1. **FOUNDATION SPRING**
2. **Getting started**

* Khi ứng dụng khởi chạy, các component cần được khởi tạo và giới thiệu cho nhau. Spring giới thiệu 1 *container*  như là phần core của spring thường được gọi là *Spring appilcation context* thực hiện tạo và quản lý các component của ứng dụng
* Các component (thưởng được gọi là các *bean*) sẽ được nối với nhau bên trong *Spring application Context.* Việc thực hiện nối các *bean* với nhau được thực hiện dựa trên khuân mẫu (pattern) *Dependence Injection (DI):* Thay vì các component phải khởi tạo và quản lý vòng đời của các bean thì các DI application sẽ dựa trên các thực thể riêng biệt để tạo và duy trì tất cả các component và sẽ tiêm (*inject*) chúng cho bean cần.
* @Configuration: chỉ thị cho spring biết rằng đây là một class cấu hình để cùng cấp bean cho *Spring Application Context.*
* @Bean : được đặt trên các method chỉ thị rằng các object được trả ra từ hàm này sẽ được thêm vào như là bean trong application contex( mặc định bean IDs sẽ cùng tên với tên hàm)
* Cấu trúc của 1 spring project có thể bao gồm các file hoặc thư mục :
  + *mvnw và mvnw.cmd:* là các script maven
  + *pom.xml:* Maven build specification.
  + *XXXApplication.java:* main class của Spring Boot, thực hiện việc khởi động project
  + *Application.properties:* file cấu hình của ứng dụng
  + *Static:* thư mục chứa nội dung tĩnh (ảnh, stylesheet, …) mà ta cần phục vụ cho brower.
  + *Template:* thư mục chứa file sử dụng để render nội dung cho brower
  + *XXXApplicationTest.*java: class test đơn giản để đảm bảo rằng Spring application contex khơi tạo thành công
* *@SpringBootApplication:* là sự tổng hợp của 3 annotation:
  + *@SpringBootConfiguration:* thiết kết rằng class này là một class cấu hình.
  + *@EnableAutoConfiguration:* cho phép cấu hình tự động của Spring Boot.
  + *@ComponentScan:* cho phép Spring sẽ tự động tìm kiếm và đăng kí các class có đánh dấu bằng các annotation (@Component, @Service…) là các bean trong Spring Appliaction Context.
* *Các annotation @Service, @Component, @Repository, @Controller*: về cơ bản đểu có tác dụng giống nhau: đánh dấu class là một component của Spring. Mục đích chính của việc sử dụng các annotation này là để đánh dấu vai trò của class.
* *DEVTOOL*:
  + *Một số tiện ích*:
    - *Tự động khởi động ứng dụng khi có code thay đổi*
    - *Tự động làm mới trình duyệt khi có thay đổi về tài nguyên*
    - *Tự đông vô hiệu hóa bộ đệm mẫu*
    - *Cài đặt H2 Console khi có H2 DataBase được sử dụng*
  + Tuy nhiên nó chỉ thích hợp cho môi trường dev, khi deploy lên môi trường production cần vô hiệu hóa.

1. **Developing Web Application**

* Controller đóng vai trò quan trọng trong Spring MVC. Công việc của nó là xử lý các HTTP request để: hoặc trả về 1 view để render HTML hoặc trả về dữ liệu trực tiếp thông qua body của response.
* *@RequestMapping:* đánh dấu trên level class, xác định loại request mà controller sẽ xử lý.
* *@GetMapping:* được đánh dấu trên method cùng với *@RequestMapping* xác định HTTP method request nhận được từ đường dẫn “/design” và sẽ xử lý chúng
* *@ModelAttribute: được sử dụng trong spring MVC* đánh dấu một hàm, hàm này sẽ thực hiện thêm các giá trị cho model.

1. **Securing Spring**

* Spring Security đưa ra một vài lựa chọn cho việc cấu hình lưu trữ user bao gồm:
  + In-memory user store
  + JDBC-base user store
  + LDAP-backend user store
  + Custome user detail service
* Ta có thể sử dụng một trong các cấu hình này bằng cách overrider lại hàm *configure()* được định nghĩa trong WebSecurityConfigurerAdapter

1. **Working with configuration properties**

* Có 2 loại cấu hình trong Spring:
  + Bean wiring: khai báo 1 component sẽ được tạo như là 1 bean trong spring application context và cách chúng được inject cho các bean khác
  + Property Injection: thiết lập các thuộc tính vào bean trong Spring applocation context
* *@Bean:*  các method được dánh dấu vừa có thể tạo bean vừa có thể set các giá trị cho nó
* *Sping environment abtraction:* thực hiện “trìu tượng hóa” các tài nguyên ban đầu làm cho các bean cần đến tài nguyên đó có thể tiêu thụ chúng, nó sẽ tổng hợp các nguyên vào một nguồn duy nhất. Spring environment thực hiên lấy từ nhiều nguồn bao gồm:
  + JVM system properties
  + Operating system environment variable
  + Command-line argument
  + Application property configuration file
* Cấu hình data source:

spring:  
datasource:  
url: jdbc:mysql://localhost/tacocloud  
username: tacodb  
password: tacopassword  
driver-class-name: com.mysql.jdbc.Driver

* Datasource bean được tự động cấu hình các tham số theo Tomcat’s JDBC nếu có chứa trong classpath, nếu không Spring Boot sẽ tự động tìm kiếm và sử dụng 1 trong 2 implement DB connection pool:
  + HikariCP
  + Commons DBCP 2
* *Server.port*: thiết lập port cho servler container
* *welcome: ${spring.application.name}*: sử dụng giá trị được cấu hình từ tham số khác
* *@ConfigurationProperties (prefix=””):* sử dụng để inject các giá trị từ file cấu hình (application.yml, application.properties) vào bean
* *@Profile:* sử dụng để thiết kế bean áp dụng theo profile xác định

1. **Integrated Spring**
2. **Creating REST serivce**

* ***@****RestController*: sử dụng với 2 mục đích:
  + Đánh dấu class như là một bean component
  + Xử lý các request và trả về kết quả trong body (bằng với việc sử dụng @Controller trong class và @ResponseBody trên các method trong class).
* *@CrossOrigin:* sử dụng để bỏ qua việc chặn CORS (Cross-Origin Resource Share)
* *@GetMapping (“/{id}”):* xử lý GET request với path “desing/{id}” với id là một giá trị xác định.
* *@PathVariable(“id”) :* là giá trị thực sự của id trong @GetMapping
* *(@RequestBody Class object):* body của request sẽ được convert thành object
* *@ResposeStatus(HTTPStatus.OK, CREATED…) :*  mã trả về nếu thực hiện thành công
* *@PutMapping vs @PathMapping:* Cả hai đều dùng trong trường hợp thay đổi dữ liệu. Thông thường method PUT được dùng trong trường hợp thay thế toàn bộ dữ liệu, còn nếu muốn cập nhật một số trường ta sẽ dùng method Path
* *@DeleteMapping:* khai báo 1 hàm dùng để xáo dữ liệu. nếu tài nguyên không tồn tại thì EmptyResultDataAccessException sẽ được ném ra, tuy nhiên ta không nên xư lý gì khi catch nó cả, thường mã trả về là *HTTPStatus.NO\_CONTENT = 204*

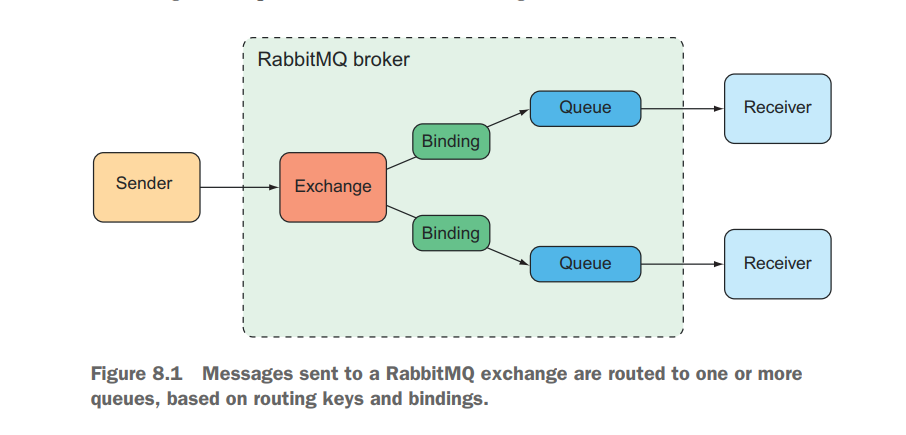
1. **Sending message asynchronously**

* Việc xử lý các message bất đồng bộ thông thường sẽ phải dùng đến các message borker, trong đó có một số loại phổ biến : kafka, RabitMQ, ActiveMQ, JMS

### 6.1 JMS

* Để push các thông điệp có thể dùng 2 hàm được hỗ trợ sẵn: *send() và convertAndSend(),* trong đó hàm convertAndSend() chấp nhận một String hoặc Object, sau đó Object sẽ được convert thành 1 *Message* trước khi nó gửi đi.
* Một số dạng convert messge được Spring hỗ trợ bao gồm:
  + MappingJackson2MessageConverter: sử dụng thư viện Jackson để convert thông điệp từ JSON và thành dạng JSON
  + MarshallingMessageConverter: sử dụng JAXB để convert dữ liệu từ và thàng XML
  + MessagingMessageConverter: convert từ 1 Message từ và thành 1 message sử dụng *MessageConverter*  co payload và sử dụng JSMHeader để convert header thành dạng tiêu chuẩn
  + SimpleMessageConverter: đây là dạng mặc định sẽ được sử dụng convert:
    - String <->TextMessage ,
    - byte <->ByteMessage
    - Maps<-> MapMessage
    - Serializable <-> ObjectMessage
* Một số trường hợp ta cần sử đổi dữ liệu trước khi thưc hiện push đi, có thể sử dụng các hàm custom
* Có 2 mô hình để nhận message từ : pull model à push model (JMS template sử dụng hình thức pull model)
  + Pull model: gọi method để yêu cầu message, thread sẽ block cho đến khi message hiện hữu
  + Push model: định nghĩa một listener sẽ được gọi bất cứ khi nó có message, cách này được ưu tiên sử dụng hơn do không phải block thread nhưng có nhược điểm sẽ có thể bị tải cao nếu message đến quá nhanh
* Với trường hợp pull model: JMS cung cấp hàm receive() và receiveAndConvert():
  + *Receive*(): hàm trả về 1 *Message(có chứa cả thông tin header và payload)* và cần phải convert về theo mong muốn
  + *receiveAndConvert:* sử dụng trong trường hợp ta chỉ quan tâm đến pay load
* Trong trường hợp push model:
  + Để tạo một listener tương tác với JMS message, ta chỉ cần thêm annotation *@JmsListener* trên một method

### RabbitMQ và AMQP



* RabbitMQ là một phiên bản nổi bật của AMQP, khác với JMS nó được đánh địa chỉ với tên của *exchange* và *routingkey* làm cho mô hình giữa queue và receiver trở lên decouple
* Khi có message đến RabbitMQ Broker, nó sẽ được trao đổi (exchange) tới nơi nó đươc đánh địa chỉ, exchange sẽ có nhiệm vụ định tuyến tới một hoặc nhiều queue dựa vào:
  + Kiểu exchange
  + Binding giữa exchange và queue
  + Giá trị routing key của message
* Một vài loại exchange:
  + *Default:* là một loại exchange đặc biệt, tự động create bởi broker. Nó định tuyển thông điệp tới queue có tên trùng với routing key, tất cả các queue sẽ tự động bao bởi exchange mặc định
  + *Direct:* Tự động định tuyến thông điệp tới queue mà binding key trùng tên với routing key của message
  + *Topic:* Định tuyến một message tới 1 hoặc nhiều queue mà binding key (có thể là các ký tự đại diện) trùng khớp với routing key của message
  + *Fanout:* Định tuyến messgae tới tất cả các queue ngoại trừ các quue liên quan đến rounting key hoặc binding key
  + *Header:* tương tự như toppic, ngoại trừ việc định tuyến sẽ liên quan đến giá trị của message header
  + *Dead Letter:* Sử dụng để bắt bất cứ message nào không có nơi đến xác định( không khớp với bất cứ ràng buộc exchange-to-queue nào được định nghĩa)