# Tóm tắt

Nhà thông minh là khái niệm đã không còn quá xa lạ với chúng ta mà hiện tại đã trở thành xu thế công nghệ tất yếu, là tiêu chuẩn của nhà ở hiện đại. Chúng ta đang sống trong một thế giới của công nghệ. Một ngôi nhà hoàn hảo không chỉ sang trọng trong thiết kế, mà còn phải mang lại cảm giác thoải mái, tiện nghi và còn có khả năng hiểu được chủ nhân. Cuộc đua nhà thông minh đã và đang có những bước phát triển ở Việt Nạm, có thể kể đến như hệ thống nhà thông minh Bkav. Tuy nhiên, hiện tại ở Việt Nam, nhà thông minh chưa được phổ biến rộng, một phần vì chi phí lắp đặt, chi phí xây dựng khá cao. Nắm bắt xu hướng đó, ở luận văn này của nhóm với đề tài “Xây dựng ứng dụng nền tảng web hoặc di động tương tác với nhà thông minh qua các kịch bản” giúp chúng ta có thể tự tạo nên một hế thống nhà thông minh nhỏ, có khả năng quản lý tự động một số thiết bị trong gia đình với mức chi phí phù hợp.

Về phương pháp thực hiện, nhóm chia việc xây dựng hệ thống thành các phần: giao diện, server và bộ phận tương tác thiết bị trên Raspberry Pi. Giao diện giúp người dùng quản lý thiết bị, kịch bản trong nhà. Phía server làm nhiệm vụ lưu trữ, kiểm tra, xử lý, vận hành kịch bản từ phía người dùng yêu cầu. Bộ phận tương tác có chức năng cung cấp cho server những dịch vụ (services) để điều khiển các thiết bị được gắn trên Raspberry Pi.

Hệ thống được thử nghiệm cùng với một số loại thiết bị như: cảm biến nhiệt độ, cảm biến ánh sáng, cảm biến vật thể chuyển động, cảm biến khí gas, bóng đèn led, còi hú. Theo đánh giá của nhóm, hệ thống có thể xử lý tốt trong một số trường họp kịch bản đơn giản, tuy nhiên vẫn theo sát nhu cầu thực tế. Trong tương lai, hệ thống có thể tiếp tục được mở rộng. Nhóm hi vọng đề tài này có thể cung cấp những thông tin hữu ích cho các nghiên cứu liên quan sau này.

# Giới thiệu

## tổng quan

Trong khoảng những năm trở lại đây, chắc hẳn khái niệm IoT (Internet of Things) đã không còn xa lạ và mới mẻ với chúng ta. Ý tưởng về một thế giới mà mọi thứ trong cuộc sống được kết nối với Internet để truyền tải, trao đổi dữ liệu, từ đó người dùng có thể tương tác, điều khiển và kiểm soát mọi hoạt động trong cuộc sống thông qua những thiết bị thông minh như điện thoại hoặc máy tính bảng chỉ qua vài cái chạm tay đơn giản.

Hãy tưởng tượng một ngôi nhà trong đó có thể kết nối và tự động hóa một số thiết bị điện tử, điện gia dụng, như đèn chiếu sáng, còi báo động,… hay đến thiết bị an ninh như camera chả hạn. Ta sẽ dễ dàng kiểm soát hoạt động của ngôi nhà ngay cả khi vắng mặt từ bất cứ đâu với máy tính hay smartphone, giảm thiểu những rủi ro ngoài ý muốn. Những kịch bản tự động hóa hoạt động của các thiết bị theo ngữ cảnh đem lại tiện nghi, thoải mái và an toàn cho cuộc sống [7].

Theo Gartner, IoT dự kiến sẽ có khoảng 26 tỷ thiết bị vào năm 2020 [8]. Tuy nhiên ở thời điểm hiện tại, IoT vẫn còn là khái niệm khá mới mẻ đối với các quốc gia đang trên đà phát triển như Việt Nam ta. Để ngôi nhà của mình sở hữu những tính năng thông minh kể trên, chắc hẳn chi phí cũng không hề nhỏ. Và đúng như thế, chi phí là một trong những rào cản lớn với sự phát triển IoT ở các nước đang phát triển, trong đó có cả Việt Nam.

## Nhà thông minh với raspberry pi

Có rất nhiều tùy chọn trong việc tiếp cận với IoT nhưng không có gì tốt hơn ngoài những trải nghiệm thực tế. Một trong những nển tảng quan trọng cho việc học IoT là mạch tính toán nhỏ và đơn giản, nhắm đến khả năng tạo một chiếc máy tính nhỏ gọn, giá rẻ phục vụ cho công tác học tập, nghiên cứu và thử nghiệm. Raspberry Pi không chỉ là một nền tảng phần cứng thú vị để ứng dụng cho các dự án IoT mà còn là công cụ giúp các nhà phát triển học hỏi và hoàn thiện kĩ năng Internet kết nối vạn vật. Một trong những ứng dụng tốt nhất của Raspberry Pi chính là trở thành trung tâm điều khiển dành cho các thiết bị sử dụng nguồn điện áp thấp.

## Mục tiêu và phạm vi đề tài

Từ những nhu cầu thực tế phát sinh, nhóm mong muốn được thực hiện một hệ thống nhà thông minh được quản lý tự động bởi các kịch bản định sẵn, tương tác với ứng dụng nền tảng web hay ứng dụng thiết bị di động. Trong đó, Raspberry Pi sẽ đóng vai trò là “bộ não” trung tâm, quản lý thiết bị trong nhà, đồng thời cũng là nơi vận hành, xử lý các kịch bản người dùng.

Mục tiêu của đề tài trong việc xây dựng hệ thống này đó là:

* Có giao diện ứng dụng nền tảng web hoặc di động để người dùng thao tác, quản lý ngôi nhà của riêng mình.
* Xây dựng hệ thống (phía back-end) giúp cho việc lưu trữ, xử lý, vận hành các kịch bản người dùng định nghĩa trên ứng dụng.
* Các thao tác, điều khiển các thiết bị gắn trên Raspberry Pi hoạt động theo ý muốn của kịch bản người dùng đã định.

Vì giới hạn thời gian, hệ thống chưa thực sự hoàn thiện và tối ưu. Tuy nhiên, hệ thống vẫn có thể hoạt động tốt trên một số kịch bản đáp ứng được với thực tế và có độ phức tạp thấp. Và ở đề tài này, nhóm chỉ tập trung vào phát triển, xây dựng ứng dụng nền tảng Web, cũng như ứng dụng trên thiết bị di động, các thiết bị phần cứng chỉ là hỗ trợ, phục vụ cho thí nghiệm và đánh giá hệ thống.

## cấu trúc luận văn

Toàn bộ nội dung luận văn được nhóm trình bày trong bảy chương. Các chương này lần lượt nêu lên những kiến thức cần thiết, chi tiết cách hiện thực để xây dựng và hoàn thiện hệ thống. Ở chương cuối, nhóm đưa ra kết quả tổng kết về những vấn đề đã được giải quyết ở đề tài này, song song đó là những hạn chế còn tồn đọng cũng như là hướng phát triển trong tương lai.

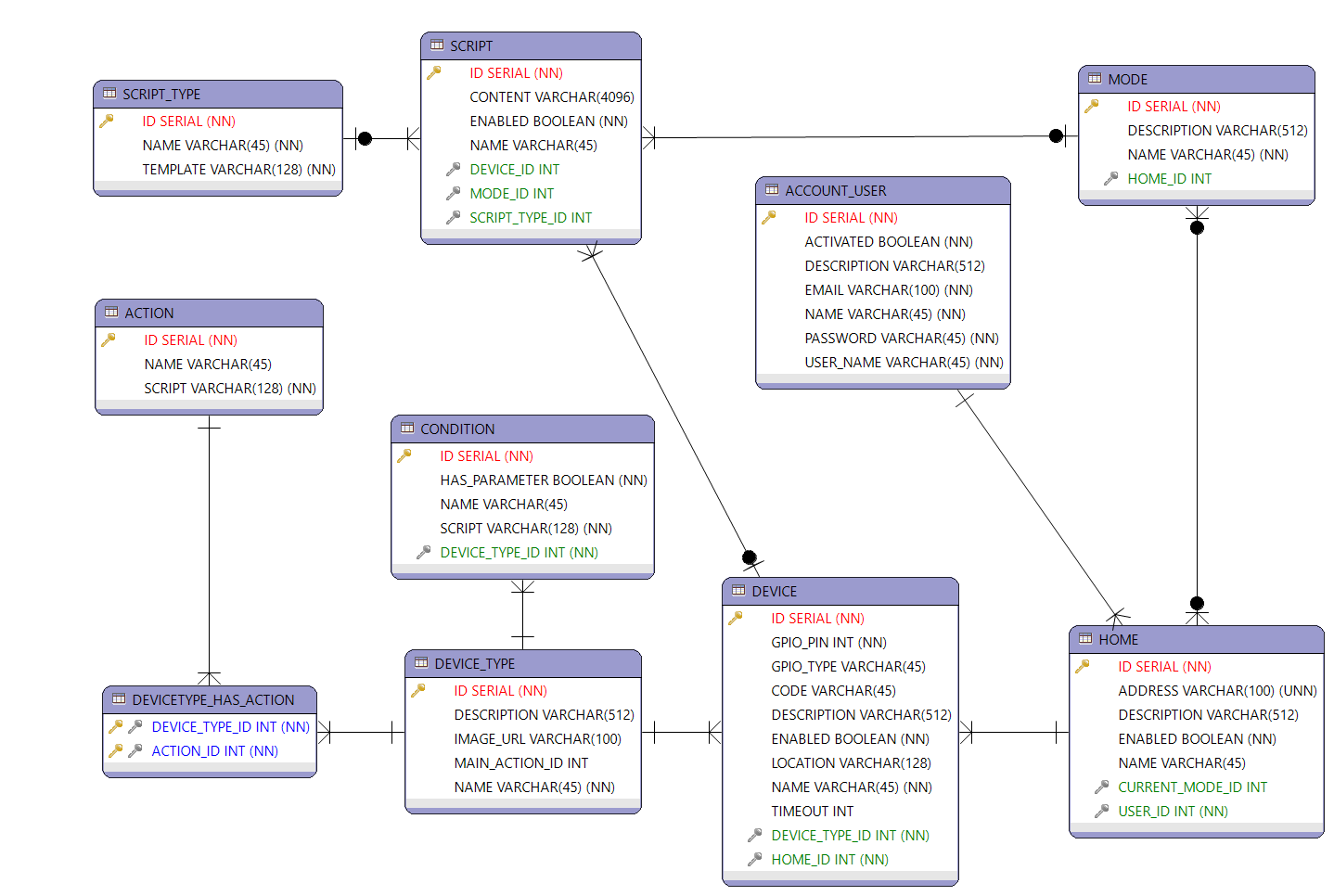
Đầu tiên ta đến với chương **Tổng quan hệ thống**, nguời đọc sẽ có cái nhìn toàn cảnh về lý do nhóm tiến hành thực hiện đề tài, vai trò và vị trí của đề tài trong xu thế phát triển nhà thông minh ở Việt Nam cũng như phạm vi của luận án này. Kế đến chương **Phân tích và thiết kế hệ thống** sẽ nêu lên những yêu cầu được đặt ra cho hệ thống, từ đó nhóm đề xuất những phương pháp thiết kể phù hợp. Sau khi có được cái nhìn tổng thể về hệ thống, chương **Tổng quan về hệ thống back-end** giúp người đọc hiểu thêm về cách xây dựng, tổ chức, thiết kế ở phía back-end. Đồng thời các công nghệ sử dụng để xây dụng back-end cũng sẽ được giới thiệu. Đồng thời để hiểu thêm về cách mà back-end tương tác với Raspberry Pi, cũng như việc điều khiển các thiết bị gắn trên nó, ta sẽ đến với chương **Module điều khiển thiết bị.** Qua đó, một dịch vụ (service) được xây dựng để back-end có thể tương tác được với các thiết bị. Tiếp đến chương **Tổng quan giao diện ứng dụng** sẽ giới thiệu cho ta về mặt giao diện ứng dụng của hệ thống đã thiết kế, cũng như cách tổ chức và cách sử dụng cho việc quản lý nhà, chế độ, thiết bị và kịch bản người dùng. Các công nghệ sử dụng để xây dựng giao diện cũng sẽ được giới thiệu tại đây. Sauk hi trình bày tổng quan hệ thống, chúng ta đi đến với chi tiết hiện thực back-end, giao diện ứng dụng, module điều khiển thiết bị. Những thông tin này sẽ được trình bày rõ trong chương kế tiếp **Hiện thực và đánh giá**. Cuối cùng, chương **Tổng kết** sẽ tóm tắt lại các kết quả đã đạt được sau quá trình thực hiện luận án, cũng như hạn chế còn chưa giải quyết, từ đó đề xuất hướng phát triển mở rộng trong tương lai.

# Thiết kế và phân tích

Phần này chúng ta sẽ thảo luận về thiết kế cơ sở dữ liệu (database) cho ứng dụng. Tiếp đến, ta sẽ đi vào giới thiệu tổng quan về các công nghệ mà nhóm đã tìm kiếm và ứng dụng trong việc xây dựng hệ thống này. Cuối cùng, dựa trên những phân tích về yêu cầu hệ thống, nhóm đề xuất và thiết kế ra mô hình kiến trúc của back-end để đáp ứng các nhu cầu đó.

## DB Design

Mục tiêu của ứng dụng đó là tạo ra và lưu trữ các kịch bản của người dùng. Do đó, việc thiết kế database đóng một vai trò quan trọng trong việc xây dựng ứng dụng này. Ở phần thiết kế database, nhóm sử dụng PostgreSQL để hiện thực trên Raspberry Pi. Mô hình quan hệ giữa các bảng trong phần thiết kế database được trình bày như ở hình 1.



Hình 1 Tổng quan mô hình cơ sở dữ liệu cho ứng dụng

Trở lại với yêu cầu được đặt ra từ hệ thống, người dùng có thể có nhiều ngôi nhà. Trong mỗi ngôi nhà sẽ điều khiển nhiều thiết bị khác nhau thuộc nhiều loại khác nhau. Mỗi thiết bị có thể có nhiều kịch bản để tự động hóa chúng. Kịch bản được đặc tả theo dạng **<điều kiện – hành động>**, nghĩa là dưới điều kiện nào đó do người dùng quyết định thì sẽ có những hành động tương ứng xảy ra. Những kịch bản sẽ hoặc thuộc dạng đơn giản (bao gồm 1 điều kiện, 1 hành động), hoặc dạng phức tạp mà người dùng có thể tự tạo theo ý muốn riêng của mình. Hơn nữa, với mỗi ngôi nhà có thể có nhiều chế độ quản lý nhưng tại 1 thời điểm chỉ có 1 chế độ được kích hoạt, ví dụ như chế độ đi vắng thì sẽ có những kịch bản riêng, còn với chế độ ở nhà sẽ là bộ kịch bản khác nhằm giúp người dùng tiện lợi trong việc quản lý căn nhà của mình ở nhiều hoàn cảnh khác nhau.

Với cách thiết kế như trên, hệ thống có những đặc điểm:

* Tính thích ứng cao khi có thay đổi yêu cầu.
* Đơn giản trong việc bảo trì và cập nhật (ví dụ như thêm thiết bị hay loại thiết bị mới).

Chi tiết về chức năng các bảng trong thiết kế được mô tả ở bảng 1. Các bảng quan trọng trong hệ thống có thể kể đến là Account\_User, Home, Mode, Device, Script dùng để chứa dữ liệu của người dùng. Còn những bảng còn lại chủ yếu chứa dữ liệu của hệ thống (master data).

Bảng 1. Chi tiết về chức năng các bảng được thiết kế trong database ứng dụng

|  |  |  |
| --- | --- | --- |
| STT | Tên bảng | Chức năng |
| 1 | Account\_User | Thông tin liên quan đến tài khoản người dùng, đã kích hoạt hay chưa, một vài thông tin cá nhân cơ bản |
| 2 | Home | Thông tin liên quan đến nhà , có đang “active” hay không, nhà đang ở chế độ nào |
| 3 | Mode | Thông tin liên quan chế độ được người dùng định nghĩa cho ngôi nhà của mình |
| 4 | Device | Thông tin các thiết bị cho từng nhà mà hệ thống quản lý như cổng GPIO, có đang “active” không v.v. |
| 5 | Device\_Type | Loại thiết bị hệ thống có thể quản lý như đèn , còi , cảm biến v.v. |
| 6 | Script | Quản lý các thông tin liên quan đến kịch bản, kịch bản người dùng được lưu xuống theo 1 cú pháp định sẵn |
| 7 | Script\_Type | Phân loại kịch bản đơn giản, được cung cấp sẵn hay phức tạp |
| 8 | Condition | Những điều kiện hợp lệ gắn với từng loại thiết bị để sử dụng khi định nghĩa kịch bản |
| 9 | Action | Những hành động hợp lệ gắn với từng loại thiết bị để sử dụng khi định nghĩa kịch bản |
| 10 | Device\_Has\_Action | Mô tả mối quan hệ giữa loại thiết bị và hành động của chúng |

## Thiết kế và kiến trúc hệ thống back-end

Hệ thống back-end được hiện thực hoàn toàn dựa trên ngôn ngữ Java và tận dụng sức mạnh từ Spring, một trong những framework được sử dụng nhiều nhất trong Java EE framework.

Spring có thể giúp chúng ta xây dựng ứng dụng một cách nhanh chóng, tốn ít thời gian để làm quen. Spring cho phép các thư viện bên ngoài (third party service) tích hợp vào ứng dụng dễ dàng. Ví dụ như ứng dụng ta có thể kết nối đến nhiều loại database khác nhau với chi phí thay đổi rất thấp hay 1 ứng dụng Spring MVC có thể chuyển thành dịch vụ cung cấp tài nguyên thông qua REST API. Một điểm cộng nữa cho Spring đó là các công nghệ front-end đều có thể làm việc cùng với nó. Với sự phổ biến của Spring trong cộng đồng phát triển ứng dụng Java nền web cùng nhiều đặc điểm nổi trội so với các frameworks khác như: Struts, Vaadin,… nhóm đã chọn Spring để phát triển hệ thống back-end này. Phần 4.2.1 sẽ cho ta cái nhìn tỗng quan về Spring framework, cùng những tính năng, ưu điểm của nó.

### Spring framework

Hai khái niệm chính của Spring framework core là "Dependency Injection - DI" và "Aspect Oriented Programming - AOP". Spring framework được sử dụng như là ứng dụng java cơ bản để đạt được kỹ thuật "loose coupling" giữa các components khác nhau bằng cách sử dụng kỹ thuật DI và hỗ trợ việc thực hiện chéo những task vụ như logging, authentication,... theo kỹ thuật AOP [1][2].

Spring framework cung cấp khá nhiều tính năng khác và số lượng lớn các module cho các mục đích cụ thể, ví dụ như web có Spring MVC, hỗ trợ security có Spring Security, tương tác với datababse có Spring JDBC, và nhiều thứ khác nữa. Ngoài ra, nó còn là một dự án open source với rất nhiều cộng đồng sử dụng, tài liệu tham khảo.



Hình 2 Tổng quan về kiến trúc Spring Framework

Hình 2 cho thấy kiến trúc tổng quan của Spring framework với nhiều module thiết kế tách biệt phục vụ cho nhiều mục đích khác nhau. Một số tính năng chính cũng như ưu điểm có thể kể đến của Spring:

* Dependency Injection hoặc Inversion of Control được sử dụng để giúp các component tách rời, độc lập với nhau. Spring container sẽ giúp gắn kết những components này lại với nhau theo đặc tả business.
* Spring MVC framework được sử dụng cho phát triển ứng dụng web rất dễ dàng với việc hỗ trợ rất tốt các tính năng web services, json,... (như RESTful web service framework).
* Hỗ trợ quản lý transaction, các thao tác với JDBC, tải file, xử lý lỗi,... rất dễ dàng bằng cách cấu hình được rút gọn, thay vào đó là sử dụng annotation.
* Làm giảm đi khối lượng code rất nhiều, chẳng hạn như việc khởi tạo đối tượng, open/close các resources,...

Với đặc trưng của ứng dụng hiện tại là dùng những kịch bản quản lý tự động thiết bị trong nhà, không cần real time (như ứng dụng chat, stream video,…) và chỉ gửi yêu cầu tại một số thời điểm, với tần suất nhỏ nên nhóm lựa chọn dùng RESTful web service, xây dựng những API cho client có thể giao tiếp và truy xuất tài nguyên từ server một cách thuận tiện. Hơn thế nữa, tận dụng sức mạnh của Spring framework, việc xây dựng các RESTFul web service trở nên dễ dàng và tiết kiệm công sức hơn rất nhiều.

### RESTful Web Service

Representational State Transfer (REST) là một kiểu kiến trúc được sử dụng trong việc giao tiếp giữa các máy tính (máy tính cá nhân và máy chủ của trang web) trong việc quản lý các tài nguyên trên internet. Hình 3 là sơ đồ thể hiện cách giao tiếp giữa client và máy chủ RESTFul web service thông qua giao thức HTTP [5].



Hình 3 Sơ đồ giao tiếp giữa client và máy chủ RESTFul Web Service

Những nguyên tắc cơ bản của một RESTFul Web Service:

* Sử dụng các phương thức HTTP một cách rõ ràng.
* Phi trạng thái.
* Hiển thị cấu trúc thư mục như URIs.

#### Sử dụng các phương thức HTTP một cách rõ ràng

Một đặc tính quan trọng của dịch Web service RESTful là sử dụng một cách rõ ràng các phương thức HTTP theo cách một giao thức được xác định bởi RFC 2616.

REST yêu cầu các nhà phát triển sử dụng phương thức HTTP một cách rõ ràng theo cách tương thích với giao thức chuẩn. Nguyên lý thiết kế REST cơ bản này thiết lập một ánh xạ 1-1 giữa các hành động tạo, đọc, cập nhật và xoá (CRUD) các quá trình vận hành và các phương thức HTTP. Theo cách ánh xạ này thì:

* Để tạo một tài nguyên trên máy chủ, ta cần sử dụng phương thức POST.
* Để truy xuất một tài nguyên, ta sử dụng GET.
* Để thay đổi trạng thái một tài nguyên hoặc để cập nhật nó, ta sử dụng PUT.
* Để huỷ bỏ hoặc xoá một tài nguyên, ta sử dụng DELETE [5].

#### Phi trạng thái

Một yêu cầu hoàn chỉnh, độc lập không đòi hỏi máy chủ thu thập được bất kỳ ngữ cảnh hoặc trạng thái của ứng dụng nào trong lúc xử lý yêu cầu. Một ứng dụng (hoặc máy khách) Web service REST chứa ở phần đầu và phần thân trang HTTP của một yêu cầu tất cả các tham số, ngữ cảnh và dữ liệu cần thiết bởi thành phần bên ngoài máy chủ để đưa ra một phản hồi. Phi trạng thái theo nghĩa này nâng cao tính hiệu quả của dịch vụ Web, đơn giản hoá thiết kế và sự thi hành của các thành phần của máy chủ vì khi máy chủ không có trạng thái sẽ huỷ bỏ nhu cầu để đồng bộ hoá các mảng dữ liệu với một ứng dụng bên ngoài [5].

#### Hiển thị cấu trúc thư mục như URIs

Các địa chỉ Web service REST nên có tính hiện thực theo nghĩa rằng chúng dễ dàng đối với người dùng. Có thể nghĩ rằng một địa chỉ đường dẫn như là giao diện tự đóng gói mà đòi hỏi ít lý giải hay tham chiếu, nếu có, đối với một nhà phát triển để hiểu nó nhắm đến điểm gì và phân phối tài nguyên liên quan. Cuối cùng, cấu trúc của địa chỉ nên rõ ràng, có thể đoán được và dễ hiểu.

Một cách để đạt được mức độ sử dụng này là xác định cấu trúc thư mục giống URIs. Loại URI này có thứ bậc, có điểm khởi nguồn tại một đường dẫn đơn giản, và có nhánh đi ra là các nhánh phụ thể hiện các vùng chính của dịch vụ. Theo định nghĩa này, một URI không chỉ là một chuỗi bị cắt không giới hạn, mà còn là một cây với các nhánh chính và nhánh dọc nối nhau tại các nút. Ví dụ:

http://www.myservice.org/discussion/topics/{topic}

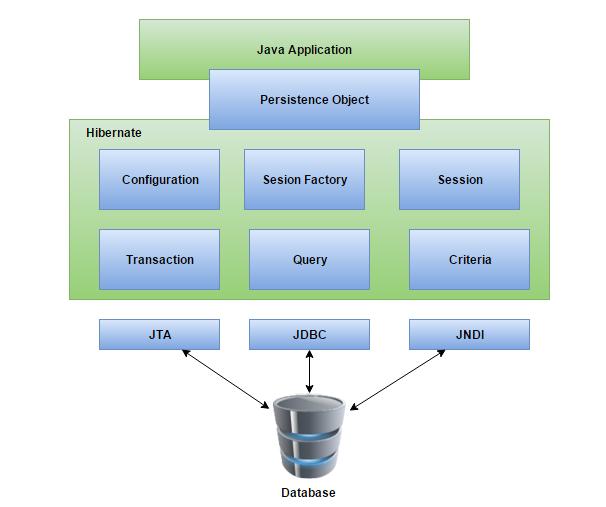
Các địa chỉ URIs nên giữ nguyên để khi tài nguyên thay đổi hoặc khi tiến hành thay đổi dịch vụ, đường liên kết cũng sẽ giữ nguyên. Việc này cho phép đánh dấu lại vị trí đang đọc. Nó cũng rất quan trọng vì mối liên quan giữa các tài nguyên mà được mã hoá trong các địa chỉ được giữ nguyên độc lập với các mối liên quan đại diện khi chúng được lưu trữ [5].

Với thiết kế database của nhóm, giữa các thực thể có mối quan hệ, liên kết với nhau chặt chẽ. Việc quản lý và thao tác với dữ liệu trực tiếp trên các database thường là công việc nặng nhọc và dễ phát sinh lỗi với các lập trình viên. Hibernate framework ra đời, giúp ta giải quyết những khó khăn trên. Ta có thể mô tả các đối tượng Java (Java objects) đại diện tương ứng cho mối quan hệ giữa các thực thể ở database và Hibernate sẽ làm công việc chuyển ánh xạ đó. Hibernate còn giúp quản lý cả kết nối (connection) với database, quản lý session,… Với những ưu điểm đó, nhóm chọn Hibernate để đơn giản hóa việc quản lý, truy xuất dữ liệu từ database. Hơn nữa, Hibernate cũng nhận được sự hỗ trợ từ Spring không nhỏ, tiết kiệm chi phí thời gian lẫn công sức trong việc xây dựng hệ thống. Phần 4.2.3 sẽ giới thiệu tổng quan về kiến trúc cũng như tính năng, điểm mạnh của Hibernate.

### Hibernate Framework

Hibernate Framework là một công cụ mã nguồn mở, dung lượng nhỏ (lightweight) và ORM (Object Relational Mapping) giúp đơn giản hóa việc phát triển ứng dụng Java để tương tác với cơ sở dữ liệu. Do Hibernate Framework là một ORM framework cho persistence layer nên khi phát triển ứng dụng, lập trình viên chỉ cần tập trung vào những layer khác (như tầng ứng dụng-business) mà không cần xem xét nhiều về persistence layer, dẫn đến tránh thao tác nhiều với database.

Cấu trúc Hibernate được thể hiện qua hình 7. Hibernate sử dụng nhiều API của Java như JDBC, Java Transaction, Java Naming and Directory Interface. JDBC cho phép bất kỳ cơ sở dữ liệu nào với một trình điều khiển JDBC đều được hỗ trợ bởi Hibernate.



Hình 4 Cấu trúc Hibernate

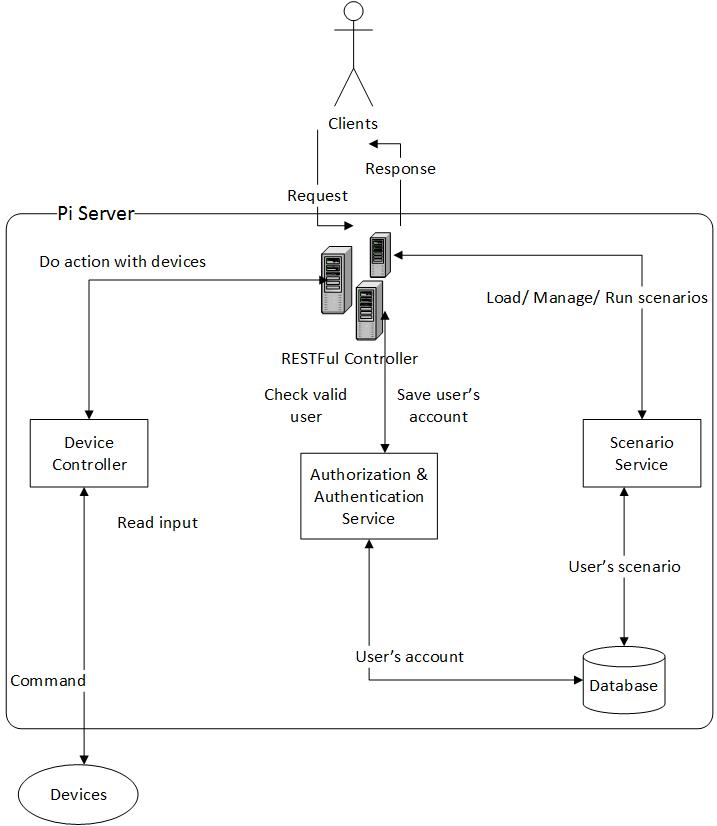
Dưới đây là một vài mô tả ngắn gọn về các thành phần trong cấu trúc Hibernate

* Cấu hình đối tượng (Configuration): nó đại diện cho một tập tin cấu hình, cung cấp thông tin về database muốn kết nối đến. Đây cũng là thành phần tạo ra sự kết nối giữa các Java class và các bảng cơ sở dữ liệu.
* SessionFactory: đối tượng này được tạo ra trong quá trình ứng dụng khởi động. Mỗi database sử dụng một tập tin cấu hình riêng biệt và chỉ có 1 đối tượng SessionFactory duy nhất. Đối tượng này có thể được truy cập đồng thời bởi nhiều thread nhưng vẫn đảm bảo tính an toàn dữ liệu (thread-safe).
* Session: đối tượng này được ứng dụng dùng để giao tiếp với database. Các đối tượng Session không nên giữ mở trong thời gian dài vì không an toàn (not thread-safe).
* Transaction: đối tượng này đại diện cho công việc nhỏ (ví dụ như cập nhật, lưu giá trị). Một session thường bao gồm nhiều transaction.
* Query: đối tượng truy vấn sử dụng SQL hoặc Hibernate Query Language (HQL) để lấy dữ liệu từ cơ sở dữ liệu.
* Criteria: kết hợp một hay nhiều tiêu chí để truy xuất một thực thể từ database thỏa mãn

Những lợi ích mà Hibernate đem lại

* Hibernate Framework là mã nguồn mở theo LGPL licence và dung lượng nhỏ.
* Đơn giản hóa việc truy nhập, kết nối
* Hibernate Framework cung cấp các thiết bị để tạo ra các bảng tự động
* Hỗ trợ hầu hết các loại database management system thông dụng hiện nay
* Cung cấp cơ chế tự động quản lý cache, cache cấp 1 và cấp 2, giúp tối ưu hóa việc truy xuất dữ liệu.

### Kiến trúc back-end



Hình 5 Kiến trúc hệ thống ở back-end

Hình 5 mô tả kiến trúc hệ thống ở back-end, bao gồm các module nhỏ như sau:

* Authorization & Authentication service: phục vụ mục đích bảo mật hệ thống, chỉ những người dùng hợp lệ (có tài khoản hợp lệ , có quyền truy xuất với tài nguyên yêu cầu ), quản lý token
* Scenario Service: quản lý trạng thái các kịch bản (có đang được kích hoạt chạy hay không) hay có thay đổi từ nhà hoặc thiết bị ảnh hưởng đến trạng thái kịch bản; quản lý việc thực thi các kịch bản một cách tự động; kiểm tra tính hợp lệ của kịch bản, xác định xem kịch bản có bị mâu thuẫn với chính nó hay với những kịch bản đã tồn tại hay không; cho phép truy xuất, tạo mới, cập nhật kịch bản.
* Device Controller: các kịch bản khi ở trong trạng thái kích hoạt, và thỏa 1 điều kiện định trước do người dùng định nghĩa thì nó sẽ thực thi những hành động tương ứng. Và module này đóng vai trò trung gian trong việc tương tác với thiết bị thật gắn trên Raspberry Pi ở hệ thống back-end, cụ thể là các kịch bản đang chạy.

Chi tiết về hiện thực các module này sẽ nằm trong mục **6.Hiện thực back-end**.

# hiện thực server back-end

## RESTFul Web Service - Cách thức giao tiếp giữa client và server

Như đã đề cập ở phần Thiết kế back-end, nhóm đã chọn dùng RESTful web service làm cách giao tiếp chính giữa client và server. Một tiện ích khi sử dụng Spring framework đó là nó có hỗ trợ sẵn **@RestController** (là 1 annotation hỗ trợ bởi Spring framework), đơn giản hóa việc tạo ra các RESTful web services.



Hình 6 Spring MVC RESTful Web services workflow

Hình 6 diễn tả luồng thực thi của Spring MVC REST, bao gồm các bước sau:

* Client gửi yêu cầu đến web service theo như một định dạng URI định sẵn và hợp lệ
* Yêu cầu đi qua Servlet Dispacher đầu tiên và nó sẽ tìm ra 1 controller phù hợp nhất để xử lý yêu cầu đó
* Yêu cầu sau khi được xử lý bởi controller sẽ được gửi trả về client dưới định dạng JSON [3].

Danh sách API có thể được tham khảo thêm ở mục **Phụ lục**.

## cách thức giao tiếp với database

Như đã đề cập ở mục thiết kế hệ thống back-end bên trên, nhóm sử dụng Hibernate framework để hỗ trợ cho các thao tác liên quan đến database. Hibernate cung cấp sẵn các hàm giúp truy xuất, lưu, cập nhật, xóa thực thể liên quan. Hình 7 thể hiện tầng thao tác dữ liệu (DAO) trong ứng dụng mà nhóm đã thiết kế và hiện thực.



Hình 7 Tổ chức của tầng truy xuất dữ liệu (DAO)

Thiết kế này giúp tăng khả năng tái sử dụng (reuse), cũng như việc quản lý, bảo trì, mở rộng hệ thống được dễ dàng hơn trong tương lai. Ý tưởng cơ bản là có 1 class BaseDao, được hiện thực đầy đủ các hàm **save, update, delete,...** còn các thực thể khác (như **Home, User, Mode, Device, Script**) thì thừa kế class BaseDao này và hiện thực thêm một số phương thức khác tùy theo nhu cầu.

Nếu những thao tác với database gây ra lỗi, dữ liệu sẽ được rollback ngay thời điểm đó (ví dụ như vi phạm ràng buộc, khóa ngoại,…), nhằm đảm bảo tính nhất quán của dữ liệu.

## giới thiệu về kịch bản (scenario)

Kịch bản là một bản phác thảo, diễn tả những hành vi mình mong muốn thiết bị trong nhà sẽ tự động thực hiện trong hoàn cảnh nhất định hay điều kiện nào đó thỏa mãn.

### Kịch bản người dùng

Kịch bản người dùng sử dụng ngôn ngữ tự nhiên để đặc tả. Lấy ví dụ như “Trong khoảng thời gian từ 18h tối đến 22h tối thì bật đèn ở hành lang lên”. Vế đầu “trong khoảng thời gian từ 18h tối đến 22h tối” đặc tả điều kiện, vế sau “bật đèn hành lang” nêu ra hành động mong muốn khi mà điều kiện trên thỏa mãn.

### Kịch bản lưu trữ (script)

Kịch bản lưu trữ có tác dụng như một bản thảo ngắn gọn của kịch bản người dùng, phục vụ mục đích lưu trữ dưới cơ sở dữ liệu là chính. Kịch bản lưu trữ được đặc tả bởi văn phạm riêng, được giới thiệu ở mục 6.4 sắp tới.

### Kịch bản hệ thống (scenario)

Kịch bản hệ thống là dạng kịch bản mà hệ thống có khả năng “đọc”, “hiểu” và xử lý. Hệ thống đọc kịch bản lưu trữ và dùng 1 cấu trúc dữ liệu riêng (sẽ được giới thiệu ở mục 6.5) để mô tả nó và xây dựng lên thành kịch bản hệ thống (lưu trên bộ nhớ máy tính).

## Văn phạm (grammar) dùng tạo ra kịch bản lưu trữ

Kịch bản của người dùng thường được đặc tả bởi ngôn ngữ tự nhiên. Với hệ thống hiện tại của nhóm, tính năng xử lý ngôn ngữ tự nhiên không được hỗ trợ, do đó vấn đề cấp thiết đặt ra đầu tiên và cũng không kém phần quan trọng, chính là đặc tả văn phạm cho kịch bản. Đặc tả văn phạm kịch bản nhằm một mặt giúp hệ thống có thể lưu trữ, mặt khác là phân định được kịch bản nào là hợp lệ và kịch bản nào không hợp lệ. Hơn thế nữa, văn phạm còn giúp hệ thống có thể “đọc”, “hiểu” và xử lý kịch bản. Tuy nhiên, công việc khó khăn là làm sao văn phạm đặc tả chính xác được kịch bản mà vẫn giữ đúng ý nghĩa của nó. Sau thời gian nghiên cứu, nhóm quyết định chọn BNF (Backus-Naur form), gồm những kí hiệu toán học để đặc tả văn phạm cho ngôn ngữ phi ngữ cảnh, thường dùng để xây dựng cú pháp các ngôn ngữ trong ngành máy tính, ví dụ như ngôn ngữ lập trình, tập lệnh... áp dụng vào việc xây dựng văn phạm kịch bản hệ thống.

Dưới đây là văn phạm mà nhóm dùng để mô tả những kịch bản lưu trữ, bao gồm từ khóa (đặt trong cặp ngoặc kép), kí hiệu, phép toán, cấu trúc (đặt trong <>):

<Scenario> ::= “[“ <ControlBlock> “]” | “[“ <Scenario> “,” <ControlBlock> “]”

| “[“ <SimpleAction> “]” | “[“ <Scenario> “,” <SimpleAction> “]”

<ControlBlock> ::= “[‘If’ ,“ <Condition> “,” <Action> “]”

| “[‘If’,” <Condition> “,” <Action> “,” <Action> “]”

| <FromToBlock>

<Condition> ::= “[“ <DeviceName> “,” <RelationalOperator> “,” <Value> “]”

<Action> ::= <Scenario>

<FromToBlock> ::= “[‘FromTo’,” **Time** “,” **Time** “,” <Action> “]”

<RelationalOperator> ::= <Equal> | <NotEqual>

| < GreaterThan > | <GreaterThanEqual>

| <LessThan> | <LessThanEqual>

<SimpleAction> ::= “[‘*TurnOn’,*” <DeviceName> “]“

| “[‘*TurnOff’,*” <DeviceName> “]”

| “[‘*Toggle’,*” <DeviceName> “]”

| “[‘*TakePicture’,*”<DeviceName> “]”

<DeviceName> ::= **String**

<Value> ::= **Long** | **Boolean**

<Equal> ::= “=”

<NotEqual> ::= “!=”

<GreaterThan> ::= “>”

<GreaterThanEqual> ::= “>=”

<LessThan> ::= “<”

<LessThanEqual> ::= “<=”

Nhóm đã chọn đặc tả văn phạm kịch bản với cú pháp nêu trên, nhằm mục đích:

* Có thể dễ dàng lưu trữ kịch bản
* Cú pháp tương tự như dạng JSON, nhóm có thể dùng parser JSON để phân tích và chuyển đổi thành kịch bản hệ thống dễ dàng hơn.

Tuy nhiên, với cú pháp trên cũng có mặt hạn chế với cú pháp như việc tạo ra kịch bản lưu trữ cần phải được xử lý cẩn thận. Nếu khâu đầu vào có sai sót thì toàn bộ khâu còn lại, như đọc và xử lý sẽ gặp vấn đề. Để khắc phục hạn chế này, nhóm đã xây dựng 1 module nhằm tạo ra kịch bản hệ thống theo cú pháp nhất định (sẽ được giới thiệu trong mục **6.8**).

Để dễ hình dung, sau đây là một ví dụ trong thực tế về 1 kịch bản mà ta đang muốn hệ thống đọc, hiểu và xử lý:

“Nếu cảm biến nhiệt phát hiện nhiệt độ nằm trong khoảng 40 đến 50 độ C thì bật còi hú ở phòng khách”

“Trong khoảng thời gian từ 18h tối đến 22h tối thì bật đèn ở hành lang lên”

Sử dụng văn phạm đã có, kịch bản lưu trữ dùng mô tả các kịch bản trên sẽ là:

“[ [‘If’, [‘*temperature\_sensor*’,’>’, 40], [ [‘If’, [‘*temperature\_sensor*’,’<’, 50], [ [‘TurnOn’, ‘buzzer\_living\_room’] ] ] ] ] ]”

“[ [ ‘FromTo’, ’18:00’, ’20:00’, [ [ ‘TurnOn’, ‘light\_lobby’ ] ] ] ]”

Với văn phạm đặc tả trên, hệ thống đã có thể phân định được kịch bản hợp lệ và không hợp lệ. Vì lý do thời gian nên nhóm chưa thể hỗ trợ nhiều dạng kịch bản hơn, nhưng việc mở rộng là hoàn toàn khả thi. Nhóm cũng đã liệt kê một số kịch bản thông dụng trên thực tế mà nhóm đã sưu tập và lặp nên (tham khảo thêm ở phần **Phụ lục**).

## Cấu trúc dữ liệu xây dựng kịch bản hệ thống

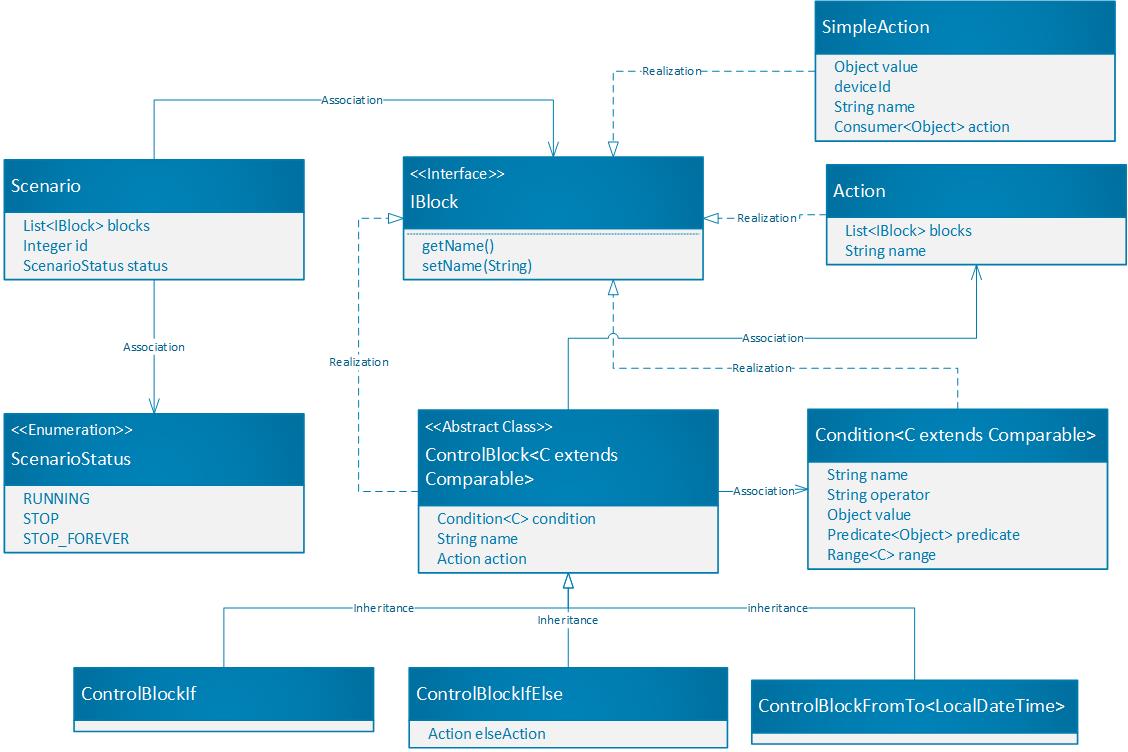
Kịch bản hệ thống sẽ được lưu trữ trong bộ nhớ hệ thống khi chạy. Hình 8 cho ta thấy cấu trúc dữ liệu của kịch bản hệ thống (scenario) bao gồm

* Thông tin trạng thái của nó, được định nghĩa bằng 1 enum ScenarioStatus là
  + RUNNING: đang được thực thi
  + STOP: đã dừng lại (tạm thời và sau đó có thể được khởi động chạy lại)
  + STOP\_FOREVER: dừng lại vĩnh viễn, trạng thái này dùng để kiểm soát những kịch bản trong hàng chờ được gỡ bỏ
* Tập hợp các khối (block) hiện thực từ interface IBlock, các khối hợp lệ và được phép nằm trong kịch bản sẽ là
  + ControlBlock (sẽ chứa tham khảo đến Condition)
  + Action
  + SimpleAction

### Class SimpleAction

Class SimpleAction đại diện cho những hành động nhỏ nhất, đơn giản nhất, có thể tương tác với thiết bị. Lấy ví dụ như hành động “Tắt đèn 1” hay “Bật còi hú ở bếp” là những hành động đơn giản.

Thuộc tính “action” có kiểu Consumer<Object> nhằm để mình truyền vào 1 biểu thức dạng Lamda Expression, cụ thể nó là 1 hàm gọi tới Device controller , tương tác với thiết bị thật. Chi tiết về phần hiện thực sẽ được giải thích trong mục Module Script Creator.



Hình 8 Cấu trúc dữ liệu kịch bản hệ thống

### Class Action

Class Action là tập hợp các hành động mong muốn thực hiện, có thể xem như là 1 kịch bản hệ thống “con” trong kịch bản hệ thống lớn bên ngoài và chứa nó. Vì thế mà cấu trúc của nó gần tương tự như là 1 kịch bản hệ thống vậy.

### Class Condition

Class Condition dại diện cho 1 điều kiện nào đó. Lấy 1 ví dụ như “Nhiệt độ lớn hơn 40 độ C thì bật còi hú 1”. Phân tích kịch bản trên thì điều kiện ở đây chính là “Nhiệt độ lớn hơn 40 độ C”. Khi đó, thuộc tính “operator” sẽ có giá trị “>”, thuộc tính “value” có giá trị “40”. Một thuộc tính đặc biệt là range, kiểu Range<C> mang giá trị dãy số (40, +infinity), phục vụ cho mục đích kiểm tra kịch bản mâu thuẫn (giới thiệu ở phần module ScenarioConflictValidation)

Thuộc tính predicate có kiểu Predicate<Object> sẽ chứa 1 biểu thức Lamda Expression, tương ứng với điều kiện mà mình mô tả trong kịch bản. Chi tiết về hiện thực sẽ nằm trong mục Module Script Creator.

### Class ControlBlock và các class kế thừa từ nó

Class ControlBlock được sinh ra nhằm mục địch kết nối 2 class: Action và Condition lại với nhau. Class ControlBlock này có 3 class kế thừa từ nó là

* ControlBlockIf: ngữ nghĩa cơ bản là “Nếu điều kiện A xảy ra thì mình sẽ thực hiện hành động B”.
* ControlBlockIfElse: nó mang nghĩa “Nếu điều kiện A xảy ra thì mình sẽ thực hiện hành động B còn không thì sẽ thực hiện hành động C”. Vì thế mà nó có thêm 1 thuộc tính là “elseAction”, diễn tả hành động mong muốn được thực thi nếu mệnh đề điều kiện không thỏa mãn.
* ControlBlockFromTo: đây là khối điều khiển chuyên dụng cho các điều kiện liên quan tới thời gian, hiểu là “Trong khoảng thời gian từ X đến Y thì mình sẽ thực hiện hành động A”. Khoảng thời gian này có thể là trong cùng một ngày, hay kéo dài từ ngày này sang ngày tới. Ví dụ “Từ 18h00 đến 22h00 thì tắt đèn 1” hay “Từ 19h00 đến 01h00 sáng hôm sau thì bật đèn hành lang”.

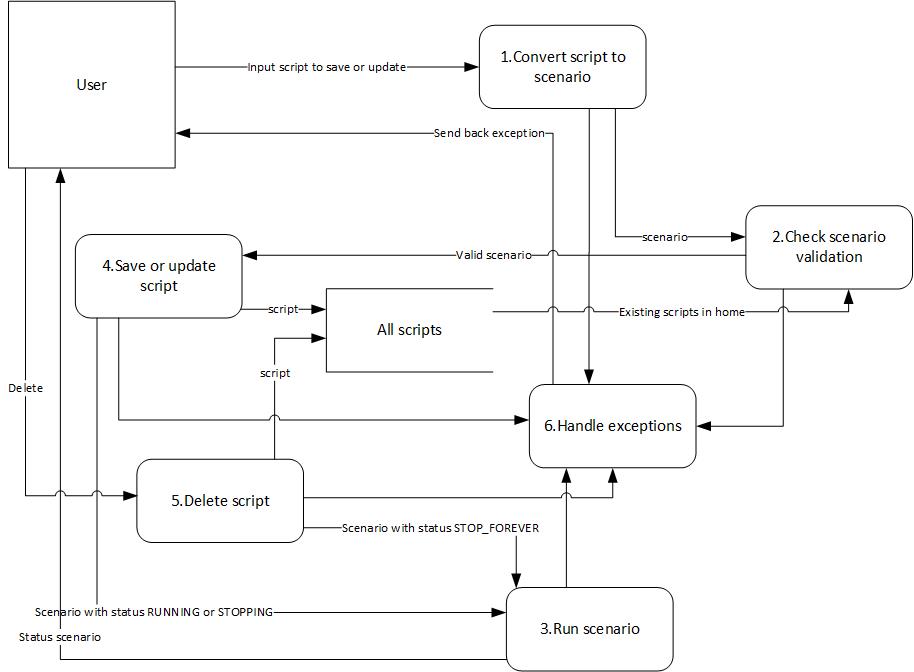
Nếu như sau này ứng dụng mở rộng và hỗ trợ thêm nhiều khối điều khiển khác, ta có thể kế thừa từ class ControlBlock này và tiếp tục hiện thực nó một cách dễ dàng.

## Sơ đồ mô tả luồng dữ liệu trong hệ thống khi thêm, sửa, xóa kịch bản người dùng

Một trong những chức năng chính của hệ thống đó là giúp người dùng quản lý thiết bị trong nhà một cách tự động, theo kịch bản được định sẵn. Hình 9 mô tả sơ đồ luồng dữ liệu của hệ thống back-end khi mà người dùng thêm, sửa hay xóa một kịch bản.

Các tiến trình xử lý lần lượt theo các bước sau:

* Khi người dùng thêm, sửa 1 kịch bản qua giao diện, phía client sẽ gửi về cho server thông tin kịch bản đó dưới dạng kịch bản lưu trữ.
* Kịch bản lưu trữ được chuyển đổi thành kịch bản hệ thống, được lưu trong bộ nhớ dưới 1 cấu trúc dữ liệu riêng bởi module **1.Convert script to scenario** (sẽ đi vào chi tiết ở mục module scenario creator).
* Kịch bản hệ thống trên được kiểm tra tính hợp lệ, cũng như đảm bảo không mâu thuẫn (conflict) với những kịch bản khác hiện có trong ngôi nhà ấy bởi module **2.Check scenario validation** (sẽ đi vào chi tiết ở mục module Scenario Validation). Nếu như kịch bản không hợp lệ, thông báo lỗi sẽ được gửi trả về ngay cho người dùng.
* Sau khi kịch bản được kiểm tra tính hợp lệ, nó sẽ được lưu hay cập nhật vào cơ sở dữ liệu (mình sẽ lưu ở dạng kịch bản lưu trữ).
* Việc cuối cùng là hệ thống sẽ “đọc” và xử lý kịch bản ấy. Việc quản lý đó được module **3.Run scenario** đảm nhiệm (sẽ đi vào chi tiết ở mục module Scenario Runner). Trạng thái của kịch bản mới sẽ được quyết định bởi người dùng, nó có thể là chưa được chạy (STOP) hay được chạy (RUNNING). Thông thường khi 1 kịch bản mới được thêm vào hệ thống sẽ mang trạng thái được chạy.
* Trong trường hợp mà người dùng muốn xóa một kịch bản, hệ thống back-end kiểm tra dưới cơ sở dữ liệu và gửi truy vấn xóa. Sau đó, hệ thống cập nhật lại trạng thái của kịch bản ấy là đã dừng vĩnh viễn (STOP\_FOREVER) và từ đó kịch bản này không còn có ảnh hưởng tới các thiết bị trong nhà nữa.



Hình 9 Sơ đồ luồng dữ liệu khi thêm/sửa/xóa kịch bản

## module parser chuyển đổi kịch bản lưu trữ thành kịch bản hệ thống (scenario creator)

Kịch bản lưu trữ chỉ phục vụ cho mục đích lưu dữ liệu. Khi hệ thống muốn đọc, hiểu, xử lý và chạy những kịch bản ấy thì nó cần được chuyển sang dạng kịch bản hệ thống. Để phục vụ cho nhu cầu đó, module scenario creator được sinh ra. Đầu vào của nó là một kịch bản lưu trữ và đầu ra sẽ là kịch bản hệ thống. Kịch bản hệ thống sẽ được dùng trong phần kiểm tra tính hợp lệ và chạy kịch bản như đã đề cập ở mục tổng quan phía trên.

Kịch bản lưu trữ được thể hiện dưới dạng mảng JSON. Sở dĩ nhóm chọn cách thể hiện này là vì thư viện parse JSON đã có sẵn, việc tận dụng nó sẽ giúp giảm thời gian hiện thực hệ thống này. Mỗi một mảng JSON tương đương với 1 khối: điều kiện (condition) hay hành động (action) hay hành động đơn giản (simple action). Và cũng trong module này, nhóm đã áp dụng nhiều kĩ thuật mang tên Lamda Expression được hỗ trợ trong Java 8 để hiện thực việc xử lý 1 điều kiện hay là 1 hành động đơn giản tương tác với thiết bị thật.

Khi xử lý 1 điều kiện, nhóm dùng Predicate class để hiện thực nó. Predicate như một mệnh đề luận lý trả về giá trị hoặc đúng, hoặc sai, rất phù hợp cho việc kiểm tra điều kiện. Một vài điều kiện cơ bản mà hệ thống hiện có đó là: kiểm tra xem đèn có được bật hay không, kiểm tra ngày hay đêm từ cảm biến ánh sáng, thực hiện so sánh nhiệt độ thu được từ cảm biến nhiệt với một giá trị người dùng mong muốn… Mỗi predicate sẽ nhận việc gọi đến device controller và kiểm tra giá trị trả về từ controller ấy. Nói cách khác thì khi hệ thống kiểm tra một điều kiện nào đó thì sẽ chỉ cần kiểm tra thuộc tính predicate bên trong cấu trúc kịch bản mà không phải gọi riêng lẽ một device controller nào. Nó giúp cho việc code được ngắn gọn và “sạch sẽ” hơn. Tương tự như việc xử lý 1 hành động đơn giản, nó có 1 thuộc tính “action” kiểu Consumer. Khi mà muốn thực hiện hành động, mình chỉ cần gọi phương thức trên biến “action” đó là đủ.

## module hỗ trợ xây dựng kịch bản tùy ý (script builder)

Hệ thống không chỉ cung cấp cho người dùng những kịch bản theo mẫu định sẵn, mà còn hỗ trợ cả kịch bản tùy biến. Để dễ dàng hơn cho người dùng định nghĩa 1 kịch bản tùy biến, nhóm tạo ra 1 module là Script Builder. Cú pháp của kịch bản người dùng tùy biến gần giống như ngôn ngữ tự nhiên, nhưng lại có khả năng chuyển đổi sang dạng kịch bản lưu trữ dễ dàng. Dưới đây là văn phạm mô tả cú pháp của kịch bản tùy biến.

<Scenario> ::= <ControlBlock> | <Scenario> “.” <ControlBlock>

| <SimpleAction> | <Scenario> “.” <SimpleAction>

<ControlBlock> ::= “*If”* <Condition> “.” <Action> “.*endIf()*”

| “*If”* <Condition> “*.”* <Action> “.*Else().”* <Action> “.*endIf()*”

| <FromToBlock>

<Condition> ::= “(“ <DeviceName> “,” <RelationalOperator> “,” <Value> “)”

<Action> ::= <Scenario>

<FromToBlock> ::= “*FromTo(“* **Time**  “*,”* **Time** “)*.”* <Action> “.*endFromTo()*”

<RelationalOperator> ::= <Equal> | <NotEqual>

| < GreaterThan > | <GreaterThanEqual>

| <LessThan> | <LessThanEqual>

<SimpleAction> ::= “action( ‘*TurnOn’,”* <DeviceName> “)”

| “action( ‘*TurnOff’,”* <DeviceName> “)”

| “action( ‘*Toggle’,”* <DeviceName> “)”

| “action( ‘*TakePicture,”* <DeviceName> “)”

<DeviceName> ::= **String**

<Value> ::= **Long** | **Boolean**

<Equal> ::= “=”

<NotEqual> ::= “!=”

<GreaterThan> ::= “>”

<GreaterThanEqual> ::= “>=”

<LessThan> ::= “<”

<LessThanEqual> ::= “<=”

Lấy ví dụ về 1 kịch bản người dùng là: “Nếu nhiệt độ thu được từ cảm biến nhiệt gần cửa sổ lớn hơn 40 độ thì bật đèn phòng và bật đèn nhà bếp”.

Kịch bản người dùng tùy biến được viết lại theo văn phạm trên sẽ là:

If( ‘temp\_sensor\_near\_window’, ‘>’ , ‘40’)

.action(‘TurnOn’, ‘light\_room’)

.action(‘TurnOn’, ‘light\_kitchen’)

.endIf()

Nhiệm vụ của client là sẽ mang kịch bản người dùng tùy biến trên gửi lên server, server sẽ dùng module ScriptBuilder để mà chuyển đổi sang kịch bản lưu trữ. Đi vào chi tiết module ScriptBuilder thì nó gồm 1 bộ biên dịch, nhằm thực thi một đoạn code được lưu dưới dạng chuỗi. Sở dĩ nhóm đề xuất văn phạm, cú pháp trên cho kịch bản tùy biến vì nó chính là đoạn code thu nhỏ, sử dụng ngôn ngữ Java. Nhiệm vụ của bộ biên dịch là đọc kịch bản đó giống như đọc và thực thi 1 đoạn code Java. Kết quả trả về từ đoạn code (hay kịch bản tùy biến) sẽ là một kịch bản lưu trữ. Bằng cách này, nhóm có thể cung cấp cho người dùng những cú pháp khác linh hoạt hơn và tiện lợi hơn (syntatic sugar), giúp người dùng định nghĩa một kịch bản tùy biến dễ dàng và thuận tiện nhất.

Trước khi hiện thực module ScriptBuilder này, nhóm cũng đã tìm hiểu sơ qua các Rule Engine hỗ trợ xây dựng kịch bản. Theo tìm hiểu của nhóm, Rule Engine sẽ hỗ trợ cho ta định nghĩa ra các quy tắc cứng, đã được định nghĩa sẵn (thông thường là business rule) dành cho hệ thống trong một số trường hợp cụ thể nào đó. Khi ta muốn thay đổi các quy tắc ấy thì cần khởi chạy lại hệ thống để có hiệu lực. Chính vì thế mà Rule Engine không phù hợp với ứng dụng nhóm muốn phát triển. Có thể lý giải rằng các kịch bản người dùng đặt ra không phải là các business rule của hệ thống. Hơn nữa, những kịch bản ấy có tính linh hoạt, người dùng có thể cập nhật nội dung mới và nó tự động có hiệu lực ngay sau đó.

## Giới thiệu về tính hợp lệ của kịch bản

### Thế nào là kịch bản tự mâu thuẫn (self-conflict script)

Kịch bản tự mâu thuẫn là kịch bản chứa các điều kiện mâu thuẫn trong nội tại chính nó. Lấy ví dụ: Nếu như nhiệt độ thu được từ cảm biến nhiệt ở phòng khách lớn hơn 40 độ và bé hơn 30 độ thì bật đèn 1. Dễ dàng nhận thấy rằng 2 điều kiện “nhiệt độ lớn hơn 40 độ” và “nhiệt độ bé hơn 30 độ” mâu thuẫn lẫn nhau khi giá trị đó cùng thu thập được từ cùng 1 thiết bị. Vì thế, chúng ta có thể kết luận rằng kịch bản nêu trên là kịch bản tự mâu thuẫn. Nhưng nếu điều kiện nhiệt độ bé hơn 30 độ thu được từ 1 cảm biến nhiệt khác thì kịch bản trên không được gọi là tự mâu thuẫn.

### Thế nào là kịch bản mâu thuẫn (conflict) ?

Kịch bản này được gọi là mâu thuẫn với kịch bản kia nếu như cả 2 kịch bản có tồn tại 2 hành động đơn giản (simple action) trái ngược nhau (counter-action) nhưng điều kiện xảy ra hành động trên lại giống nhau. Hai hành động được gọi là trái ngược nhau nếu hành vi của chúng ngược với nhau và chúng cùng là hành động đơn giản (simple action). Lấy ví dụ, ta có 2 kịch bản sau

* Kịch bản 1: Nếu đèn 1 bật thì đèn 2 tắt.
* Kịch bản 1: Nếu đèn 1 bật thì đèn 2 bật.

Ta nhận thấy, 2 hành động “đèn 2 bật” và “đèn 2 tắt” là 2 hành động trái ngược nhau. Hơn nữa, điều kiện xảy ra hành động trái ngược nhau trên là “đèn 1 bật”. Có thể kết luận rằng 2 kịch bản trên là mâu thuẫn với nhau.

### Thế nào là kịch bản có khả năng mâu thuẫn (potential conflict)?

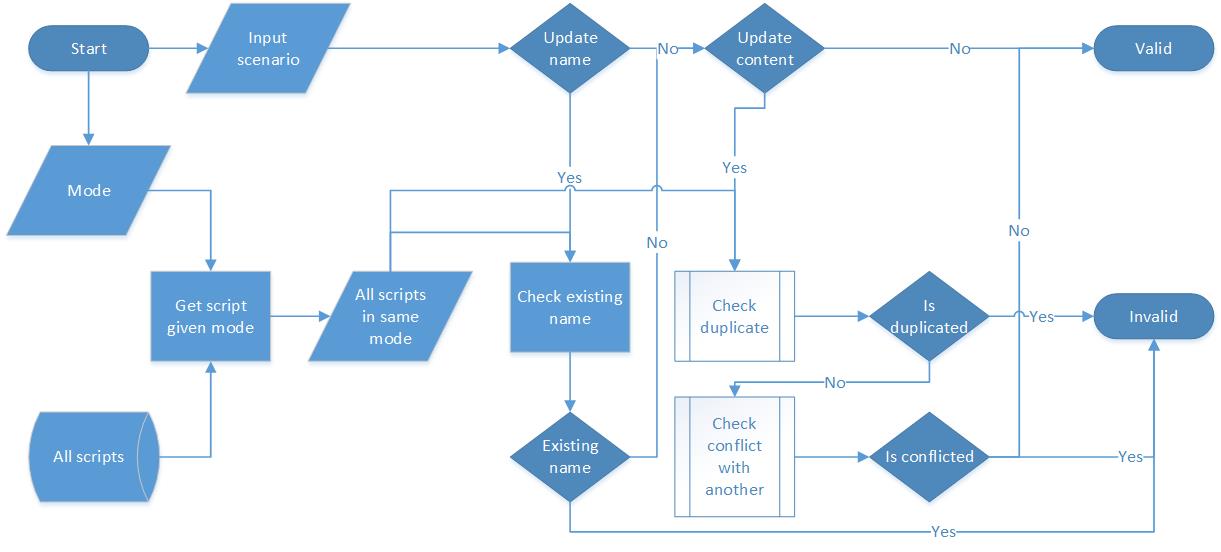
Hai kịch bản được gọi là có khả năng mâu thuẫn nếu như giữa 2 kịch bản ấy tồn tại ít nhất một mâu thuẫn (theo khái niệm từ mục 6.9.2) và mâu thuẫn đó chỉ xảy ra trong một số điều kiện nhất định. Sau đây là ví dụ về kịch bản có khả năng mâu thuẫn.

* Giả sử ta có kịch bản là “từ (12h40, 15h30) thì tắt đèn 1” và một kịch bản khác yêu cầu là bật đèn 1 nhưng có thời gian thực hiện không xác định (ví dụ: khi có gas, khi trời sáng, khi có người, khi nhiệt độ >, <,...). Hai kịch bản trên vẫn có khả năng mâu thuẫn vì giữa chúng tồn tại sự mâu thuẫn giữa hành động “bật” và “tắt” đèn 1 và thời gian thực hiện kịch bản thứ hai là không xác định, nó có thể là thời điểm nào đó trong ngày và cùng lúc điều kiện kịch bản 1 đang thực thi thì điều kiện kịch bản 2 cũng được thỏa mãn.
* Ví dụ như ta có kịch bản 1 là “Nếu nhiệt độ lớn hơn 35 độ và nếu đèn 1 tắt thì đèn 2 bật” và kịch bản 2 là “Nếu đèn 1 tắt thì đèn 2 tắt”. Ta nhận thấy ở kịch bản 1 có cặp <điều kiện, hành động> là <khi đèn 1 tắt, đèn 2 bật> mâu thuẫn với <khi đèn 1 tắt, đèn 2 tắt> ở kịch bản 2. Nhưng thực sự chỉ khi nhiệt độ lớn hơn 35 độ thì sự mâu thuẫn mới xảy ra, bình thường thì ta nói 2 kịch bản trên có khả năng mâu thuẫn mà thôi.
* Trường hợp điều kiện cùng khoảng giá trị cũng là một dạng có khả năng mâu thuẫn. Lấy 1 ví dụ như “nhiệt độ trong khoảng (30,40) độ C thì thực hiện bật đèn 1” và “nhiệt độ trong khoảng (35,45) độ C thì tắt đèn 1”. Ta thấy là 2 khoảng giá trị trên có trùng lắp lẫn nhau và 2 hành động lại đối nghịch nhau. Hai kịch bản trên có khả năng mâu thuẫn.
* Hoặc trùng khoảng thời gian: tương tự như khoảng giá trị trên, còn bây giờ là khoảng thời gian. Ví dụ “từ (12h40, 15h30) thì bật đèn 1” và “từ (15h, 16h) tắt đèn 1” cũng là những kịch bản có khả năng mâu thuẫn.

### Định nghĩa về kịch bản hợp lệ

Kịch bản hợp lệ là kịch bản không trùng tên hay trùng nội dung hoàn toàn với một trong những kịch bản, không tự mâu thuẫn với chính nó, cũng như không mâu thuẫn hay có khả năng mâu thuẫn với các kịch bản khác đã có ở cùng chế độ (mode) của ngôi nhà đang xét. Trong tương lai, nhóm đề xuất kịch bản có khả năng mâu thuẫn vẫn là kịch bản hợp lệ và chúng sẽ được quản lý bởi độ ưu tiên riêng biệt.

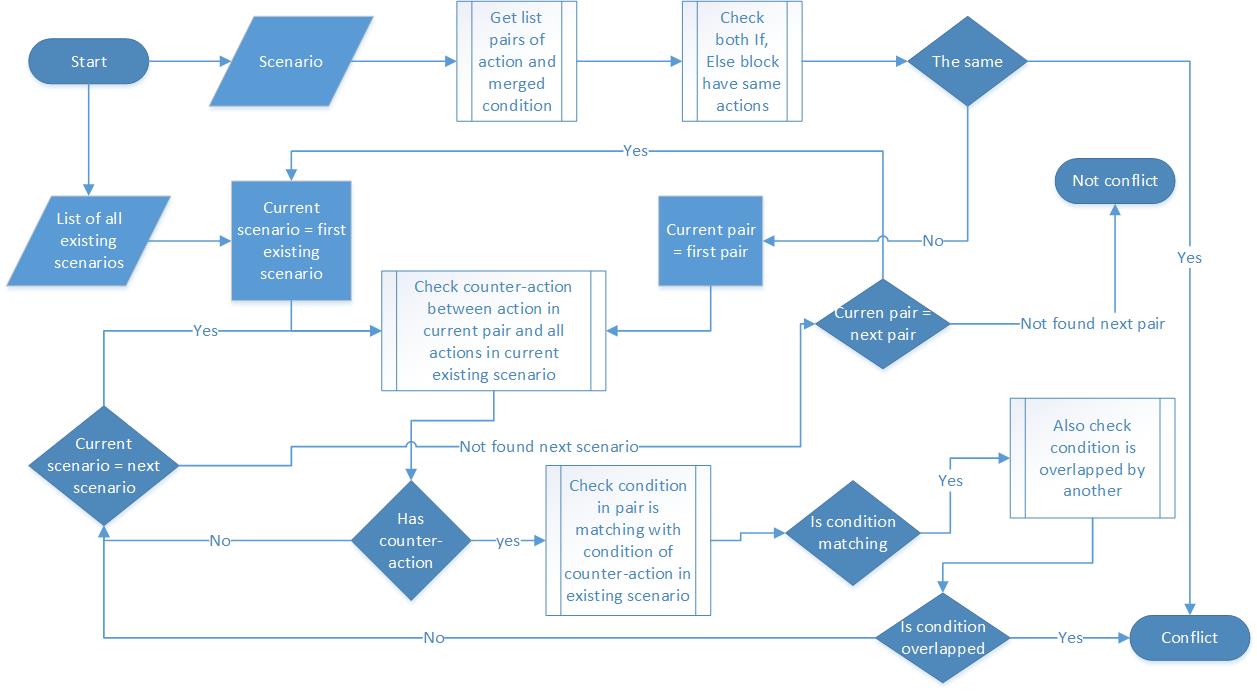
### Thuật toán kiểm tra kịch bản hợp lệ



Hình 10 Flowchart thể hiện cách kiểm tra kịch bản hợp lệ

Đầu vào thuật toán nhận một kịch bản cần kiểm tra tính hợp lệ (gọi là “input scenario”), còn đầu ra của thuật toán sẽ là giá trị luận lý đúng hoặc sai, tương ứng kịch bản hợp lệ hoặc không hợp lệ. Hình 10 thể hiện thuật toán kiểm tra tính hợp lệ của kịch bản. Dựa trên thông tin về chế độ (mode) khi kịch bản được thêm vào hệ thống thì ta sẽ lấy tất cả kịch bản trong cùng chế độ ấy lên từ cơ sở dữ liệu. Mục đích của công việc này là kiểm tra xem tên của kịch bản mới (nếu có) có trùng với những kịch bản có sẵn hay không. Nếu trùng tên xảy ra thì kết luận rằng kịch bản đầu vào không hợp lệ. Kế tiếp, ta cần kiểm tra rằng nội dung kịch bản có bị trùng với kịch bản nào có sẵn chưa (xét trùng về mặt ngữ nghĩa chứ không chỉ là về mặt chuỗi lưu trữ kịch bản có giống nhau hoàn toàn hay không). Cuối cùng, thuật toán kiểm tra kịch bản mâu thuẫn sẽ được thực thi (chi tiết thuật toán được mô tả ở mục sau). Nếu kịch bản đầu vào không xảy ra mâu thuẫn với các kịch bản sẵn có thì kết quả trả về từ thuật toán là hợp lệ, trường hợp khác sẽ là không hợp lệ.

### Thuật toán kiểm tra kịch bản mâu thuẫn



Hình 11 Flowchart thể hiện cách kiểm tra kịch bản mâu thuẫn

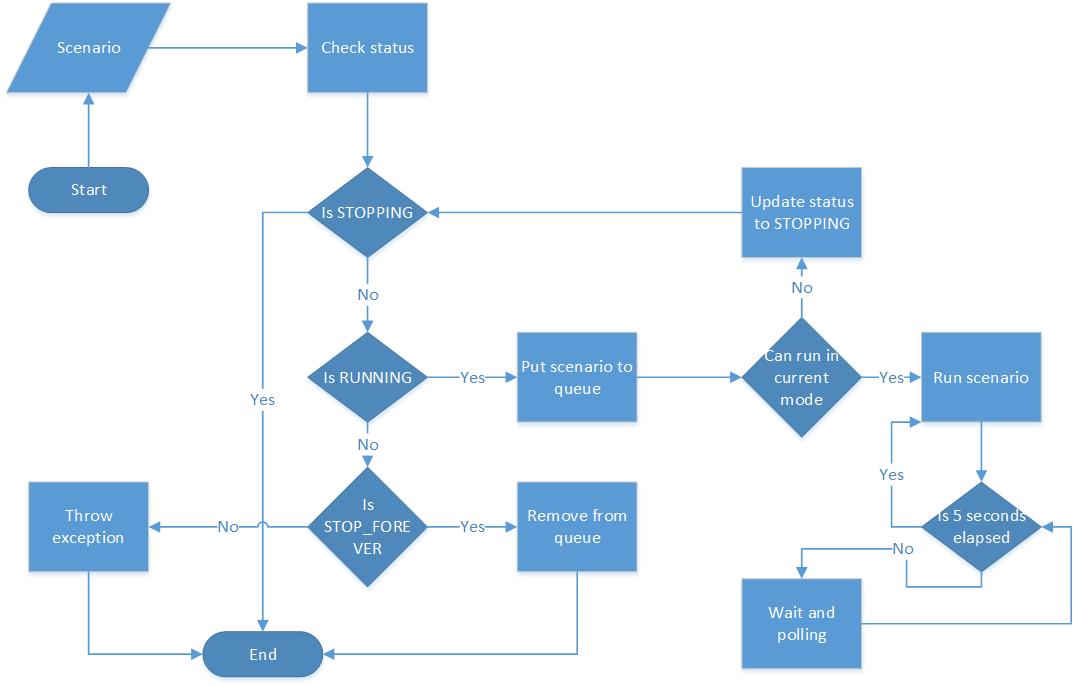
Như đã đề cập về khái niệm kịch bản mâu thuẫn, trong thực tế nếu sự mâu thuẫn giữa các kịch bản tồn tại, hệ thống sẽ gặp nhiều rắc rối. Do đó, việc xử lý, ngăn chặn kịch bản mâu thuẫn là thiết yếu phải có của hệ thống. Hình 11 mô tả tổng quan về thuật toán kiểm tra kịch bản mâu thuẫn. Đầu vào của thuật toán là kịch bản hệ thống muốn kiểm tra, cùng với danh sách kịch bản đã tồn tại trong cùng chế độ.

Bước đầu, ta cần trích xuất ra từ kịch bản đầu vào những cặp <điều kiện gộp, hành động đơn giản> (<merged condition, simple action>) giúp cho việc so sánh. Nếu như kịch bản có tất cả 3 hành động đơn giản (simple action) thì ta sẽ có ít nhất 3 cặp <điều kiện gộp, hành động đơn giản> như trên. Một lưu ý nhỏ là <điều kiện gộp> có thể không tồn tại. Một điều kiện bên ngoài có thể gộp được với điều kiện lồng trong nó khi và chỉ khi 2 điều kiện đó cùng thuộc về 1 thiết bị nào đó, hoặc là điều kiện về thời gian. Lấy ví dụ:

* Kịch bản 1: “Nếu nhiệt độ thu được từ cảm biến nhiệt gần cửa sổ lớn hơn 40 độ và bé hơn 50 độ thì bật đèn phòng”. Điều kiện gộp ở đây sẽ là nhiệt độ nằm trong khoảng 40 độ và 50 độ.
* Kịch bản 2: “Nếu thời gian từ 0h đến 6h thì tắt đèn phòng, đồng thời trong khoảng thời gian đó, nếu thời gian từ 4h đến 5h thì bật thêm đèn hành lang”. Điều kiện gộp ở đây cho hành động bật đèn hành lang sẽ là “từ 4h đến 5h”.

Tiếp theo, thuật toán còn kiểm tra cả việc nếu như kịch bản có dạng “If/Then/Else” thì hành động ở khối “Then” và khối “Else” không được giống nhau (giống về mặt ngữ nghĩa). Nếu không, thuật toán sẽ xếp loại kịch bản này là mâu thuẫn. Sau các bước trên, ta sẽ phải lặp trên tất cả kịch bản hiện tại, với mỗi kịch bản, ta so sánh các hành động đơn giản trong kịch bản ấy với từng hành động trong cặp <điều kiện gộp, hành động đơn giản>. Nếu 2 điều kiện đó là trái ngược nhau về mặt ngữ nghĩa (ví dụ một hành động kêu bật đèn phòng và một hành động khác kêu tắt đèn phòng) thì ta xét tiếp về điều kiện gộp của chúng có “trùng nhau” hay không. Khái niệm xét “trùng” điều kiện đã được nêu rõ ở mục **6.9.2**. Nếu duyệt đến hết danh sách kịch bản hiện tại và không tìm thấy dấu hiệu kịch bản mâu thuẫn, thuật toán trả về kết quả kịch bản không mâu thuẫn. Trường hợp ngược lại thì sẽ trả về là kịch bản mâu thuẫn và lỗi sẽ được thông báo cho người dùng trên giao diện.

## module quản lý trạng thái các kịch bản (scenario runner)



Hình 12 Flowchart thể hiện cách quản lý trạng thái các kịch bản

Việc xây dựng kịch bản, cách thức tổ chức, cấu trúc dữ liệu kịch bản, … đều là tiền đề cho việc thực thi kịch bản đó. Nhóm đã phải đắn đó trong việc chọn cách thiết kế cấu trúc dữ liệu để việc quản lý, cũng như thực thi kịch bản dễ dàng và suôn sẻ hơn. Đáp ứng cho nhu cầu quản lý ấy, nhóm tạo ra module Scenario Runner. Đầu vào của module này sẽ là một kịch bản hệ thống với mong muốn là hệ thống sẽ thực thi, “chạy” kịch bản ấy. Hình 12 mô tả thuật toán quản lý trạng thái các kịch bản trong hệ thống. Đầu vào thuật toán nhận kịch bản hệ thống với trạng thái xác định. Nếu trạng thái là dừng (STOP), module lập tức cập nhật ngay trạng thái đó vào kịch bản tương ứng trong hàng chờ hệ thống và làm nó dừng thật sự. Nếu trạng thái là chạy (RUNNING), ta cần kiểm tra thêm 1 điều kiện đó là kịch bản này có thể chạy ở chế độ hiện tại hay không. Nếu có thì cứ sau khoảng thời gian mặc định, tầm 5 giây, kịch bản được kích hoạt hàm chạy một lần. Và ngược lại, nếu kịch bản không thuộc về chế độ hiện tại của nhà này, trạng thái kịch bản sẽ cập nhật thành dừng và nằm trong hàng chờ hệ thống. Trường hợp mà người dùng mong muốn xóa đi kịch bản, trạng thái của nó sẽ trở thành dừng vĩnh viễn và được loại bỏ khỏi hàng chờ.

Module Scenario Runner quản lý mỗi kịch bản bởi 1 thread khác nhau, nói cách khác là chịu trách nhiệm xử lý multi-thread. Mỗi kịch bản cứ sau khoảng 5 giây sẽ được chạy lại một lần, nhằm đảm bảo mọi cập nhật mới nhất với kịch bản ấy có hiệu lực gàn như lập tức. Kịch bản thì có chứa thông tin về trạng thái: chạy, dừng, dừng vĩnh viễn. Nhiệm vụ module này cũng là kiểm soát trạng thái các kịch bản ấy.

Sau khi một kịch bản mới được thêm vào hệ thống, trạng thái của nó sẽ là chạy (RUNNING). Khi kịch bản ấy được cập nhật, trạng thái của nó cũng được cập nhật lại. Nếu như người dùng muốn cho phép (enable) hoặc vô hiệu hóa (disable) ngôi nhà, hay thiết bị có liên quan kịch bản ấy thì thao tác đó cũng làm ảnh hưởng đến trạng thái của kịch bản. Ví dụ người dùng cho phép dùng (enable) thiết bị ấy thì các kịch bản liện quan thiết bị đó sẽ mang trạng thái chạy (RUNNING). Khi người dùng vô hiệu hóa thiết bị, các kịch bản liên quan sẽ chuyển sang trạng thái dừng (STOP). Tương tự với việc cho phép sử dụng nhà hay vô hiệu hóa nhà thì kịch bản thuộc nhà ấy cũng có trạng thái chạy hay dừng tương ứng.

Tuy nhiên, vấn đề quản lý multi-thread hiệu quả khi số lượng kịch bản tăng lên cũng là một khó khăn mà nhóm đang đối mặt và sẽ được đề cập tới trong mục thảo luận.

## module hỗ trợ bảo mật, xác thực và phân quyền (authorization & authentication)

@Tùng

Kết thúc chương này, ta có được một cái nhìn chi tiết hơn, sâu hơn về phần thiết kế hệ thống back-end. Back-end đóng vai trò là trung tâm, là “bộ não” của toàn hệ thống, lưu trữ các dữ liệu quan trọng của người dùng, … cho nên khâu thiết kế, xây dựng và hiện thực cần tiến hành kĩ lưỡng, cân nhắc đến nhiều vấn đề liên quan. Hệ thống back-end được xây dựng theo hướng module hóa, mỗi module đảm nhận nhiệm vụ chuyên biệt. Trong đó, quan trọng nhất chính là các module chuyển đổi sang kịch bản hệ thống, module kiểm tra tính hợp lệ và module quản lý, khởi chạy các kịch bản. Tuy rằng hệ thống chưa thực sự hoàn thiện, việc quản lý tài nguyên, quản lý kịch bản chưa được tối ưu, dạng kịch bản chưa phong phú, đa dạng, … nhưng với bước khởi điểm này cùng những định hướng mở rộng trong tương lai thì hệ thống sẽ có tính khả thi và thực tiễn cao hơn.

# Phụ lục

## Một số kịch bản thông dụng

**Conditions đang có:**

* Đèn bật, đèn tắt
* Còi bật, còi tắt
* Nhiệt độ >, <, >=, <=
* Có gas
* Có người
* Trời sáng, trời tối

**Actions đang có:**

* Bật đèn, tắt đèn
* Bật còi, tắt còi

**If - Then scenarios**

* **If** (đèn ngủ bật) **Then** (tắt đèn phòng khách và nhà bếp)
* **If** (đèn cầu thang 1 bật) **Then** (tắt đèn cầu thang 2)
* **If** (đèn cầu thang 2 bật) **Then** (tắt đèn cầu thang 1)
* **If** (còi bật) **Then** (bật tắt cả các đèn)
* Chú thích: Khi còi bật => có sự cố xảy ra => bật đèn để thoát hiểm
* **If** (nhiệt độ > 50) **Then** (bật còi và bật tất cả đèn)
* **If** (có gas) **Then** (bật còi và bật tất cả các đèn)
* **If** (có người trước cửa và trời tối) **Then** (Bật đèn và hú còi)
* **If** (có người trước cửa và trời sáng) **Then** (Hú còi)
* **If** (có người vào nhà và trời tối) **Then** (Hú còi và bật tất cả đèn)
* **If** (có người vào nhà và trời sáng) **Then** (Hú còi)
* Chú thích: Giống như chuông reo khi có người đến
* **If** (có người lên cầu thang) **Then** (bật đèn cầu thang)
* **If** (trời sáng) **Then** (tắt đèn)
* **If** (trời tối) **Then** (bật đèn)

**From - To scenarios**

* **From** (20h) **To** (6h) **Do** (Bật đèn ngủ)
* **From** (6h) **To** (20h) **Do** (Tắt đèn ngủ)
* **From** (20h) **To** (6h) **Do** (When (có người) Then (Hú còi + Bật đèn) )
* **From** (20h) **To** (6h) **Do** (When (có gas) Then (Hú còi + Bật đèn) )
* **From** (20h) **To** (6h) **Do** (**If** (nhiệt độ > 50) **Then** (Hú còi + Bật đèn) )
* **From** (20h) **To** (6h) **Do** (**If** (còi hú) **Then** (Bật tất cả các đèn) )

## Danh sách API cung cấp cho client

Danh sách API được liệt kê theo bảng sau

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Page/ Function** | **Method Type** | **URL** | **Input** | **Output** |
| Get list categories given home | GET | homes/{homeId}/device-types | homeId: Long | HTTP Status OK 200 if successful  categories: List<Category> |
| Get all devices given category | GET | homes/{homeId}/device-types/{deviceTypeId}/devices | homeId: Long  deviceTypeId: Long | HTTP Status OK 200 if successful  Return: List<Device> |
| Enabled/  Disabled device | PATCH | homes/{homeId}/device-types/{deviceTypeId}/devices/  {deviceId} | deviceId: Long  homeId: Long  deviceTypeId: Long  Pass json: {"enabled": true}  to update enabled = true | HTTP Status OK 204 if successful |
| Delete device in home | DELETE | /homes/{homeId}/device-types/{deviceTypeId}/devices/{deviceId} | deviceId: Long  homeId: Long  deviceTypeId: Long | HTTP Status OK 204 if successful |
| Get list scripts given mode | GET | devices/{deviceId}/modes/  {modeId}/scripts | deviceId: Long  modeId: Long | HTTP Status OK 204 if successful  Return: List<Script> |
| Update script of one specific mode | PATCH | devices/{deviceId}/modes/  {modeId}/scripts/{scriptId} | deviceId: Long  modeId: Long  script: Script | HTTP Status OK 204 if successful |
| Delete script in one specific mode | DELETE | devices/{deviceId/modes/  {modeId}/scripts/{scriptId} | scriptId: Long | HTTP Status OK 204 if successful |
| Add script to device given mode | POST | devices/{deviceId}/modes/  {modeId}/scripts | deviceId: Long  modeId: Long  script: Script | HTTP Status OK 201 if successful  Also return the URI of new object. E.g devices/5/modes/1/  scripts/3 |
| Add device of specific category | POST | homes/{homeId}/device-types/{deviceTypeId}/devices | homeId: Long  categoryId: Long  device: Device | HTTP Status OK 201 if successful  Also return the URI of new object. E.g  homes/5/device-types/3/  devices/4 |
| Get all valid GPIO pins | GET | homes/all-gpios |  | HTTP Status OK 200 if successful.  Return: List<Integer> |
| Get home | GET | homes/{homeId} | homeId: Long | HTTP Status OK 200 if successful.  Return: Home |
| Delete home | DELETE | homes/{homeId} | homeId: Long | HTTP Status OK 204 if successful |
| Add new home  (also add default mode ) | POST | homes | home:Home | HTTP Status OK 201 if successful  Also return the URI of new object. E.g  homes/1 |
| Update home (Enabled/ Disable / Current mode….) | PATCH | homes/{homeId} | homeId: Long  home: Home  Pass json: {"enabled": true}  to update enabled = true | HTTP Status OK 204 if successful |
| Get list home of one user | GET | homes |  | HTTP Status OK 200 if successful  Return: List<Home> |
| Get list mode in home | GET | homes/{homeId}/modes | homeId: Long | HTTP Status OK 200 if successful  Return: List<Mode> |
| Delete mode | DELETE | homes/{homeId}/modes/  {modeId} | homeId: Long  modeId: Long | HTTP Status OK 204 if successful |
| Add new mode | POST | homes/{homeId}/modes | mode:Mode  homeId: Long | HTTP Status OK 201 if successful  Also return the URI of new object. E.g  homes/3/modes/3 |
| Update existing mode | PATCH | homes/{homeId}/modes/  {modeId} | homeId: Long  modeId: Long  mode: Mode | HTTP Status OK 204 if successful |
| Login | POST | /login | Add headers:  X-Username  X-Password | HTTP Status:  200 OK  401 Unauthorized |
| Logout | POST | /logout | Add header:  X-Auth-Token | Http Status:  200 OK  401 Unauthorized |
| Sign up | POST | /users/signup | User information**:**  Username  Password  (encrypted )  Fullname  Email | Return: User’s activation link  Return code:  -1: ERROR\_WHEN\_ADD\_USER  -2: USERNAME\_ALREADY\_EXISTED  -3: EMAIL\_ALREADY\_EXISTED  > 0: Successful  Http status:  200 OK  400 Bad request |
| Activate user | GET | /users/activation/{userId} | User id | Http status:  200 OK  400 Bad request |
| Get user information | GET | /users | X-Auth-Token | Http status:  200 Ok  400 Bad request  Return: User model |
| Get list device id in script | POST | /homes/{homeId}/scripts | Script object with content and script type id | Http status:  200 OK  400 BAD REQUEST  Return list device id in that script. |

# Thao luan

## Giải pháp xử lý các kịch bản có khả năng mâu thuẫn

Tại thời điểm viết luận văn này, các kịch bản có khả năng mâu thuẫn sẽ không được phép tồn tại trong hệ thống. Nhóm muốn đề xuất một phương pháp có thể xử lý tình huống này khi mở rộng hệ thống trong tương lai, đó là cho phép người dùng thiết lập thêm độ ưu tiên cho kịch bản trong trường hợp có khả năng mâu thuẫn. Giả sử trong một điều kiện nào đó, 2 kịch bản có khả năng mâu thuẫn thực sự mâu thuẫn nhau, ta hoàn toàn có thể dựa trên độ ưu tiên để xem xét kịch bản nào được chạy trong trường hợp trên, bằng cách chọn ra kịch bản có độ ưu tiên cao hơn. Tuy nhiên trong điều kiện bình thường, việc xét độ ưu tiên khi tạo ra một kịch bản mới có thể không mang lại nhiều ý nghĩa và khiến người dùng bị rối. Đứng từ khía cạnh người dùng, hệ thống có thể được xem là thông minh hơn nếu như có khả năng tự thiết lập một độ ưu tiên mặc định cho một số dạng kịch bản, dựa trên ý nghĩa, nhu cầu thực tế của kịch bản đó. Người dùng có thể tự do thay đổi giá trị mặc định đó dựa trên mục đích riêng của họ.

### Độ ưu tiên mặc định cho một số loại kịch bản

Sau khi tham khảo từ một số nhu cầu thực tế, nhóm đề xuất ra một số dạng kịch bản có thể được xét giá trị ưu tiên mặc định. Nếu như kịch bản chứa một trong những điều kiện thuộc dạng khẩn cấp sau thì chúng lần lượt có thứ tự ưu tiên như sau (số thứ tự càng thấp thì độ ưu tiên càng cao):

1. Temperature > N (N là một giá trị nào đó).

2. Phát hiện có gas.

3. Phát hiện người chuyển động trong thời gian đêm khuya.

4. Các dạng kịch bản khác.

Có thể giải thích cho việc đưa ra quyết định trên như sau

* Nếu nhiệt độ vượt ngưỡng => Khả năng có cháy xảy ra cao, dẫn đến độ nguy hiểm tính mạng cao => Ưu tiên số 1.
* Trường hợp có gas, tức có khả năng xảy ra cháy, nhưng thật sự vẫn chưa cháy => Ưu tiên số 2.
* Trường hợp này có khả năng là trộm => dẫn đến có thể bị mất đồ nhưng khả năng gây chết người thấp => Ưu tiên số 3.

Nếu kịch bản có chứa nhiều điều kiện dạng trên thì độ ưu tiên mặc định của kịch bản ấy chỉnh là độ ưu tiên cao nhất của một trong các điều kiện liệt kê ở trên.

### Cách xử lý một số trường hợp trùng độ ưu tiên

Như đã thảo luận ở mục trên, việc xử lý kịch bản có khả năng mâu thuẫn có thể giải quyết bằng độ ưu tiên. Nhưng một số trường hợp, chúng có cùng độ ưu tiên (người dùng không có ý muốn thay đổi độ ưu tiên mặc định) thì ta phải xử lý ra sao? Nhóm xin liệt kê và giải thích qua một vài ví dụ.

Ví dụ 1:

Kịch bản 1: Nếu có gas thì tắt đèn phòng.

Kịch bản 2: Nếu có gas thì bật còi hú gần cửa ra vào và đồng thời kiểm tra nếu cảm biến ánh sáng cho kết quả bây giờ là ban ngày thì bật đèn phòng.

Nhìn sơ qua, 2 kịch bản trên có khả năng mâu thuẫn nhau và đang có cùng độ ưu tiên mặc định là 2. Trường hợp này, nhóm đề xuất nếu như kịch bản nào có độ chi tiết hơn thì nó sẽ được xem là có độ “ưu tiên” cao hơn. Lý do là vì đứng từ góc nhìn người đặt ra kịch bản, việc thực thi 1 kịch bản có mức chi tiết hơn sẽ đáp ứng mong đợi người dùng nhiều hơn.

Ví dụ 2:

Kịch bản 1: Từ 22h đến 6h thì tắt đèn phòng.

Kịch bản 2: Từ 5h đến 7h thì bật đèn phòng.

Trường hợp này, nhóm đề xuất kịch bản 2 có độ ưu tiên cao hơn. Có thể giải thích rằng là do kịch bản bật đèn từ 5h đến 7h có khoảng giá trị nhỏ hơn từ 22h -> 6h, mang nghĩa kịch bản 2 có mức độ xuất hiện không thường xuyên cao hơn kịch bản 1 cho nên có ưu tiên cao hơn.

Nếu xảy ra một trường hợp khác không nằm trong các trường hợp liệt kê mục này, hệ thống có thể thông báo cho người dùng, yêu cầu họ nhập giá trị ưu tiên cho các kịch bản có khả năng mâu thuẫn trên.

## Giải pháp tối ưu việc quản lý, vận hành các kịch bản dựa trên kĩ thuật multi-threading

Kịch bản là một khái niệm quan trọng trong hệ thống này. Như ta đã biết, kịch bản được tạo ra, được hệ thống chuyển đổi, kiểm tra hợp lệ,…. và cuối cùng là chạy nó. Với thiết kế hệ thống hiện tại, mỗi kịch bản được quản lý bởi 1 thread riêng biệt. Khi kịch bản ở trạng thái chạy, cứ cách một khoảng thời gian ngắn, kịch bản ấy sẽ được kiểm tra lại điều kiện và thực thi hành động tương ứng, điều đó diễn ra thành vòng lặp vô tận. Có thể để ý thấy, mỗi thread không luôn luôn phải ở trạng thái sẵn sàng (active) mà nó chỉ cần hoạt động sau mỗi khoảng thời gian ngắn. Từ đó dẫn đến việc lãng phí một lượng lớn tài nguyên hệ thống (ở đây là lãng phí xử lý CPU và cả bộ nhớ, memory) chỉ phục vụ cho mục đích chờ đợi là chủ yếu. Thử tưởng tượng nếu hệ thống có tầm 20 kịch bản, đồng nghĩa có 20 threads tại thời điểm này, trong khi bộ nhớ của Rapsberry Pi khá thấp, CPU cũng không đủ mạnh, việc xử lý đa luồng và mỗi luồng là một kịch bản có thể sẽ làm giảm hiệu năng toàn hệ thống.

Sau thời gian tìm hiểu và thảo luận, nhóm xin đề xuất một thuật toán cải tiến nhỏ giúp cho việc quản lý trên hiệu quả hơn. Ta sẽ vẫn giữ mỗi thread quản lý một kịch bản, tuy nhiên điểm khác biệt đó là việc chạy kịch bản chỉ được thực hiện 1 lần khi tạo, hay 1 lần sau khi cập nhật mới nội dung kịch bản. Trong module “Scenario Runner”, ta hiện thực thêm 1 hashmap “scenario mapping” chả hạn, để ánh xạ kịch bản với thread quản lý nó. Nếu như có bất kì thay đổi nào tới kịch bản, dựa trên hashmap “scenario mapping” trên để thông báo đến thread tương ứng và chạy lại kịch bản mới ấy. Với cách hiện thực này, việc lãng phí tài nguyên CPU sẽ giảm đi đáng kể và chỉ còn chi phí khi lưu thread trên bộ nhớ thôi.

## Giải pháp xây dựng hệ thống trên cloud phục vụ nhiều người dùng

@Cập nhật sau

# Mot so cau hoi co the dat ra

Vd minh muon support them device moi thi minh lam the nao ?

An toan du lieu ?

Kha nang chiu dung cua he thong ?

Tại sao dùng Restful WS mà ko dùng WebSocket hay cách khác để communicate với clients ? ( có thể trả lời là 1 phần đề tài yêu càu, phần khác mình phải tìm hiểu thêm

For more real time interaction, or real time transfer or streaming of data, HTTP and REST aren’t the best suited protocol and abstraction combination. This is where Sockets and WebSockets shine.

https://www.pubnub.com/blog/2015-01-05-websockets-vs-rest-api-understanding-the-difference/

)

# Tài liệu tham khảo

[1] <http://www.javadevchannel.com/2015/07/nhung-cau-hoi-pho-bien-ve-spring.html> (29/11/16)

[2] <https://projects.spring.io/spring-framework/> (29/11/16)

[3] <https://www.genuitec.com/spring-frameworkrestcontroller-vs-controller/> (30/11/16)

[4] <http://hibernate.org/orm/> (30/11/16)

[5] Alex Rodriguez, 2013. *Căn bản về RESTFul Web Services.* Trích 30/11/16

<https://www.ibm.com/developerworks/vn/library/ws-restful/>

[6] <http://www.accesa.eu/2015/resources/javafx-and-restful-web-services-communication>

[7] <http://www.athlsolutions.com/web/ve-chung-toi/tin-tuc-su-kien/xu-huong-tat-yeu-cua-nha-thong-minh> . Trích từ nguồn <http://www.pcworld.com.vn> , tham khảo vào 11/12/16

[8] STAMFORD Conn, 2016. *Gartner says worldwide IoT*. Trích 25/05/16

<http://www.gartner.com/newsroom/id/3291817>