# **Giới thiệu bản thân**

Chào anh/chị, em là Lê Tạ Hà Phương - một lập trình viên vừa tốt nghiệp. Em được đào tạo full stack tại trung tâm CodeGym. Tại đây, với những kiến thức mà em được học, sẽ là cơ sở để em có thể tự học hỏi và phát triển bản thân hơn nữa. Bên cạnh đó với kinh nghiệm làm việc trong những công ty của Nhật Bản, em đã có cơ hội tiếp cận và làm việc trong môi trường doanh nghiệp với những quy trình chặt chẽ. Em tin rằng mình sẽ dễ thích nghi và hoà nhập với công ty cũng như sớm nắm bắt được công việc. Em là người cởi mở, thích học hỏi và có tính cầu toàn trong công việc. Với những ưu điểm trên thì e có thể dễ dàng tiếp thu kiến thứ, học hỏi cái mới. Tuy nhiên, đôi khi tính cầu toàn đó cũng thỉnh thoảng đem lại cho e chút trở ngại, em muốn công việc của mình phải tốt nhất có thể trong khả năng nên đôi khi sẽ làm cho công việc chậm lại. Đến với buổi phỏng vấn hôm nay, em mong sẽ nhận được cơ hội để ứng tuyển vào vị trí mà Quý công ty đang tuyển dụng.

# **API**

API là viết tắt của "Application Programming Interface" (giao diện lập trình ứng dụng), là một tập hợp các quy tắc, giao thức và công cụ mà cho phép các ứng dụng khác nhau có thể trao đổi dữ liệu và tương tác với nhau.

API là một cách để các ứng dụng tương tác với nhau mà không cần biết chi tiết bên trong của nhau. Nó cung cấp một giao diện đơn giản để các nhà phát triển có thể sử dụng để kết nối các ứng dụng của họ với các ứng dụng khác và sử dụng các chức năng của chúng.

API thường được sử dụng trong phát triển phần mềm, đặc biệt là trong các ứng dụng web và di động. Các API được sử dụng để tương tác với các dịch vụ web khác nhau, như các trang web mạng xã hội, cơ sở dữ liệu, hệ thống thanh toán, v.v. Nó cho phép các ứng dụng phát triển nhanh chóng và kết nối với nhiều dịch vụ khác nhau mà không cần phải xây dựng lại chúng từ đầu.

# **Regex**

Regex là viết tắt của Regular Expression, có nghĩa là biểu thức chính quy. Đây là một chuỗi các ký tự đặc biệt được định nghĩa để tạo nên các mẫu (pattern) để tìm kiếm, thay thế hoặc phân tích cú pháp các chuỗi và đoạn ký tự. Regex là một công cụ cực mạnh cho xử lí chuỗi trong nhiều ngôn ngữ lập trình như PHP, JavaScript, Python…

# **Webservice**

Là tập hợp các giao thức và tiêu chuẩn được sử dụng để trao đổi dữ liệu giữa các ứng dụng hoặc các hệ thống.

* Là phương thức giao tiếp ứng dụng qua mạng
* Là ứng dụng hoặc thành phần ứng dụng để giao tiếp
* Là tập hợp các tiêu chuẩn hoặc giao thức để trao đổi thông tin giữa hai thiết bị hoặc ứng dụng

# **Cookie và Local storage**

* Server có thể tuy cập vào cookie nhưng không thể truy cập vào local storage
* Cookie có thể chữa 2MB dữ liệu còn local storage có thể chưa 5MB. Vì vậy cookie không nên chứa những phần dữ liệu dung lượng lớn
* Dữ liệu được lưu trong cookie có hạn sử dụng còn dữ liệu trên local storage không hết hạn chỉ có 1 cách là dọn dẹp local storage bằng cách thủ công bằng code hoặc người dùng dọn dẹp bộ nhớ của trình duyệt thủ công
* Trừ khi sử dụng javascript thì khác khó khăn để đọc và viết cookie với document.cookie còn với local storage thì nó thực thi lại web storage API dễ dàng để đọc và ghi dữ liệu
* Tóm lại:
  + Nếu bạn muốn lưu dữ liệu và có thể được truy cập từ cả client và server thì dùng cookie
  + Nếu bạn lưu data có dung lượng lớn hoặc không có hạn sử dụng thì dùng local storage
  + Nếu bạn muốn có method để dễ dàng sử dụng để truy cập và sửa chữa thi dùng local storage

# **Spring Core**

1. **IOC**

Inversion of control là cơ chế đảo ngược quyền điều khiển. Nó giúp làm thay đổi luồng điều khiển của chương trình một cách linh hoạt. Thường dùng với Dependency Injection

Dependency Injection là 1 kỹ thuật, 1 design pattern cho phép xoá bỏ sự phụ thuộc

Dependency Injection (DI) là một mô hình lập trình trong đó đối tượng được cung cấp với các phụ thuộc của nó (dependencies) từ bên ngoài, thay vì nó tự tạo ra các phụ thuộc đó. Trong DI, các phụ thuộc của một đối tượng được truyền vào đối tượng đó thông qua hàm khởi tạo, phương thức hoặc các thuộc tính.

Mục đích chính của DI là giảm sự phụ thuộc giữa các đối tượng và tăng tính linh hoạt của chương trình. Nó cho phép bạn thay đổi các phụ thuộc của một đối tượng mà không cần thay đổi mã nguồn của đối tượng đó. Điều này rất hữu ích trong việc phát triển các ứng dụng lớn hoặc các hệ thống phức tạp, nơi sự phụ thuộc giữa các thành phần có thể trở nên rất phức tạp và khó kiểm soát.

1. **Spring IOC**

IOC container là thành phần thực hiện IOC

Tạo các đôi tượng, lắp rắp, cấu hình và quản lý vòng đời (Đọc thông tin từ các file XML và thực thi chúng)

Sử dụng DI để quản lý các thành phần, đối tượng để tạo nên 1 ứng dụng (Các thành phần đôi tượng này gọi là spring Bean)

Có 2 kiểu IOC là: BeanFactory và ApplicationContext

Cả 2 đều là các interface thực hiện IoC container. ApplicationContext được xây dựng như BeanFactory nhưng có thêm 1 số chức năng mở rộng như tích hợp với Spring AOP, xử lý massage, context cho web application

Khi chúng ta định nghĩa một Spring Bean trong file cấu hình của ứng dụng, chúng ta chỉ định tên hoặc kiểu dữ liệu của các phụ thuộc mà Bean đó cần sử dụng. Spring Container sẽ tự động cung cấp các phụ thuộc đó khi tạo ra Bean đó.

Việc Spring Container quản lý các đối tượng và phụ thuộc của chúng giúp cho việc tạo ra các đối tượng và quản lý các phụ thuộc của chúng trở nên dễ dàng và linh hoạt hơn. Nó cũng giúp cho việc kiểm thử và tái sử dụng các đối tượng trở nên dễ dàng hơn.

1. **Spring bean**

Spring bean là các object trong Spring framework, được khởi tạo thông qua spring container. Bất kỳ class nào POJO nào cũng có thể là Spring Bean nếu nó được cấu hình và khởi tạo thông qua container bằng việc cung cấp các thông tin cấu hình

Có các Bean scope sau:

* + Singleton: Chỉ duy nhất một thể hiện của Bean sẽ được tạo cho mỗi container. Đây là scope mặc định cho spring bean. Khi sử dụng scope này cần chắc chắn rằng các bean không có thuộc tính được share
  + Prototype: Một thể hiện của bean sẽ được tạo ra cho mỗi lần yêu cầu
  + Request: giống vơi prototype scope, tuy nhiên nó dùng cho ứng dụng web, một thể hiện của bean sẽ được tạo cho môi http request
  + Session: mỗi thể hiện của bean sẽ tạo cho mỗi http session
  + Global-session: được sử dụng để tạo global session bean cho các ứng dụng portlet

1. **Spring AOP**

Aspect Oriented Programming (AOP) là 1 kỹ thuật lập trình dùng để tách logic chương trình thành các phần riêng biệt.

Trong Spring AOP có 4 loại advice được hỗ trợ:

* + Before advice: chạy trước khi method được thực thi
  + After returning advice: chạy sau khi method trả về một kết quả
  + After throwing advice: chạy khi method ném ra 1 exception
  + Around advice: chạy khi method được thực thi (Bao gồm cả 3 loại advice trên)

Các khái niệm trong AOP

* + Join point: là các điểm trong chương trình (ví dụ: điểm thực thi, điểm xử lý exception…) spring chỉ hỗ trợ method excution join point.
  + Advice: một hành động thực hiện ở join point
  + Pointcut: là expression language giúp kết nối với join point
  + Introduction: cho phép introduce với các new interface với bất kỳ object advice nào.
  + Target object: Object sẽ được advice
  + Aspect: là một class bao gồm các advice và các join point
  + Interceptor: là một aspect chỉ có duy nhất một advice
  + AOP porxy: dùng để cài đặt các aspect
  + Weaving: tiến trình nôi các aspect với các object, type để tạo nên advised object

# **Design Pattern**

Lầ các khuôn mẫu thiết kế

Có 3 loại design pattern là: Creational, Structural, Behavioural

* + Creational (5 mẫu): tập hợp các giải pháp liên quan đến việc khởi tạo object như thế nào
  + Structural (7 mẫu): tập hợp các giải pháp liên quan đến thiết lập kết cấu, liên hệ giữa các đối tượng
  + Behavioral (11 mẫu): tập hợp các giải pháp liên quan đến các hành vi của đối tượng và giao tiếp giữa các đối tượng khác nhau

Các design pattern trong framework spring

Tài liệu tham khảo:

* + Sách: head first design pattern, design patterns for dummies
  + Link: [www.tutorialspoint.com/designpattern](http://www.tutorialspoint.com/designpattern), sourcemaking.com/design patterns

Tìm hiểu thêm

* + Anti-pattern
  + Clean Code
  + SOLID principle
  + Refactoring

Một số design pattern thường gặp

* + **Singleton**

Được sử dụng khá phổ biến. Rất nhiều framework sử dụng loại này. Pattern này được sử dụng khi ta muốn tạo một object từ một class và muốn chắc chắn rằng chỉ có một object được tạo ra từ nó. Implement cho design pattern này:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class SingletonSample {  private static SingletonSample instance = null;    private SingletonSample() {  }    public static SingletonSample getInstance() {  if(instance == null) {  instance = new SingletonSample();  }  return instance;  }  } |

Một constructor được khởi tạo private để tránh truy cập từ bên ngoài vào. Cũng cần tạo một biến static và method getInstance() đảm bảo rằng chỉ một instance của class được tạo ra

Nên sử dụng thêm từ khoá **synchronized**(Thêm sau static và trước kiểu dữ liệu) để đảm bảo chỉ có một luồng có thể thực thi phương thức getInstance tại 1 thời điểm. Việc này đảm bảo tính đơn nhất của dối tượng singleton và tránh tình trạng tạo ra nhiều đối tượng singleton

* + **Initialization on demand holder**

Cái này khá giông với Singleton bên trên nhưng nó có một ưu điểm tốt hơn đó là khi làm việc với thread thì pattern này sẽ giúp thread safe, trường hợp của singleton nếu không sử lý đồng bộ có thể tạo 2 íntance khác nhau. Implement cho initialization on demand holder là:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class SingletonSample {  private SingletonSample() {  }  public static SingletonSample getInstance() {  return SingletonSampleHolder.INSTANCE;  }    private static class SingletonSampleHolder {  private static final SingletonSample INSTANCE = new SingletonSample();  }  } |

Đối tượng này sẽ không khởi tạo cho đến khi method getInstance được gọi, với ưu điểm này thì giúp thread safe

* + **Strategy và factory pattern**

2 pattern được sử dụng rất phổ biến

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | // class biểu diễn cho cái nhà  public class House implements Building {  public String getType(){  return "house"  }  }    // class biểu diễn cho lâu đài  public class Edifice implements Building {  public String getType(){  return "edifice"  }  }    public class BuildingFactory {  private static Map<String, Building> instances;    static {  instances = new HashMap<>();    instances.put("house", new House());  instances.put("edifice", new Edifice());  }    public static <T extends Building> T getBuilding(String type){  return (T) instances.get(type);  }  }  // Khởi tạo object  Building building = BuildingFactory.getBuilding("house"); |

Nếu cần một loại building nào đấy, ta chỉ cần truyền vào loại và nó sẽ trả về object của loại đó hoặc null nếu như không có instance cho loại này, hữu ích khi dùng đa hình

* + **Builder**

Nhiều class khi tạo một object cần phải truyền vào rất nhiều tham số, trong trường hợp như vậy thì khi ta sử dụng constructor hoặc sử dụng get, set đều khiến code trở nên rối và dài dòng. Builder pattern sẽ giúp ta giải quyết vấn đề này.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95 | // Ta có một class Product  public class Product {  private String id;  private String name;  private String description;  private Double value;    private Product(Builder builder) {  setId(builder.id);  setName(builder.name);  setDescription(builder.description);  setValue(builder.value);  }    public static Builder newProduct() {  return new Builder();  }    public String getId() {  return id;  }    public void setId(String id) {  this.id = id;  }    public String getName() {  return name;  }    public void setName(String name) {  this.name = name;  }    public String getDescription() {  return description;  }    public void setDescription(String description) {  this.description = description;  }    public Double getValue() {  return value;  }    public void setValue(Double value) {  this.value = value;  }    // định nghĩa một inner class bên trong class product    public static final class Builder {  private String id;  private String name;  private String description;  private Double value;    private Builder() {  }    public Builder id(String id) {  this.id = id;  return this;  }    public Builder name(String name) {  this.name = name;  return this;  }    public Builder description(String description) {  this.description = description;  return this;  }    public Builder value(Double value) {  this.value = value;  return this;  }    public Product build() {  return new Product(this);  }  }  }  // Khi muốn tạo một object ta chỉ cần khai báo như sau  Product product = Product.newProduct()  .id(1l)  .description("TV 46'")  .value(2000.00)  .name("TV 46'")  .build();  // nhìn vào có thể thấy code khá dễ hiểu trong việc mô tả các trường trong trường hợp cần nhiều biến cần truyền vào |

* + **Template method**

Pattern này được áp dụng trong trường hợp chúng ta có nhiều method chung nhưng khác nhau về hành vi của chúng, pattern này thì hoàn toàn dựa trên tính đa hình

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public abstract class Animal {  public abstract void makeSound();  public abstract void eatFood();  public abstract void sleep();    public void doEveryday(){  makeSound();  eatFood();  sleep();  }  }    public class Dog extends Animal {  public void makeSound(){  //bark!  }    public void eatFood(){  //eat dog food  }    public void sleep(){  //sleep a lot!  }  } |

* + **State**

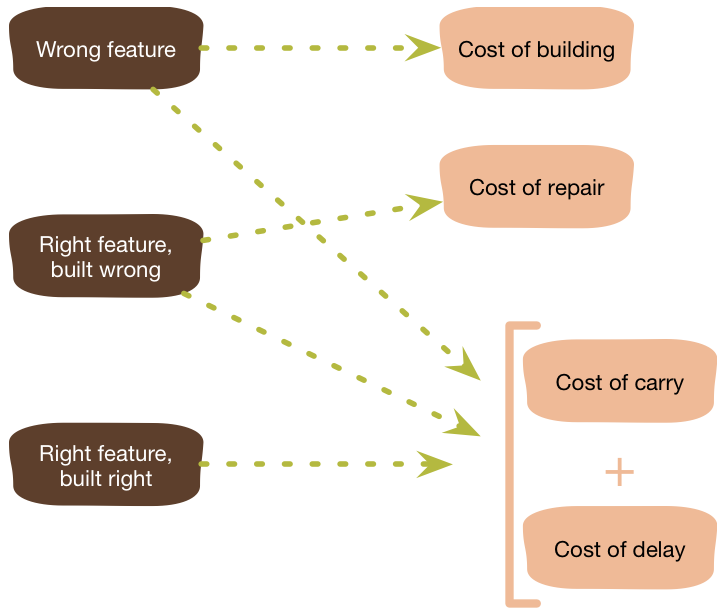
Rất nhiều object có trạng thái riêng của chúng. Ví dụ như đài radio có 2 trạng thái là bật và tắt. Chúng ta sẽ biểu diễn dươi dạng hương đối tượng

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | // một interface cho trạng thái của radio  public interface RadioState {  void execute(Radio radio);  }    // class radio  public class Radio {  private boolean on;  private RadioState state;    public Radio(RadioState state){  this.state = state;  }    public void execute(){  state.execute(this);  }    public void setState(RadioState state){  this.state = state;  }    public void setOn(boolean on){  this.on = on;  }    public boolean isOn(){  return on;  }    public boolean isOff(){  return !on;  }  }  // 2 class cho trạng thái của radio  public class OnRadioState implements RadioState {  public void execute(Radio radio){  //throws exception if radio is already on  radio.setOn(true);  }  }    public class OffRadioState implements RadioState {  public void execute(Radio radio){  //throws exception if radio is already off  radio.setOn(false);  }  }    // Khi thực thi kết quả như sau  Radio radio = new Radio(new OffRadioState()); //initial status  radio.setState(new OnRadioState());  radio.execute(); //radio on  radio.setState(new OffRadioState());  radio.execute(); //radio off |

# **Design principles**

Là những nguyên tắc thiết kế cơ bản, để giúp ta thiết kế một sản phẩm dễ dàng để phát triển, dễ dàng để bảo trì, hay đơn giản là có thể vận hành tốt.

* **DRY**
  + Viết đầy đủ Don’t repeat yourself
  + Nội dung là đừng lặp lại các đoạn code, hãy xử lý ở nhiều nơi khác nhau. Áp dụng trong code, documentation, database schemas…
  + Giúp cho maintain code tốt hơn, giải quyết vấn đề thay đổi logic code của một đoạn xử lý dễ dàng hơn sau này
* **KISS**
  + Viết đầy đủ Keep It Simple, Stupid
  + Đề cặp đến việc tính đơn giản nên được đặt là mục tiêu cảu việc thiết kế hệ thống, và hầu hết các hệ thống sẽ làm việc tốt nhất khi nó được giữ ở trạng thái đơn giản thay vì phức tạp.
* **YAGNI**
  + Viết đầy đủ You aren’t gonna need it
  + Miêu tả rằng bạn không nên đưa vào những chức năng cho đến khi chúng thực sự cần thiết, hay chỉ tiến hành implement những chức năng mà bạn cảm thấy là mình cần đến nó, chứ không phải là bạn cảm thấy sau này có thể sẽ cần đến nó.



* + Cost of building: Khi bạn lầm chức năng mà cuối cùng không cần đến nó. Nó khiến bạn tốn nhiều effort trong việc lên thiết kế, code, test…
  + Cost of repair: Khi chức nnagw mà bạn hướng đến là cần thiết, nhưng bạn lại implement theo một cách không hợp lý. Nó sẽ khiến bạn tốn effort để lên kế hoạch lại, code lại, và test lại chức năng đã làm, bởi nó không thực sự là những gì bận cần
  + Cost of delay: Bạn đang mất thời gian vào một chức năng mà mình chưa cần đến ở thời điểm hiện tại, nó kéo theo việc những chức năng cần thiết ở thời điểm hiện tại không thể được hoàn thiện và release sớm.
  + Cost of carry: Dù trong bất kỳ trường hợp nào, bạn cũng sẽ gặp phải vấn đề này. Bạn đang thêm một lượng code mới vào trong project của mình, khiến cho hệ thống phức tạp hơn và sẽ mất công để maintain, modify, debug hơn
* **Boy Scout Rule**
  + Đừng làm 1 đoạn code đã có sãn trở nên tồi tệ hơn
* **Separation of Concerns**
  + Phân tách phụ thuộc hoặc chia tách quan hệ
* **SOLID**
  + **Single Responsibility Principle:** mỗi phần của code chỉ nên phụ trách 1 nhiệm vụ. Nếu bạn có 1 phần code phụ trách 1 lúc nhiều nhiệm vụ, bạn nên chia nó thành nhiều phần nhỏ hơn với mỗi phần một nhiệm vụ riêng biệt.
  + **Open Closed Principle:** Code của bạn nên mở để mở rộng nhưng đóng cho chỉnh sửa. Nghĩa là bạn có thể thêm tính năng mới vào code mà không được thay đổi code đã tổn tại
  + **Liskov Substitution Principle:** Bạn có thể dùng class con mỗi khi bạn muốn dùng class cha. Có nghĩa là class con nên là phiên bản tốt hơn class cha và không nên gây lỗi code sử dụng class cha.
  + **Interface Segregation Principle:** Bạn nên tạp ra các interfaces nhỏ cho nhưng nhóm chức năng liên quan cụ thể.
  + **Dependency Inversion Principle:** lớp cấp cao không nên phụ thuộc vào lớp cấp thấp nhưng cả hai nên phụ thuộc và lớp trừu tượng. Nghĩa là code của bạn không nên phụ thuộc vào lớp cụ thể hoặc chức năng, nên phụ thuộc vào các lớp trừu tượng.