**Trường Đại Học Công Nghiệp TP. Hồ Chí Minh**

**Khoa Công Nghệ Thông Tin**

Logo

Description automatically generated with low confidence

**MÔN: KIẾN TRÚC VÀ THIẾT KẾ PHẦN MỀM**

**ĐỀ TÀI:**

**HỆ THỐNG QUẢN LÝ LỚP HỌC TÍN CHỈ**

**Thành viên nhóm:**

|  |  |  |
| --- | --- | --- |
| **STT** | **MSSV** | **Họ tên** |
| 1 | 20074301 | Trần Quốc Việt |
| 2 | 19431331 | Huỳnh Duy Kha |
| 3 | 20075801 | Vũ Thái Dương |

**Giảng viên:**

**Võ Văn Hải**

TP. Hồ Chí Minh, tháng 05, năm 2024

**MỤC LỤC**

1. **Cơ sở lý thuyết**
   1. Kiến trúc phần mềm.
   2. Các nguyên tắc cơ bản..
   3. Các kiến trúc phần mềm phổ biến và trình bày các hiểu biết về kiến trúc đó.
2. **Mô Tả yêu cầu**
   1. Yêu cầu ban đầu (Yêu cầu hệ thống)..
   2. Giả thuyết mới.
3. **Biện luận kiến trúc**
   1. Lý do chọn kiến trúc.
   2. Mô hình hóa trình bày kiến trúc.
   3. Mô phỏng, lựa chọn deploy ứng dụng.
   4. Các Diagram.
4. **Demo**
   1. Trình bài triển khai dự án.
   2. Hình ảnh minh họa.
5. **Kết luận**

Tài liệu tham khảo.

**CHƯƠNG I CƠ SỞ LÝ THUYẾT**

* 1. **Khái niệm về kiến trúc phần mềm.**

Kiến trúc: là sự tổ chức cơ bản của một hệ thống, thể hiện qua các thành phần của nó, mối quan hệ giữa các thành phần với nhau và với môi trường, cùng các nguyên tắc điều chỉnh thiết kế và sự phát triển của nó. Kiến trúc phần mềm (Software architecture) đề cập đến cấu trúc cơ bản của một hệ thống phần mềm và quy tắc của việc tạo ra những cấu trúc và hệ thống. Mỗi cấu trúc bao gồm sự sắp xếp của các yếu tố phần mềm, mối quan hệ giữa các yếu tố, và tính chất của các yếu tố đó.

* 1. **Các nguyên tắc cơ bản.**

Tách biệt mối quan tâm (Separation of Concerns): Phân chia hệ thống thành các phần nhỏ hơn có mối quan tâm riêng biệt, giúp dễ dàng quản lý và thay đổi.

Tính mô-đun (Modularity): Thiết kế hệ thống sao cho các phần có thể được phát triển, sửa đổi và thử nghiệm độc lập.

Tính cân nhắc (Balance): Đảm bảo sự cân nhắc giữa các yếu tố như hiệu suất, bảo mật, tính sẵn sàng và chi phí.

Tính khả dụng (Usability): Đảm bảo hệ thống dễ sử dụng và thân thiện với người dùng cuối.

Tính linh hoạt (Flexibility): Hệ thống cần có khả năng thích ứng và mở rộng để đáp ứng yêu cầu mới và thay đổi.

Tính bảo trì (Maintainability): Thiết kế sao cho hệ thống dễ bảo trì, sửa đổi và nâng cấp.

Tính mở (Openness): Sử dụng các tiêu chuẩn mở và giao thức để tạo ra hệ thống có khả năng tương tác và tích hợp tốt với các hệ thống khác.

Tính hiệu suất (Performance): Đảm bảo hệ thống đáp ứng các yêu cầu về thời gian xử lý và tài nguyên sử dụng.

Tính bảo mật (Security): Bảo vệ hệ thống và dữ liệu khỏi các mối đe dọa bằng cách triển khai các biện pháp bảo mật phù hợp.

Tính mở rộng (Scalability): Thiết kế sao cho hệ thống có thể mở rộng để đáp ứng nhu cầu tăng trưởng của người dùng mà không ảnh hưởng đến hiệu suất.

Loose Coupling: nếu các lớp (classes) phu thuộc vào nhau, chúng sẽ ảnh hưởng nhau. Càng ít các lớp phụ thuộc (Low coupling hay loose coupling), thì việc thay đổi chúng càng dễ dàng.

High Cohesion: mức độ mà các element của một tổng thể thuộc về nhau. Cohesion ngược lại so với Coupling, các components của classes nên có tính gắn kết cao.

Locality: Các thay đổi, bảo trì, tiện ích mở rộng chỉ mang tính cục bộ. Điều này dẫn đến không gây hại cho toàn bộ môi trường.

Removeable: Các component phần mềm phải có thể dễ dàng tháo gỡ.

Small Components: lý tưởng nhất là hệ thống phần mềm chỉ gồm các thành phần nhỏ (small component), mỗi thành phần chỉ làm một nhiệm vụ.

* 1. **Các kiến trúc phần mềm phổ biến**

Có thể phân ra thành hai loại chính:

\* **Kiến trúc monolithic**:

Toàn bộ ứng dụng được phát triển và triển khai như một đơn vị duy nhất. Tất cả các thành phần của ứng dụng (giao diện người dùng, logic nghiệp vụ, cơ sở dữ liệu) được kết hợp trong một "bản ghi" lớn. Gồm các kiến trúc :

* Layered architecture (kiến trúc lớp):

Layered architecture là một kiến trúc phổ biến trong phát triển phần mềm, trong đó hệ thống được chia thành các tầng (layers) khác nhau, mỗi tầng chịu trách nhiệm cho một loại hoạt động cụ thể.

Các tầng thường được tổ chức theo một cách hợp lý, từ tầng giao diện người dùng (presentation layer) đến tầng xử lý logic (business logic layer), và sau đó đến tầng truy cập dữ liệu (data access layer)…

Cấu trúc tầng giúp tách biệt và tổ chức rõ ràng các thành phần của hệ thống, tạo điều kiện thuận lợi cho việc phát triển, bảo trì và mở rộng ứng dụng.

* Pipeline architecture (kiến trúc ống dẫn):

Pipeline architecture tập trung vào việc sắp xếp các xử lý thành một chuỗi liên tiếp (pipeline), trong đó đầu ra của mỗi bước xử lý trở thành đầu vào của bước tiếp theo.

Mỗi bước xử lý trong pipeline thường là một module hoặc thành phần nhỏ thực hiện một công việc cụ thể.

Kiến trúc này thường được sử dụng trong các ứng dụng xử lý dữ liệu hoặc các quy trình xử lý thông tin, nơi mà dữ liệu cần được xử lý theo một chuỗi các bước khác nhau.

* Microkernel architecture (kiến trúc vi nhân):

Microkernel architecture tập trung vào việc chia nhỏ hệ thống thành các module nhỏ và đơn giản, gọi là "microkernels", mỗi microkernel thực hiện một chức năng cụ thể.

Các microkernel giao tiếp với nhau thông qua giao diện tiêu chuẩn, cho phép tính năng linh hoạt và mở rộng.

Đặc điểm quan trọng của kiến trúc này là sự phân tách giữa các tính năng cốt lõi của hệ thống và các tính năng phụ trợ, giúp giảm thiểu sự phức tạp và tăng tính linh hoạt của hệ thống.

Các kiến trúc của phân loại monolithic có ưu điểm là dễ dàng triển khai, toàn bộ ứng dụng được triển khai như một đơn vị duy nhất, việc bảo trì và debug cũng trở nên dễ dàng hơn. Tuy nhiên lại khó mở rộng và khả năng tái sử dụng thấp.

**\* Kiến trúc Distributed(kiến trúc phân tán)**:

Ứng dụng được chia thành các dịch vụ độc lập nhau, mỗi dịch vụ thực hiện một chức năng cụ thể, các dịch vụ này có thể triển khai trên các máy chủ khác nhau và giao tiếp với nhau qua mạng . Gồm các kiến trúc:

* Service-based architecture (kiến trúc dựa trên dịch vụ):

Kiến trúc dựa trên dịch vụ tập trung vào việc tạo ra các dịch vụ độc lập, có thể tái sử dụng và có thể được ghép nối với nhau để tạo thành các ứng dụng hoàn chỉnh.

Mỗi dịch vụ thực hiện một chức năng cụ thể và giao tiếp với các dịch vụ khác qua các giao diện dịch vụ xác định.

Service-based architecture dễ dàng mở rộng và bảo trì. Các dịch vụ có thể được phát triển, triển khai và mở rộng một cách độc lập. Tăng khả năng tái sử dụng mã nguồn và tài nguyên.

* Event-driven architecture (Kiến trúc hướng sự kiện):

Kiến trúc hướng sự kiện dựa trên việc phát hiện và phản ứng với các sự kiện xảy ra trong hệ thống.

Các thành phần trong hệ thống giao tiếp với nhau thông qua các sự kiện và phản hồi với các hành động phù hợp.

Khả năng mở rộng cao và dễ dàng tích hợp các thành phần mới. Phù hợp với các hệ thống yêu cầu phản ứng nhanh với các thay đổi hoặc sự kiện xảy ra. Tính linh hoạt cao, giúp hệ thống dễ dàng thích nghi với các thay đổi trong môi trường làm việc

* Space-based architecture (Kiến trúc dựa trên không gian):

Kiến trúc dựa trên không gian sử dụng không gian dữ liệu phân tán để lưu trữ trạng thái và giao tiếp giữa các thành phần của hệ thống.

Dữ liệu và các thành phần xử lý được phân tán qua các nút khác nhau trong một không gian dữ liệu chung.

Tăng khả năng chịu lỗi và khả năng mở rộng do việc phân tán dữ liệu và xử lý. Giảm tắc nghẽn và cải thiện hiệu suất thông qua việc phân tán tải công việc. Thích hợp cho các ứng dụng yêu cầu xử lý đồng thời và hiệu suất cao.

* Service-orirented architecture (kiến trúc hướng dịch vụ):

SOA là một mô hình kiến trúc mà các chức năng được đóng gói dưới dạng các dịch vụ độc lập.

Các dịch vụ này có thể được phát triển, triển khai và quản lý một cách riêng biệt, nhưng tương tác với nhau qua các giao thức chuẩn.

Khả năng tái sử dụng cao và khả năng tích hợp với các hệ thống hiện có. Tính linh hoạt cao trong việc phát triển và triển khai các dịch vụ mới. Đòi hỏi việc xác định rõ ràng các giao diện dịch vụ và quản lý dịch vụ một cách hiệu quả.

* Microservices architecture:

Kiến trúc microservices phân chia ứng dụng thành các dịch vụ nhỏ, độc lập và có thể triển khai riêng rẽ.

Mỗi microservice đảm nhận một chức năng cụ thể và giao tiếp với các microservice khác thông qua các giao thức nhẹ như HTTP/REST hoặc messaging.

Khả năng mở rộng và bảo trì tốt hơn do mỗi microservice có thể được phát triển, triển khai và mở rộng độc lập. Dễ dàng áp dụng các công nghệ và ngôn ngữ lập trình khác nhau cho từng microservice. Tăng khả năng chịu lỗi do sự cô lập giữa các microservice; nếu một microservice gặp sự cố, nó không ảnh hưởng đến toàn bộ hệ thống.

**CHƯƠNG II MÔ TẢ YÊU CẦU**

* 1. **Yêu cầu ban đầu**

Một trường đại học cần quản lý hệ thống đăng ký học phần của sinh viên theo theo cơ chế tín chỉ. Các sinh viên sẽ được biên chế ở các khoa theo mỗi ngành học của mình đăng ký. Trong mỗi học kỳ, nhà trường sẽ chọn một số môn học của từng ngành học cho sinh viên đăng ký học. Sinh viên đăng ký theo học vào một lớp nhất định (lớp tín chỉ) với số sinh viên được đăng ký tối đa được qui định cho từng lớp. Môn học được mở có thể là môn học tự chọn Mỗi môn học có một hoặc nhiều môn học tiên quyết. Sinh viên sẽ không được đăng ký môn học mà có môn tiên quyết mình chưa học. Mỗi học kỳ sinh viên được đăng ký tối đa 30 tín chỉ. Nếu quá số này, hệ thống sẽ không cho đăng ký. Để phòng ngừa trường hợp sinh viên đăng ký xong rồi hủy bỏ, nhà trường yêu cầu sinh viên phải xác nhận trước khi đăng ký. Vào ngày mở đăng ký, sinh viên sẽ đăng nhập vào hệ thống và sẽ nhìn thấy danh sách các môn học mà mình có khả năng đăng ký. Sinh viên chọn các môn học và tiến hành đăng ký. Trường hợp các lớp đã đầy, sinh viên sẽ được đưa vào một danh sách dự bị để nhà trường cân nhắc có mở thêm lớp hay không. Nếu không mở thêm lớp, sinh viên sẽ bị hủy đăng ký môn đó. Sau khi đăng ký thành công, một email thông báo sẽ được gửi cho sinh viên xác nhận việc đăng ký và nhận quyết định đóng học phí. Ngoài ra, hệ thống còn có khả năng cho phép sinh viên xem thông tin học tập của mình (số tín chỉ đã đạt, số môn đã học, điểm môn học, điểm trung bình tích lũy…), thời khóa biểu theo tuần và các tiện ích khác. Sau khi số tín chỉ đã đạt theo từng ngành, sinh viên có quyền đăng ký xét tốt nghiệp. Nếu mọi tiêu chuẩn đều thỏa mãn, sinh viên sẽ được cấp bằng tốt nghiệp và sẽ được đưa vào danh sách các cựu sinh viên. Thông tin về bằng cấp sẽ được công khai trên trang web của nhà trường. Thông tin của cựu sinh viên sẽ được lưu giữ để theo dõi quá trình làm việc (nếu sinh viên đồng ý), làm các cuộc survey, cũng như nhiều hoạt động khác.

Từ đề bài trên ta có thể phân tích ra các yêu cầu:

* **Quản lý đăng ký học phần theo cơ chế tín chỉ**: Yêu cầu: Hệ thống cần phải cho phép sinh viên đăng ký các học phần dựa trên số tín chỉ được qui định cho mỗi học kỳ.
* **Quản lý đăng ký theo ngành học và khoa**: Yêu cầu: Sinh viên được phân vào các khoa và đăng ký học phần theo ngành học của mình.
* **Quản lý lớp học và số lượng sinh viên đăng ký**: Yêu cầu: Hệ thống cần quản lý số lượng sinh viên được đăng ký vào mỗi lớp học, và nếu lớp đầy, cần xử lý danh sách dự bị và hủy bỏ đăng ký nếu không mở thêm lớp.
* **Quản lý môn học và môn học tiên quyết**: Yêu cầu: Hệ thống cần theo dõi các môn học tiên quyết và không cho phép sinh viên đăng ký môn học mà họ chưa hoàn thành môn tiên quyết.
* **Xác nhận đăng ký và thông báo cho sinh viên**: Yêu cầu: Sau khi đăng ký, sinh viên cần được xác nhận đăng ký và nhận thông báo qua email.
* **Quản lý thông tin cá nhân và học tập của sinh viên**: Yêu cầu: Sinh viên cần có khả năng xem thông tin học tập của mình, bao gồm số tín chỉ đã đạt, số môn đã học, điểm môn học, điểm trung bình tích lũy và thời khóa biểu.
* **Quản lý quy trình xét tốt nghiệp và cấp bằng**: Yêu cầu: Hệ thống cần hỗ trợ quá trình xét tốt nghiệp và cấp bằng tốt nghiệp cho sinh viên đã đủ điều kiện.
* **Quản lý thông tin cựu sinh viên**: Yêu cầu: Hệ thống cần lưu trữ thông tin của cựu sinh viên để theo dõi quá trình làm việc, thực hiện các cuộc khảo sát và các hoạt động khác.
  1. **Giả thuyết mới.**

Các giả thuyết mới:

- Mỗi học phần sẽ được gán một số tín chỉ và hệ thống sẽ kiểm tra số tín chỉ đã đăng ký của sinh viên để đảm bảo không vượt quá ngưỡng tín chỉ cho phép.

- Hệ thống cần theo dõi thông tin sinh viên, bao gồm ngành học và khoa, để họ chỉ có thể đăng ký các môn học phù hợp với ngành của mình.

- Hệ thống cần theo dõi số lượng sinh viên đăng ký vào mỗi lớp, thông báo khi lớp đầy, và tự động xử lý danh sách dự bị.

- Hệ thống cần có cơ sở dữ liệu về môn học và môn học tiên quyết, và kiểm tra điều kiện này khi sinh viên đăng ký.

- Hệ thống cần có cơ chế gửi email tự động khi sinh viên đăng ký thành công.

- Hệ thống cần lưu trữ và cập nhật thông tin này liên tục, cũng như cung cấp giao diện để sinh viên truy cập.

- Hệ thống cần theo dõi số tín chỉ đã đạt và các điều kiện khác để tự động xác nhận và thông báo cho sinh viên khi họ đủ điều kiện xét tốt nghiệp.

- Hệ thống cần có cơ sở dữ liệu riêng để lưu trữ thông tin của cựu sinh viên và cung cấp giao diện để quản lý thông tin này.

**CHƯƠNG III BIỆN LUẬN KIẾN TRÚC**

**3.1 Kiến trúc layered (kiến trúc phân lớp).**

Kiến trúc layered (kiến trúc phân lớp) còn được goi là kiểu kiến trúc n-tiered, là một trong những kiến trúc phổ biến nhất từ những năm 90 cho tới ngày nay. Mỗi lớp trong kiến trúc này có chức năng riêng và tương tác với lớp ngay trên hoặc dưới nó. Các mô hình sử dụng kiến trúc này bao gồm MVC, MVVM, ...

Các thành phần (component) trong kiến trúc phân lớp được tổ chức thành các lớp logic ngang, mỗi lớp thực hiện một vai trò cụ thể trong ứng dụng (ví dụ: logic trình bày - User interface hoặc logic nghiệp vụ - Business). Mặc dù không có con số cụ thể về số lượng và loại lớp phải có, kiến trúc phân lớp thường bao gồm bốn lớp tiêu chuẩn: presentation, business, persistence, và database.

Lớp Presentation: Gồm các thành phần như giao diện người dùng, trình duyệt, ứng dụng di động, hoặc các thành phần tương tác với người dùng khác. Có chức năng hiển thị thông tin và kết quả từ lớp Logic cho người dùng và chuyển dữ liệu người dùng nhập vào lớp Logic.

Lớp Business: Đôi khi còn được gọi là lớp Service. Có nhiệm vụ xử lý các logic nghiệp vụ, nhận yêu cầu từ lớp Presentation, xử lý nó và gửi yêu cầu tương ứng đến lớp Persistence.

Lớp Persistence: Đảm nhiệm việc gửi các yêu cầu đến lớp Database để thực hiện các thao tác liên quan đến dữ liệu.

Lớp Database: Bao gồm cơ sở dữ liệu và các thành phần liên quan như hệ quản trị cơ sở dữ liệu. Có nhiệm vụ quản lý cơ sở dữ liệu, thực hiện thao tác đọc và ghi dữ liệu và triển khai các truy vấn và lưu trữ dữ liệu theo cách được định nghĩa từ lớp Persistence.

Trong kiến trúc phân lớp, các lớp có thể ở trạng thái mở hoặc đóng. Hình minh hoạ ở dưới thể hiện kiến trúc phân lớp đóng. Các lớp cô lập (layers of isolation) được tạo ra nhằm đảm bảo sự độc lập giữa các lớp với nhau, thay đổi của lớp này không ảnh hưởng tới lớp khác.

**3.2 Lý do chọn kiến trúc.**

FrameWork đơn giản và dễ học, phổ biến và dễ hiểu, giúp cho các thành viên trong nhóm dễ dàng nắm bắt và triển khai. Dễ quản lý hơn khi hệ thống được chia thành các lớp riêng biệt.

Tính mô-đun hóa cao vì chia hệ thống thành các tầng riêng biệt, mỗi tầng chịu trách nhiệm một chức năng cụ thể. Điều này giúp dễ dàng phát triển, bảo trì và nâng cấp từng phần của hệ thống mà không ảnh hưởng đến các phần khác.

Tái sử dụng mã, các thành phần trong mỗi lớp có thể được tái sử dụng trong các dự án khác, giảm thiểu thời gian phát triển.

Tính linh hoạt, mỗi lớp có thể được thay thế hoặc nâng cấp độc lập mà không cần phải thay đổi toàn bộ hệ thống, giúp tăng tính linh hoạt trong việc phản ứng với các yêu cầu mới hoặc thay đổi.

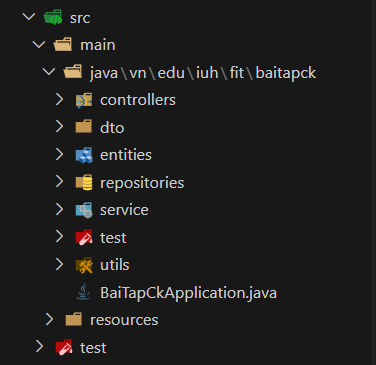
Sự phụ thuộc giảm đi vì chức năng của mỗi tầng là riêng biệt so với các tầng khác.

Việc kiểm thử dễ dàng hơn do các thành phần được phân tách; mỗi thành phần có thể được kiểm thử một cách riêng lẻ.

Kiến trúc phân lớp là lựa chọn tốt cho các ứng dụng hoặc trang web vừa và nhỏ. Nó cũng là một lựa chọn kiến trúc tốt, đặc biệt là như một điểm khởi đầu, cho các tình huống có ngân sách và thời gian hạn chế. Chi phí vận hành khá thấp.

Kiến trúc phân lớp cũng là một lựa chọn tốt khi một kiến trúc sư đang phân tích nhu cầu và yêu cầu kinh doanh và chưa chắc chắn về kiểu kiến trúc nào sẽ phù hợp nhất.

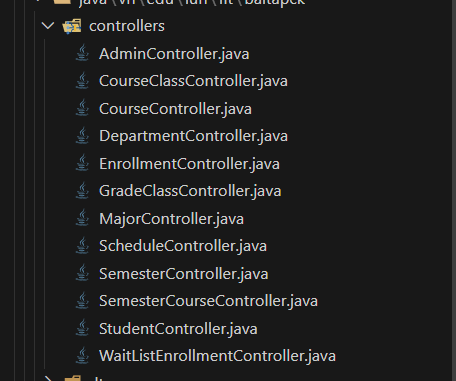
**3.3 Mô hình hóa kiến trúc:**

****

**1. Presentation Layer (Lớp trình bày):**

Thư mục: controllers

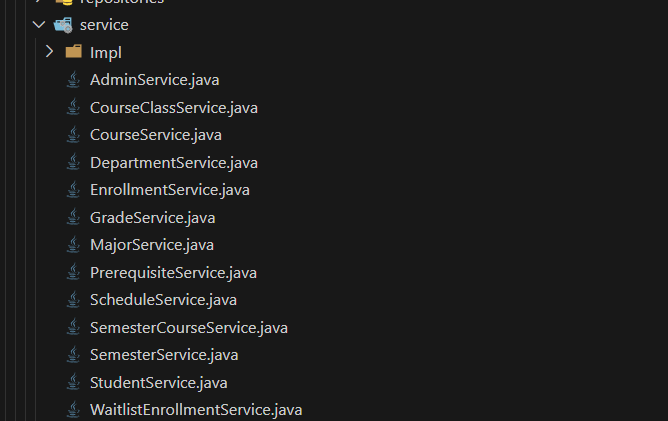
Nhiệm vụ: Xử lý các yêu cầu từ người dùng, gửi yêu cầu đến các dịch vụ, và trả về kết quả hoặc trang giao diện. Chi tiết: controllers: Chứa các lớp điều khiển (controller classes) chịu trách nhiệm tiếp nhận các yêu cầu HTTP, xử lý dữ liệu đầu vào, và chuyển các yêu cầu này đến các lớp dịch vụ thích hợp. Sau đó, các controller này sẽ trả lại phản hồi cho người dùng.

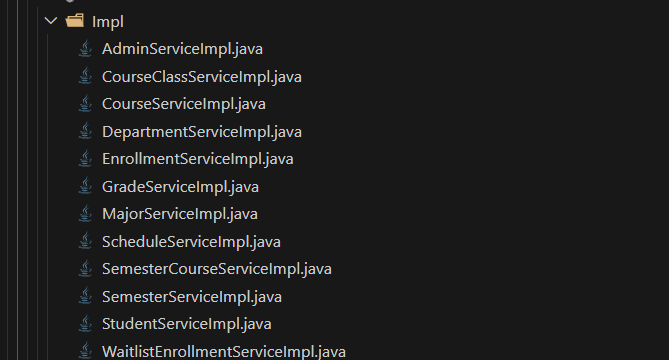


**2. Application Layer (Lớp ứng dụng):**

Thư mục: service

Nhiệm vụ: Chứa logic nghiệp vụ và xử lý các yêu cầu từ lớp trình bày. Chi tiết: service: Chứa các lớp dịch vụ (service classes) chịu trách nhiệm thực hiện các quy tắc nghiệp vụ và xử lý logic cần thiết để đáp ứng các yêu cầu của người dung.

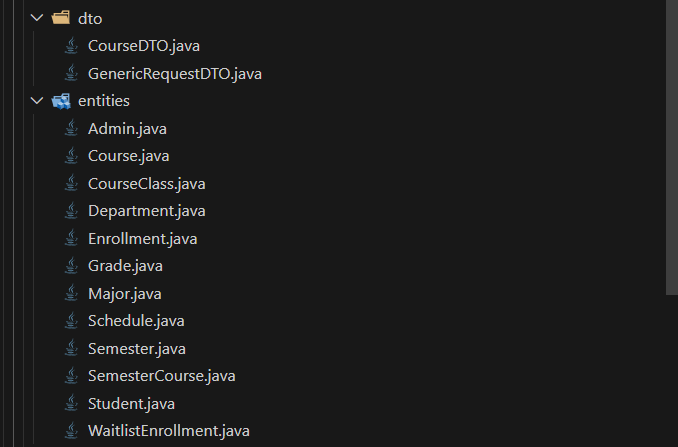




**3. Business Logic Layer (Lớp logic nghiệp vụ):**

Thư mục: dto, entities

Nhiệm vụ: Định nghĩa các đối tượng truyền dữ liệu và các thực thể nghiệp vụ. Chi tiết: dto (Data Transfer Objects): Chứa các lớp DTO giúp chuyển dữ liệu giữa các tầng khác nhau mà không bao gồm logic nghiệp vụ. Các DTO này thường được sử dụng để đảm bảo rằng chỉ những dữ liệu cần thiết được truyền tải. entities: Chứa các lớp thực thể (entity classes) mô hình hóa các đối tượng nghiệp vụ và ánh xạ chúng đến các bảng trong cơ sở dữ liệu.

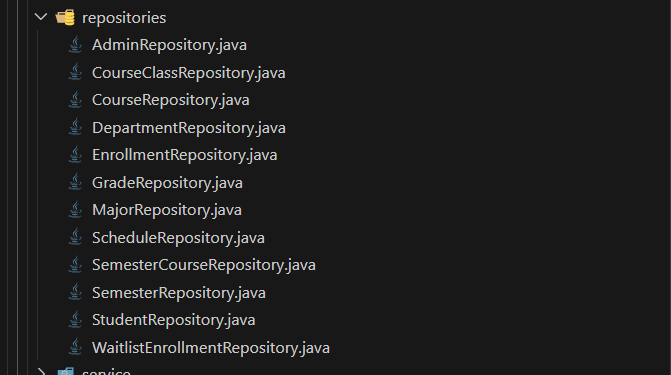


**4. Data Access Layer (Lớp truy cập dữ liệu):**

Thư mục: repositories

Nhiệm vụ: Xử lý các thao tác với cơ sở dữ liệu.

Chi tiết: repositories: Chứa các lớp kho (repository classes) chịu trách nhiệm tương tác với cơ sở dữ liệu. Các repository này thường cung cấp các phương thức CRUD (Create, Read, Update, Delete) và có thể sử dụng các framework ORM (Object-Relational Mapping) như Hibernate hoặc JPA.



**3.4 Mô phỏng, deploy ứng dụng.**

**1. Môi trường phát triển:**

VsCode: phiên bản 1.89.1 x64

Intellij IDEA: phiên bản 2023.2.1

Backend: java

FontEnd: react native

Hệ điều hành window

**2. Môi trường kiểm thử:**

Jenkin

**3. Deploy ứng dụng:**

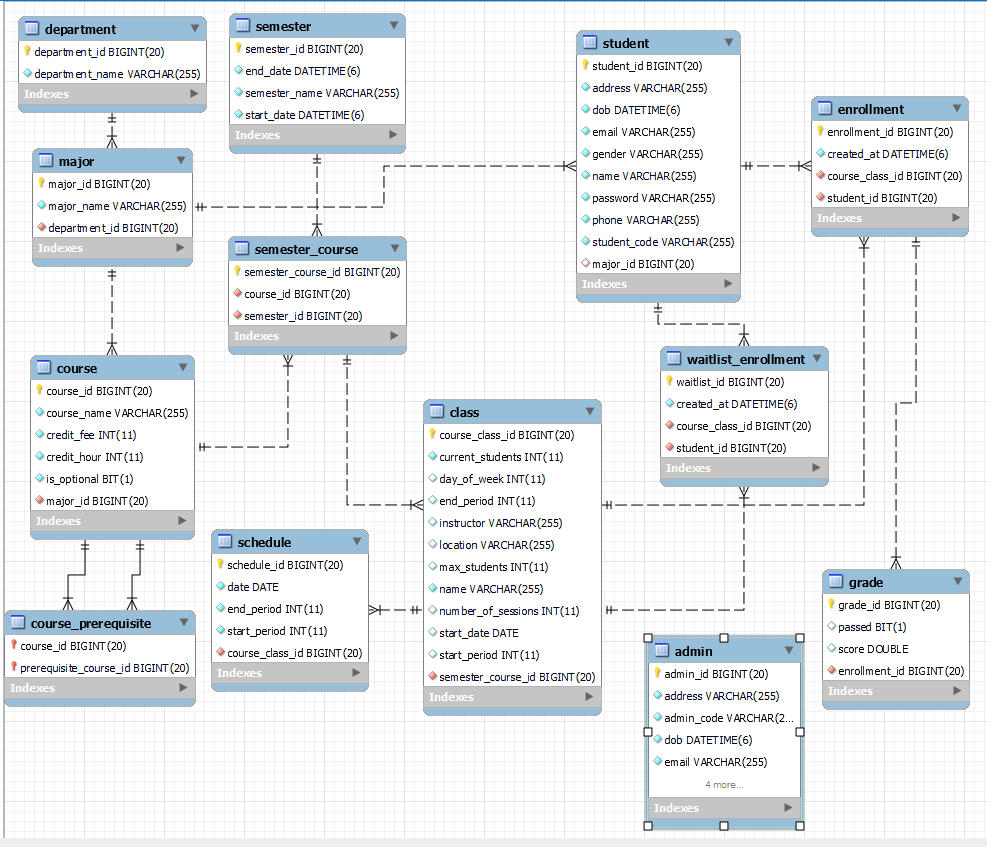
Sử dụng Docker để tạo và quản lý các container ứng dụng. Docker cung cấp một cách tiện lợi để đóng gói ứng dụng và các phụ thuộc của nó vào các container độc lập, giúp đảm bảo tính di động và khả năng chạy đa nền tảng.

Khi triển khai, chúng tôi sử dụng Docker để xây dựng các image của ứng dụng và sau đó chạy các container từ những image này trên môi trường triển khai. Việc sử dụng Docker giúp giảm thiểu sự phức tạp trong quá trình triển khai, đồng thời cung cấp một môi trường cô đọng và nhất quán giữa môi trường phát triển và triển khai.

**3.5 Các Diagram (deploy diagram, package diagram, component diagram)**

**1. Use Case Diagram:**

**2. Class Diagram:**

****

**3. Sequence Diagram:**

**4. Deployment Diagram:**

**CHƯƠNG IV DEMO**

**4.1 Trình bày triển khai dự án**

**1. Giới thiệu về chương trình:**

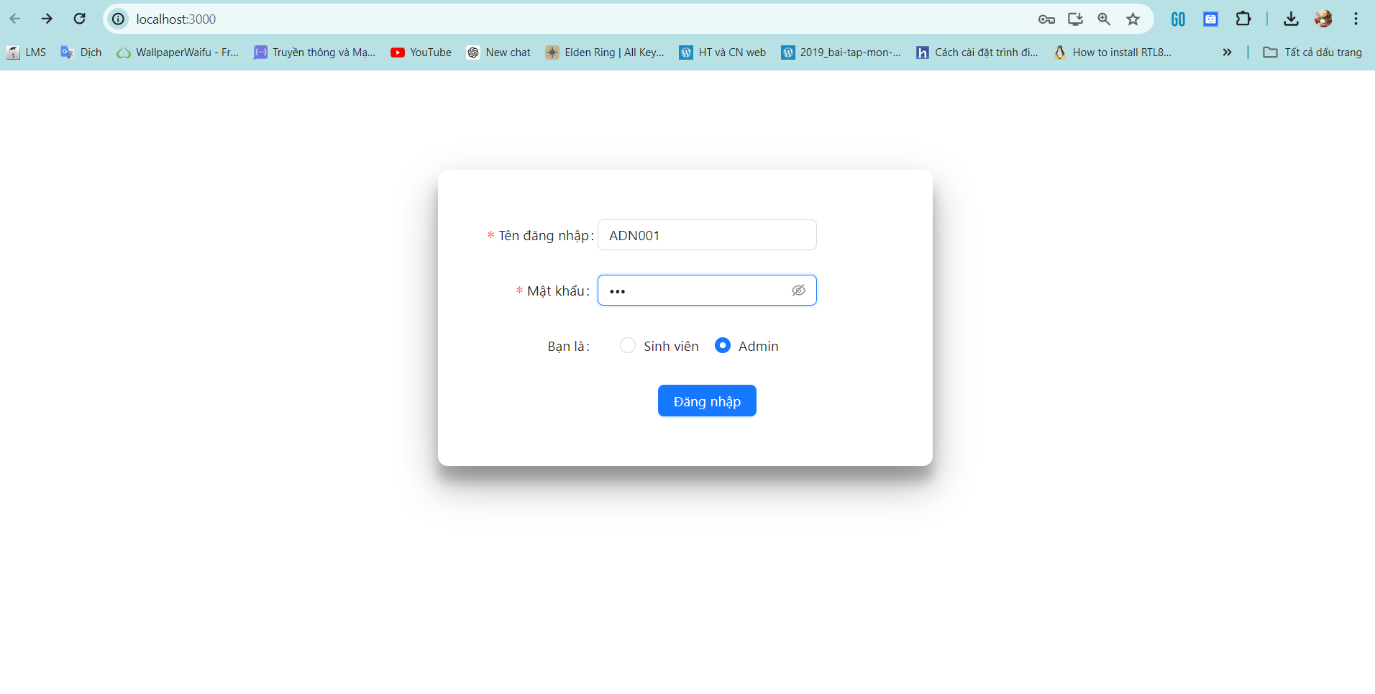
**2. Môi trường phát triển:**

**3. Quá trình phát triển:**

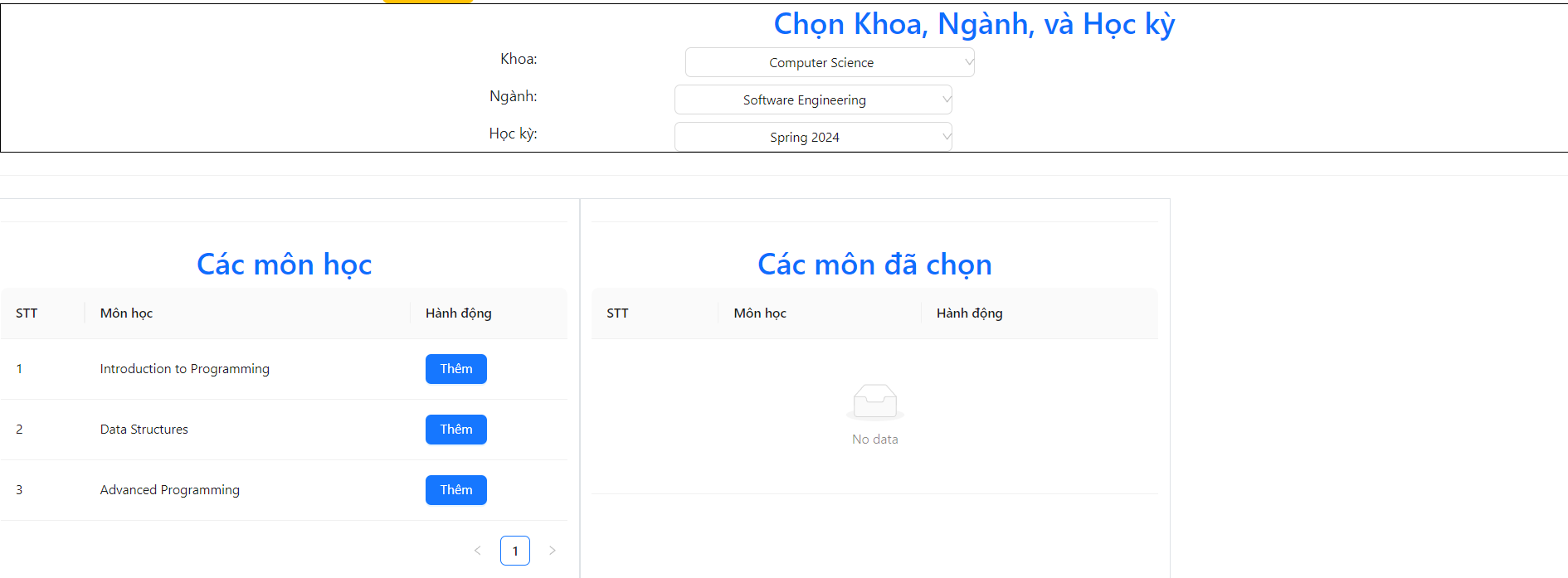
**4. Triển khai:**

**4.2 Hình ảnh demo**

**1. Giao diện đăng nhập**

****

**2.Giao diện admin quản lí học phần**

****

**CHƯƠNG V KẾT LUẬN**

**TÀI LIỆU THAM KHẢO**