TÀI LIỆU HƯỚNG DẪN PHÁT TRIỂN DỰ ÁN

Cài đặt Công cụ Phát triển

Docker & Docker Desktop

Tải và cài đặt Docker Desktop:

- Truy cập: https://www.docker.com/products/docker-desktop/
- Tải file installer phù hợp với hệ điều hành
- Chạy file `Docker Desktop Installer.exe`
- Trong quá trình cài đặt, chọn **"Use WSL 2 instead of Hyper-V"** (khuyến nghị)
- Khởi động lại máy tính sau khi cài đặt

Kiểm tra cài đặt:

```
docker --version
docker-compose --version
```

PHPStorm

Cài đặt PHPStorm:

- Truy cập: https://www.jetbrains.com/phpstorm/download/
- Tải phiên bản phù hợp với hệ điều hành
- Chạy file installer và làm theo hướng dẫn
- Kích hoạt license (sử dụng education license)
- Mở dự án VietMarket bằng PHPStorm

Kết nối MySQL trong PHPStorm:

1. Mở **Database tool window** (góc phải trên của PHPStorm)
2. Click nút **New (+)** → **Data Source** → **MySQL**
3. Cấu hình kết nối MySQL Docker:
   - **Host:** localhost
   - **Port:** 3306
   - **Database:** laravel
   - **User:** admin
   - **Password:** secret
   - **Driver:** MySQL 9
4. Test connection và save

_Lưu ý: MySQL phải chạy trong Docker container với port mapping 3306:3306

Cấu trúc dự án

```
project-root/
├── backend/                # Laravel 12 API
├── frontend/               # React + Vite UI
├── database/               # Test data
├── docs/                   # Tài liệu
│   └── mysql/
├── docker-compose.yml      # Orchestration
├── .env.docker            # Environment variables
└── README.md
```

Lệnh Git cơ bản

```
# Kiểm tra trạng thái
git status
git diff

# Staging và commit
git add <file>
git add .
git commit -m "message"
git commit --amend  # Sửa commit cuối

# Đồng bộ repository
git fetch origin     # Lấy thông tin mới từ remote
git pull origin main # Fetch + merge vào branch hiện tại
git push origin feature/branch-name

# Stash - lưu tạm thay đổi
git stash save "work in progress"
git stash list
git stash pop       # Khôi phục stash cuối
git stash apply     # Áp dụng stash nhưng không xóa

# Reset và revert
git reset --soft HEAD~1    # Undo commit, giữ changes
git reset --hard HEAD~1    # Undo commit, xóa changes
git revert <commit-hash>   # Tạo commit mới để undo
```

1.1. Mô hình Git Flow

Nhóm dự án áp dụng mô hình **Gitflow** để quản lý các nhánh Git một cách hiệu quả. Trong mô hình này, có 5 loại nhánh chính với vai trò khác nhau:

- **Nhánh** `main` (nhánh chính): chứa mã nguồn *production-ready* – luôn ở trạng thái sẵn sàng triển khai. Mỗi phiên bản phát hành ổn định (stable release) sẽ nằm trên `main` và thường được gắn tag phiên bản tương ứng (ví dụ: `v1.0.0`).

- **Nhánh** `develop` (nhánh phát triển): là nhánh tích hợp cho các tính năng sẽ có trong bản phát hành tiếp theo. Mã nguồn ở `develop` luôn phản ánh trạng thái phát triển mới nhất và sẵn sàng cho lần release kế tiếp. Nhánh `develop` được tạo một lần từ `main` khi khởi đầu dự án và duy trì suốt vòng đời phát triển.

- **Nhánh** `feature/*` (nhánh tính năng): được tách ra từ nhánh `develop` để phát triển từng tính năng mới hoặc cải tiến. Lập trình viên sẽ tạo một nhánh `feature/*` cho mỗi tính năng, thực hiện coding, commit và push trên nhánh đó. Khi tính năng hoàn tất (đã code xong và tự kiểm thử ổn định), lập trình viên tạo Pull Request (PR) để nhóm review và hợp nhất (merge) nhánh vào nhánh `develop`.

  > Lưu ý: nhánh feature chỉ merge vào develop, **không** merge trực tiếp vào **main**, nhằm đảm bảo **main** luôn chứa mã đã phát hành. Sau khi merge, có thể xóa nhánh feature để dọn dẹp.

- **Nhánh** `release/*` (nhánh phát hành): được tách ra từ `develop` khi chuẩn bị một bản phát hành mới. Ví dụ: khi kết thúc giai đoạn phát triển cho phiên bản **v1.0.0** (mọi tính năng dự kiến đã được merge vào **develop**), nhóm sẽ tạo nhánh `release/v1.0.0` từ **develop**. Trên nhánh release, **chỉ** thực hiện các bước hoàn thiện phiên bản: sửa lỗi nhỏ, tối ưu hiệu năng, viết thêm tài liệu hoặc chỉnh sửa số phiên bản... Tuyệt đối không thêm tính năng mới trên nhánh release. Khi mọi thứ đã ổn định, tester xác nhận không còn lỗi nghiêm trọng, **nhánh release được hợp nhất vào** `main` **và được gắn tag phiên bản** (ví dụ tag **v1.0.0**). Đồng thời, những cập nhật trên nhánh release cũng phải được hợp nhất ngược trở lại **develop** để **develop** bao gồm những sửa lỗi vừa thực hiện, sẵn sàng cho chu kỳ phát triển kế tiếp. Sau đó, nhánh release được xóa đi. (Trong trường hợp phát hiện xung đột khi merge, cần xử lý thủ công và commit bổ sung để đảm bảo **develop** và **main** đồng bộ).

- **Nhánh** `hotfix/*` (nhánh vá lỗi nóng): được sử dụng khi cần sửa chữa **ngay lập tức** một lỗi nghiêm trọng trên phiên bản đang chạy ở môi trường sản xuất (production). Nhánh hotfix được tách trực tiếp từ nhánh **main** tại điểm tag phiên bản đang gặp lỗi (ví dụ phiên bản `v1.0.0`). Lập trình viên sẽ thực hiện sửa lỗi trên nhánh này. Sau khi hoàn tất việc fix và tester xác nhận lỗi đã được khắc phục, nhánh hotfix được merge vào **main** và gắn tag phiên bản vá lỗi (ví dụ: `v1.0.1`). Đồng thời, thay đổi này cũng phải được merge vào **develop** để đội ngũ phát triển có mã nguồn cập nhật nhất. (Nếu song song đang có nhánh release chuẩn bị cho phiên bản kế tiếp, có thể merge hotfix đó vào nhánh release thay vì develop, sau đó phiên bản kế tiếp sẽ bao gồm luôn bản vá này). Nhánh hotfix sau đó cũng được xóa đi.

1.2. Tóm tắt tương tác giữa các nhánh trong Gitflow

- Mã nguồn mới được phát triển trên các nhánh `feature/*` tách từ `develop`, sau khi xong sẽ merge vào `develop`.
- Khi chuẩn bị phát hành, tạo nhánh `release/*` từ `develop` để thử nghiệm và sửa lỗi, xong xuôi merge vào `main` **(**đồng thời merge ngược lại `develop`).
- Nếu có sự cố khẩn cấp trên bản live, tạo nhánh hotfix từ `main`, fix xong merge vào lại main và `develop`.

2. Quản lý phiên bản (Version Management)

2.1. Các giai đoạn phiên bản

Để quản lý phiên bản, nhóm dự án sẽ phải tuân theo quy tắc **Semantic Versioning** (phiên bản ngữ nghĩa) với định dạng `MAJOR.MINOR.PATCH`.

> Ví dụ: `1.2.0` trong đó **1** là phiên bản lớn (Major), **2** là phiên bản nhỏ (Minor), 0 là bản vá (Patch). Mỗi khi phát hành, phiên bản sẽ được tăng tương ứng: tăng Patch cho lần vá lỗi, tăng Minor cho lần thêm tính năng tương thích ngược, và tăng Major khi có thay đổi phá vỡ tương thích cũ.

Quá trình phát hành phần mềm được chia thành các giai đoạn phiên bản chính sau, giúp kiểm soát chất lượng trước khi ra mắt bản ổn định:

- **Snapshot** (bản snapshot): Đây là phiên bản dựng thử trong giai đoạn rất sớm (tiền alpha). Bản snapshot thường là *bản dựng nội bộ* (internal build) hoặc nightly build, chứa những tính năng đang phát triển dở dang. Mục đích của snapshot là để nhóm phát triển xem trước tích hợp hoặc demo cho nội bộ.
- **Alpha** (bản alpha): Là giai đoạn đầu tiên của kiểm thử phần mềm. Phiên bản alpha thường đã có một số chức năng cơ bản nhưng **chưa hoàn thiện, có thể còn nhiều lỗi** và thiếu ổn định. Bản alpha thường được cung cấp cho một nhóm người dùng giới hạn (như đội kiểm thử nội bộ) để tìm kiếm lỗi. Cuối giai đoạn alpha sẽ thực hiện **đóng băng tính năng** (feature freeze), không bổ sung thêm chức năng mới, đảm bảo phần mềm đã hoàn chỉnh về chức năng trước khi sang beta.
- **Beta** (bản beta): Sau alpha là phiên bản beta. Về cơ bản **đã đầy đủ tính năng (feature-complete)** và bước vào giai đoạn kiểm thử mở rộng. Bản beta thường được phát hành rộng rãi hơn (cho một nhóm người dùng lớn hơn hoặc công chúng tình nguyện dùng thử) nhằm thu thập phản hồi. Mục tiêu của beta là phát hiện thêm lỗi trong điều kiện sử dụng thực tế và sửa các lỗi đó trước khi phát hành ứng viên.
- **Release Candidate (RC)** (bản ứng cử phát hành): Đây là phiên bản "*ứng viên*" cho phát hành chính thức, được tạo ra sau một hoặc nhiều bản beta ổn định. Bản RC đã qua nhiều vòng sửa lỗi, **độ ổn định cao** và gần như sẵn sàng để trở thành bản chính thức. Nguyên tắc là nếu **không có lỗi nghiêm trọng nào được phát hiện ở bản RC**, nó sẽ được dùng làm bản phát hành cuối cùng.
- **Stable release** (bản phát hành ổn định): Là phiên bản chính thức dành cho người dùng cuối, sau khi đã trải qua toàn bộ các giai đoạn thử nghiệm trên. Bản stable release thường chính là RC cuối cùng đạt yêu cầu (không còn lỗi nghiêm trọng). Lúc này phần mềm được coi là đủ chất lượng để phát hành rộng rãi.

Việc áp dụng Semantic Versioning giúp người dùng dễ dàng nhận biết mức độ thay đổi giữa các phiên bản, đảm bảo quản lý phụ thuộc chính xác, và giảm thiểu rủi ro khi cập nhật phiên bản mới.

2.2. Gắn thẻ (tag) phiên bản

Mỗi phiên bản phát hành (dù là alpha, beta, hay stable) đều nên được gắn tag trong Git để dễ dàng theo dõi. Tag thường được đặt với tiền tố `v` để biểu thị "version". Ví dụ:

- Tag bản alpha: `v1.0.0-alpha` (có thể thêm số hoặc ngày nếu có nhiều bản alpha, ví dụ `v1.0.0-alpha.2` cho alpha lần 2).
- Tag bản beta: `v1.0.0-beta` (tương tự có thể là `v1.0.0-beta.1`**,** nhiều bản beta).
- Tag bản RC: `v1.0.0-rc` (hoặc kèm số: `v1.0.0-rc.1`**).**
- Tag bản phát hành chính thức: `v1.0.0` (bỏ hậu tố, chỉ ghi số phiên bản đầy đủ).

- Tag bản vá/hotfix: `v1.0.1`, `v1.0.2` … (tăng số Patch).

2.3. Cách tạo tag trong Git

Để đánh dấu phiên bản, bạn dùng lệnh `git tag`. Thông thường nên sử dụng *annotated tag* để lưu thông tin người tạo và mô tả. Ví dụ, sau khi merge xong nhánh release vào `main`, PM hoặc người chịu trách nhiệm phát hành chạy:

```
git checkout main
git tag -a v1.0.0 -m "Release phiên bản 1.0.0"
git push origin main --tags
```

Lệnh trên tạo một tag `v1.0.0` trên commit hiện tại của `main` kèm thông điệp mô tả, rồi đẩy tag lên GitHub. Tương tự, khi có bản vá lỗi, ví dụ v1.0.1, sau khi merge hotfix vào `main` thì gắn tag `v1.0.1` và push tag. Việc gắn tag giúp định danh rõ ràng các phiên bản trong lịch sử commit. Trên GitHub, nhóm có thể vào mục **Releases** để chuyển các tag này thành bản phát hành, kèm mô tả release notes nếu cần.

3. Các lệnh Git thường dùng

Trong quá trình làm việc với Git, các thành viên sẽ thường xuyên sử dụng các lệnh Git cho việc tạo nhánh, cập nhật mã, thực hiện commit, merge... Dưới đây là một số lệnh thông dụng minh họa cho quy trình Gitflow của dự án:

3.1. Khởi tạo repo & thiết lập nhánh chính (Repo owner thực hiện)

```
# Khởi tạo repository Git (nếu dùng repo đã tạo trên GitHub thì git clone về)
git init

# Kết nối đến remote GitHub
git remote add origin <repo-url>

# Lấy mã nguồn nhánh main (nếu repo đã có sẵn README ban đầu)
git pull origin main

# Tạo nhánh develop từ nhánh main
git checkout -b develop main

# Đẩy nhánh develop lên remote và thiết lập track
git push -u origin develop
```

Giải thích: Tạo nhánh `develop` từ `main` để bắt đầu quy trình Gitflow. Cờ `-b` dùng để tạo mới và checkout. Lệnh `push -u origin develop` giúp thiết lập upstream cho nhánh develop.

3.2. Tạo nhánh tính năng (feature branch) (Dev thực hiện khi nhận task)

Ví dụ dev cần phát triển tính năng "đăng nhập người dùng":

```
# Tạo và chuyển sang nhánh feature mới từ develop
git checkout -b feature/login develop
```

Lệnh trên tạo nhánh tên `feature/login` tách từ `develop`. Dev sẽ làm việc trên nhánh này, thêm code, và commit các thay đổi.

3.3. Thực hiện thay đổi, commit và push

Giả sử dev đã chỉnh sửa, thêm file cho tính năng đăng nhập, giờ lưu lại thành một commit:

```
# Kiểm tra những file đã thay đổi
git status

# Chọn các file đã chỉnh sửa để đưa vào staging (hoặc dùng `git add .` để đưa tất cả vào)
git add src/auth/*

# Tạo commit với thông điệp ý nghĩa
git commit -m "Add user login feature with JWT authentication"
```

```
# Đẩy commit trên nhánh feature lên remote GitHub
git push -u origin feature/login
```

Giải thích: Cờ `-u` lần đầu push nhánh mới giúp liên kết nhánh local với nhánh remote cùng tên. Sau lệnh này, nhánh `feature/login` sẽ xuất hiện trên GitHub.

3.4. Tạo Pull Request (trên GitHub)

Thông thường dev sẽ vào trang GitHub, chuyển đến repository và sẽ thấy gợi ý “Compare & pull request” cho nhánh vừa push. Dev nhấp vào đó để mở PR, chọn *base* là `develop` và compare là nhánh `feature/login`. Viết mô tả PR bao gồm: chi tiết tính năng, ảnh chụp màn hình (nếu UI), liên kết Issue (ví dụ “Closes #12”), gán người review (ví dụ BA, tester) và người approve (PM nếu cần). Sau khi tạo, PR sẽ nằm ở thẻ **Pull Requests** chờ được review.

3.5. Cập nhật nhánh develop (pull changes)

Khi một PR được merge, các dev khác nên cập nhật nhánh `develop` local của mình để đồng bộ với remote. Sử dụng lệnh:

```
# Chuyển sang nhánh develop local
git checkout develop

# Kéo về những commit mới (nếu PR của người khác vừa merge)
git pull origin develop
```

Lệnh này giúp mọi người luôn có mã nguồn mới nhất trên nhánh phát triển.

3.6. Tạo nhánh release (chuẩn bị phát hành)

Khi sẵn sàng đóng phiên bản, PM hoặc dev lead tạo nhánh release từ `develop`:

- **Tạo nhánh release**:

  ```
  # Chuyển từ develop sang nhánh release/v1.0.0
  git checkout -b release/v1.0.0 develop

  # Đẩy nhánh release lên remote
  git push -u origin release/v1.0.0
  ```

  Trên nhánh release, nhóm chỉ commit sửa lỗi nhỏ hoặc cập nhật document. Mỗi lần sửa tương tự: `git add`, `git commit -m "Fix ..."` rồi `git push`.
- **Đánh dấu Snapshot:** Khi đã đóng code và sẵn sàng để thử tích hợp liên tục (CI), tạo một **snapshot tag** để dễ quay lại trạng thái tạm thời:

  ```
  git tag -a v1.0.0-snapshot.20250611 -m "Snapshot trước alpha"
  git push origin v1.0.0-snapshot.20250611
  ```

  `snapshot.YYYYMMDD` giúp chỉ rõ thời điểm chụp nhanh (daily build).
- **Đánh dấu Alpha:** Sau khi chạy qua CI/CD và ổn định về cơ bản, tạo **alpha tag** để gửi cho nhóm nội bộ thử nghiệm:

  ```
  git tag -a v1.0.0-alpha.1 -m "Alpha 1: nội bộ"
  git push origin v1.0.0-alpha.1
  ```

  Tăng số `alpha.N` mỗi lần có build mới sau khi fix lỗi nhỏ.
- **Đánh dấu Beta:** Khi alpha đã trải qua một số vòng thử và ổn định, tạo **beta tag** để mở rộng QA hoặc cho một nhóm người dùng giới hạn:

  ```
  git tag -a v1.0.0-beta.1 -m "Beta 1: thử nghiệm rộng"
  git push origin v1.0.0-beta.1
  ```

  Tương tự, tăng `beta.N` sau mỗi lượt feedback.
- **Đánh dấu Release Candidate (RC):** Khi tất cả lỗi nghiêm trọng được fix, chuẩn bị cho bản chính thức, tạo **rc tag**:

  ```
  git tag -a v1.0.0-rc.1 -m "RC 1: ứng viên phát hành"
  git push origin v1.0.0-rc.1
  ```

  Mỗi lần có sửa cuối cùng trên `release/v1.0.0`, tăng `rc.N` cho đến khi đạt chất lượng để merge xuống `main`.

3.7. Hợp nhất vào nhánh main (phát hành)

Sau khi tester duyệt bản release, tiến hành hợp nhất vào `main` để phát hành:

```
git checkout main

# đồng bộ với remote, phòng trường hợp có hotfix trước đó
git pull origin main

# merge nhánh release vào main (phát hành)
git merge --no-ff release/v1.0.0

# gắn tag phiên bản phát hành
git tag -a v1.0.0 -m "Release v1.0.0"

# đẩy commit merge và tag lên remote
git push origin main --tags
```

Giải thích: merge release -> main và gắn tag phiên bản. Tiếp theo, cần merge ngược về develop:

```
git checkout develop

# lấy cập nhật mới nhất (nếu có commit khác trong lúc chờ)
git pull origin develop

# hợp nhất những sửa chữa từ release vào lại develop
git merge --no-ff release/v1.0.0

git push origin develop
```

Cuối cùng, dọn dẹp nhánh release:

```
# xóa nhánh release local
git branch -d release/v1.0.0

# xóa nhánh release trên remote
git push origin --delete release/v1.0.0
```

Tương tự cho các bản 1.1.0, 1.2.0 chỉ khác tên nhánh và số tag.

3.8. Tạo nhánh hotfix (vá lỗi khẩn cấp)

Khi có lỗi sản xuất cần sửa ngay, tạo nhánh hotfix từ `main`:

```
# Tạo nhánh hotfix để sửa lỗi (ví dụ lỗi crash)
git checkout -b hotfix/fix-crash main
git push -u origin hotfix/fix-crash
```

Lập trình viên sửa code, commit và push như bình thường. Sau khi kiểm tra xong, hợp nhất vào `main`:

```
git checkout main

# đảm bảo main up-to-date
git pull origin main

# merge bản vá vào main
git merge --no-ff hotfix/fix-crash
git tag -a v1.0.1 -m "Hotfix v1.0.1 - fix crash on delete"

# đẩy lên và phát hành bản vá
git push origin main --tags
```

Đồng thời, hợp nhất vào `develop` để cập nhật mã nguồn phát triển:

```
git checkout develop
git pull origin develop
git merge --no-ff hotfix/fix-crash
git push origin develop
```

Cuối cùng, xóa nhánh hotfix đã dùng xong:

```
git branch -d hotfix/fix-crash
git push origin --delete hotfix/fix-crash
```

Giải thích: Việc merge hotfix vào cả main và develop là bắt buộc để đồng bộ fix cho nhánh phát triển. Tag v1.0.1 được tạo để phát hành bản vá.

Các lệnh trên chỉ là ví dụ điển hình. Trên thực tế, nhóm có thể dùng thêm nhiều lệnh khác (rebase, cherry-pick, stash...) tùy tình huống, nhưng với quy trình Gitflow căn bản, những lệnh trên đủ để thực hiện các hoạt động hằng ngày. Quan trọng là luôn đảm bảo làm việc trên đúng nhánh, thường xuyên **pull** cập nhật, và tuân thủ nguyên tắc merge qua Pull Request có review để giữ chất lượng code.

4. Sử dụng GitHub hiệu quả trong nhóm

Ngoài việc quản lý mã nguồn với Git, nhóm dự án còn tận dụng các công cụ hỗ trợ của GitHub để nâng cao hiệu quả cộng tác và chất lượng sản phẩm.

Pull Requests (Xem xét & tích hợp mã)

Pull Request (PR) là trung tâm của quy trình tích hợp code của nhóm. Mỗi khi một lập trình viên hoàn thành xong một nhánh tính năng, họ mở PR để hợp nhất vào `develop` (hoặc vào `release` / `main` tùy giai đoạn). Nhóm thiết lập một số quy tắc cho Pull Requests nhằm đảm bảo chất lượng và trách nhiệm chung.

- **Mỗi PR cần có mô tả chi tiết:** Template PR yêu cầu điền rõ *"Đã làm những gì", "Liên quan đến issue nào", "Cách kiểm thử"*. Dev phải mô tả đầy đủ giúp người review nắm nhanh nội dung thay đổi. Nếu PR liên quan đến giao diện, product designer phải đính kèm ảnh chụp màn hình hoặc GIF minh họa. Với các thay đổi lớn về kiến trúc hoặc logic hệ thống, system architect cần bổ sung mô tả kỹ thuật hoặc trình bày lý do thực hiện giải pháp.
- **Gán người review:** Việc gán reviewer thường yêu cầu ít nhất một dev khác xem xét code, và BA hoặc tester kiểm tra trên góc độ nghiệp vụ và chất lượng. Product designer sẽ review nếu PR có liên quan tới giao diện người dùng, còn system architect sẽ xem xét các thay đổi ảnh hưởng đến kiến trúc hoặc luồng xử lý chính. GitHub cho phép thiết lập *Code Owners* hoặc *reviewer bắt buộc*, ví dụ các PR vào develop yêu cầu tối thiểu một approval từ dev lead hoặc SA.
- **Thảo luận trên PR:** người review để lại comment inline trên từng đoạn code hoặc tổng thể, người mở PR phải phản hồi từng ý kiến và chỉnh sửa nếu cần, sau đó push commit cập nhật. Nguyên tắc "*Code review xong mới được merge*" được áp dụng nghiêm túc kể cả với các thay đổi nhỏ. Môi trường PR của GitHub giúp nhóm lưu lại đầy đủ lịch sử thảo luận, tránh lặp lại tranh cãi sau này.
- **Kết nối PR với Issue và Projects:** PR cũng là cầu nối giữa công việc và mã nguồn: khi mở PR, dev liên kết với Issue (bằng "Closes `#xxx`") và chọn thẻ tương ứng trong Projects. Khi PR được merge, Issue sẽ tự động đóng và thẻ được chuyển sang Done thông qua automation. Trong quy trình này, product designer có thể gắn mình vào PR để theo dõi và xác nhận giao diện được triển khai đúng thiết kế ban đầu, còn system architect có thể kiểm tra các ảnh hưởng side-effect hoặc tính nhất quán hệ thống.
- **Merge và xoá nhánh:** Việc merge và xoá nhánh được thực hiện bởi người mở PR sau khi đã đủ điều kiện (review OK, CI pass), thông qua nút **Merge** trên GitHub (chọn merge commit hoặc squash tùy trường hợp). Sau khi merge, nhóm chọn **Delete branch** để dọn repo, và lập trình viên tự xóa nhánh local nếu cần.