes)
24 heures
2 - 4 semaines
Product Backlog
Sprint Backlog
Sprint review (2-4 heures)
Sprint retrospective (1,5-3 heures)
SPRINT
Aucun changement de périmètre !
Incrément
Scrum est basé sur des itérations de blocs de temps.
Où interviennent les opérations IT ?

SCRUM

- Rôles dans Scrum

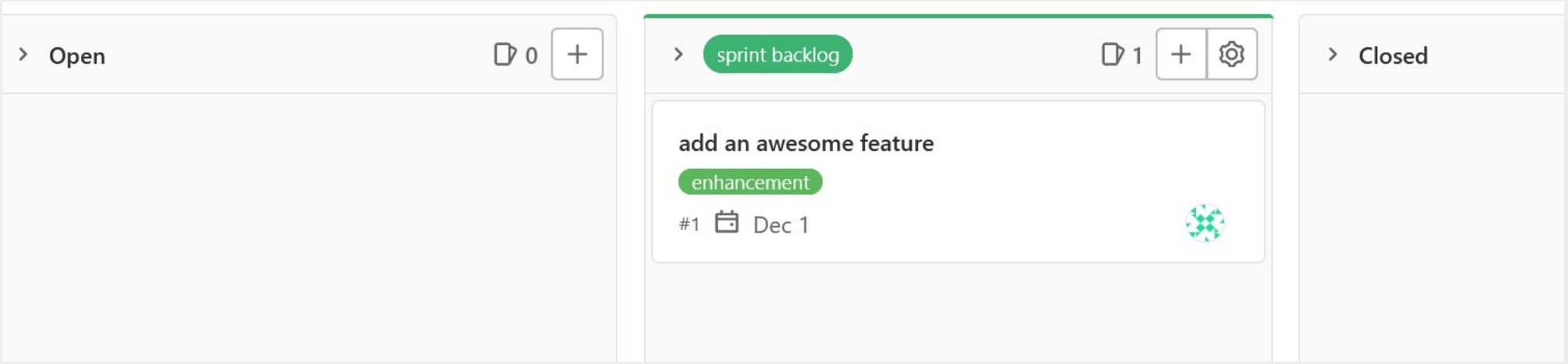
  - Le « Product Owner », qui possède la vision du produit et qui exprime le besoin, doit pouvoir observer régulièrement l'avancée du projet au rythme de la diffusion de livrables ; il est soit client, soit en interface avec le client

  - L' « Agile Process Owner » utilise les principes et pratiques Agile et Scrum pour concevoir, gérer et mesurer les processus individuels

  - Le « Scrum Master » (Dev) et L' « Agile Service Manager » (Ops) sont les garants du cadre méthodologique de l' équipe. Ils doivent veiller à ce que les processus agiles mis en place soient respectés. Ils ne sont pas nécessairement chef de projet

SCRUM / plannification

- Créer des labels pour catégoriser le KANBAN, les issues et les epics
  - Projet / Groupe → Project / Group Information → Labels
  - Créer au moins un label pour la colone **« sprint backlog » Issues → Board**
  - Créer des labels classiques « Bug » « Feature request » « Hot Fix », ...



SCRUM / plannification

- Créer des issues sous forme de « **users stories** »
  - Une « **user story** » est une description de tâche rédigée du **point de vue de l'utilisateur final**
  - Un « **epic** » décrit une grande user story à décomposer en user stories
  - On cherche à respecter les critères **INVEST**

    - **I** pour Indépendante : au moins sur le sprint en cours.
    - **N** pour négociable : le contenu fait l'objet d'une **concertation**. on écrit une user-story en **une seule phrase.**
    - **V** pour valeur : chaque user story doit apporter de la valeur ajoutée aux clients / utilisateurs
    - **E** pour Estimable : dans le temps **« due date »** ou en complexité **« points de sprint »**
    - **S** pour Suffisamment petite : découpage fin pour livraison au sein d'un seul Sprint.
    - **T** pour Testable : On doit pouvoir **déduire un test** de l'énoncé (cf TDD ou BDD).

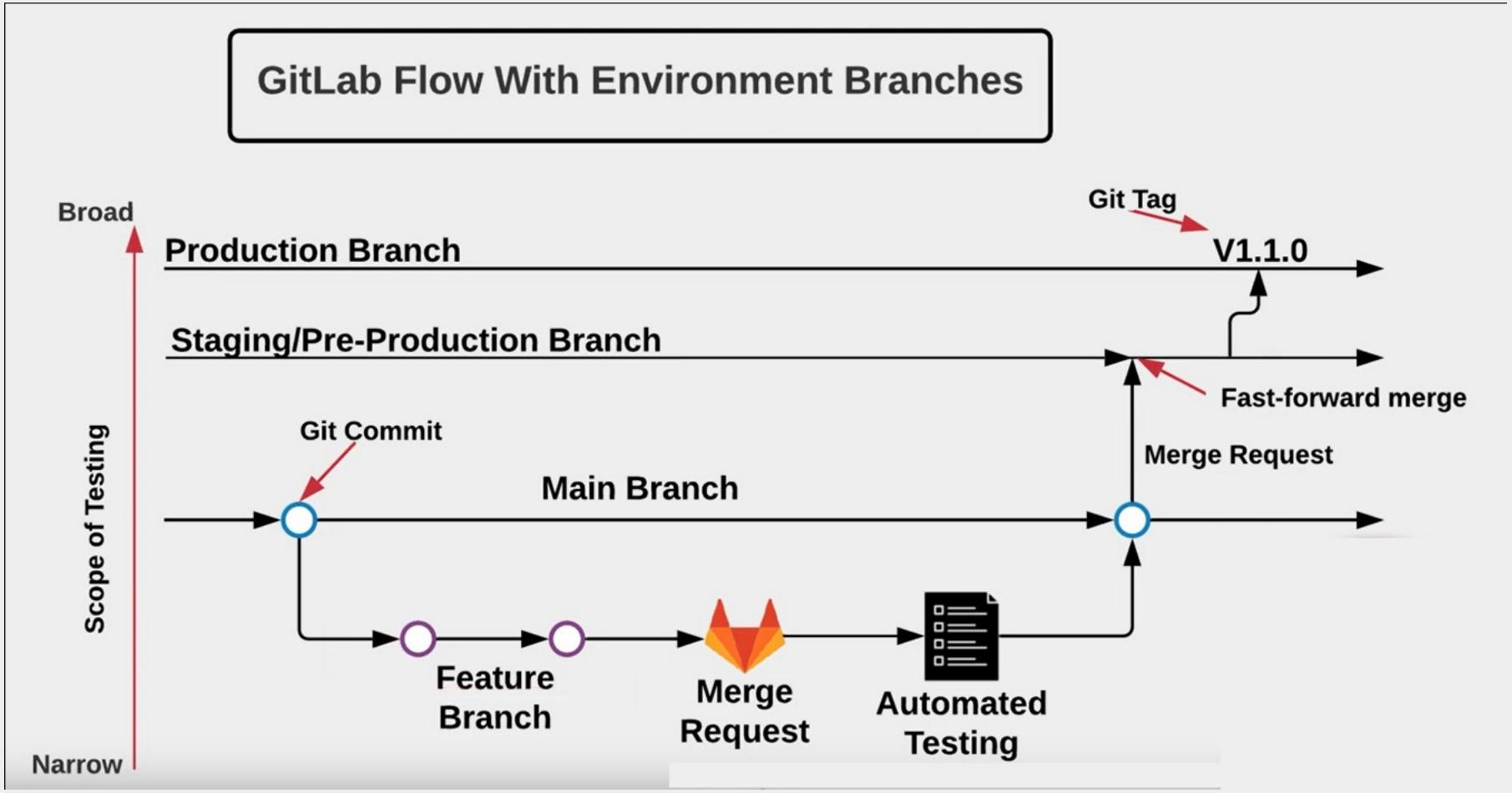
SCRUM / plannification

- Créer des issues sous forme de « **users stories** »
  - Pour cela, créer des templates de description d'issues dans **.gitlab/issue_templates/xxxx.md**
  - **Projet / groupe → Settings → general → default.md (v14.8+)**

```
### Given Context ...
<!-- Why ? Who ? Where ? When ? -->

### Action To Do ...
<!-- How ? -->

### Expected Result
<!-- What ? -->
<!-- Metrics ? -->

<!-- Quick Actions -->
/label ~"enhancement"
```

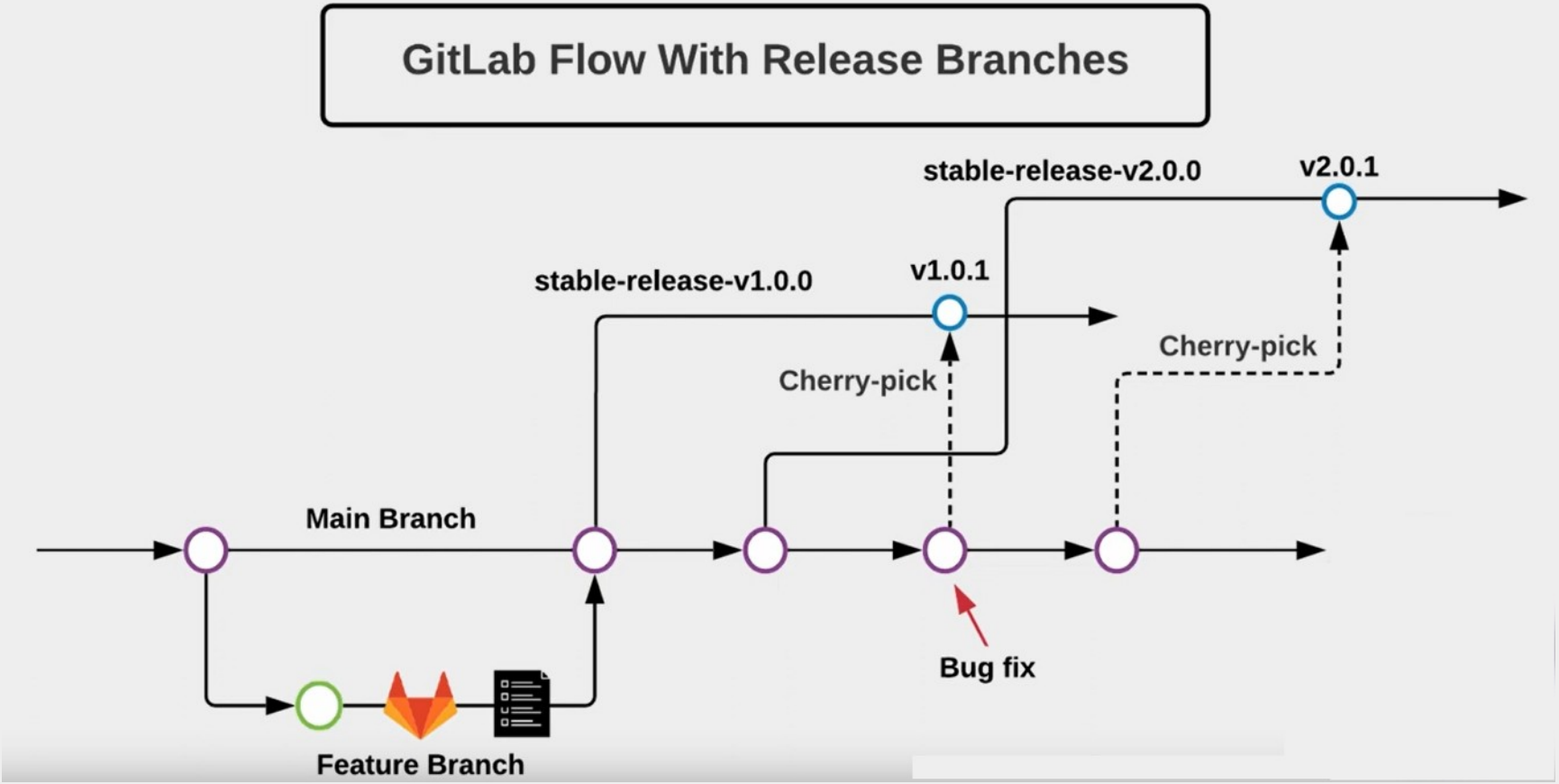
GITLAB FLOW

- Cas pour les Services : branches d'environnement à long terme



GITLAB FLOW

- Cas pour les Produits : branches de releases à long terme



GITLAB FLOW

- Workflow
  - Créer une **merge request** à partir d'une issue
    - crée automatiquement la branche de fonctionnalité
    - préenregistre un message pour fermer l'issue si fusionnée
  - Modifier ses fichiers et pousser la branche de fonctionnalité pour soumettre au **pipeline CI / CD**
  - Renseigner un responsable de revue de code et procéder à la **revue de code**
  - Si la fusion est décidée, **fusionner** la branche de fonctionnalité dans la branche principale
    - possibiité de supprimer la branche distante en cas de succès
    - possibilité de « squasher », i.e condenser les commits de la branche en un seul, + message de synthèse.
  - Une fois fusionnée, récupérer le code sur la branche principale en local
  - … et supprimer la branche de fonctionnalité locale