# LỜI CẢM ƠN

# 🙞🏵🙜

Chúng em xin chân thành cảm ơn Ban giám hiệu trường Đại học Nông Lâm TP Hồ Chí Minh, quý thầy cô trường Đại học Nông Lâm TP Hồ Chí Minh

Xin chân thành cảm ơn cô Trần Thị Thanh Nga đã tận tình hướng dẫn, chỉ bảo cho chúng em trong suốt thời gian thực hiện đề tài

Xin chân thành cảm ơn quý thầy cô khoa Công nghệ thông tin đã tận tình giúp đỡ chúng em trong suốt những năm học qua và trong thời gian thực hiện luận văn tốt nghiệp

Xin tỏ lòng biết ơn sâu sắc đến cha mẹ cùng các anh chị em trong gia đình đã tạo mọi điều kiện cho chúng em học tập

Chân thành cảm ơn tất cả các bạn khoa Công nghệ thông tin trường Đại học Nông Lâm đã tận tình giúp đỡ, chỉ dẫn cho chúng tôi trong suốt thời gian thực tập tốt nghiệp

TP HCM, ngày tháng năm

NHÓM THỰC HIỆN

LUẬN VĂN TỐT NGHIỆP

Luận văn cử nhân được bảo vệ tại HỘI ĐỒNG CHẤM LUẬN VĂN CỬ NHÂN TRƯỜNG ĐẠI HỌC NÔNG LÂM TP HCM ngày … tháng … năm …

*Nhận xét của giáo viên hướng dẫn*

*Nhận xét của giáo viên phản biện*

**SINH VIÊN THỰC HIỆN:**

Họ tên sinh viên:

Ngày tháng năm sinh:

Chuyên ngành: Công nghệ thông tin

Điện thoại liên lạc:

Email:

Họ tên sinh viên:

Ngày tháng năm sinh:

Chuyên ngành: Công nghệ thông tin

Điện thoại liên lạc:

Email:

QUI CÁCH VIẾT TÀI LIỆU

--- phần này giải thích các phần được in đậm, in nghiêng, gạch chân … trong tài liệu---

Mục lục

Danh mục bằng hình ảnh

Danh mục bảng

CHƯƠNG 1: MỞ ĐẦU

# Đặt vấn đề

Ngày nay, đời sống của mọi người ngày càng được cải thiện và nâng cao, mọi người ngày càng có nhu cầu hưởng thụ cuộc sống cao hơn. Bên cạnh đó, người dân cần đi làm để trang trải cuộc sống, vì vậy họ không thường xuyên ở nhà. Vấn đề về an ninh, tiết kiệm năng lượng, phòng chống cháy nổ,… cần được quan tâm nhiều hơn.

Nhà thông minh dùng công nghệ để kết nối các thiết bị trong căn nhà và giúp bạn kiểm soát hoàn toàn hoạt động trong ngôi nhà theo ý muốn thông qua smartphone hay bộ điều khiển trung tâm. Từ ánh sáng, nhiệt độ, an ninh đến các chi tiết khác ảnh hưởng đến cuộc sống thường ngày của gia đình bạn. Công nghệ nhà thông minh có thể giúp gia đình tiết kiệm chi phí sinh hoạt cũng như đem lại sự thoải mái và yên tâm mỗi khi bạn đi vắng.

Do đó chúng tôi quyết định lựa chọn đề tài Smart Home để thực hiện.

# Mục đích, đối tượng và phạm vi nghiên cứu

## Mục đích nghiên cứu

định hướng để giải quyết những vấn đề cấp bách.

Xây dựng hệ thống nhà thông minh giúp người dùng dễ dàng điều khiển các thiết bị điện trong gia đình bằng smartphone qua các chế độ như:

+ điều khiển bằng giọng nói

+ điều khiển bằng button

+ điều khiển bằng chế độ tự động thông qua các cảm biến

+ điều khiển qua chế độ hẹn giờ

## Đối tượng và phạm vi nghiên cứu

* đối tượng nghiên cứu:
* Công cụ lập trình android studio ngôn ngữ lập trình kotlin
* Công nghê Nodejs và socketIo
* Database MogoDB và Mogu
* ngôn ngữ Arduino, nodeMCU esp8266 và các cảm biến
* Phạm vi nghiên cứu:
* Xây dựng ứng dụng Smart Home trên điện thoại di động, và webside theo cơ chế realtime.
* Xây dựng trang addmin thống kê và quản lý các tài khoản, thiết bị điện trong nhà của user.

# Kết quả, mục tiêu cần đạt được.

* Ứng dụng app mobile “*smart home*” cho phép người dùng điều khiển các thiết bị trong nhà thông qua giọng nói, chế độ tự động bật tắt và cài đặt giờ bật tắt.
* Hệ thống admin quản lý user, phòng, thiets bị của user.
* Hệ thống server nodejs kết nối các client

CHƯƠNG 2: KIẾN THỨC CƠ BẢN

# Tìm hiểu về Kotlin

## Tổng quan về công cụ Kotlin

Kotlin là một ngôn ngữ lập trình chạy trên máy ảo Java và cũng có thể được biên dịch thành mã nguồn JavaScript hoặc sử dụng cơ sở hạ tầng trình biên dịch LLVM. Phát triển chính của nó là từ một nhóm các lập trình viên JetBrains có trụ sở tại Saint Petersburg, Nga. Trong khi cú pháp không tương thích với Java, việc thực thi JVM của thư viện chuẩn Kotlin được thiết kế để tương thích với mã Java và dựa vào mã Java từ Thư viện lớp Java hiện có, chẳng hạn như khung bộ sưu tập. Kotlin sử dụng suy luận kiểu tích cực để xác định loại giá trị và biểu thức cho loại nào bị bỏ sót. Điều này làm giảm độ dài ngôn ngữ tương đối so với Java, yêu cầu thông số kỹ thuật loại thường hoàn toàn dư thừa trước phiên bản 10.

Kể từ Android Studio 3.0 (được phát hành vào tháng 10 năm 2017), Kotlin được Google hỗ trợ đầy đủ để sử dụng với hệ điều hành Android của họ và được bao gồm trực tiếp trong gói cài đặt của IDE thay thế cho trình biên dịch Java chuẩn. Trình biên dịch Kotlin Android cho phép người dùng chọn giữa nhắm mục tiêu Java 6, Java 7 hoặc mã byte bytecode tương thích Java 8.

### **Lịch sử**

Vào tháng 7 năm 2011, JetBrains đã giới thiệu Project Kotlin, một ngôn ngữ mới cho JVM, vốn đã được phát triển trong một năm. JetBrains dẫn đầu Dmitry Jemerov nói rằng hầu hết các ngôn ngữ không có các tính năng mà họ đang tìm kiếm, ngoại trừ Scala. Tuy nhiên, ông trích dẫn thời gian biên dịch chậm của Scala như là một sự thiếu hụt rõ ràng. Một trong những mục tiêu đã nêu của Kotlin là biên dịch nhanh như Java. Vào tháng 2 năm 2012, JetBrains đã mở nguồn dự án theo giấy phép Apache 2.

Tên đến từ đảo Kotlin, gần St. Petersburg. Andrey Breslav đã đề cập rằng nhóm đã quyết định đặt tên cho nó sau khi một hòn đảo giống như Java được đặt tên theo đảo Java của Indonesia (mặc dù ngôn ngữ lập trình Java có lẽ được đặt tên theo cà phê.)

JetBrains hy vọng rằng ngôn ngữ mới sẽ thúc đẩy doanh số bán hàng IntelliJ IDEA.

Kotlin v1.0 được phát hành vào ngày 15 tháng 2 năm 2016. Đây được coi là bản phát hành chính thức đầu tiên ổn định và JetBrains đã cam kết khả năng tương thích ngược dài hạn bắt đầu với phiên bản này.

Tại Google I / O 2017, Google đã công bố hỗ trợ hạng nhất cho Kotlin trên Android.

Kotlin v1.2 được phát hành vào ngày 28 tháng 11 năm 2017. Mã chia sẻ giữa tính năng nền tảng JVM và Javascript mới được thêm vào bản phát hành này.

### **Kotlin và Android**

Lập trình Android là một trong những lĩnh vực quan trọng mà JetBrains hướng đến với Kotlin. Ngôn ngữ mang đến tính tương thích ngược với Java 6 và 7, các phiên bản của Java hầu hết đều tương thích chặt chẽ với Android. JetBrains cũng hy vọng Kotlin sẽ được sử dụng trong các lĩnh vực khác chẳng hạn như các ứng dụng lớn và phức tạp, đề cao hiệu suất.

Các nhà phát triển không có lựa chọn thay thế cho việc phát triển ứng dụng Android bằng ngôn ngữ Java. Mặc dù được sử dụng rộng rãi, nhưng trong quá trình hoạt động ngôn ngữ Java sinh ra rất nhiều file rác. Java 8 đã giải quyết một số vấn đề ngôn ngữ và đặc biệt hơn là với Java 10. Để có được nhiều lợi ích từ việc chỉnh sửa trong hai phiên bản này, bạn phải đặt SDK tối thiểu sang Android 24 chỉ để sử dụng Java 8. Kotlin nhắm đến việc lấp đầy khoảng trống đó của một ngôn ngữ hiện đại đang thiếu cho nền tảng Android.

Có một vài nguyên lý cốt lõi mà giúp Kotlin tồn tại đó là:

Nhỏ gọn để giảm số lượng mã boilerplate cần viết. Mã code dễ đọc và dễ hiểu hơn. An toàn, tránh toàn bộ các lớp lỗi như các null pointer exceptions. Linh hoạt cho việc xây dựng ứng dụng phía máy chủ, ứng dụng Android hoặc mã lối vào đang chạy trong trình duyệt. Khả năng tương tác để tận dụng các khuôn khổ và thư viện hiện có của JVM với khả năng tương tác Java 100%.

### **Ưu và nhược điểm của Kotlin**

#### **Ưu điểm**

Dù không thiếu các ngôn ngữ biên dịch sang Java bytecode, nhưng có một vài yếu tố làm cho Kotlin nổi bật hơn cả:

##### Có thể thay thế cho Java

Một trong những thế mạnh lớn nhất của Kotlin như là một ứng viên để thay thế cho Java là khả năng tương tác rất tốt giữa Java và Kotlin—bạn có thể thậm chí có code Java và Kotlin tồn tại song song trong cùng dự án, và tất cả mọi thứ vẫn sẽ được biên dịch một cách hoàn hảo. Dưới đây, bạn có thể thấy một ví dụ về một dự án bao gồm một Activity Java và một Actitivy Kotlin.

Trong thực tế, một khi dự án kết hợp Kotlin và Java được biên dịch, người dùng sẽ không thể biết những phần nào của dự án của bạn được viết bằng Java, và những phần nào được viết bằng Kotlin. Bởi vì các lớp Kotlin và Java có thể tồn tại song song trong cùng một dự án, nên có thể bắt đầu sử dụng Kotlin mà không cần phải làm bất cứ điều gì to tát cả giống như chuyển đổi toàn bộ dự án sang Kotlin hoặc bắt đầu một dự án mới để bạn có thể thử Kotlin.

Vì Kotlin là hoàn toàn tương thích với Java, nên cũng có thể sử dụng phần lớn các thư viện Java và các framework trong dự án Kotlin của bạn—thậm chí nâng cao các framework dựa vào chú thích xử lý.

##### Dễ học

Kotlin nhằm mục đích là nâng cao hơn so vớiJava, chứ không phải hoàn toàn viết lại, rất nhiều các kỹ năng đã có trong việc code Java của bạn vẫn được áp dụng đối với dự án Kotlin.

Kotlin cũng được thiết kế để có thể dễ học cho các nhà phát triển Java. Những nhà phát triển Java sẽ cảm thấy rằng hầu hết cú pháp của Kotlin đều quen thuộc; ví dụ, các code được sử dụng để tạo ra một lớp mới trong Kotlin là rất giống với Java:

class MainActivity : AppCompatActivity() {

Kotlin cũng được thiết kế trực quan và dễ đọc, do đó, ngay cả khi bạn gặp một số code khác biệt, thì bạn vẫn có thể để biết được ý nghĩa về những gì code này làm.

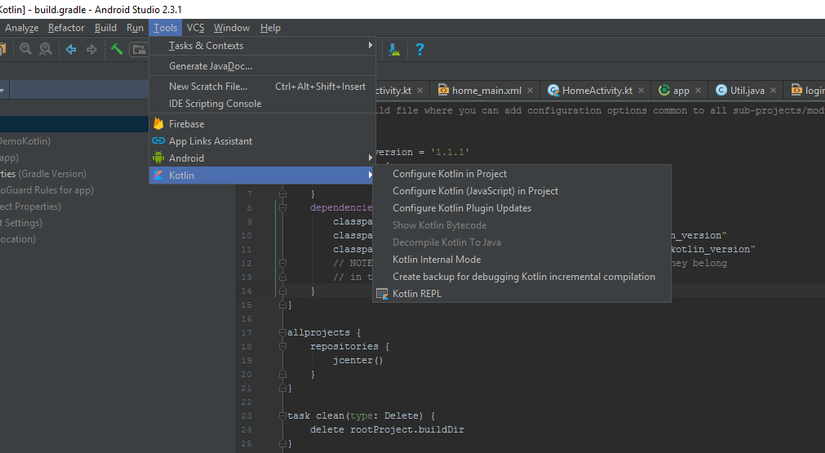
##### Kết hợp những gì tốt nhất của lập trình hàm và thủ tục

Hiện đang có một số lý thuyết lập trình được sử dụng rộng rãi, nhưng khi nói đến các câu hỏi về "phương pháp nào là tốt nhất", thì không dễ để có câu trả lời. Mỗi kiểu lập trình có tập hợp điểm mạnh và điểm yếu của nó, do đó, mặc dù không thiếu các kịch bản mà lập trình hàm có một lợi thế, thì cũng có rất nhiều vấn đề nơi mà một cách tiếp cận lập trình thủ tục sẽ hiệu quả hơn.

Vậy tại sao bạn cần phải lựa chọn giữa hàm và thủ tục? Giống như nhiều ngôn ngữ lập trình hiện đại khác, Kotlin nhằm mục đích mang lại cho bạn những gì tốt nhất của cả hai bằng cách kết hợp các khái niệm và các yếu tố của lập trình thủ tục và hàm.

##### Android Studio hỗ trợ

Kotlin được phát triển bởi JetBrains, công ty đứng sau IntelliJ—IDE mà Android Studio dựa trên nó. Không có gì bất ngờ, rằng Android Studio hỗ trợ tốt cho Kotlin. Một khi bạn đã cài đặt plugin Kotlin, Android Studio làm cho việc cấu hình Kotlin trong dự án của bạn trở nên đơn giản giống như mở một vài menu.



Một khi bạn đã thiết lập plugin Kotlin cho Android Studio, IDE của bạn sẽ không gặp vấn đề gì việc hiểu, biên dịch và chạy code Kotlin. Android Studio cũng cung cấp việc gỡ lỗi, tự động hoàn tác, điều hướng code, unit testing, và tái cấu trúc cho Kotlin.

Một khi dự án Android Studio của bạn đã được cấu hình để hỗ trợ cho Kotlin, bạn thậm chí có thể chuyển đổi toàn bộ một tập tin mã nguồn Java thành một tập tin Kotlin, với chỉ một vài cú nhấp chuột.

##### Code ngắn gọn hơn

Nếu bạn so sánh một lớp Java và một lớp Kotlin cho ra cùng một kết quả, thì cái được viết trong Kotlin thường sẽ gọn gàng hơn nhiều và nhỏ gọn hơn so với những gì được viết bằng Java. Và như mọi nhà phát triển đều biết, code ít hơn có nghĩa là ít lỗi hơn!

Ví dụ, các code Java sau tạo ra một Activity bao gồm một floating action button (FAB), mà khi chạm vào, hiển thị một snackbar có chứa các thông báo hữu ích This is a snackbar.

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);

setSupportActionBar(toolbar);

FloatingActionButton myfab = (FloatingActionButton) findViewById(R.id.myfab);

myfab.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View view) {

Snackbar.make(view, "This is a snackbar", Snackbar.LENGTH\_LONG)

.setAction("Action", null).show();

}

});

}

}

Kotlin đạt được điều tương tự bằng lượng code ít hơn, đặc biệt là khi nói đến việc tạo FAB và thiết lập onClickListener:

class MainActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_main)

val toolbar = findViewById(R.id.toolbar) as Toolbar

setSupportActionBar(toolbar)

val myfab = findViewById(R.id.myfab) as FloatingActionButton

myfab.setOnClickListener { view ->

Snackbar.make(view, "This is a snackbar", Snackbar.LENGTH\_LONG)

.setAction("Action", null).show()

}

}

}

Cả hai đoạn code cho ra kết quả giống nhau:



Kotlin đặc biệt tốt trong việc giảm số lượng code mà bạn cần phải viết, làm cho việc viết code trong Kotlin trở thành một trải nghiệm thú vị hơn nhiều, so với việc viết code trong nhiều ngôn ngữ khác như Java.

Đặc biệt, các extension của Kotlin Android (mà chúng ta sẽ khám phá trong phần hai) cho phép bạn nhập tham chiếu đến một View vào một tập tin Activity, từ đó, bạn có thể làm việc với giao diện như thể nó là một phần của Activity đó. Điều này có nghĩa là bạn không còn phải xác định mỗi View bằng cách sử dụng findViewById, mà có thể chuyển đổi code chẳng hạn như:

TextView text = (TextView) findViewById(R.id.myTextView); text.setText("Hello World");

Thành gọn gàng hơn nhiều:

myTextView.setText("Hello World")

#### **Nhược điểm**

Không có ngôn ngữ lập trình nào là hoàn hảo cả, do đó, mặc dù Kotlin có rất nhiều thứ để cung cấp cho các nhà phát triển Android, nó vẫn có một số nhược điểm mà bạn cần lưu ý:

##### ****Thêm thời gian chạy Runtime****

Thư viện tiêu chuẩn của Kotlin và runtime sẽ làm tăng kích thước tập tin .apk của bạn. Mặc dù nó chỉ tương đương với khoảng 800KB, nhưng nếu ứng dụng của bạn đã lớn sẵn rồi thì 800KB phụ có thể làm nó phình to và khiến người dùng nghĩ lại trước khi tải về ứng dụng của bạn.

##### ****Không hẳn là code dễ đọc đối với beginer****

Mặc dù cú pháp ngắn gọn của Kotlin là một trong những thế mạnh lớn nhất của ngôn ngữ, nhưng bạn có thể thấy một số khó khăn ban đầu, đơn giản bởi vì có rất nhiều thứ đang được thực hiện trong một số lượng nhỏ code đó. Java có thể dài dòng hơn, nhưng ngược lại tất cả mọi thứ đều rõ ràng, có nghĩa là những người không quen code Java có xu hướng dễ dàng hơn để hiểu so Kotlin.

Ngoài ra, nếu sử dụng không hợp lý, quá tải toán tử của Kotlin có thể dẫn đến số code đó có thể khó khăn để đọc.

##### ****Thiếu hỗ trợ chính thức****

Kotlin có thể được trợ tốt trong Android Studio, nhưng cần ghi nhớ rằng Kotlin không được xác nhận chính thức là của Google.

Ngoài ra, tính năng tự động hoàn tác và biên dịch trong Android Studio có xu hướng chạy hơi chậm khi bạn làm việc với Kotlin so với một dự án thuần Java.

##### ****Cộng đồng nhỏ hơn và ít có sẵn trợ giúp****

Vì Kotlin là một ngôn ngữ tương đối mới, nên cộng đồng Kotlin vẫn còn khá nhỏ, đặc biệt là so với cộng đồng của ngôn ngữ khác như Java. Nếu bạn chuyển đổi sang Kotlin, thì bạn có thể không có được truy cập vào cùng một số hướng dẫn, bài đăng trên blog, và tài liệu hướng dẫn sử dụng, và có thể gặp ít hỗ trợ từ cộng đồng về những nơi như các diễn đàn và Stack Overflow. Tại thời điểm bài viết, tìm kiếm cho Kotlin trong Stack Overflow trả về các bài viết chỉ hơn 4.600 được dán nhãn Kotlin—so với hơn 1.000.000 bài viết có chứa các từ khóa Java.

### **Đặc điểm**

#### **Khai báo biến**

* Có 2 từ khóa khai báo biến trong Kotlin là var và val

val a: Int = 1

val b = 1 // `Int` type is inferred

val c: Int // Type required when no initializer is provided

c = 1 // definite assignment

var x = 5 // `Int` type is inferred

x += 1

* Từ khóa var sử dụng khi giá trị của biến thay đổi, val sử dụng khi giá trị của biến không thay đổi
* Từ khóa val giống như readonly trong C# hoặc final trong Java
* Biến val phải được khởi tạo lúc khai báo
* Từ khóa Void trong java hay C# sẽ được thay thế bằng Unit trong Kotlin

#### **Null Safety**

Một trong những cạm bẫy phổ biến trong các ngôn ngữ lập trình, bao gồm cả Java là cho phép 1 thành phần nào đó được Null. Nếu không chắc chắn thành phần đó được phép Null hay không sẽ rất dễ xảy ra lỗi không lường trước được, gây nguy hiểm cho hệ thống của bạn. Cụ thể, trong java sẽ gây ra 1 exception là NullPoiterException hoặc viết ngắn gọi là NPE.

Kotlin nhằm mục đích xóa bỏ NullPoiterException trong code của chúng ta. Ngay khi khai báo biến với Kotlin, bạn đã phải chỉ rõ biến đó có được phép Null hay không. Có 2 trường hợp được phép đó là : không thể Null và có thể Null

* Khai báo 1 biến cho phép Null

var b: String? = "abc" // có thêm dấu ? sau kiểu của biến

b = null // compilation ok

val l = b.length // not safe

* Khai báo 1 biến không được phép Null

var a: String = "abc"

a = null // compilation error

val l = a.length // safe

Chúng ta có thể thấy Kotlin đã khắc phục được expcetion NPE trong Java. Các biến cho phép Null hay không được phép Null đã được xác định ngay trong quá trình khai báo biến, các IDE sẽ giúp chúng ta phát hiện ra lỗi ngay khi compile. Việc phát hiện ra lỗi sớm khi compile sẽ tốt hơn so với khi runtime. Nó giúp hệ thống của chúng ta an toàn hơn.

##### String template

String trong kotlin có thể chứa các biểu thức template, tức là những kết quả trả về hoặc biến có thể được nối vào trong 1 String. Một biểu thức template bắt đầu với ($$ và tên :

val apples = 4

valbananas = 3

println(“I have $apples apples.”)

println(“I have $apples apples and ” + (apples + bananas) + “ fruits.”)

println(“I have $apples apples and ${apples + bananas} “ fruits.”)

#### **Kế thừa và Override**

* Mặc định class trong Kotlin đều là final (tức là không được phép kế thừa)

open class Base {

open fun v() {}

fun nv() {}

}

class Derived() : Base() {

override fun v() {}

}

* Trong class final (tức là không có từ khóa “open”) thì việc khai báo các hàm, thuộc tính open bị cấm
* Ở 1 lớp con khác, hàm được đánh dấu là override thì chính nó là open (lại được kế thừa từ lớp khác), nếu muốn chống override thì lại sử dụng từ khóa final

open class AnotherDerived() : Base() {

final override fun v() {}

}

* Trong interface, các member mặc định là “open”

interface B {

fun f() { print("B") } // interface members are 'open' by default

fun b() { print("b") }

}

* Kotlin implement interface :

class C() : A(), B {

// The compiler requires f() to be overridden:

override fun f() {

super<A>.f() // call to A.f()

super<B>.f() // call to B.f()

}

}

#### **Chỉ định truy cập (Visibility Modifiers)**

* Có 4 loại Visivility Modifiers trong Kotlin : private, protected, internal, và public
* Nếu không chỉ rõ thì mặc định là public
* Các chỉ định truy cập : private, protected và public giống cách sử dụng trọng java
* Riêng internal ta có thể sử dụng trong cùng module

#### **Truyền tham số trong hàm**

* Kotlin cho phép thay đổi vị trí tham số trong hàm

fun main(args : Array<String>) {

greet(firstName = "Frasensco", lastName = "Merini")

greet(lastName = "John", firstName = "Stamos")

greet("Borat", "Ismail")

greet("Crystal", lastName = "Stamos")

call("Xavier", age = 20, location = "Portugal")

}

fun greet(firstName : String, lastName : String){

println("Good morning $firstName $lastName")

}

fun call(name : String, location : String, age : Int){

println("Call $name who lives at $location and he is $age old")

}

Out put :

Good morning Frasensco Merini

Good morning Stamos John

Good morning Borat Ismail

Good morning Crystal Stamos

Call Xavier who lives at Portugal and he is 20 old

## DataBinding trong Kotlin

### **Tổng quan**

The Data Binding Library là một thư viện hỗ trợ cho phép bạn liên kết các thành phần giao diện người dùng trong bố cục của bạn với các nguồn dữ liệu trong ứng dụng của bạn bằng cách sử dụng định dạng khai báo thay vì theo chương trình.Tìm hiểu về Internet Of Things (IoT)

Các bố trí thường được định nghĩa trong các hoạt động với mã gọi các phương thức khung UI. Ví dụ, mã bên dưới gọi hàm findViewById () để tìm một widget TextView và liên kết nó với thuộc tính userName của biến viewModel:

findViewById<TextView>(R.id.sample\_text).apply {  
    text = viewModel.userName  
}

Ví dụ sau đây cho thấy cách sử dụng Thư viện ràng buộc dữ liệu để gán văn bản cho tiện ích trực tiếp trong tệp bố cục. Điều này loại bỏ sự cần thiết phải gọi bất kỳ mã Java nào được hiển thị ở trên. Lưu ý việc sử dụng cú pháp @ {} trong biểu thức gán:

<TextView  
    android:text="@{viewmodel.userName}" />

Các thành phần ràng buộc trong tệp bố cục cho phép bạn xóa nhiều cuộc gọi khuôn khổ giao diện người dùng trong các hoạt động của mình, giúp chúng trở nên đơn giản và dễ bảo trì hơn. Điều này cũng có thể cải thiện hiệu suất ứng dụng của bạn và giúp ngăn chặn rò rỉ bộ nhớ và ngoại lệ con trỏ null.

### Get started

Tìm hiểu cách để môi trường phát triển của bạn sẵn sàng làm việc với Thư viện ràng buộc dữ liệu, bao gồm hỗ trợ mã ràng buộc dữ liệu trong Android Studio.

Thư viện ràng buộc dữ liệu cung cấp cả tính linh hoạt và khả năng tương thích rộng — đó là thư viện hỗ trợ, do đó bạn có thể sử dụng nó với các thiết bị chạy Android 4.0 (API cấp 14) trở lên.

Bạn nên sử dụng Plugin Android mới nhất cho Gradle trong dự án của bạn. Tuy nhiên, tính năng ràng buộc dữ liệu được hỗ trợ trên phiên bản 1.5.0 trở lên. Để biết thêm thông tin, hãy xem cách cập nhật Plugin Android cho Gradle.

#### **Xây dựng môi trường**

Để bắt đầu với ràng buộc dữ liệu, hãy tải xuống thư viện từ Kho lưu trữ hỗ trợ trong trình quản lý SDK Android. Để biết thêm thông tin, see [Update the IDE and SDK Tools](https://developer.android.com/studio/intro/update.html).

Để định cấu hình ứng dụng của bạn để sử dụng ràng buộc dữ liệu, hãy thêm phần tử dataBinding vào tệp build.gradle của bạn trong mô-đun ứng dụng, như được hiển thị trong ví dụ sau:

android {  
    ...  
    dataBinding {  
        enabled = true  
    }  
}

#### **Hỗ trợ Android Studio cho ràng buộc dữ liệu**

Android Studio hỗ trợ nhiều tính năng chỉnh sửa cho mã ràng buộc dữ liệu. Ví dụ, nó hỗ trợ các tính năng sau cho biểu thức ràng buộc dữ liệu:

* Tô sáng cú pháp
* Gắn cờ các lỗi cú pháp ngôn ngữ biểu thức
* Hoàn thành mã XML
* Tài liệu tham khảo, bao gồm điều hướng (chẳng hạn như điều hướng đến một tuyên bố) và tài liệu nhanh chóng
* Ngăn xem trước trong Layout sữa bố cục hiển thị giá trị mặc định của biểu thức ràng buộc dữ liệu, nếu được cung cấp. Ví dụ, ngăn xem trước hiển thị giá trị my\_default trên tiện ích TextView được khai báo trong ví dụ sau:

<TextView android:layout\_width="wrap\_content"  
    android:layout\_height="wrap\_content"  
    android:text="@{user.firstName, default=my\_default}"/>

Nếu bạn chỉ cần hiển thị giá trị mặc định trong giai đoạn thiết kế của dự án, bạn có thể sử dụng các thuộc tính công cụ thay vì các giá trị biểu thức mặc định, như được mô tả trong Tham chiếu thuộc tính công cụ.

1. **Trình biên dịch liên kết dữ liệu mới cho các lớp ràng buộc**

Phiên bản plugin Android Gradle 3.1.0-alpha06 bao gồm một trình biên dịch liên kết dữ liệu mới tạo ra các lớp ràng buộc. Trình biên dịch mới từng bước tạo ra các lớp ràng buộc, trong hầu hết các trường hợp, tăng tốc quá trình xây dựng. Để tìm hiểu thêm về các lớp ràng buộc, hãy xem các lớp ràng buộc được tạo ra.

Các phiên bản trước của trình biên dịch liên kết dữ liệu đã tạo ra các lớp ràng buộc trong cùng một bước để biên dịch mã được quản lý của bạn. Nếu mã được quản lý của bạn không biên dịch được, bạn có thể nhận được nhiều lỗi báo cáo rằng các lớp ràng buộc không được tìm thấy. Trình biên dịch liên kết dữ liệu mới ngăn chặn các lỗi này bằng cách tạo ra các lớp ràng buộc trước khi trình biên dịch được quản lý xây dựng ứng dụng của bạn.

Để kích hoạt trình biên dịch liên kết dữ liệu mới, hãy thêm tùy chọn sau vào tệp gradle.properties của bạn:

android.databinding.enableV2=true

You can also enable the new compiler in your gradle command by adding the following parameter:

-Pandroid.databinding.enableV2=true

Các thay đổi hành vi sau áp dụng khi bạn bật trình biên dịch liên kết dữ liệu mới:

* Plugin Android Gradle tạo các lớp ràng buộc cho bố cục của bạn trước khi biên dịch mã được quản lý của bạn.
* Nếu bố cục được bao gồm trong nhiều cấu hình tài nguyên đích, thư viện ràng buộc dữ liệu sử dụng android.view.View làm loại chế độ xem mặc định cho tất cả các chế độ xem chia sẻ cùng một id tài nguyên nhưng không xem loại.
* Các lớp ràng buộc cho các mô-đun thư viện được biên dịch và đóng gói vào tệp lưu trữ Android (AAR) tương ứng. Các mô-đun ứng dụng phụ thuộc vào các mô-đun thư viện đó không còn cần phải tạo lại các lớp ràng buộc. Để biết thêm thông tin về các tệp AAR, hãy xem Tạo Thư viện Android.
* Bộ điều hợp ràng buộc của mô-đun không còn có thể thay đổi hành vi của bộ điều hợp phụ thuộc của mô-đun. Bộ điều hợp ràng buộc chỉ ảnh hưởng đến mã trong mô-đun riêng và người tiêu dùng của mô-đun.

### **Layouts and binding expressions**

Ngôn ngữ biểu thức cho phép bạn viết các biểu thức xử lý các sự kiện được các khung nhìn gửi đi. Thư viện ràng buộc dữ liệu tự động tạo ra các lớp cần thiết để liên kết các khung nhìn trong bố trí với các đối tượng dữ liệu của bạn.

Các tệp bố cục ràng buộc dữ liệu hơi khác một chút và bắt đầu bằng thẻ bố cục gốc được theo sau bởi phần tử dữ liệu và phần tử gốc của khung nhìn. Yếu tố chế độ xem này là nội dung gốc của bạn trong tệp bố cục không ràng buộc. Đoạn mã sau cho thấy một tệp sơ đồ bố trí mẫu:

<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android">  
   <data>  
       <variable name="user" type="com.example.User"/>  
   </data>  
   <LinearLayout  
       android:orientation="vertical"  
       android:layout\_width="match\_parent"  
       android:layout\_height="match\_parent">  
       <TextView android:layout\_width="wrap\_content"  
           android:layout\_height="wrap\_content"  
           android:text="@{user.firstName}"/>  
       <TextView android:layout\_width="wrap\_content"  
           android:layout\_height="wrap\_content"  
           android:text="@{user.lastName}"/>  
   </LinearLayout>  
</layout>

biến user trong dữ liệu mô tả thuộc tính có thể được sử dụng trong layout này.

<variable name="user" type="com.example.User" />

Biểu thức trong bố cục được viết trong thuộc tính thuộc tính bằng cách sử dụng cú pháp "@ {}". Ở đây, văn bản TextView được đặt thành thuộc tính firstName của biến user:

<TextView android:layout\_width="wrap\_content"  
          android:layout\_height="wrap\_content"  
          android:text="@{user.firstName}" />

#### **Data object**

Loại đối tượng này có dữ liệu không bao giờ thay đổi. Nó là phổ biến trong các ứng dụng để có dữ liệu được đọc một lần và không bao giờ thay đổi sau đó. Cũng có thể sử dụng một đối tượng theo một tập hợp các quy ước, chẳng hạn như việc sử dụng các phương thức accessor trong Java, như trong ví dụ sau:

data class User(val firstName: String, val lastName: String)

Từ quan điểm ràng buộc dữ liệu, hai lớp này tương đương nhau. Biểu thức @ {user.firstName} được sử dụng cho thuộc tính android: text truy cập trường firstName trong lớp trước và phương thức getFirstName () trong lớp sau. Ngoài ra, nó cũng được giải quyết cho firstName () nếu phương thức đó tồn tại.

#### **Binding data**

Từ quan điểm ràng buộc dữ liệu, hai lớp này tương đương nhau. Biểu thức @ {user.firstName} được sử dụng cho thuộc tính android: text truy cập trường firstName trong lớp trước và phương thức getFirstName () trong lớp sau. Ngoài ra, nó cũng được giải quyết cho firstName () nếu phương thức đó tồn tại.

override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
  
    val binding: ActivityMainBinding = DataBindingUtil.setContentView(  
            this, R.layout.activity\_main)  
  
    binding.user = User("Test", "User")  
}

Khi chạy, ứng dụng sẽ hiển thị Kiểm tra trong giao diện người dùng. Ngoài ra, bạn có thể lấy chế độ xem bằng cách sử dụng LayoutInflater, như minh họa trong ví dụ sau:

val binding: ActivityMainBinding = ActivityMainBinding.inflate(getLayoutInflater())

Nếu bạn đang sử dụng các mục ràng buộc dữ liệu bên trong một bộ điều hợp Fragment, ListView hoặc RecyclerView, bạn có thể thích sử dụng phương thức inflate () của các lớp kết buộc hoặc lớp DataBindingUtil, như trong ví dụ mã sau:

val listItemBinding = ListItemBinding.inflate(layoutInflater, viewGroup, false)  
// or  
val listItemBinding = DataBindingUtil.inflate(layoutInflater, R.layout.list\_item, viewGroup, false)

#### **Expression language**

##### Common features

Ngôn ngữ biểu thức trông giống như các biểu thức được tìm thấy trong mã được quản lý. Bạn có thể sử dụng các toán tử và từ khóa sau đây trong ngôn ngữ biểu thức:

* Mathematical + - / \* %
* String concatenation +
* Logical && ||
* Binary & | ^
* Unary + - ! ~
* Shift >> >>> <<
* Comparison == > < >= <=
* instanceof
* Grouping ()
* Literals - character, String, numeric, null
* Cast
* Method calls
* Field access
* Array access []
* Ternary operator ?:

Examples:

* android:text="@{String.valueOf(index + 1)}"  
  android:visibility="@{age < 13 ? View.GONE : View.VISIBLE}"  
  android:transitionName='@{"image\_" + id}'

##### Missing operations

Các hoạt động sau bị thiếu từ cú pháp biểu thức mà bạn có thể sử dụng trong mã được quản lý:

* this
* super
* new
* Explicit generic invocation

##### Null coalescing operator

Toán tử kết hợp null (??) chọn toán hạng bên trái nếu nó không phải là null hoặc bên phải nếu giá trị cũ là null.

android:text="@{user.displayName ?? user.lastName}"

Đây là chức năng tương đương với:

android:text="@{user.displayName != null ? user.displayName : user.lastName}"

##### Property Reference

Một biểu thức có thể tham chiếu một thuộc tính trong một lớp bằng cách sử dụng định dạng sau, giống như đối với các trường, getters và các đối tượng ObservableField:

android:text="@{user.lastName}"

##### Avoiding null pointer exceptions

Mã ràng buộc dữ liệu được tạo tự động kiểm tra các giá trị null và tránh các ngoại lệ con trỏ null. Ví dụ, trong biểu thức @ {user.name}, nếu người dùng là null, user.name được gán giá trị mặc định của nó là null. Nếu bạn tham chiếu user.age, trong đó tuổi là kiểu int, thì ràng buộc dữ liệu sử dụng giá trị mặc định là 0.

##### Cnollectios

Các bộ sưu tập phổ biến, chẳng hạn như mảng, danh sách, danh sách thưa thớt và bản đồ, có thể được truy cập bằng toán tử [] để thuận tiện.

<data>  
    <import type="android.util.SparseArray"/>  
    <import type="java.util.Map"/>  
    <import type="java.util.List"/>  
    <variable name="list" type="List<String>"/>  
    <variable name="sparse" type="SparseArray<String>"/>  
    <variable name="map" type="Map<String, String>"/>  
    <variable name="index" type="int"/>  
    <variable name="key" type="String"/>  
</data>  
…  
android:text="@{list[index]}"  
…  
android:text="@{sparse[index]}"  
…  
android:text="@{map[key]}"

##### String literals

Bạn có thể sử dụng dấu nháy đơn để bao quanh giá trị thuộc tính, cho phép bạn sử dụng dấu ngoặc kép trong biểu thức, như trong ví dụ sau:

android:text='@{map["firstName"]}'

Cũng có thể sử dụng dấu ngoặc kép để bao quanh giá trị thuộc tính. Khi làm như vậy, chuỗi chữ nên được bao quanh với dấu ngoặc kép ':

android:text="@{map[`firstName`]}"

##### Resources

Bạn có thể truy cập tài nguyên trong một biểu thức bằng cách sử dụng cú pháp sau:

android:padding="@{large? @dimen/largePadding : @dimen/smallPadding}"

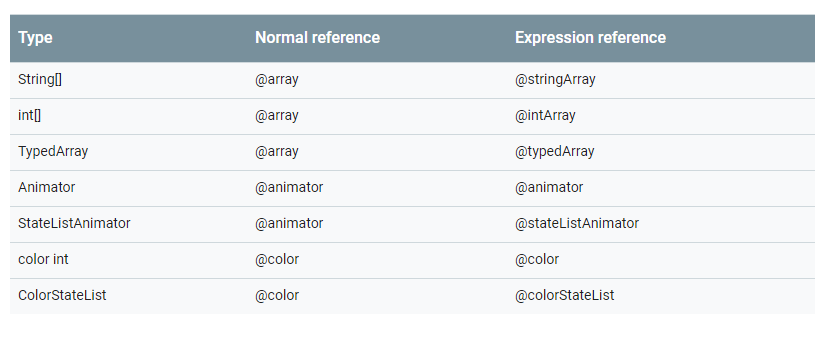
Các chuỗi định dạng và số nhiều có thể được đánh giá bằng cách cung cấp các tham số:

android:text="@{@string/nameFormat(firstName, lastName)}"  
android:text="@{@plurals/banana(bananaCount)}"

Khi một số nhiều có nhiều tham số, tất cả các thông số phải được chuyển:

Have an orange  
  Have %d oranges  
  
android:text="@{@plurals/orange(orangeCount, orangeCount)}"

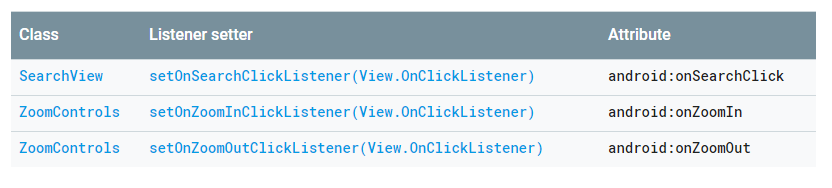
Một số tài nguyên yêu cầu đánh giá loại rõ ràng, như được hiển thị trong bảng sau:



#### **Event handling**

Ràng buộc dữ liệu cho phép bạn viết các sự kiện xử lý biểu thức được gửi đi từ các khung nhìn (ví dụ, phương thức onClick ()). Tên thuộc tính sự kiện được xác định bằng tên của phương thức trình nghe với một vài ngoại lệ. Ví dụ: View.OnClickListener có phương thức onClick (), vì vậy thuộc tính cho sự kiện này là android: onClick.

Có một số trình xử lý sự kiện chuyên biệt cho sự kiện nhấp cần một thuộc tính khác với android: onClick để tránh xung đột. Bạn có thể sử dụng các thuộc tính sau để tránh các loại xung đột sau:



Bạn có thể sử dụng các cơ chế sau để xử lý sự kiện:

* Tham chiếu phương thức: Trong các biểu thức của bạn, bạn có thể tham chiếu các phương thức phù hợp với chữ ký của phương thức nghe. Khi một biểu thức đánh giá tham chiếu phương thức, Ràng buộc dữ liệu bao bọc tham chiếu phương thức và đối tượng chủ sở hữu trong trình lắng nghe và đặt trình lắng nghe đó trên khung nhìn đích. Nếu biểu thức đánh giá là null, ràng buộc dữ liệu không tạo ra một trình lắng nghe và đặt một bộ lắng nghe rỗng thay thế.
* Các ràng buộc của trình nghe: Đây là các biểu thức lambda được đánh giá khi sự kiện xảy ra. Ràng buộc dữ liệu luôn tạo ra một người nghe, mà nó đặt trên khung nhìn. Khi sự kiện được gửi đi, người nghe sẽ đánh giá biểu thức lambda.

##### Method references

Sự kiện có thể được ràng buộc với các phương thức xử lý trực tiếp, tương tự như cách android: onClick có thể được gán cho một phương thức trong một hoạt động. Một lợi thế lớn so với thuộc tính View onClick là biểu thức được xử lý tại thời gian biên dịch, vì vậy nếu phương thức không tồn tại hoặc chữ ký của nó không chính xác, bạn nhận được một lỗi thời gian biên dịch.

Sự khác biệt chính giữa các tham chiếu phương thức và các ràng buộc của trình lắng nghe là việc thực hiện trình lắng nghe thực sự được tạo ra khi dữ liệu bị ràng buộc, không phải khi sự kiện được kích hoạt. Nếu bạn muốn đánh giá biểu thức khi sự kiện xảy ra, bạn nên sử dụng tính năng gắn listener.

Để gán một sự kiện cho trình xử lý của nó, sử dụng một biểu thức ràng buộc bình thường, với giá trị là tên phương thức để gọi. Ví dụ, hãy xem xét đối tượng dữ liệu bố cục ví dụ sau:

class MyHandlers {  
    fun onClickFriend(view: View) { ... }  
}

The binding expression can assign the click listener for a view to the onClickFriend() method, as follows:

<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android">  
   <data>  
       <variable name="handlers" type="com.example.MyHandlers"/>  
       <variable name="user" type="com.example.User"/>  
   </data>  
   <LinearLayout  
       android:orientation="vertical"  
       android:layout\_width="match\_parent"  
       android:layout\_height="match\_parent">  
       <TextView android:layout\_width="wrap\_content"  
           android:layout\_height="wrap\_content"  
           android:text="@{user.firstName}"  
           android:onClick="@{handlers::onClickFriend}"/>  
   </LinearLayout>  
</layout>

##### Listener bindings

Các ràng buộc của trình nghe là các biểu thức ràng buộc chạy khi một sự kiện xảy ra. Chúng tương tự như các tham chiếu phương thức, nhưng chúng cho phép bạn chạy các biểu thức ràng buộc dữ liệu tùy ý. Tính năng này khả dụng với Plugin Android Gradle dành cho Gradle phiên bản 2.0 trở lên.

Trong các tham chiếu phương thức, các tham số của phương thức phải khớp với các tham số của trình nghe sự kiện. Trong các ràng buộc của trình nghe, chỉ giá trị trả về của bạn phải khớp với giá trị trả về mong đợi của người nghe (trừ khi nó được mong đợi là void). Ví dụ, hãy xem xét lớp trình bày sau có phương thức onSaveClick ():

class Presenter {  
    fun onSaveClick(task: Task){}  
}

Sau đó, bạn có thể ràng buộc sự kiện bấm vào phương thức onSaveClick (), như sau:

<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android">  
    <data>  
        <variable name="task" type="com.android.example.Task" />  
        <variable name="presenter" type="com.android.example.Presenter" />  
    </data>  
    <LinearLayout android:layout\_width="match\_parent" android:layout\_height="match\_parent">  
        <Button android:layout\_width="wrap\_content" android:layout\_height="wrap\_content"  
        android:onClick="@{() -> presenter.onSaveClick(task)}" />  
    </LinearLayout>  
</layout>

Khi gọi lại được sử dụng trong một biểu thức, ràng buộc dữ liệu sẽ tự động tạo trình nghe cần thiết và đăng ký nó cho sự kiện. Khi khung nhìn kích hoạt sự kiện, ràng buộc dữ liệu sẽ đánh giá biểu thức đã cho. Như trong các biểu thức ràng buộc thông thường, bạn vẫn nhận được sự an toàn của chuỗi và dữ liệu ràng buộc trong khi các biểu thức nghe này đang được đánh giá.

Trong ví dụ trên, chúng tôi đã không xác định tham số lượt xem được chuyển đến onClick (Chế độ xem). Các liên kết Listener cung cấp hai lựa chọn cho các tham số listener: bạn có thể bỏ qua tất cả các tham số cho phương thức hoặc đặt tên tất cả chúng. Nếu bạn muốn đặt tên cho các tham số, bạn có thể sử dụng chúng trong biểu thức của bạn. Ví dụ, biểu thức trên có thể được viết như sau:

android:onClick="@{(view) -> presenter.onSaveClick(task)}"

Hoặc nếu bạn muốn sử dụng tham số trong biểu thức, nó có thể hoạt động như sau:

class Presenter {  
    fun onSaveClick(view: View, task: Task){}  
}

android:onClick="@{(theView) -> presenter.onSaveClick(theView, task)}"

Bạn có thể sử dụng biểu thức lambda với nhiều tham số:

class Presenter {  
    fun onCompletedChanged(task: Task, completed: Boolean){}  
}

<CheckBox android:layout\_width="wrap\_content" android:layout\_height="wrap\_content"  
      android:onCheckedChanged="@{(cb, isChecked) -> presenter.completeChanged(task, isChecked)}" />

Nếu sự kiện bạn đang nghe trả về một giá trị có loại không bị vô hiệu, các biểu thức của bạn cũng phải trả lại cùng một loại giá trị. Ví dụ: nếu bạn muốn nghe sự kiện nhấp chuột dài, biểu thức của bạn sẽ trả lại boolean.

class Presenter {  
    fun onLongClick(view: View, task: Task): Boolean { }  
}

android:onLongClick="@{(theView) -> presenter.onLongClick(theView, task)}"

Nếu biểu thức không thể được đánh giá do các đối tượng null, ràng buộc dữ liệu trả về giá trị mặc định cho loại đó. Ví dụ: null cho các loại tham chiếu, 0 cho int, false cho boolean, v.v.

Nếu bạn cần sử dụng một biểu thức với một vị từ (ví dụ, ternary), bạn có thể sử dụng void làm biểu tượng.

android:onClick="@{(v) -> v.isVisible() ? doSomething() : void}"

##### Avoid complex listeners

Biểu thức người nghe rất mạnh mẽ và có thể làm cho mã của bạn rất dễ đọc. Mặt khác, người nghe có chứa các biểu thức phức tạp làm cho bố cục của bạn khó đọc và duy trì. Các biểu thức này phải đơn giản như truyền dữ liệu sẵn có từ giao diện người dùng đến phương thức gọi lại của bạn. Bạn nên triển khai bất kỳ logic nghiệp vụ nào bên trong phương thức gọi lại mà bạn đã gọi ra từ biểu thức người nghe.

#### Imports, variables, and includes

Thư viện ràng buộc dữ liệu cung cấp các tính năng như nhập, biến và bao gồm. Nhập khẩu giúp dễ dàng tham khảo các lớp bên trong tệp bố cục của bạn. Các biến cho phép bạn mô tả một thuộc tính có thể được sử dụng trong các biểu thức ràng buộc. Bao gồm cho phép bạn sử dụng lại bố cục phức tạp trên ứng dụng của mình.

##### Imports

Nhập khẩu cho phép bạn dễ dàng tham chiếu các lớp bên trong tệp bố cục của bạn, giống như trong mã được quản lý. Không thể sử dụng các phần tử nhập hoặc nhiều hơn trong phần tử dữ liệu. Ví dụ mã sau nhập khẩu lớp View vào tệp layout:

<data>  
    <import type="android.view.View"/>  
</data>

Việc nhập lớp Chế độ xem cho phép bạn tham chiếu nó từ các biểu thức ràng buộc của bạn. Ví dụ sau đây cho thấy cách tham chiếu các hằng số VISIBLE và GONE của lớp View:

<TextView  
   android:text="@{user.lastName}"  
   android:layout\_width="wrap\_content"  
   android:layout\_height="wrap\_content"  
   android:visibility="@{user.isAdult ? View.VISIBLE : View.GONE}"/>

##### Type aliases

Khi có xung đột tên lớp, một trong các lớp có thể được đổi tên thành bí danh. Ví dụ sau đổi tên lớp View trong gói com.example.real.estate thành Vista:

<import type="android.view.View"/>  
<import type="com.example.real.estate.View"  
        alias="Vista"/>

Bạn có thể sử dụng Vista để tham chiếu com.example.real.estate.View và View có thể được sử dụng để tham khảo android.view.View bên trong tệp bố cục.

##### Import other classes

Các loại được nhập có thể được sử dụng làm tham chiếu kiểu trong các biến và biểu thức. Ví dụ sau cho thấy Người dùng và Danh sách được sử dụng làm loại biến:

<data>  
    <import type="com.example.User"/>  
    <import type="java.util.List"/>  
    <variable name="user" type="User"/>  
    <variable name="userList" type="List<User>"/>  
</data>

Bạn cũng có thể sử dụng các loại đã nhập để truyền một phần biểu thức. Ví dụ sau mô tả thuộc tính kết nối cho một kiểu người dùng:

<TextView  
   android:text="@{((User)(user.connection)).lastName}"  
   android:layout\_width="wrap\_content"  
   android:layout\_height="wrap\_content"/>

Các loại được nhập cũng có thể được sử dụng khi tham chiếu các trường tĩnh và các phương thức trong các biểu thức. Đoạn mã sau nhập khẩu lớp MyStringUtils và tham chiếu phương thức viết hoa của nó:

<data>  
    <import type="com.example.MyStringUtils"/>  
    <variable name="user" type="com.example.User"/>  
</data>  
…  
<TextView  
   android:text="@{MyStringUtils.capitalize(user.lastName)}"  
   android:layout\_width="wrap\_content"  
   android:layout\_height="wrap\_content"/>

Cũng giống như trong mã được quản lý, java.lang. \* Được nhập tự động.

##### Variables

Bạn có thể sử dụng nhiều phần tử biến bên trong phần tử dữ liệu. Mỗi phần tử biến mô tả một thuộc tính có thể được đặt trên bố trí được sử dụng trong các biểu thức ràng buộc trong tệp bố cục. Ví dụ sau tuyên bố biến người dùng, hình ảnh và ghi chú:

<data>  
    <import type="android.graphics.drawable.Drawable"/>  
    <variable name="user" type="com.example.User"/>  
    <variable name="image" type="Drawable"/>  
    <variable name="note" type="String"/>  
</data>

Các loại biến được kiểm tra tại thời gian biên dịch, vì vậy nếu một biến thực hiện quan sát hoặc là một tập hợp quan sát được, cần được phản ánh trong loại. Nếu biến là lớp cơ sở hoặc giao diện không triển khai giao diện Quan sát, thì các biến không được quan sát.

Khi có các tệp bố cục khác nhau cho các cấu hình khác nhau (ví dụ: ngang hoặc dọc), các biến được kết hợp. Không được có xung đột định nghĩa biến giữa các tệp bố cục này.

Lớp ràng buộc được tạo ra có một setter và getter cho mỗi biến được mô tả. Các biến lấy giá trị mã được quản lý mặc định cho đến khi setter được gọi là - null cho kiểu tham chiếu, 0 cho int, false cho boolean, v.v.

Một biến đặc biệt có tên là bối cảnh được tạo ra để sử dụng trong các biểu thức ràng buộc khi cần thiết. Giá trị của ngữ cảnh là đối tượng Context từ phương thức getContext () của khung nhìn gốc. Biến ngữ cảnh bị ghi đè bởi một khai báo biến rõ ràng với tên đó.

##### Includes

Các biến có thể được chuyển thành ràng buộc của bố cục bao gồm từ bố cục chứa bằng cách sử dụng không gian tên ứng dụng và tên biến trong thuộc tính. Ví dụ sau đây cho thấy các biến người dùng được bao gồm từ các tệp bố cục name.xml và tệp contact.xml:

<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android"  
        xmlns:bind="http://schemas.android.com/apk/res-auto">  
   <data>  
       <variable name="user" type="com.example.User"/>  
   </data>  
   <LinearLayout  
       android:orientation="vertical"  
       android:layout\_width="match\_parent"  
       android:layout\_height="match\_parent">  
       <include layout="@layout/name"  
           bind:user="@{user}"/>  
       <include layout="@layout/contact"  
           bind:user="@{user}"/>  
   </LinearLayout>  
</layout>

Ràng buộc dữ liệu không hỗ trợ bao gồm dưới dạng con trực tiếp của phần tử hợp nhất. Ví dụ: bố cục sau không được hỗ trợ:

<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android"  
        xmlns:bind="http://schemas.android.com/apk/res-auto">  
   <data>  
       <variable name="user" type="com.example.User"/>  
   </data>  
   <merge><!-- Doesn't work -->  
       <include layout="@layout/name"  
           bind:user="@{user}"/>  
       <include layout="@layout/contact"  
           bind:user="@{user}"/>  
   </merge>  
</layout>

### **Work with observable data objects**

Khả năng quan sát đề cập đến khả năng của một đối tượng để thông báo cho người khác về những thay đổi trong dữ liệu của nó. Thư viện ràng buộc dữ liệu cho phép bạn tạo các đối tượng, trường hoặc bộ sưu tập có thể quan sát được.

Bất kỳ đối tượng đồng bằng cũ nào cũng có thể được sử dụng để liên kết dữ liệu, nhưng việc sửa đổi đối tượng không tự động khiến giao diện người dùng cập nhật. Ràng buộc dữ liệu có thể được sử dụng để cung cấp cho các đối tượng dữ liệu của bạn khả năng thông báo cho các đối tượng khác, được gọi là người nghe, khi dữ liệu của nó thay đổi. Có ba loại lớp quan sát khác nhau: đối tượng, trường và bộ sưu tập.

Khi một trong các đối tượng dữ liệu quan sát được ràng buộc với giao diện người dùng và thuộc tính của các thay đổi đối tượng dữ liệu, giao diện người dùng được cập nhật tự động.

#### **Observable fields**

Một số công việc có liên quan đến việc tạo các lớp thực hiện giao diện Quan sát được, điều này không thể đáng giá nếu bạn chỉ có một vài thuộc tính. Trong trường hợp này, bạn có thể sử dụng lớp Quan sát chung và các lớp cụ thể nguyên thủy sau đây để làm cho các trường có thể quan sát được:

* [ObservableBoolean](https://developer.android.com/reference/android/databinding/ObservableBoolean.html)
* [ObservableByte](https://developer.android.com/reference/android/databinding/ObservableByte.html)
* [ObservableChar](https://developer.android.com/reference/android/databinding/ObservableChar.html)
* [ObservableShort](https://developer.android.com/reference/android/databinding/ObservableShort.html)
* [ObservableInt](https://developer.android.com/reference/android/databinding/ObservableInt.html)
* [ObservableLong](https://developer.android.com/reference/android/databinding/ObservableLong.html)
* [ObservableFloat](https://developer.android.com/reference/android/databinding/ObservableFloat.html)
* [ObservableDouble](https://developer.android.com/reference/android/databinding/ObservableDouble.html)
* [ObservableParcelable](https://developer.android.com/reference/android/databinding/ObservableParcelable.html)

Trường có thể quan sát là các đối tượng có thể quan sát được có một trường duy nhất. Các phiên bản nguyên thủy tránh boxing và unboxing trong các hoạt động truy cập. Để sử dụng cơ chế này, hãy tạo thuộc tính chung cuối cùng trong ngôn ngữ lập trình Java hoặc thuộc tính chỉ đọc trong Kotlin, như trong ví dụ sau:

class User {  
    val firstName = ObservableField<String>()  
    val lastName = ObservableField<String>()  
    val age = ObservableInt()  
}

Để truy cập giá trị trường, sử dụng phương thức accessor set () và get (), như sau:

user.firstName = "Google"  
val age = user.age

#### **Observable collections**

Một số ứng dụng sử dụng cấu trúc động để giữ dữ liệu. Các bộ sưu tập quan sát cho phép truy cập vào các cấu trúc này bằng cách sử dụng một khóa. Lớp ObservableArrayMap hữu ích khi khóa là một kiểu tham chiếu, chẳng hạn như String, như trong ví dụ sau:

ObservableArrayMap<String, Any>().apply {  
    put("firstName", "Google")  
    put("lastName", "Inc.")  
    put("age", 17)  
}

Trong bố trí, bản đồ có thể được tìm thấy bằng cách sử dụng các phím chuỗi, như sau:

<data>  
    <import type="android.databinding.ObservableMap"/>  
    <variable name="user" type="ObservableMap<String, Object>"/>  
</data>  
…  
<TextView  
    android:text="@{user.lastName}"  
    android:layout\_width="wrap\_content"  
    android:layout\_height="wrap\_content"/>  
<TextView  
    android:text="@{String.valueOf(1 + (Integer)user.age)}"  
    android:layout\_width="wrap\_content"  
    android:layout\_height="wrap\_content"/>

Lớp ObservableArrayList hữu ích khi khóa là một số nguyên, như sau:

ObservableArrayList<Any>().apply {  
    add("Google")  
    add("Inc.")  
    add(17)  
}

Trong bố trí, danh sách có thể được truy cập thông qua các chỉ mục, như trong ví dụ sau:

<data>  
    <import type="android.databinding.ObservableList"/>  
    <import type="com.example.my.app.Fields"/>  
    <variable name="user" type="ObservableList<Object>"/>  
</data>  
…  
<TextView  
    android:text='@{user[Fields.LAST\_NAME]}'  
    android:layout\_width="wrap\_content"  
    android:layout\_height="wrap\_content"/>  
<TextView  
    android:text='@{String.valueOf(1 + (Integer)user[Fields.AGE])}'  
    android:layout\_width="wrap\_content"  
    android:layout\_height="wrap\_content"/>

#### **Observable objects**

class User : BaseObservable() {  
  
    @get:Bindable  
    var firstName: String = ""  
        set(value) {  
            field = value  
            notifyPropertyChanged(BR.firstName)  
        }  
  
    @get:Bindable  
    var lastName: String = ""  
        set(value) {  
            field = value  
            notifyPropertyChanged(BR.lastName)  
        }  
}

Ràng buộc dữ liệu tạo ra một lớp có tên là BR trong gói mô-đun chứa các ID của các tài nguyên được sử dụng để gắn kết dữ liệu. Chú thích Bindable tạo ra một mục trong tệp lớp BR trong khi biên dịch. Nếu lớp cơ sở cho các lớp dữ liệu không thể thay đổi, giao diện quan sát có thể được thực hiện bằng cách sử dụng một đối tượng PropertyChangeRegistry để đăng ký và thông báo cho người nghe một cách hiệu quả.

### **Generated binding classes**

Thư viện ràng buộc dữ liệu tạo ra các lớp ràng buộc được sử dụng để truy cập các biến và khung nhìn của bố cục. Trang này cho thấy cách tạo và tùy chỉnh các lớp ràng buộc được tạo ra.

Lớp ràng buộc được tạo ra liên kết các biến bố cục với các khung nhìn trong bố trí. Tên và gói của lớp ràng buộc có thể được tùy chỉnh. Tất cả các lớp ràng buộc được tạo ra kế thừa từ lớp ViewDataBinding.

Một lớp ràng buộc được tạo cho mỗi tệp bố cục. Theo mặc định, tên của lớp được dựa trên tên của tệp sơ đồ bố trí, chuyển đổi nó thành trường hợp Pascal và thêm hậu tố Ràng buộc vào nó. Tên tệp bố cục ở trên là activity\_main.xml do đó lớp được tạo tương ứng là ActivityMainBinding. Lớp này chứa tất cả các ràng buộc từ các thuộc tính bố trí (ví dụ, biến người dùng) đến các khung nhìn của bố cục và biết cách gán các giá trị cho các biểu thức ràng buộc.

#### **Create a binding object**

Đối tượng ràng buộc sẽ được tạo ngay sau khi tăng bố cục để đảm bảo rằng hệ thống phân cấp khung nhìn không được sửa đổi trước khi nó liên kết với các khung nhìn với các biểu thức trong bố trí. Phương pháp phổ biến nhất để liên kết đối tượng với bố cục là sử dụng các phương thức tĩnh trên lớp ràng buộc. Bạn có thể thổi phồng hệ thống phân cấp khung nhìn và liên kết đối tượng với nó bằng cách sử dụng phương thức inflate () của lớp ràng buộc, như trong ví dụ sau:

override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
  
    val binding: MyLayoutBinding = MyLayoutBinding.inflate(layoutInflater)  
}

Có một phiên bản thay thế của phương thức inflate () lấy đối tượng ViewGroup ngoài đối tượng LayoutInflater, như trong ví dụ sau:

val binding: MyLayoutBinding = MyLayoutBinding.inflate(getLayoutInflater(), viewGroup, false)

Nếu bố cục được tăng lên bằng cách sử dụng một cơ chế khác, nó có thể bị ràng buộc riêng biệt, như sau:

val binding: MyLayoutBinding = MyLayoutBinding.bind(viewRoot)

Sometimes the binding type cannot be known in advance. In such cases, the binding can be created using the[DataBindingUtil](https://developer.android.com/reference/android/databinding/DataBindingUtil.html) class, as demonstrated in the following code snippet:

val rootView = LayoutInflater.from(this).inflate(layoutId, parent, attachToParent)  
val binding: ViewDataBinding? = DataBindingUtil.bind(viewRoot)

Nếu bạn đang sử dụng các mục ràng buộc dữ liệu bên trong một bộ điều hợp Fragment, ListView hoặc RecyclerView, bạn có thể thích sử dụng phương thức inflate () của các lớp kết buộc hoặc lớp DataBindingUtil, như trong ví dụ mã sau:

val listItemBinding = ListItemBinding.inflate(layoutInflater, viewGroup, false)  
// or  
val listItemBinding = DataBindingUtil.inflate(layoutInflater, R.layout.list\_item, viewGroup, false)

#### **Views with IDs**

Thư viện ràng buộc dữ liệu tạo ra một trường bất biến trong lớp ràng buộc cho mỗi chế độ xem có ID trong bố cục. Ví dụ, Thư viện ràng buộc dữ liệu tạo ra các trường firstName và lastName của kiểu TextView từ layout sau:

<layout xmlns:android="http://schemas.android.com/apk/res/android">  
   <data>  
       <variable name="user" type="com.example.User"/>  
   </data>  
   <LinearLayout  
       android:orientation="vertical"  
       android:layout\_width="match\_parent"  
       android:layout\_height="match\_parent">  
       <TextView android:layout\_width="wrap\_content"  
           android:layout\_height="wrap\_content"  
           android:text="@{user.firstName}"  
   android:id="@+id/firstName"/>  
       <TextView android:layout\_width="wrap\_content"  
           android:layout\_height="wrap\_content"  
           android:text="@{user.lastName}"  
  android:id="@+id/lastName"/>  
   </LinearLayout>  
</layout>

Thư viện trích xuất các khung nhìn bao gồm các ID từ phân cấp khung nhìn trong một lần truyền. Cơ chế này có thể nhanh hơn gọi phương thức findViewById () cho mỗi khung nhìn trong layout.

Các ID không cần thiết vì chúng không có ràng buộc dữ liệu, nhưng vẫn còn một số trường hợp truy cập vào các khung nhìn vẫn cần thiết từ mã.

#### **Variables**

Thư viện Binding dữ liệu tạo ra các phương thức accessor cho mỗi biến được khai báo trong layout. Ví dụ, bố cục sau tạo ra các phương thức setter và getter trong lớp liên kết cho các biến người dùng, hình ảnh và ghi chú:

<data>  
   <import type="android.graphics.drawable.Drawable"/>  
   <variable name="user" type="com.example.User"/>  
   <variable name="image" type="Drawable"/>  
   <variable name="note" type="String"/>  
</data>

#### **ViewStubs**

Không giống như chế độ xem thông thường, các đối tượng ViewStub bắt đầu dưới dạng chế độ xem ẩn. Khi chúng được hiển thị hoặc được nói rõ ràng để thổi phồng, chúng thay thế chính chúng trong bố cục bằng cách tăng bố cục khác.

Vì ViewStub về cơ bản biến mất khỏi hệ thống phân cấp khung nhìn, nên khung nhìn trong đối tượng ràng buộc cũng phải biến mất để cho phép được yêu cầu bởi bộ sưu tập rác. Bởi vì các khung nhìn là cuối cùng, một đối tượng ViewStubProxy chiếm vị trí của ViewStub trong lớp ràng buộc được tạo ra, cho phép bạn truy cập vào ViewStub khi nó tồn tại và cũng có thể truy cập vào hệ thống phân cấp xem tăng cao khi ViewStub bị thổi phồng.

Khi thổi phồng một bố cục khác, một ràng buộc phải được thiết lập cho bố cục mới. Vì vậy, ViewStubProxy phải lắng nghe ViewStub OnInflateListener và thiết lập các ràng buộc khi cần thiết. Vì chỉ có một người nghe có thể tồn tại tại một thời điểm nhất định, ViewStubProxy cho phép bạn thiết lập một OnInflateListener, mà nó gọi sau khi thiết lập ràng buộc.

#### **Immediate Binding**

Khi một biến hoặc đối tượng quan sát thay đổi, ràng buộc được lên lịch để thay đổi trước khung tiếp theo. Tuy nhiên, có những lúc, khi ràng buộc phải được thực thi ngay lập tức. Để bắt buộc thực hiện, sử dụng phương thức executePendingBindings ().

#### **Advanced Binding**

##### Dynamic Variables

Đôi khi, lớp ràng buộc cụ thể không được biết. Ví dụ, một hoạt động RecyclerView.Adapter chống lại các bố trí tùy ý không biết lớp ràng buộc cụ thể. Nó vẫn phải gán giá trị ràng buộc trong khi gọi hàm onBindViewHolder ().

Trong ví dụ sau, tất cả các bố cục mà RecyclerView liên kết để có một biến mục. Đối tượng BindingHolder có một phương thức getBinding () trả về lớp cơ sở ViewDataBinding.

override fun onBindViewHolder(holder: BindingHolder, position: Int) {  
    item: T = mItems.get(position)  
    holder.binding.setVariable(BR.item, item);  
    holder.binding.executePendingBindings();  
}

#### **Background Thread**

Bạn có thể thay đổi mô hình dữ liệu của mình trong một chuỗi nền miễn là nó không phải là một bộ sưu tập. Ràng buộc dữ liệu bản địa hóa từng biến / trường trong quá trình đánh giá để tránh bất kỳ vấn đề tương tranh nào.

#### **Custom binding class names**

Theo mặc định, một lớp ràng buộc được tạo ra dựa trên tên của tệp sơ đồ bố trí, bắt đầu bằng một chữ hoa, xóa dấu gạch dưới (\_), viết hoa chữ cái sau và điền vào từ Ràng buộc. Lớp được đặt trong gói databinding trong gói module. Ví dụ, tệp layout file contact\_item.xml tạo lớp ContactItemBinding. Nếu gói mô-đun là com.example.my.app, thì lớp ràng buộc được đặt trong gói com.example.my.app.databinding.

Các lớp ràng buộc có thể được đổi tên hoặc đặt trong các gói khác nhau bằng cách điều chỉnh thuộc tính lớp của phần tử dữ liệu. Ví dụ, bố cục sau tạo lớp liên kết ContactItem trong gói databinding trong mô-đun hiện tại:

<data class="ContactItem">  
    …  
</data>

Bạn có thể tạo lớp ràng buộc trong một gói khác bằng cách đặt trước tên lớp với một dấu chấm. Ví dụ sau tạo lớp liên kết trong gói module:

<data class=".ContactItem">  
    …  
</data>

Bạn cũng có thể sử dụng tên gói đầy đủ nơi bạn muốn tạo lớp ràng buộc. Ví dụ sau tạo lớp liên kết ContactItem trong gói com.example:

<data class="com.example.ContactItem">  
    …  
</data>

### **Binding adapters**

Các bộ điều hợp ràng buộc chịu trách nhiệm thực hiện các cuộc gọi khung thích hợp để đặt các giá trị. Một ví dụ là thiết lập một giá trị thuộc tính như gọi phương thức setText (). Một ví dụ khác là thiết lập một trình lắng nghe sự kiện như gọi phương thức setOnClickListener ().

Thư viện ràng buộc dữ liệu cho phép bạn chỉ định phương thức được gọi để đặt giá trị, cung cấp logic ràng buộc của riêng bạn và chỉ định loại đối tượng được trả về bằng cách sử dụng bộ điều hợp.

#### **Setting attribute values**

Bất cứ khi nào một giá trị ràng buộc thay đổi, lớp ràng buộc được tạo ra phải gọi một phương thức setter trên khung nhìn với biểu thức ràng buộc. Bạn có thể cho phép Thư viện ràng buộc dữ liệu tự động xác định phương thức, khai báo rõ ràng phương thức hoặc cung cấp logic tùy chỉnh để chọn phương thức.

##### Automatic method selection

Đối với một thuộc tính có tên là ví dụ, thư viện sẽ tự động cố gắng tìm phương thức setExample (arg) chấp nhận các kiểu tương thích làm đối số. Không gian tên của thuộc tính không được xem xét, chỉ có tên thuộc tính và kiểu được sử dụng khi tìm kiếm một phương thức.

Ví dụ, với biểu thức android: text = "@ {user.name}", thư viện tìm kiếm một phương thức setText (arg) chấp nhận kiểu được trả về bởi user.getName (). Nếu kiểu trả về của user.getName () là String, thư viện sẽ tìm một phương thức setText () chấp nhận một đối số String. Nếu biểu thức trả về một int thay vào đó, thư viện sẽ tìm kiếm một phương thức setText () chấp nhận một đối số int. Biểu thức phải trả về kiểu đúng, bạn có thể truyền giá trị trả về nếu cần.

Liên kết dữ liệu hoạt động ngay cả khi không có thuộc tính nào tồn tại với tên đã cho. Sau đó, bạn có thể tạo các thuộc tính cho bất kỳ setter nào bằng cách sử dụng ràng buộc dữ liệu. Ví dụ, lớp hỗ trợ DrawerLayout không có bất kỳ thuộc tính nào, nhưng có rất nhiều người định cư. Bố trí sau đây sẽ tự động sử dụng phương thức setScrimColor (int) và setDrawerListener (DrawerListener) làm bộ setter cho ứng dụng: scrimColor và ứng dụng: thuộc tính drawerListener, tương ứng:

<android.support.v4.widget.DrawerLayout  
    android:layout\_width="wrap\_content"  
    android:layout\_height="wrap\_content"  
    app:scrimColor="@{@color/scrim}"  
    app:drawerListener="@{fragment.drawerListener}">

##### Specify a custom method name

Một số thuộc tính có các trình cài đặt không khớp với tên. Trong những tình huống này, một thuộc tính có thể được liên kết với bộ setter bằng cách sử dụng chú thích BindingMethods. Chú thích được sử dụng với một lớp và có thể chứa nhiều chú thích BindingMethod, một chú thích cho mỗi phương thức được đổi tên. Các phương thức ràng buộc là các chú thích có thể được thêm vào bất kỳ lớp nào trong ứng dụng của bạn. Trong ví dụ sau, thuộc tính android: tint được liên kết với phương thức setImageTintList (ColorStateList), không phải với phương thức setTint ():

@BindingMethods(value = [  
    BindingMethod(  
        type = android.widget.ImageView::class,  
        attribute = "android:tint",  
        method = "setImageTintList")])

Hầu hết thời gian, bạn không cần phải đổi tên người định cư trong các lớp khung công tác Android. Các thuộc tính đã được triển khai bằng cách sử dụng quy ước tên để tự động tìm các phương thức khớp.

##### Provide custom logic

Một số thuộc tính cần logic ràng buộc tùy chỉnh. Ví dụ, không có setter liên quan cho thuộc tính android: paddingLeft. Thay vào đó, phương thức setPadding (trái, trên, phải, dưới) được cung cấp. Một phương thức adapter ràng buộc tĩnh với chú thích BindingAdapter cho phép bạn tùy chỉnh cách một setter cho một thuộc tính được gọi.

Các thuộc tính của các lớp khung công tác Android đã được tạo chú thích BindingAdapter. Ví dụ, ví dụ sau đây cho thấy bộ điều hợp ràng buộc cho thuộc tính paddingLeft:

@BindingAdapter("android:paddingLeft")  
fun setPaddingLeft(view: View, padding: Int) {  
    view.setPadding(padding,  
                view.getPaddingTop(),  
                view.getPaddingRight(),  
                view.getPaddingBottom())  
}

Các kiểu tham số là quan trọng. Tham số đầu tiên xác định loại chế độ xem được liên kết với thuộc tính. Tham số thứ hai xác định loại được chấp nhận trong biểu thức ràng buộc cho thuộc tính đã cho.

Bộ điều hợp ràng buộc rất hữu ích cho các loại tùy chỉnh khác. Ví dụ: trình tải tùy chỉnh có thể được gọi từ chuỗi công nhân để tải hình ảnh.

Các bộ điều hợp ràng buộc mà bạn định nghĩa sẽ ghi đè các bộ điều hợp mặc định do khung công tác Android cung cấp khi có xung đột.

Bạn cũng có thể có bộ điều hợp nhận nhiều thuộc tính, như được hiển thị trong ví dụ sau:

@BindingAdapter("imageUrl", "error")  
fun loadImage(view: ImageView, url: String, error: Drawable) {  
    Picasso.get().load(url).error(error).into(view)  
}

Bạn có thể sử dụng bộ điều hợp trong bố cục của bạn như trong ví dụ sau. Lưu ý rằng @ drawable / venueError đề cập đến tài nguyên trong ứng dụng của bạn. Xung quanh tài nguyên với @ {} làm cho nó trở thành một biểu thức ràng buộc hợp lệ.

<ImageView app:imageUrl="@{venue.imageUrl}" app:error="@{@drawable/venueError}" />

Bộ điều hợp được gọi nếu cả hai imageUrl và lỗi được sử dụng cho một đối tượng ImageView và imageUrl là một chuỗi và lỗi là một Drawable. Nếu bạn muốn bộ điều hợp được gọi khi bất kỳ thuộc tính nào được đặt, bạn có thể đặt cờ requireAll tùy chọn của bộ điều hợp thành sai, như được hiển thị trong ví dụ sau:

@BindingAdapter(value = ["imageUrl", "placeholder"], requireAll = false)  
fun setImageUrl(imageView: ImageView, url: String, placeHolder: Drawable) {  
    if (url == null) {  
        imageView.setImageDrawable(placeholder);  
    } else {  
        MyImageLoader.loadInto(imageView, url, placeholder);  
    }  
}

Các phương pháp bộ điều hợp ràng buộc có thể tùy ý lấy các giá trị cũ trong các trình xử lý của chúng. Phương pháp lấy giá trị cũ và mới phải khai báo tất cả các giá trị cũ cho các thuộc tính trước, sau đó là các giá trị mới, như được hiển thị trong ví dụ bên dưới:

@BindingAdapter("android:paddingLeft")  
fun setPaddingLeft(view: View, oldPadding: Int, newPadding: Int) {  
    if (oldPadding != newPadding) {  
        view.setPadding(padding,  
                    view.getPaddingTop(),  
                    view.getPaddingRight(),  
                    view.getPaddingBottom())  
    }  
}

Trình xử lý sự kiện chỉ có thể được sử dụng với các giao diện hoặc các lớp trừu tượng với một phương thức trừu tượng, như trong ví dụ sau:

@BindingAdapter("android:onLayoutChange")  
fun setOnLayoutChangeListener(  
        view: View,  
        oldValue: View.OnLayoutChangeListener?,  
        newValue: View.OnLayoutChangeListener?  
) {  
    if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.HONEYCOMB) {  
        if (oldValue != null) {  
            view.removeOnLayoutChangeListener(oldValue)  
        }  
        if (newValue != null) {  
            view.addOnLayoutChangeListener(newValue)  
        }  
    }  
}

Sử dụng trình xử lý sự kiện này trong bố cục của bạn như sau:

<View android:onLayoutChange="@{() -> handler.layoutChanged()}"/>

Khi một người nghe có nhiều phương thức, nó phải được chia thành nhiều người nghe. Ví dụ, View.OnAttachStateChangeListener có hai phương thức: onViewAttachedToWindow (View) và onViewDetachedFromWindow (View). Thư viện cung cấp hai giao diện để phân biệt các thuộc tính và trình xử lý cho chúng:

/ Translation from provided interfaces in Java:  
@TargetApi(Build.VERSION\_CODES.HONEYCOMB\_MR1)  
interface OnViewDetachedFromWindow {  
    fun onViewDetachedFromWindow(v: View)  
}  
  
@TargetApi(Build.VERSION\_CODES.HONEYCOMB\_MR1)  
interface OnViewAttachedToWindow {  
    fun onViewAttachedToWindow(v: View)  
}

Bởi vì việc thay đổi một người nghe cũng có thể ảnh hưởng đến người nghe khác, bạn cần một bộ điều hợp hoạt động cho cả thuộc tính hoặc cho cả hai. Bạn có thể đặt requireAll thành false trong chú thích để chỉ định rằng không phải mọi thuộc tính đều phải được gán một biểu thức ràng buộc, như trong ví dụ sau:

@BindingAdapter(  
        "android:onViewDetachedFromWindow",  
        "android:onViewAttachedToWindow",  
        requireAll = false  
)  
fun setListener(view: View, detach: OnViewDetachedFromWindow?, attach: OnViewAttachedToWindow?) {  
    if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.HONEYCOMB\_MR1) {  
        val newListener: View.OnAttachStateChangeListener?  
        newListener = if (detach == null && attach == null) {  
            null  
        } else {  
            object : View.OnAttachStateChangeListener {  
                override fun onViewAttachedToWindow(v: View) {  
                    attach?.onViewAttachedToWindow(v)  
                }  
  
                override fun onViewDetachedFromWindow(v: View) {  
                    detach?.onViewDetachedFromWindow(v)  
                }  
            }  
        }  
  
        val oldListener: View.OnAttachStateChangeListener? =  
                ListenerUtil.trackListener(view, newListener, R.id.onAttachStateChangeListener)  
        if (oldListener != null) {  
            view.removeOnAttachStateChangeListener(oldListener)  
        }  
        if (newListener != null) {  
            view.addOnAttachStateChangeListener(newListener)  
        }  
    }  
}

Ví dụ trên hơi phức tạp hơn bình thường vì lớp View sử dụng phương thức addOnAttachStateChangeListener () và removeOnAttachStateChangeListener () thay vì phương thức setter cho OnAttachStateChangeListener. Lớp android.databinding.adapters.ListenerUtil giúp theo dõi các trình lắng nghe trước đó để chúng có thể bị loại bỏ trong bộ điều hợp ràng buộc.

Bằng cách chú thích các giao diện OnViewDetachedFromWindow và OnViewAttachedToWindow với @TargetApi (VERSION\_CODES.HONEYCOMB\_MR1), trình tạo mã ràng buộc dữ liệu biết rằng người nghe chỉ nên được tạo khi chạy trên Android 3.1 (API level 12) và cao hơn, cùng phiên bản được addOnAttachStateChangeListener hỗ trợ ( ) phương pháp.

#### **Object conversions**

##### Automatic object conversion

Khi một đối tượng được trả về từ một biểu thức ràng buộc, thư viện chọn phương thức được sử dụng để thiết lập giá trị của thuộc tính. Đối tượng được truyền đến một kiểu tham số của phương thức đã chọn. Hành vi này thuận tiện trong các ứng dụng sử dụng lớp ObservableMap để lưu trữ dữ liệu, như trong ví dụ sau:

<TextView  
   android:text='@{userMap["lastName"]}'  
   android:layout\_width="wrap\_content"  
   android:layout\_height="wrap\_content" />

Đối tượng userMap trong biểu thức trả về một giá trị, được tự động truyền vào kiểu tham số được tìm thấy trong phương thức setText (CharSequence) được sử dụng để đặt giá trị của thuộc tính android: text. Nếu kiểu tham số không rõ ràng, bạn phải bỏ kiểu trả về trong biểu thức.

##### Custom conversions

Trong một số trường hợp, yêu cầu chuyển đổi tùy chỉnh giữa các loại cụ thể. Ví dụ: thuộc tính android: background của chế độ xem dự kiến có thể Vẽ, nhưng giá trị màu được chỉ định là số nguyên. Ví dụ sau đây cho thấy một thuộc tính mong đợi một Drawable, nhưng một số nguyên được cung cấp thay vào đó:

<View  
   android:background="@{isError ? @color/red : @color/white}"  
   android:layout\_width="wrap\_content"  
   android:layout\_height="wrap\_content"/>

Bất cứ khi nào một Drawable được mong đợi và một số nguyên được trả về, int sẽ được chuyển đổi thành một ColorDrawable. Việc chuyển đổi có thể được thực hiện bằng cách sử dụng một phương thức tĩnh với chú thích BindingConversion, như sau:

@BindingConversion  
fun convertColorToDrawable(color: Int) = ColorDrawable(color)

Tuy nhiên, các loại giá trị được cung cấp trong biểu thức ràng buộc phải nhất quán. Bạn không thể sử dụng các loại khác nhau trong cùng một biểu thức, như được hiển thị trong ví dụ sau:

<View  
   android:background="@{isError ? @drawable/error : @color/white}"  
   android:layout\_width="wrap\_content"  
   android:layout\_height="wrap\_content"/>

### **Bind layout views to Architecture Components**

Thư viện AndroidX bao gồm các thành phần kiến ​​trúc mà bạn có thể sử dụng để thiết kế các ứng dụng mạnh mẽ, có thể kiểm tra và bảo trì. Thư viện ràng buộc dữ liệu hoạt động liền mạch với các thành phần kiến ​​trúc để đơn giản hóa hơn nữa sự phát triển của giao diện người dùng của bạn. Bố cục trong ứng dụng của bạn có thể liên kết với dữ liệu trong Cấu phần Kiến trúc, đã giúp bạn quản lý vòng đời bộ điều khiển giao diện người dùng và thông báo về những thay đổi trong dữ liệu.

#### **Use LiveData to notify the UI about data changes**

Sử dụng LiveData để thông báo cho UI về thay đổi dữ liệu

Bạn có thể sử dụng các đối tượng LiveData làm nguồn ràng buộc dữ liệu để tự động thông báo cho UI về những thay đổi trong dữ liệu. Để biết thêm thông tin về Cấu phần Kiến trúc này, hãy xem Tổng quan về LiveData.

Không giống như các đối tượng thực hiện Quan sát — chẳng hạn như các trường có thể quan sát — các đối tượng LiveData biết về vòng đời của các nhà quan sát đã đăng ký với các thay đổi dữ liệu. Kiến thức này cho phép nhiều lợi ích, được giải thích trong Những lợi thế của việc sử dụng LiveData. Trong Android Studio phiên bản 3.1 trở lên, bạn có thể thay thế các trường có thể quan sát bằng các đối tượng LiveData trong mã ràng buộc dữ liệu của bạn.

Để sử dụng một đối tượng LiveData với lớp ràng buộc của bạn, bạn cần xác định một chủ sở hữu vòng đời để xác định phạm vi của đối tượng LiveData. Ví dụ sau chỉ định hoạt động là chủ sở hữu vòng đời sau khi lớp ràng buộc đã được khởi tạo:

class ViewModelActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        // Inflate view and obtain an instance of the binding class.  
        val binding: UserBinding = DataBindingUtil.setContentView(this, R.layout.user)  
  
        // Specify the current activity as the lifecycle owner.  
        binding.setLifecycleOwner(this)  
    }  
}

Bạn có thể sử dụng thành phần ViewModel, như được giải thích trong Sử dụng ViewModel để quản lý dữ liệu liên quan đến giao diện người dùng, để liên kết dữ liệu với bố cục. Trong thành phần ViewModel, bạn có thể sử dụng đối tượng LiveData để chuyển đổi dữ liệu hoặc hợp nhất nhiều nguồn dữ liệu. Ví dụ sau đây cho thấy cách chuyển đổi dữ liệu trong ViewModel:

class ScheduleViewModel : ViewModel() {  
    val userName: LiveData  
  
    init {  
        val result = Repository.userName  
        userName = Transformations.map(result) { result -> result.value }  
    }  
}

#### **Use ViewModel to manage UI-related data**

Thư viện ràng buộc dữ liệu hoạt động liền mạch với các thành phần ViewModel, hiển thị dữ liệu mà bố cục quan sát và phản ứng với các thay đổi của nó. Sử dụng các thành phần ViewModel với Thư viện ràng buộc dữ liệu cho phép bạn di chuyển logic UI ra khỏi bố trí và vào các thành phần, dễ kiểm tra hơn. Thư viện ràng buộc dữ liệu đảm bảo rằng các khung nhìn bị ràng buộc và không bị ràng buộc từ nguồn dữ liệu khi cần thiết. Hầu hết các công việc còn lại bao gồm trong việc đảm bảo rằng bạn đang phơi bày dữ liệu chính xác. Để biết thêm thông tin về Cấu phần Kiến trúc này, hãy xem Tổng quan về ViewModel.

Để sử dụng thành phần ViewModel với Thư viện ràng buộc dữ liệu, bạn phải khởi tạo thành phần của bạn, kế thừa từ lớp Viewmodel, lấy một cá thể của lớp ràng buộc của bạn và gán thành phần ViewModel của bạn cho một thuộc tính trong lớp ràng buộc. Ví dụ sau đây cho thấy cách sử dụng thành phần với thư viện:

class ViewModelActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        // Obtain the ViewModel component.  
        UserModel userModel = ViewModelProviders.of(getActivity())  
                                                  .get(UserModel.class)  
  
        // Inflate view and obtain an instance of the binding class.  
        val binding: UserBinding = DataBindingUtil.setContentView(this, R.layout.user)  
  
        // Assign the component to a property in the binding class.  
        binding.viewmodel = userModel  
    }  
}

Trong bố cục của bạn, gán các thuộc tính và phương thức của thành phần ViewModel của bạn cho các khung nhìn tương ứng bằng cách sử dụng các biểu thức ràng buộc, như trong ví dụ sau:

<CheckBox  
    android:id="@+id/rememberMeCheckBox"  
    android:checked="@{viewmodel.rememberMe}"  
    android:onCheckedChanged="@{() -> viewmodel.rememberMeChanged()}" />

#### **Use an Observable ViewModel for more control over binding adapters**

Bạn có thể sử dụng thành phần ViewModel để triển khai Observable để thông báo cho các thành phần ứng dụng khác về các thay đổi trong dữ liệu, tương tự như cách bạn sử dụng đối tượng LiveData.

Có những tình huống mà bạn có thể thích sử dụng thành phần ViewModel để triển khai giao diện Quan sát bằng cách sử dụng các đối tượng LiveData, ngay cả khi bạn mất khả năng quản lý vòng đời của LiveData. Sử dụng thành phần ViewModel triển khai Observable cho phép bạn kiểm soát nhiều hơn các bộ điều hợp ràng buộc trong ứng dụng của bạn. Ví dụ: mẫu này cho phép bạn kiểm soát nhiều hơn các thông báo khi dữ liệu thay đổi, nó cũng cho phép bạn chỉ định phương thức tùy chỉnh để đặt giá trị của thuộc tính trong liên kết dữ liệu hai chiều.

Để thực hiện một thành phần ViewModel quan sát được, bạn phải tạo một lớp kế thừa từ lớp ViewModel và thực hiện giao diện Quan sát được. Bạn có thể cung cấp logic tùy chỉnh của mình khi người quan sát đăng ký hoặc hủy đăng ký thông báo bằng phương thức addOnPropertyChangedCallback () và removeOnPropertyChangedCallback (). Bạn cũng có thể cung cấp logic tùy chỉnh chạy khi các thuộc tính thay đổi trong phương thức notifyPropertyChanged (). Ví dụ mã sau đây cho thấy cách triển khai ViewModel quan sát:

/\*\*  
 \* A ViewModel that is also an Observable,  
 \* to be used with the Data Binding Library.  
 \*/  
open class ObservableViewModel : ViewModel(), Observable {  
    private val callbacks: PropertyChangeRegistry = PropertyChangeRegistry()  
  
    override fun addOnPropertyChangedCallback(  
            callback: Observable.OnPropertyChangedCallback) {  
        callbacks.add(callback)  
    }  
  
    override fun removeOnPropertyChangedCallback(  
            callback: Observable.OnPropertyChangedCallback) {  
        callbacks.remove(callback)  
    }  
  
    /\*\*  
     \* Notifies observers that all properties of this instance have changed.  
     \*/  
    fun notifyChange() {  
        callbacks.notifyCallbacks(this, 0, null)  
    }  
  
    /\*\*  
     \* Notifies observers that a specific property has changed. The getter for the  
     \* property that changes should be marked with the @Bindable annotation to  
     \* generate a field in the BR class to be used as the fieldId parameter.  
     \*  
     \* @param fieldId The generated BR id for the Bindable field.  
     \*/  
    fun notifyPropertyChanged(fieldId: Int) {  
        callbacks.notifyCallbacks(this, fieldId, null)  
    }  
}

### **Two-way data binding**

Sử dụng ràng buộc dữ liệu một chiều, bạn có thể đặt giá trị trên thuộc tính và đặt trình nghe phản ứng với thay đổi trong thuộc tính đó:

<CheckBox  
    android:id="@+id/rememberMeCheckBox"  
    **android:checked="@{viewmodel.rememberMe}"  
    android:onCheckedChanged="@{viewmodel.rememberMeChanged}"**  
/>

Liên kết dữ liệu hai chiều cung cấp một lối tắt cho quá trình này:

<CheckBox  
    android:id="@+id/rememberMeCheckBox"  
    **android:checked="@={viewmodel.rememberMe}"**  
/>

Ký hiệu @ = {}, quan trọng bao gồm ký hiệu "=", nhận các thay đổi dữ liệu đối với thuộc tính và lắng nghe bản cập nhật của người dùng cùng một lúc.

Để phản ứng với những thay đổi trong dữ liệu sao lưu, bạn có thể làm cho biến bố cục của bạn thực hiện Quan sát, thường là BaseObservable và sử dụng chú thích @Bindable, như được hiển thị trong đoạn mã sau:

class LoginViewModel : **BaseObservable** {  
    // val data = ...  
  
    **@Bindable**  
    fun getRememberMe(): Boolean {  
        return data.rememberMe  
    }  
  
    fun setRememberMe(value: Boolean) {  
        // Avoids infinite loops.  
        if (data.rememberMe != value) {  
            data.rememberMe = value  
  
            // React to the change.  
            saveData()  
  
            // Notify observers of a new value.  
            notifyPropertyChanged(BR.remember\_me)  
        }  
    }  
}

Bởi vì phương thức getter của thuộc tính bindable được gọi là getRememberMe (), phương thức setter tương ứng của thuộc tính tự động sử dụng tên setRememberMe ().

#### **Two-way data binding using custom attributes**

Nền tảng này cung cấp triển khai ràng buộc dữ liệu hai chiều cho các thuộc tính hai chiều phổ biến nhất và thay đổi trình nghe, mà bạn có thể sử dụng như một phần của ứng dụng của mình. Nếu bạn muốn sử dụng liên kết dữ liệu hai chiều với các thuộc tính tùy chỉnh, bạn cần phải làm việc với các chú thích @InverseBindingAdapter và @InverseBindingMethod.

Ví dụ: nếu bạn muốn bật liên kết dữ liệu hai chiều trên thuộc tính "thời gian" trong chế độ xem tùy chỉnh được gọi là MyView, hãy hoàn tất các bước sau:

B1: Chú thích phương thức đặt giá trị ban đầu và cập nhật khi giá trị thay đổi bằng

**@BindingAdapter("time")**  
@JvmStatic fun setTime(view: MyView, newValue: Time) {  
    // Important to break potential infinite loops.  
    if (view.time != newValue) {  
        view.time = newValue  
    }  
}

B2: Chú thích phương thức đọc giá trị từ khung nhìn bằng cách sử dụng @InverseBindingAdapter:

**@InverseBindingAdapter("time")**  
@JvmStatic fun getTime(view: MyView) : Time {  
    return view.getTime()  
}

Tại thời điểm này, ràng buộc dữ liệu biết phải làm gì khi thay đổi dữ liệu (nó gọi phương thức được chú thích bằng @BindingAdapter) và những gì cần gọi khi thuộc tính view thay đổi (nó gọi là InverseBindingListener). Tuy nhiên, nó không biết khi nào hoặc làm thế nào thay đổi thuộc tính.

Để làm được điều đó, bạn cần thiết lập một người nghe trên khung nhìn. Nó có thể là trình lắng nghe tùy chỉnh được liên kết với chế độ xem tùy chỉnh của bạn hoặc đó có thể là sự kiện chung, chẳng hạn như mất tập trung hoặc thay đổi văn bản. Thêm chú thích @BindingAdapter vào phương thức đặt bộ nghe cho các thay đổi trên thuộc tính:

**@BindingAdapter("app:timeAttrChanged")**  
@JvmStatic fun setListeners(  
        view: MyView,  
        **final InverseBindingListener attrChange**  
) {  
    // Set a listener for click, focus, touch, etc.  
}

Trình lắng nghe bao gồm một InverseBindingListener như một tham số. Bạn sử dụng InverseBindingListener để báo cho hệ thống ràng buộc dữ liệu mà thuộc tính đã thay đổi. Sau đó hệ thống có thể bắt đầu gọi phương thức được chú thích bằng cách sử dụng @InverseBindingAdapter, v.v.

Trong thực tế, người nghe này bao gồm một số logic không tầm thường, bao gồm cả người nghe để gắn kết dữ liệu một chiều. Ví dụ, xem bộ điều hợp cho thay đổi thuộc tính văn bản, TextViewBindingAdapter.

#### **Converters**

Nếu biến bị ràng buộc với đối tượng View cần được định dạng, dịch hoặc thay đổi bằng cách nào đó trước khi được hiển thị, có thể sử dụng đối tượng Converter.

Ví dụ: lấy đối tượng EditText hiển thị ngày:

<EditText  
    android:id="@+id/birth\_date"  
    android:text="@={Converter.dateToString(viewmodel.birthDate)}"  
/>

Thuộc tính viewmodel.birthDate chứa một giá trị kiểu Long, vì vậy nó cần phải được định dạng bằng cách sử dụng một trình biến đổi.

Bởi vì một biểu thức hai chiều đang được sử dụng, cũng cần phải là một công cụ chuyển đổi nghịch đảo để cho thư viện biết cách chuyển đổi chuỗi do người dùng cung cấp trở lại kiểu dữ liệu sao lưu, trong trường hợp này là Long. Quá trình này được thực hiện bằng cách thêm chú thích @InverseMethod vào một trong các trình biến đổi và có chú thích này tham chiếu đến trình biến đổi nghịch đảo. Ví dụ về cấu hình này xuất hiện trong đoạn mã sau:

object Converter {  
    **@InverseMethod("stringToDate")**  
    fun dateToString(  
        view: EditText, oldValue: Long,  
        value: Long  
    ): String {  
        // Converts long to String.  
    }  
  
    fun stringToDate(  
        view: EditText, oldValue: String,  
        value: String  
    ): Long {  
        // Converts String to long.  
    }  
}

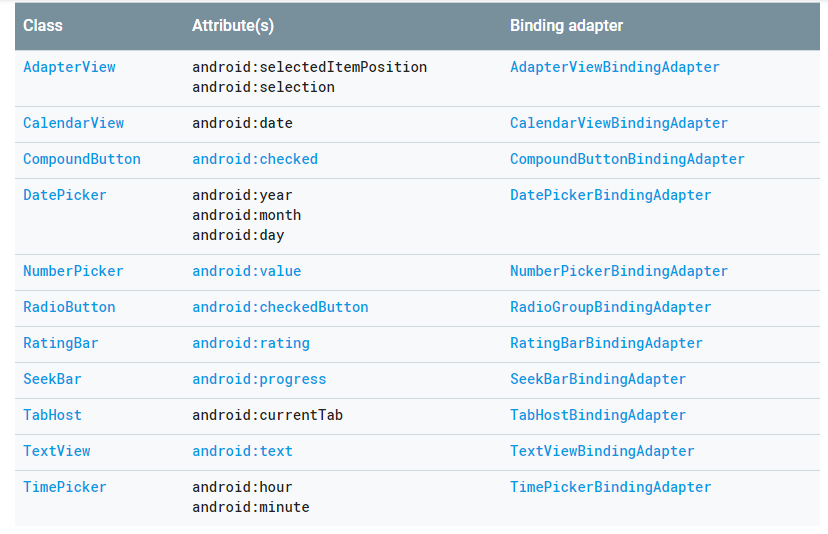
#### **Infinite loops using two-way data binding**

Be careful not to introduce infinite loops when using two-way data binding. When the user changes an attribute, the method annotated using @InverseBindingAdapter is called, and the value is assigned to the backing property. This, in turn, would call the method annotated using @BindingAdapter, which would trigger another call to the method annotated using @InverseBindingAdapter, and so on.

For this reason, it's important to break possible infinite loops by comparing new and old values in the methods annotated using @BindingAdapter.

#### **Two-way attributes**

The platform provides built-in support for two-way data binding when you use the attributes in the following table. For details on how the platform provides this support, see the implementations for the corresponding binding adapters:



# Internet of Things (IoT)

Là một kịch bản của thế giới, khi mà mỗi đồ vật, con người được cung cấp một định danh của riêng mình, và tất cả có khả năng truyền tải, trao đổi thông tin, dữ liệu qua một mạng duy nhất mà không cần đến sự tương tác trực tiếp giữa người với người, hay người với máy tính. IoT đã phát triển từ sự hội tụ của công nghệ không dây, công nghệ vi cơ điện tử và Internet[1]. Nói đơn giản là một tập hợp các thiết bị có khả năng kết nối với nhau, với Internet và với thế giới bên ngoài để thực hiện một công việc nào đó.

## tổng quan ESP8266

ESP8266 là dòng chip tích hợp Wi-Fi 2.4Ghz có thể lập trình được, rẻ tiền được sản xuất bởi một công ty bán dẫn Trung Quốc: Espressif Systems.

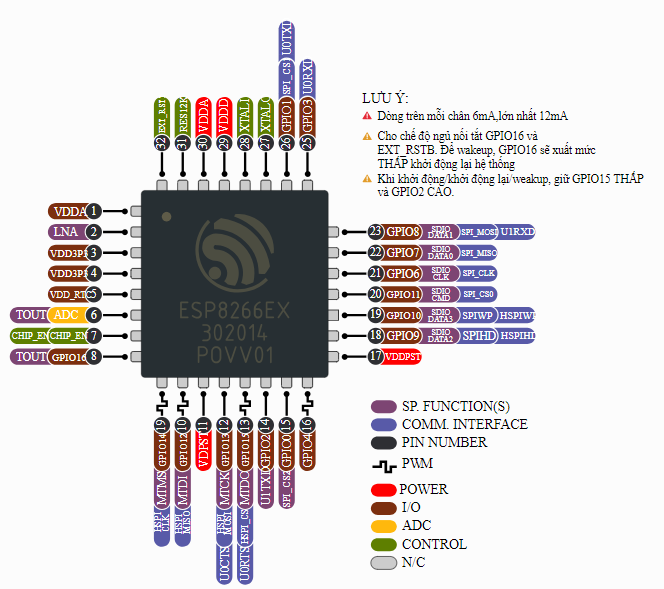
Được phát hành đầu tiên vào tháng 8 năm 2014, đóng gói đưa ra thị trường dạng Mô dun ESP-01, được sản xuất bởi bên thứ 3: AI-Thinker. Có khả năng kết nối Internet qua mạng Wi-Fi một cách nhanh chóng và sử dụng rất ít linh kiện đi kèm. Với giá cả có thể nói là rất rẻ so với tính năng và khả năng ESP8266 có thể làm được.

ESP8266 có một cộng đồng các nhà phát triển trên thế giới rất lớn, cung cấp nhiều Module lập trình mã mở giúp nhiều người có thể tiếp cận và xây dựng ứng dụng rất nhanh.

Hiện nay tất cả các dòng chip ESP8266 trên thị trường đều mang nhãn ESP8266EX, là phiên bản nâng cấp của ESP8266.

## Sơ đồ chân & Sơ đồ khối

Sơ đồ chân & sơ đồ khối ESP8266EX



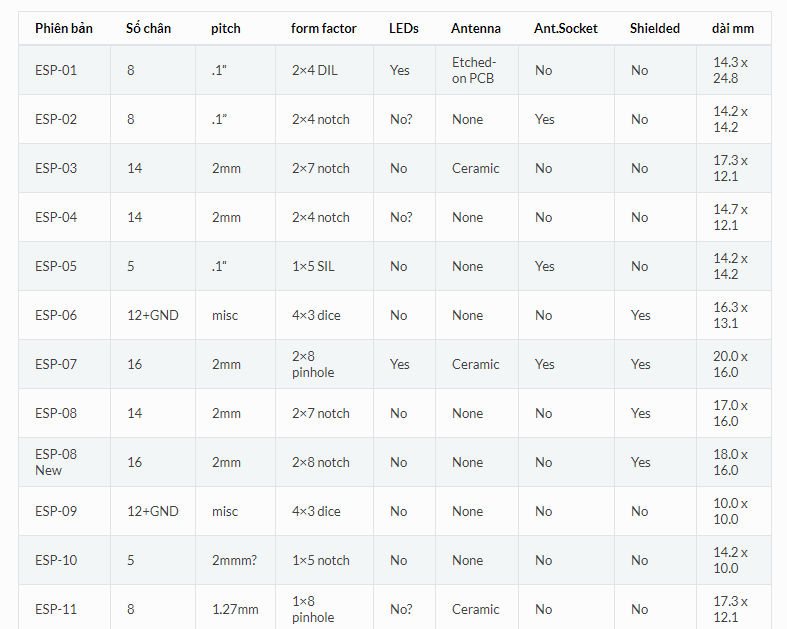
## Thông số phần cứng

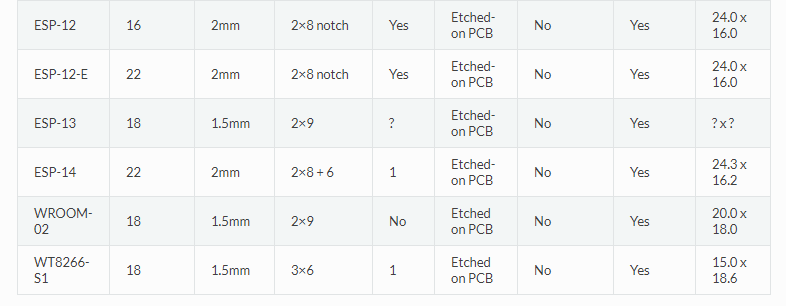
* 32-bit RISC CPU : Tensilica Xtensa LX106 running at 80 MHz
* Hổ trợ Flash ngoài từ 512KiB đến 4MiB
* 64KBytes RAM thực thi lệnh
* 96KBytes RAM dữ liệu
* 64KBytes boot ROM
* Chuẩn wifi EEE 802.11 b/g/n, Wi-Fi 2.4 GHz **Tích hợp TR switch, balun, LNA, khuếch đại công suất và matching network**Hổ trợ WEP, WPA/WPA2, Open network
* Tích hợp giao thức TCP/IP
* Hổ trợ nhiều loại anten
* 16 chân GPIO
* Hổ trợ SDIO 2.0, UART, SPI, I²C, PWM,I²S với DMA
* 1 ADC 10-bit
* Dải nhiệt độ hoạt động rộng : -40C ~ 125C

## Các loại Module ESP8266

ESP8266 cần ít nhất thêm 7 linh kiện nữa mới có thể hoạt động, trong đó phần khó nhất là Antena. Đòi hỏi phải được sản xuất, kiểm tra với các thiết bị hiện đại. Do đó, trên thị trường xuất hiện nhiều Module và Board mạch phát triển đảm đương hết để người dùng đơn giản nhất trong việc phát triển ứng dụng. Một số Module và Board phát triển phổ biến:

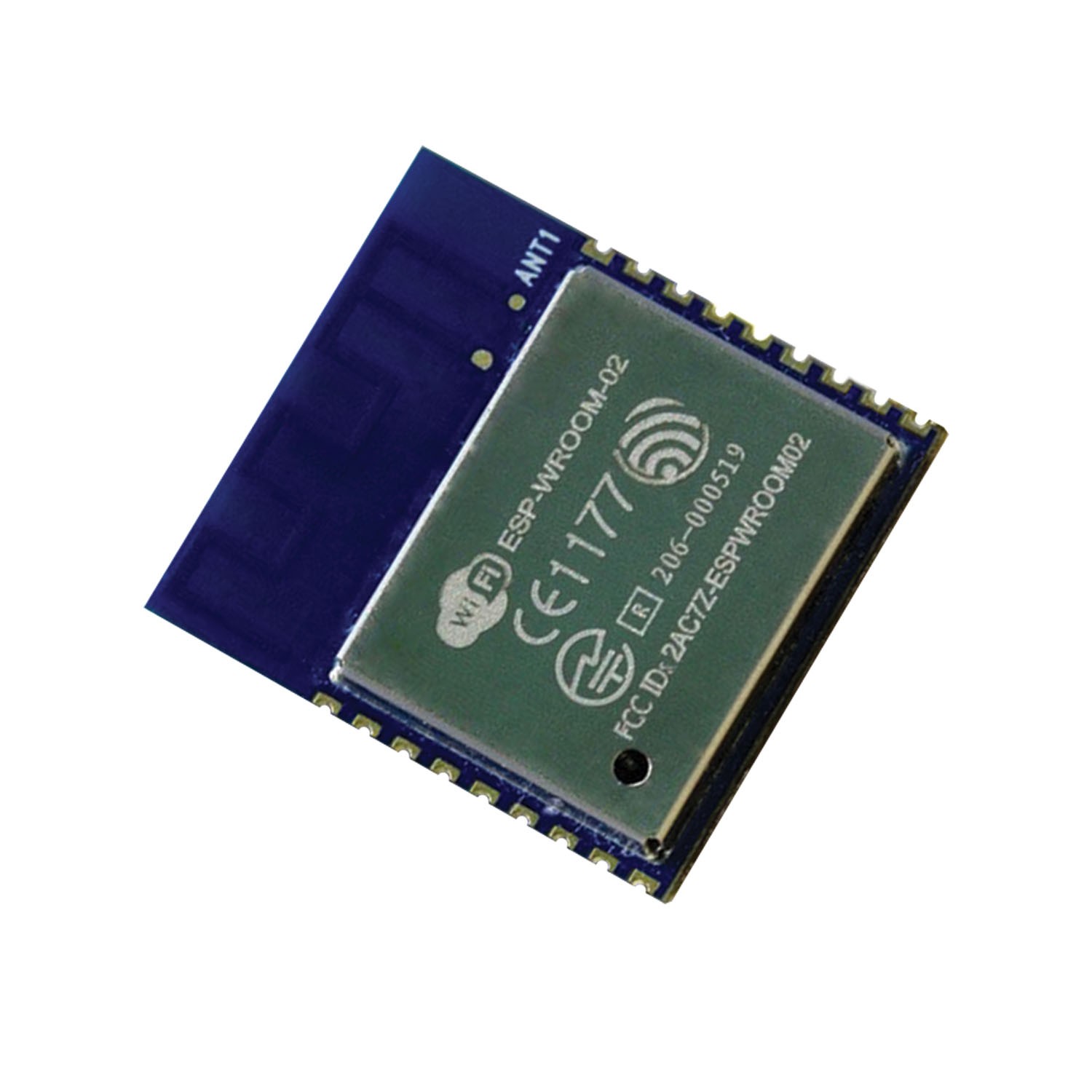
### **Bảng so sánh các thông số**



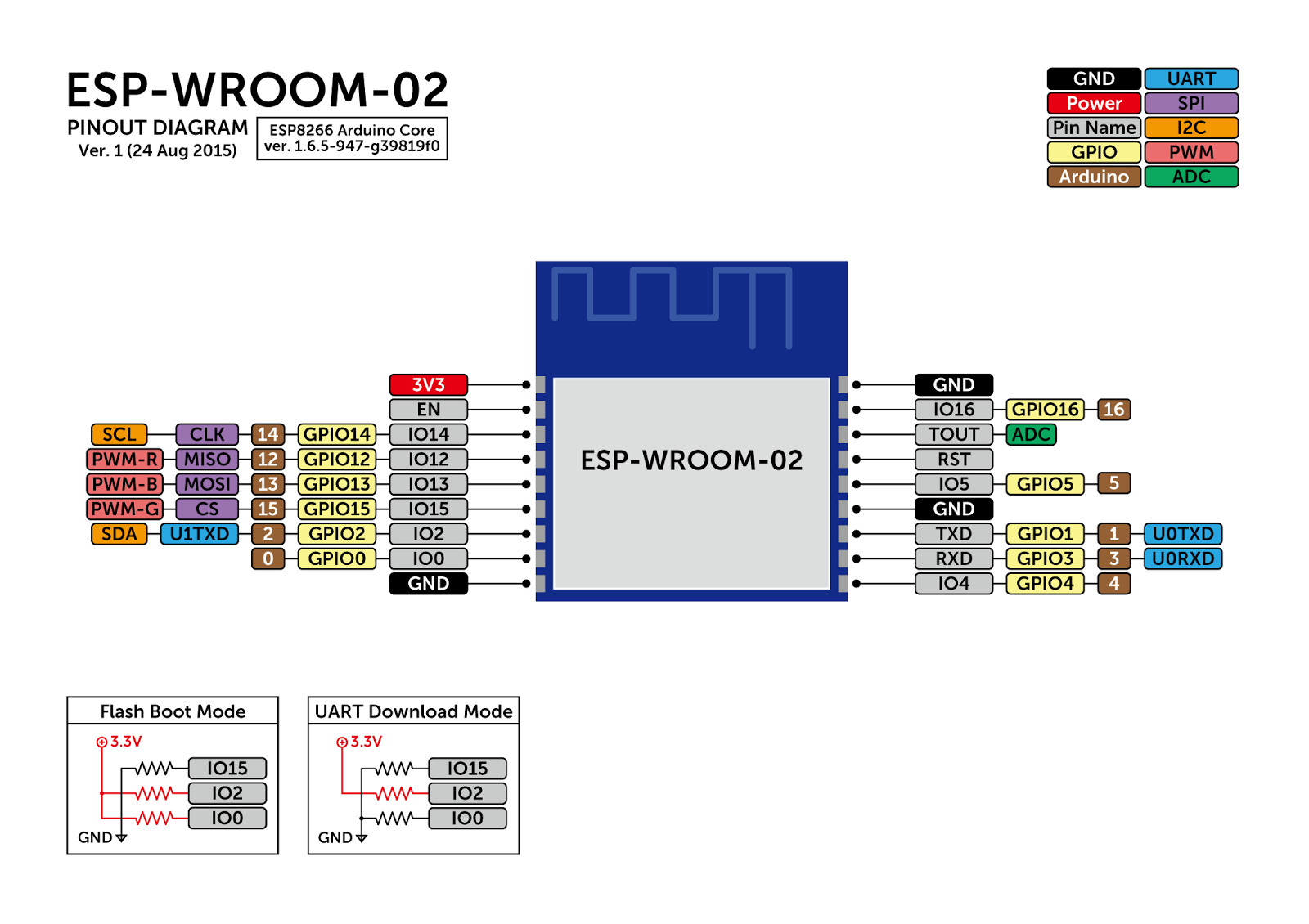


### **Một số module ESP8266 trên thị trường**

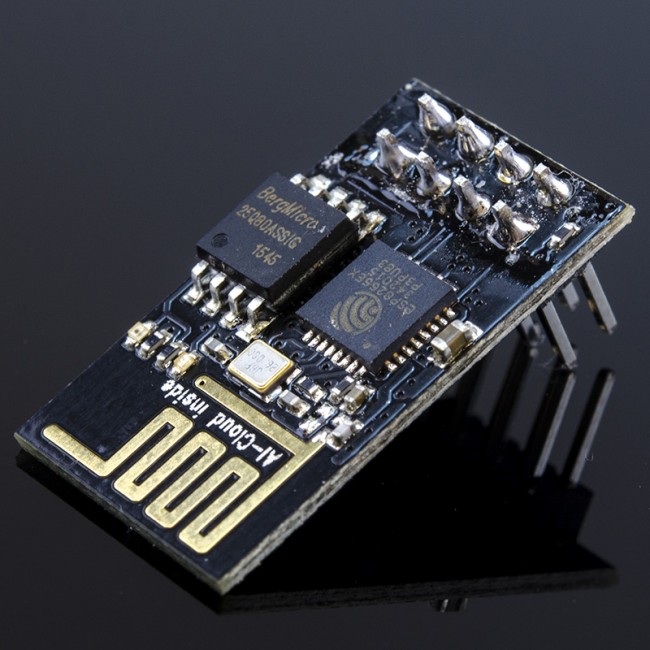
##### **ESP-WRoom-02**



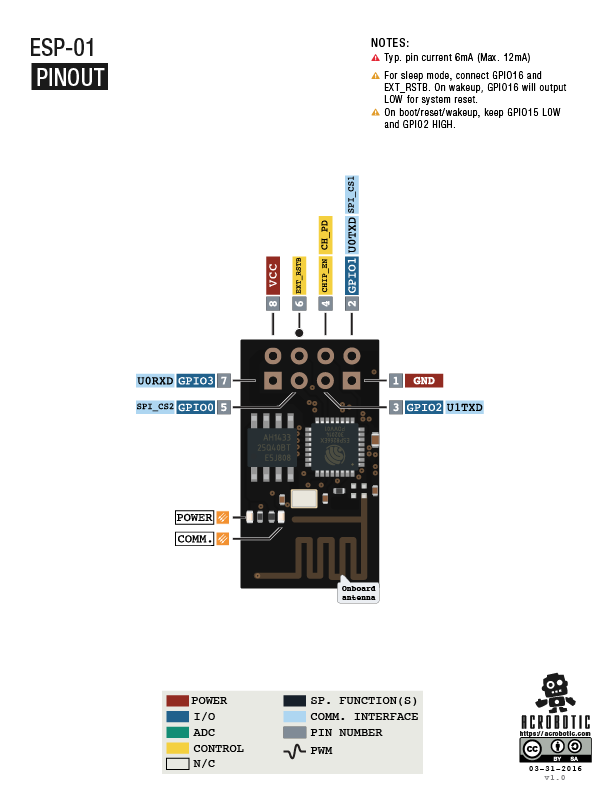
* + Tính năng:
* ESP-WROOM-02 là một module MCU Wifi 32-bit tiết kiệm năng lượng dựa trên chip ESP8266.
* Hổ trợ các chuẩn mạng không dây 802.11 b/g/n
* Tích hợp sẳn giao thức TCP/IP, 10-bit ADC, TR switch, balun, LNA, và các chuẩn giao tiếp HSPI/UART/PWM/I2C/I2S.
* Hổ trợ Wi-Fi Alliance, SRRC, FCC, CE, TELEC, IC & KCC Certified, RoHS, Halogen Free, REACH & CFSI Compliant, HTOL, ESD-HM, MSL, μHAST, HTSL
* Hổ trợ Cloud Server Development
* Custom firmware development qua SDK
* Cấu hình người dùng qua AT Instruction Set, Cloud Server và ứng dụng Afdroid/iOS
* Khoảng cách giữa các chân 2.54mm
  + Sơ đồ chân



#### **ESP-01**

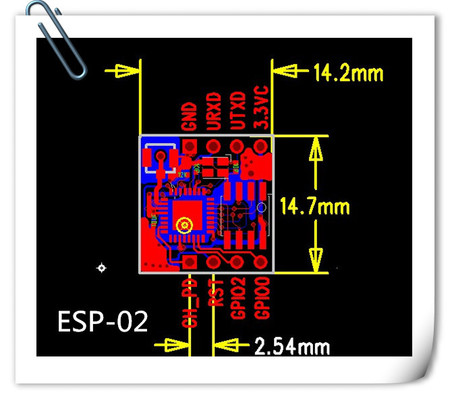


* + Tính năng
* Mạch nhỏ, gọn (24.75mm x 14.5mm)
* Điện áp làm việc 3.3v
* Tích hợp sẳn anten PCB trace trên module
* Có hai led báo hiệu : led nguồn, led TXD
* Có các chế độ: AP, STA, AT + STA
* Lệnh AT rất đơn giản, dễ dàng sử dụng
* Khoảng cách giữa các chân 2.54mm
  + Sơ đồ chân

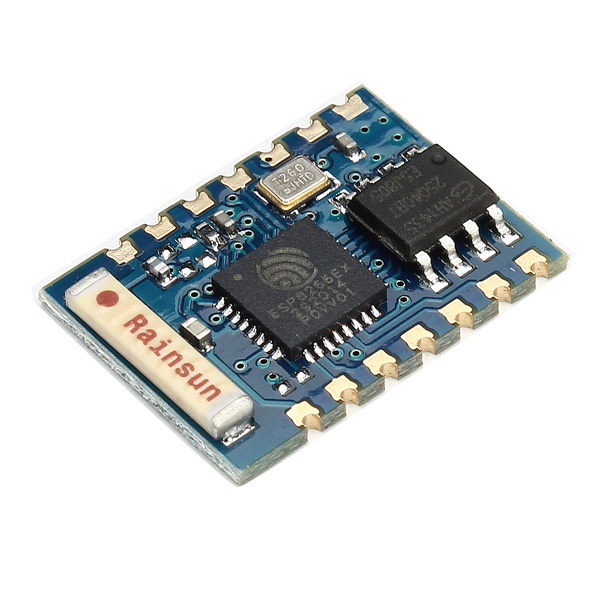


#### **ESP-02**

* + Tính năng
* Sử dụng nguồn 3.3v
* Sử dụng anten ngoài - U.FL
* Wireless network mode : station, softAP, softAP + station
* Tần số wifi hoạt động 2.4GHz, và hổ trợ bảo mật WPA/WPA2
* Dễ dàng phát triển các dự án với lệnh AT
* Khoảng cách giữa các chân 2.54mm
  + Sơ đồ chân



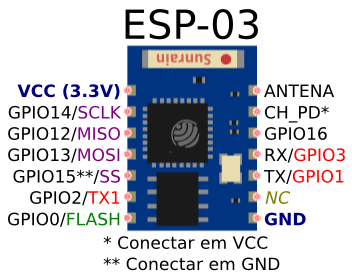
#### **ESP-03**



Tính năng

* Sử dụng nguồn 3.3v
* Tích hợp sẳn anten ceramic trên module và 1 pin
* Tiêu chuẩn wifi : 802.11b/g/n, 2.4GHz
* Wireless network mode : station, softAP, softAP + station
* Hổ trợ bảo mật WPA/WPA2
* Dễ dáng sử dụng với lệnh AT
* Khoảng cách giữa các chân 2mm

Sơ đồ chân



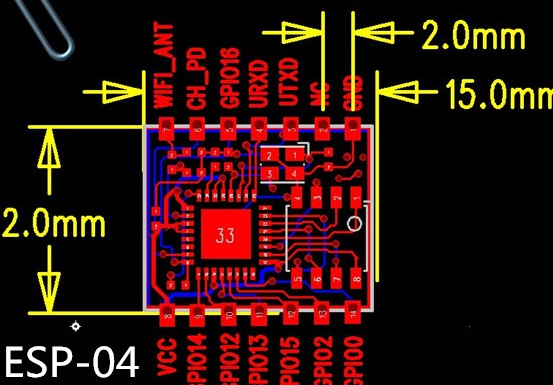
#### **ESP-04**



Tính năng

* Sử dụng nguồn 3.3v
* Wireless network mode : station, softAP, softAP + station
* Tiêu chuẩn wifi : 802.11b/g/n, với tần số 2.4GHz,và hổ trợ bảo mật WPA/WPA2
* Dễ dáng sử dụng với lệnh AT
* Khoảng cách giữa các chân 2mm

Sơ đồ chân



## NodeMCU

NodeMCU là một nền tảng IoT nguồn mở. Nó bao gồm phần mềm chạy trên ESP8266 Wi-Fi SoC từ Hệ thống Espressif và phần cứng dựa trên mô-đun ESP-12. Thuật ngữ "NodeMCU" theo mặc định đề cập đến phần vững hơn là các bộ phát triển. Phần sụn sử dụng ngôn ngữ Lua. Nó dựa trên dự án eLua, và được xây dựng trên nền tảng Espressif Non-OS SDK cho ESP8266. Nó sử dụng nhiều dự án mã nguồn mở, chẳng hạn như lua-cjson và spiffs.

### **History**

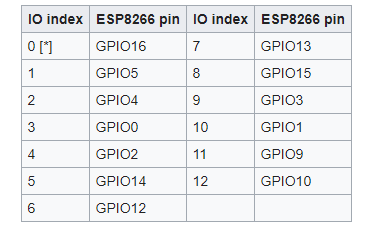
NodeMCU được tạo ra ngay sau khi ESP8266 xuất hiện. Vào ngày 30 tháng 12 năm 2013, Espressif Systems bắt đầu sản xuất ESP8266. ESP8266 là một SoC Wi-Fi tích hợp với lõi Tensilica Xtensa LX106, được sử dụng rộng rãi trong các ứng dụng IoT. NodeMCU bắt đầu vào ngày 13 tháng 10 năm 2014, khi Hong cam kết tập đầu tiên của nodemcu-firmware cho GitHub. Hai tháng sau, dự án mở rộng để bao gồm một nền tảng phần cứng mở khi nhà phát triển Huang R cam kết tệp gerber của một bảng ESP8266, có tên là devkit v0.9. Cuối tháng đó, Tuấn PM đã chuyển thư viện máy khách MQTT từ Contiki sang nền tảng ESP8266 SoC, [13] và cam kết với dự án NodeMCU, sau đó NodeMCU có thể hỗ trợ giao thức MQTT IoT, sử dụng Lua để truy cập vào môi giới MQTT. Một cập nhật quan trọng khác được thực hiện vào ngày 30 tháng 1 năm 2015, khi Devsaurus chuyển u8glib sang dự án NodeMCU, cho phép NodeMCU dễ dàng lái các màn hình LCD, Màn hình, OLED, thậm chí là VGA.

### **ESP8266 Arduino Core**

Khi Arduino.cc bắt đầu phát triển các bo mạch MCU mới dựa trên các bộ vi xử lý phi AVR như ARM / SAM MCU và được sử dụng trong Arduino Due, họ cần phải sửa đổi IDE Arduino để dễ thay đổi IDE để hỗ trợ công cụ thay thế. chuỗi để cho phép Arduino C / C ++ được biên dịch xuống các bộ xử lý mới này. Họ đã làm điều này với sự ra đời của Board Manager và SAM Core. Một "lõi" là tập hợp các thành phần phần mềm được yêu cầu bởi Board Manager và Arduino IDE để biên dịch một tệp nguồn Arduino C / C ++ xuống ngôn ngữ máy của MCU đích. Một số người đam mê ESP8266 sáng tạo đã phát triển một lõi Arduino cho ESP8266 WiFi SoC có sẵn trên trang web GitHub ESP8266 Core. Đây là cái được gọi là "ESP8266 Core cho Arduino IDE" và nó đã trở thành một trong những nền tảng phát triển phần mềm hàng đầu cho các mô-đun dựa trên ESP8266 và các bo mạch phát triển, bao gồm NodeMCUs.

Vào mùa hè năm 2015, những người sáng tạo đã từ bỏ dự án phần mềm và một nhóm những người đóng góp độc lập nhưng tận tụy đã tiếp quản. Vào mùa hè năm 2016, NodeMCU bao gồm hơn 40 mô-đun khác nhau. Do hạn chế về tài nguyên, người dùng cần phải chọn các mô-đun có liên quan cho dự án của họ và xây dựng một chương trình cơ sở phù hợp với nhu cầu của họ.

### **Pins of NodeMCU**



### **Code examples**

#### **Connect to an AP**

print(wifi.sta.getip())

*--nil*

wifi.setmode(wifi.STATION)

wifi.sta.config{ssid="SSID",pwd="password"}

*-- for older versions of the firmware wifi.sta.config("SSID","password")*

*-- wifi.sta.connect() not necessary because wifi.sta.config sets auto-connect = true*

tmr.create():alarm(1000, 1, **function**(cb\_timer)

**if** wifi.sta.getip() == **nil** **then**

print("Connecting...")

**else**

cb\_timer:unregister()

print("Connected, IP is "..wifi.sta.getip())

**end**

**end**)

#### **Control GPIO**

ledPin = 1

swPin = 2

gpio.mode(ledPin,gpio.OUTPUT)

gpio.write(ledPin,gpio.HIGH)

gpio.mode(swPin,gpio.INPUT)

print(gpio.read(swPin))

#### **HTTP request**

*-- A simple HTTP client*

conn = net.createConnection(net.TCP, 0)

conn:on("receive", **function**(sck, payload) print(payload) **end**)

conn:on("connection", **function**(sck)

sck:send("GET / HTTP/1.1**\r\n**Host: nodemcu.com**\r\n**"

.. "Connection: keep-alive**\r\n**Accept: \*/\***\r\n\r\n**")

**end**)

conn:connect(80, "nodemcu.com")

http.get("http://nodemcu.com", **nil**, **function**(code, data)

**if** (code < 0) **then**

print("HTTP request failed")

**else**

print(code, data)

**end**

**end**)

#### **HTTP server**

*-- a simple HTTP server*

srv = net.createServer(net.TCP)

srv:listen(80, **function**(conn)

conn:on("receive", **function**(sck, payload)

print(payload)

sck:send("HTTP/1.0 200 OK**\r\n**Content-Type: text/html**\r\n\r\n**<h1> Hello, NodeMCU.</h1>")

**end**)

conn:on("sent", **function**(sck) sck:close() **end**)

**end**)

#### **Connect to MQTT Broker**

*- init mqtt client with keepalive timer 120sec*

m = mqtt.Client("clientid", 120, "user", "password")

*-- setup Last Will and Testament (optional)*

*-- Broker will publish a message with qos = 0, retain = 0, data = "offline"*

*-- to topic "/lwt" if client don't send keepalive packet*

m:lwt("/lwt", "offline", 0, 0)

m:on("connect", **function**(con) print ("connected") **end**)

m:on("offline", **function**(con) print ("offline") **end**)

*-- on publish message receive event*

m:on("message", **function**(conn, topic, data)

print(topic .. ":" )

**if** data ~= **nil** **then**

print(data)

**end**

**end**)

*-- for secure: m:connect("192.168.11.118", 1880, 1)*

m:connect("192.168.11.118", 1880, 0, **function**(conn) print("connected") **end**)

*-- subscribe topic with qos = 0*

m:subscribe("/topic",0, **function**(conn) print("subscribe success") **end**)

*-- or subscribe multiple topic (topic/0, qos = 0; topic/1, qos = 1; topic2 , qos = 2)*

*-- m:subscribe({["topic/0"]=0,["topic/1"]=1,topic2=2}, function(conn) print("subscribe success") end)*

*-- publish a message with data = hello, QoS = 0, retain = 0*

m:publish("/topic","hello",0,0, **function**(conn) print("sent") **end**)

m:close();

*-- you can call m:connect again*

#### **UDP client and server**

*-- a udp server*

s=net.createServer(net.UDP)

s:on("receive",**function**(s,c) print(c) **end**)

s:listen(5683)

*-- a udp client*

cu=net.createConnection(net.UDP)

cu:on("receive",**function**(cu,c) print(c) **end**)

cu:connect(5683,"192.168.18.101")

cu:send("hello")

# LẬP TRÌNH NODE JS

## NodeJS

Nodejs là một an asynchronous event driven JavaScript runtime, Node được thiết kế để xây dựng các ứng dụng mạng có thể mở rộng.

Nó được phát triển bởi Ryan Dahl vào năm 2009, được bảo trợ bởi công ty Joyent, trụ sở tại California, Hoa Kỳ. Phần Core của Nodejs được viết hầu hết bằng C++ nên cho tốc độ xử lý và hiệu năng khá cao.. Định nghĩa NodeJS bởi tài liệu chính thức của nó như sau:

“*NodeJS là một platform được xây dựng trên môi trường thực thi của Chrome JavaScript để dễ dàng xây dựng các ứng dụng mạng nhanh và có khả năng mở rộng. NodeJS sử dụng một mô hình hướng sự kiện, mô hình I/O bất đồng bộ giúp làm cho nó nhẹ và hiệu quả, hoàn hảo cho các ứng dụng dữ liệu theo thời gian thực chạy trên các thiết bị phân tán*”

## Đặc tính của NodeJS

1. **Cực kỳ nhanh**: Node.js được xây dựng trên Công cụ JavaScript V8 của Google Chrome, do đó thư viện của nó rất nhanh trong việc thực thi mã.

**I/O là Asynchronous và Event Driven**: Tất cả các API của thư viện Node.js là bất đồng bộ, tức là không chặn. Vì vậy, một máy chủ dựa trên Node.js không bao giờ chờ đợi một API để trả về dữ liệu. Máy chủ chuyển sang API tiếp theo sau khi gọi nó và cơ chế thông báo của Sự kiện của Node.js giúp máy chủ nhận được phản hồi từ cuộc gọi API trước đó. Nó cũng là một lý do mà nó rất nhanh.

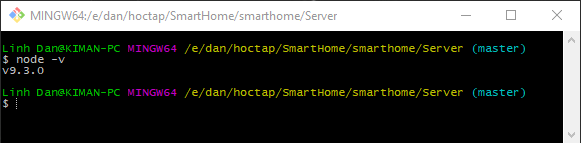
1. **Đơn luồng**: Node.js tuân theo mô hình luồng đơn với vòng lặp sự kiện.
2. **Khả năng mở rộng cao**: Node.js có khả năng mở rộng cao vì cơ chế sự kiện giúp máy chủ phản hồi theo cách không chặn.
3. **Không có bộ đệm**: Node.js giảm thời gian xử lý tổng thể trong khi tải lên tệp âm thanh và video. Các ứng dụng Node.js không bao giờ đệm bất kỳ dữ liệu nào. Các ứng dụng này đơn giản xuất dữ liệu theo khối.
4. **Mã nguồn mở**: Node.js có một cộng đồng mã nguồn mở đã tạo ra nhiều module xuất sắc để bổ sung thêm các khả năng cho các ứng dụng Node.js.
5. **Giấy phép**: Node.js được phát hành theo giấy phép MIT.

## Cài đặt NodeJS

Truy cập vào địa chỉ: <https://nodejs.org/en/download/>

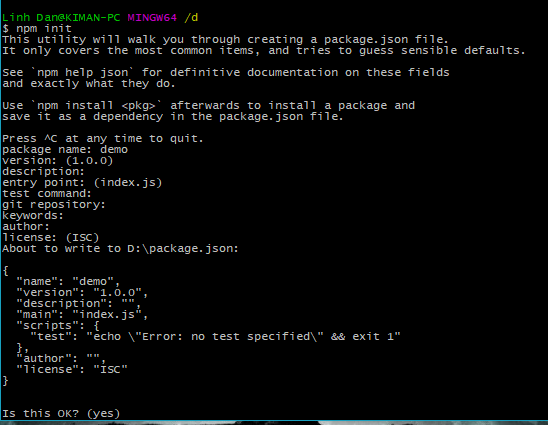
Chọn phiên bản thích hợp và tải về và thực hiện cài đặt theo mặc định cho tới khi kết thúc quá trình cài đặt.

Sau khi cài hoàn tất, mở cmd rồi di chuyển tới thư mục chứa Node và Nhập lệnh node –v để xem phiên bản.



## Tạo và thực thi chương trình

Để tạo một project, trong cmd di chuyển tới thư mục đích rồi Nhập lệnh npm init chờ tới khi trên cmd hiển thị package name: rồi nhập tên server vào, lưu ý, tên phải viết thường và không chứa ký tự đặc biệt. Sau đó có thể nhấn enter để bỏ qua các trường khác (không bắt buộc nhập, cho tới khi thấy dòng “Is this OK? (yes)”, nhấn yes để xác nhận khởi tạo project.



Vào thư mục chứa project để kiểm tra sẽ có file package.json có nội dung:

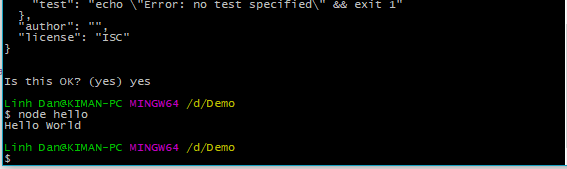
|  |
| --- |
| {  "name": "demo",  "version": "1.0.0",  "description": "",  "main": "index.js",  "scripts": {  "test": "echo \"Error: no test specified\" && exit 1"  },  "author": "",  "license": "ISC"  } |

Tạo chương trình Hello World:

Trong project, tạo file hello.js có nội dung:

|  |
| --- |
| console.log(‘Hello World’); |

Trong cmd, di chuyển vào trong thư mục rồi Nhập lệnh ‘node hello’ để thực thi và xem kết quả:



## Module

Nodejs được xây dựng như một interface là các module, cho phép chia sẻ các socket giữa các tiến trình để cho phép load balancing trên core của bạn.

Trong hệ thống module của Node.js, mỗi tập tin được coi là một module riêng biệt.

Để sử dụng module, cần gọi tới module bằng lệnh require() hoặc require.resolve () (Để lấy tên tệp chính xác sẽ được tải khi require ()). Ngữ nghĩa của hàm require() của Node.js được thiết kế để hỗ trợ một số cấu trúc thư mục hợp lý. Các chương trình quản lý package như dpkg, rpm, và npm hy vọng sẽ có thể xây dựng các gói có sẵn từ các module của Node.js mà không cần sửa đổi.

Để có một module thực thi mã nhiều lần, hãy xuất một hàm và gọi hàm đó bằng cách: module.exports.name = myModule

Ngoài ra, trên các hệ thống package không phân biệt dạng chữ hay hệ điều hành, các tên tệp được xác định khác nhau có thể trỏ đến cùng một tệp, nhưng bộ nhớ cache sẽ vẫn coi chúng là các module khác nhau và sẽ tải lại tệp nhiều lần. Ví dụ, require(‘./model/mode’) và require(‘./model/Mode’) trả về hai đối tượng khác nhau, bất kể ‘./model/mode’ và ‘./model/Mode’ là cùng một tệp.

Node.js có một số module được biên dịch thành nhị phân gọi là module lõi (core module). Các core module được định nghĩa trong nguồn của Node.js và nằm trong thư mục lib/

Các core module luôn được ưu tiên tải nếu số nhận dạng của chúng được require(). Ví dụ, require ('http') sẽ luôn trả về HTTP module được tích hợp, ngay cả khi có một tệp khác có tên đó.

Nếu module không phải core module, khi require tới module đó cần được bắt đầu bằng ‘./’ hoặc ‘../’, Node.js sẽ bắt đầu tại thư mục cha của mô-đun hiện tại và thêm /node\_modules và tải module từ vị trí đó. Node sẽ không nối thêm node\_modules vào đường dẫn đã kết thúc trong node\_modules.

Để tải thêm một số module từ hệ thống package của Nodejs, cần sử dụng npm. Các module được cài đặt sẽ nằm trong thư mục node\_modules và được khai báo trong file package.json.

## NPM

NPM viết tắt của Node Package Manager là một công cụ (chương trình) quản lý các thư viện lập trình Javascript cho Node.js, công cụ này là thật sự cần thiết cho thế giới mã nguồn mở. Trong cộng đồng Javascript, các lập trình viên chia sẻ hàng trăm nghìn các đoạn code giúp cho các dự án mới tránh phải viết lại các thành phần cơ bản, các thư viện lập trình hay thậm chí cả các framework. Mỗi đoạn code này có thể phụ thuộc vào rất nhiều các mã nguồn mở khác.

NPM đã được tích hợp sẵn khi cài đặt NodeJS.

Để cài đặt một thư viện nào đó, chỉ cần mở cửa sổ CMD và thực thi lệnh: npm install <package\_name> hoặc npm install <nhiều package\_name>

Để gỡ cài đặt thư viện nào đó, thực thi lệnh npm uninstall <package\_name> hoặc npm uninstall <nhiều package\_name>.

Trong đề tài, module được thêm để sử dụng:

|  |  |  |
| --- | --- | --- |
| Tên module | Phiên bản | Chức năng trong đề tài |
| body-parser | 3.0.0 | Đọc dữ liệu từ form sử dụng method POST trên website |
| ejs | 2.6.1 | View engine sử dụng viết front-end |
| express | 4.16.3 | Tạo server |
| jsonwebtoken | 8.2.1 | Tạo token mã hóa thông tin |
| md5 | 2.2.1 | Mã hóa thông tin |
| mongoose | 5.1.1 | Quản trị cơ sở dữ liệu mongoDB |
| node-persist | 3.0.1 | Tạo file dữ liệu cục bộ |
| node-schedule | 1.3.0 | Tạo lịch làm việc cho chế độ của người dùng |
| nodemailer | 4.6.4 | Gửi email |
| socket.io | 2.1.0 | Kết nối thời gian thực client-server |

Ngoài ra còn có một số module khác như: base64url, connect-flash, cookie-parser, email-templates, fs, jimp, formidable, path, randomstring, underscore và các module kèm theo của các module này.

## Express module

### Express

Express là một web application framework, cung cấp một API đơn giản để xây dựng các trang web, ứng dụng web và các đầu cuối. Với ExpressJS, bạn không cần phải lo lắng về các giao thức, quy trình cấp thấp,...

Express được phát triển bởi TJ Holowaychuk và được duy trì bởi nền tảng Node.js và nhiều cộng tác viên mã nguồn mở.

### Route

Route là một thành phần cực kỳ quan trọng của một website, nó giúp website biết được người dùng truy cập đến nơi nào của trang web, từ đó phản hồi lại một cách thích hợp. Trong Express, route được tích hợp sẵn và dễ dàng sử dụng.

Ví dụ: Tạo file server.js để tạo server và require các module và sử dụng middleware của các module.

|  |
| --- |
| var express = require('express'),  session = require('express-session'),  cookieParser = require('cookie-parser'),  bodyParser = require('body-parser'),  flash = require('connect-flash'),  path = require('path'),  app = express(),  server = require('http').createServer(app),  port = process.env.PORT || 3000;  //tạo đường dẫn tĩnh để chia sẻ các file trong thư mục public  app.use(express.static(path.join(\_\_dirname, '/public')));  //cho phép sử dụng session trên trang web  app.use(session({secret: “mySecretKey”, resave: true, saveUninitialized: true, cookie: {maxAge : 1000\*60\*300}}));  //cho phép sử dụng cookie  app.use(cookieParser(config.cookie\_secret\_key));  //cho phép sử dụng đọc dữ liệu gửi lên dạng POST  app.use(bodyParser.urlencoded({extended: true}));  app.use(bodyParser.json());  //sử dụng flash mesage  app.use(flash());  // Khai báo engine sử dụng cho trang web  app.set('view engine', 'ejs');  //Khai báo các đường dẫn tới các thư mục chứa các trang web  app.set('views', [path.join(\_\_dirname + '/views')]);  //Khai báo router của user  var userRouter = require('./routers/user');  //sử dụng router user đã khai báo bằng đường dẫn /  app.use(‘/’, userRouter);  require('./msocket')(io, schedule);  //cho server lắng nghe ở port (port 3000 hoặc port trên môi trường cung cấp)  server.listen(port, function() {  console.log("Waiting statement...");  }); |

Express cung cấp các phương thức get, post, put, delete để tạo REST API.

|  |
| --- |
| const express = require('express');  var router = express.Router();  router.get('/', (req, res)=>{  res.send('Get method');  //do something  });  router.post('/', (req, res)=>{  res.send('Post method');  //do something  });  router.put('/', (req, res)=>{  res.send('Put method');  //do something  });  router.delete('/', (req, res)=>{  res.send('Delete method');  //do something  }); |

Tạo route có tham số: Express cho phép tạo các route có một hay nhiều tham số. Các tham số được khai báo bằng “:nameParams” và cách nhau bằng dấu “/”

|  |
| --- |
| router.get('/:id/:page', (req, res)=>{  const id = req.params.id;  const page = req.params.page;  res.json({id: id, page: page});  }) |

## Socket.IO module

### Socket.IO

Socket.IO là một thư viện cho phép giao tiếp theo thời gian thực, hai chiều và dựa trên sự kiện giữa trình duyệt và máy chủ. Nó bao gồm:

* Một máy chủ Node.js
* Thư viện máy khách Javascript cho trình duyệt (cũng có thể được chạy từ Node.js)

Socket.IO KHÔNG phải là một WebSocket implementation. Mặc dù Socket.IO thực sự sử dụng WebSocket như một phương tiện, nhưng nó bổ sung thêm một số metadata cho mỗi gói: packet type, namespace và ack id khi một thông báo xác nhận là cần thiết. Đó là lý do tại sao một máy khách WebSocket sẽ không thể kết nối thành công với một máy chủ Socket.IO, và một khách hàng Socket.IO sẽ không thể kết nối với máy chủ WebSocket.

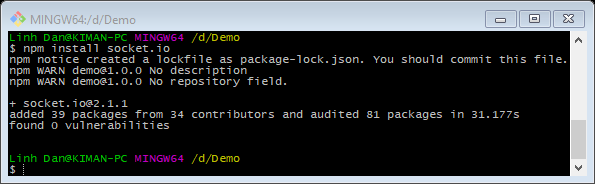
### Các đặc tính của socket.IO

1. **Độ tin cậy**: Được kết nối được thiết lập ngay
2. **Hỗ trợ tự động kết nối lại**: Máy khách bị ngắt kết nối sẽ thử kết nối lại mãi mãi, cho đến khi máy chủ có sẵn một lần nữa.
3. **Phát hiện ngắt kết nối**: Một cơ chế heartbeat mechanism được thực hiện ở cấp Engine.IO, cho phép cả máy chủ và ứng dụng khách biết khi nào bên kia không trả lời nữa.
4. **Hỗ trợ nhị phân**: Mọi cấu trúc dữ liệu tuần tự đều có thể được emit
5. **Hỗ trợ Multiplexing**: Cho phép tạo một số Namespace, sẽ hoạt động như các communication channel riêng biệt nhưng sẽ chia sẻ cùng một kết nối cơ bản.
6. **Hỗ trợ Room**: Trong mỗi Namespace, có thể xác định các channel tùy ý, được gọi là Room, mà các socket có thể join và leave. Sau đó, bạn có thể phát sóng tới bất kỳ room cụ thể nào và tiếp cận mọi socket đã join vào nó.

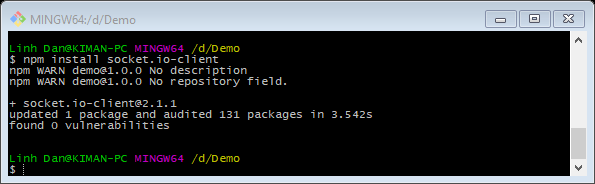
## Cài đặt

Nhập lệnh trong cmd

Cài đặt Socket.IO Server: npm install socket.io



Cài đặt Socket.IO Client: npm install socket.io-client



### Một số lệnh emit trong socket.io:

* socket.emit(‘name’, something): Gửi cho client
* socket.broadcast.emit(‘name’, something): Gửi cho tất cả client ngoại trừ người gửi
* socket.to(‘room1’).emit(‘name’, something): Gửi cho tất cả client trong room ‘room1’ ngoại trừ người gửi.
* socket.to (‘room1’). to (‘room2’). emit (‘name’, something): Gửi cho tất cả khách hàng trong ‘room1’ và room ‘room2’, ngoại trừ người gửi.
* io.of(‘namespaceRoom’).emit(‘name’, something): gửi cho tất cả các máy khách trong namespace 'namespaceRoom', bao gồm cả người gửi.
* io.to(‘${socketId}’).emit(‘name’, something): gửi đến socketid cá nhân (tin nhắn riêng).
* io.emit(‘name’, something): gửi tới tất cả khách hàng được kết nối.

Các socket cần ‘on’ để nhận các tin được emit tới: socket.on(‘name’, (something)=>{//do something})

Ví dụ:

Server:

|  |
| --- |
| var express = require('express'),  app = express(),  server = require('http').createServer(app),  io = require('socket.io')(server);  io.on('connection', (socket)=>{  socket.on('message', (data)=>{  socket.emit('hello' ,'Hello '+data);  });  }); |

Client website:

Trên website, để có thể sử dụng socket, cần khai báo thư viện socket.io trong script:

<script src=“/socket.io/socket.io.js”></script>

Và sau đó tạo kết nối tới server bằng cách viết trong script:

//kết nối tới server, không có namespace, gửi kèm token:

var socket = io({query: {‘token’ : myToken}});

|  |
| --- |
| <script src="/socket.io/socket.io.js"></script>  <script>  let token =localStorage.getItem(‘token’);  var socket = io({query : {'token' : token}});  socket.emit('join-room', token);  </script> |

Server sẽ nhận được kết nối ngay khi người dùng truy cập vào trang web có socket.

Dữ liệu người dùng kèm theo sẽ được socket lưu trong biến gọi là handshake. Để lấy được token, chỉ cần gọi socket.handshake.query.token

## Xử lý bất đồng bộ trong NodeJS

Cơ chế bất đồng bộ trong NodeJS đôi khi gây ra những khó khăn cho người lập trình.

Để xử lý bất đồng bộ, trong NodeJS sử dụng callback, Promise và Async/Await

### Callback

Callback là một đoạn mã thực thi được chuyển như một đối số cho mã khác, được dự kiến sẽ gọi lại (thực thi) đối số vào một thời điểm thuận tiện nào đó.

Nói một cách dễ hiểu, callback tức là truyền một đoạn code (**Hàm A**) này vào một đoạn code khác (**Hàm B**). Tới một thời điểm nào đó, Hàm A sẽ được hàm B gọi lại (**callback**).

Ví dụ:

|  |
| --- |
| Admin.authen = (email, password, callback)=>{  Admin.findOne().populate({path : 'user', match : {'email': email}}).  exec((error, data) => {  if (error) {  console.log(error);  return callback(error, null);  }  if (!data || typeof data === undefined) {  return callback(msg.error.login\_incorrect, null);  }  if (data && data.length !== 0) {  if (data.user.password !== password) {  return callback(msg.error.login\_incorrect, null);  } else {  var token = jwt.sign(JSON.stringify({'\_id' : data.\_id.toString(), 'email' : data.user.email, status: data.status, admin : data.user.admin, role : data.role}), config.admin\_secret\_key, {algorithm: 'HS256'});  return callback(null, {'token': token, 'id' : data.\_id});  }  }  });  } |

Tuy nhiên callback dễ gặp phải vấn đề callback hell.

### Promise

Trong phiên bản ES6, JavaScript đã được bổ sung thêm Promise. Nó là một thay thế tuyệt vời cho callbacks và hầu hết cộng đồng nhanh chóng chuyển sang sử dụng nó để thay thế cho callbacks. Code mới gần giống với code cũ, kết quả là trông dễ theo dõi và bảo trì hơn.

Ví dụ:

|  |
| --- |
| var getAllDevice = () => {  return new Promise((resolve, reject)=>{  Device.find({}).  exec().then(data) => {  resolve(data);  }).catch(e=>{  reject(e);  });  });  }; |

Tuy nhiên các vấn đề của callback vẫn chưa được giải quyết triệt để.

### Async/Await

Trong phiên bản ES7 gần đây nhất, Async/Await đã được bổ sung để việc viết code bất đồng bộ trong JavaScript tốt hơn, code dễ nhìn hơn và dễ sử dụng hơn.

|  |
| --- |
| var find = async (data)=>{  let admin = await Admin.findOne({\_id : data.\_id});  return admin;  } |

Tuy nhiên, không phải lúc nào cũng nên sử dụng async-await vì có thể làm chậm tiến trình không cần thiết khi các phương thức không thực sự cần chờ nhau. Ngoài ra, có những trường hợp mà Async/Await không sử dụng được và chúng ta phải sử dụng Promises.

## QUẢN TRỊ DỮ LIỆU VỚI MONGODB

## MongoDB

MongoDB là một cơ sở dữ liệu tài liệu mã nguồn mở cung cấp hiệu suất cao, tính sẵn sàng cao, và mở rộng quy mô tự động. Có các đặc tính:

1. **Hiệu năng cao**: MongoDB cung cấp độ bền dữ liệu hiệu năng cao
2. **Ngôn ngữ truy vấn phong phú**: MongoDB hỗ trợ một ngôn ngữ truy vấn phong phú để hỗ trợ các hoạt động đọc và ghi (CRUD) cũng như:

* Tập hợp dữ liệu
* Tìm kiếm văn bản và Truy vấn không gian địa lý.

1. **Khả năng mở rộng ngang**:

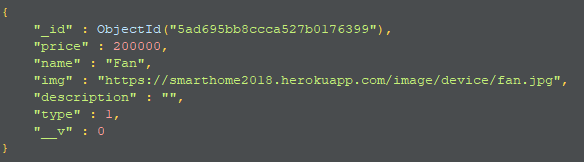
MongoDB cung cấp khả năng mở rộng ngang như là một phần của chức năng cốt lõi của nó

Hỗ trợ cho nhiều công cụ lưu trữ: MongoDB hỗ trợ nhiều công cụ lưu trữ. Ngoài ra, MongoDB cung cấp API công cụ pluggable storage cho phép các bên thứ ba phát triển các công cụ lưu trữ cho MongoDB.

## Cấu trúc dữ liệu trong Mongodb

Một bảng trong mySQL tương ứng là một Collection trong mongoDB.

Một bản ghi trong MongoDB là một document, là một cấu trúc dữ liệu gồm các field và value. Các document của MongoDB tương tự như các đối tượng JSON. Giá trị của các field có thể là: document, array và array document.



Hình 1. Ví dụ về document và collection trong mongoDB

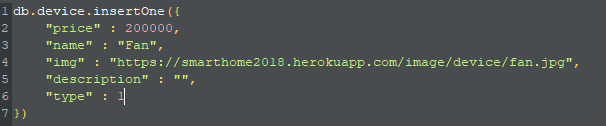
## CRUD trong Mongodb

Để thêm dữ liệu hoặc tạo một collection mới, sử dụng lệnh:

db.collection.insertOne()

hoặc db.collection.insertMany()

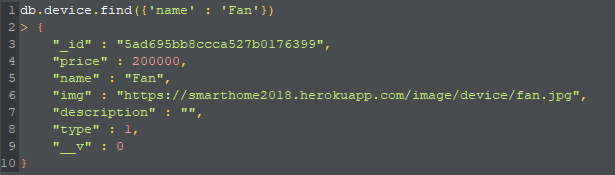
Ví dụ:



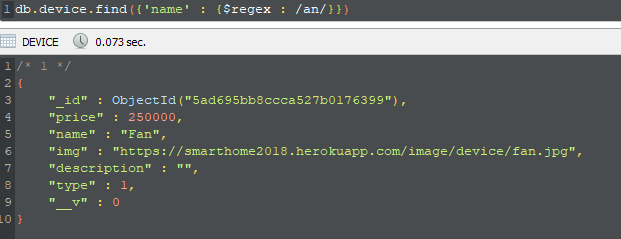
Đọc dữ liệu trong database, sử dụng lệnh:

db.collection.find()

Ví dụ:



Ngoài ra, nếu muốn tìm dữ liệu theo một chuỗi, có thể sử dụng regex:



Cập nhật dữ liệu trong database, sử dụng lệnh:

db.collection.updateOne()

db.collection.updateMany()

db.collection.replaceOne()

Khi update, nếu muốn chỉ thay đổi các field mong muốn, sử dụng $set trong query. Nếu không sử dụng $set, document đó sẽ được update lại toàn bộ các field.

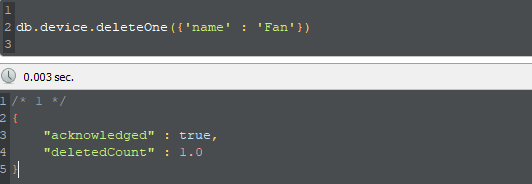


Xóa dữ liệu trong database, sử dụng lệnh:

db.collection.deleteOne()

db.collection.deleteMany()

Ví dụ:



## Kết nối mongoDB và NodeJS

## Mongoose module

Mongoose là một công cụ mô hình hóa đối tượng MongoDB được thiết kế để hoạt động trong một môi trường bất đồng bộ.

Các kiểu dữ liệu được phép trong mongoose:

* String
* Number
* Date
* Buffer
* Boolean
* Mixed
* ObjectId
* Array
* Decimal128
* Map

## CRUD trong mongoose

Tất cả mọi thứ trong Mongoose bắt đầu với một Schema. Mỗi Schema ánh xạ tới một collection trong mongoDB và định nghĩa các document trong collection đó bằng đối tượng là model.

#### **Tạo collection**

Để tạo Collection, trước hết cần kết nối mongoose tới CSDL mongoDB.

|  |
| --- |
| var mongoose = require('mongoose'),  uri = 'mongodb://masteradmin\_linhdan:nlu123456@ds040877.mlab.com:40877/smarthome',  options = {  poolSize: 20,  reconnectTries: 30,  reconnectInterval: 500  };  db = mongoose.createConnection(uri, options); |

Sau khi tạo kết nối, tạo định nghĩa cho một Schema.

|  |
| --- |
| var schemaDevice = new mongoose.Schema({  name : {type : String, text : true},  img : {type : String, default : '/image/device/no-images.png'},  description : {type : String},  price : {type : mongoose.Schema.Types.Double, default : 0},  type : {type : Number}  }, { toJSON : {virtuals: true}});  var Device = db.model('Device', schemaDevice, 'DEVICE'); |

Nếu có một collection nào tham chiếu tới collection này, có thể sử dụng virtual để tạo reference tới collection đó.

|  |
| --- |
| schemaDevice.virtual('keyOnOff', {ref : 'KeyOnOffDevice', localField : '\_id', foreignField : 'device', justOne : false}); |

Trong đó:

* ref là tên collection tham chiếu
* localField: field được tham chiếu (là field trong Schema đang set). Tại mongoDB, đối tượng \_id được tạo một cách tự động.
* foreignField: field tham chiếu

#### **Tìm kiếm document**

Mongoose hỗ trợ nhiều phương thức tìm kiếm, với các tham số truyền vào tương tự trong mongoDB.

Các phương thức:

* Model.find()
* Model.findById()
* Model.findByIdAndDelete()
* Model.findByIdAndRemove()
* Model.findByIdAndUpdate()
* Model.findOne()
* Model.findOneAndDelete()
* Model.findOneAndRemove()
* Model.findOneAndUpdate()

Ví dụ:

|  |
| --- |
| Device.find({'name': ‘Fan’).exec((error, data) => {  if(error){  console.error(error);  } else if (!error && data) {  console.log(data);  //do something  }  }); |

Khi muốn thực hiện lệnh join như trong mySQL, trong mongoose hỗ trợ phương thức populate, thay cho phương thức aggregate trong mongoDB.

Ví dụ: khi muốn lấy đối tượng keyOnOff tham chiếu tới collection Device của các document có name chứa từ là “ight” thì thực hiện lệnh:

|  |
| --- |
| Device.find({'name': {$regex: ‘ight’}}).populate({path: ‘keyOnOff’}).exec((error, data) => {  if(error){  console.error(error);  } else if (!error && data) {  console.log(data);  //do something  }  }); |

Nếu muốn join nhiều collection thì sẽ thêm lệnh .populate() vào kế tiếp lệnh populate đã có. Nếu muốn join một collection mà collection được join tham chiếu tới, thì bên trong phương thức populate()có thể Các phương thức khác của mongoose có thể dùng khi thực hiện lệnh:

* populate: join collection.
* where: điều kiện so sánh, tương đương với các param trong phương thức find()
* match: giá trị cần so sánh.
* select: các field cần lấy.
* sort: sắp xếp các document
* limit: giới hạn số document lấy ra.
* skip: bỏ qua số document được tìm thấy tính từ collection đầu tiên

#### **Thêm document**

Tạo một đối tượng model sau đó thực hiện lệnh save() để lưu.

|  |
| --- |
| let mDevice = new Device();  mDevice.name = data.name;  mDevice.img = data.img;  mDevice.description = data.description;  mDevice.price = data.price;  mDevice.type = data.type;  mDevice.save((error2, result)=>{  if(error2){  console.log(error2);  } else {  console.log(result)  }  }); |

Đối tượng result trả về là đối tượng Device đã tạo với đầy đủ các field trong database.

#### **Sửa document**

Khi muốn cập nhật lại dữ liệu, sử dụng $set để thay đổi các giá trị được truyền vào. Nếu muốn cập nhật nhiều document, cần chọn field làm key thích hợp và thêm options {multi : true}.

|  |
| --- |
| Device.update({price :{$lt : 200000}, type : 2}, {$set : data}, {multi : true}).exec((error2, result) => {  if(error2){  console.log(error2);  } else {  console.log(result)  }  }); |

#### **Xóa document**

|  |
| --- |
| Device.remove({'\_id' : \_id}).  exec((error) => {  if(error){  console.log(error);  } else {  console.log(id)  }  }); |

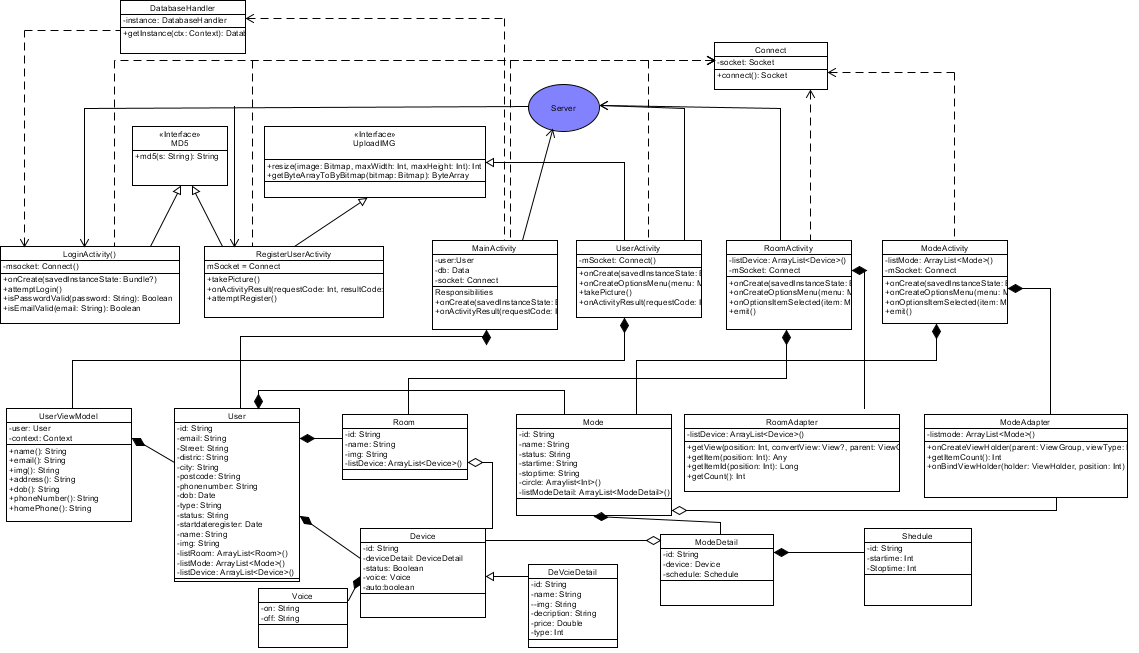
chương 3 PHÂN TÍCH THIẾT KẾ

# USE CASE

**Danh sách các Actor**

* User: Người dùng cuối
* Admin: Quản trị viên của hệ thống

# 3.4 Class diagram

* 

## 3.4.1 Lớp User

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu dữ liệu | Mô tả |
| id | String | Id khách hang |
| Email | String | Email khách hang |
| password | String | Mật khẩu |
| city | String | Thành phố |
| district | String | Quận/Huyện |
| street | String | Tên đường |
| postcode | Int | Mã tỉnh |
| phonenumber | String | Số điện thoại di động |
| homePhong | String | Số điện thoại bàn |
| dob | Date | Ngày sinh |
| Type | String | Loại khách hàng |
| Status | Boolean | Trạng thái(true hoạt động, false không) |
| Startdateregister | Date | Ngày đăng ký |
| Name | String | Tên khách hàng |
| Img | String | Hình ảnh khách hàng |

## 3.4.2 Lớp Room

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu dữ liệu | Mô tả |
| id | String | Id phòng |
| name | String | Tên phòng |
| Img | String | Hình ảnh của phòng |
| listDevice | arrayList<Device>() | các thiết bị trong phòng |

## 3.4.3 Lớp Mode

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu dữ liệu | Mô tả |
| id | String | Id chế độ |
| name | String | Tên chế độ |
| status | Boolean | Trạng thái chế độ(true chế độ mở, false chế độ tắt) |
| Startime | Int | Thời gian chế độ mở |
| Stoptime | Int | Thời gian chế độ tắt |
| Circle | ArrayList<Int>() | Vòng lập theo tuần |
| listModeDetail | ArrayList<ModeDetail>() | Danh sách chi tiết các thiết bị trong chế độ này |

## 3.4.4 Lớp ModeDetail

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu dữ liệu | Mô tả |
| id | String | Id ModeDetail |
| Device | Device | Thiết bị |
| Schedule | Schedule | Lịch bật tắt của thiết bị |

## 3.4.5 Lớp Device

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu dữ liệu | Mô tả |
| id | String | Id thiết bị |
| deviceDetail | DeviceDetail | Chi tiết thiết bị |
| Name | String | Tên thiết bị |
| Status | Boolean | Trạng thái(true mở, false tắt) |
| Voice | Voice | Giọng nói điều khiển thiết bị |
| Auto | Boolean | Chế độ tự động(true tự động, false không) |

## 3.4.6 Lớp Schedule

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu dữ liệu | Mô tả |
| Ontime | Int | Thời gian mở của thiết bị |
| Offtime | Int | Thời gian tắt của thiết bị |

## 3.4.7 Lớp Voice

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu dữ liệu | Mô tả |
| On | String | Từ khóa để mở thiết bị |
| Off | String | Từ khóa để tắt thiết bị |

## 3.4.8 Lớp DeviceDetail

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu dữ liệu | Mô tả |
| Id | String | Id loại thiết bị |
| Name | String | Tên loại thiết bị |
| Img | String | Hình ảnh thiết bị |
| Description | String | Mô tả thiết bị |
| Price | Double | Giá tiền loại thiết bị |
| Type | Int | Loại thiết bị |