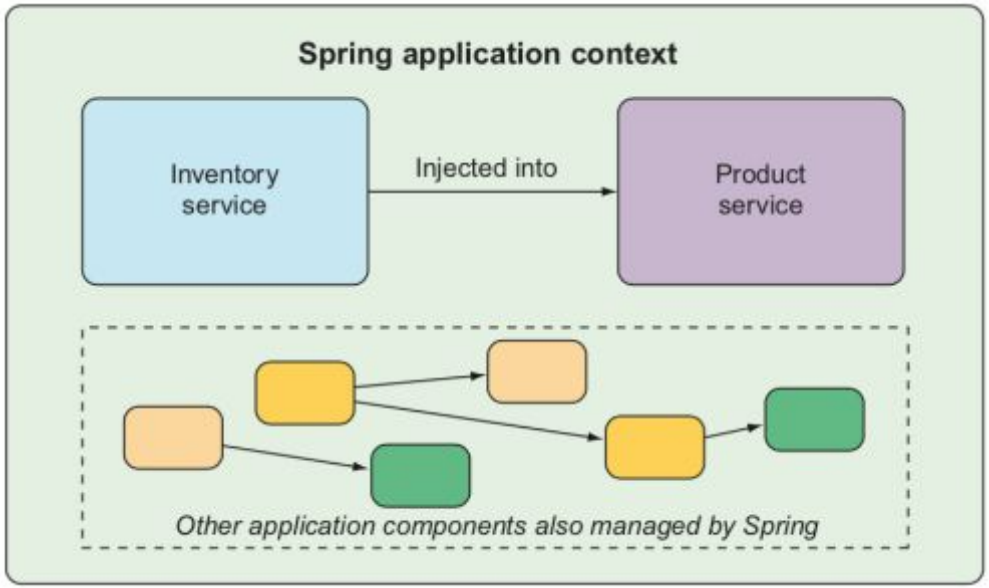
Build a Web application with Spring Framework

Spring là gì?

- Các ứng dụng phức tạp thường có nhiều thành phần. Mỗi thành phần có 1 chức năng, kết hợp với nhau để thực hiện nhiệm vụ chung.
- Spring là một Java framework với nền tảng là Spring container (Spring application context), dùng để tạo vào quản lý các thành phần của ứng dụng.
- Các thành phần ứng dụng (beans) được kết nối với nhau trong lõi của Spring để hình thành ứng dụng hoàn chỉnh.
- Hành động kết nối các thành phần với nhau dựa trên một kỹ thuật là Dependency Injection (DI).

Dependency Injection - DI

- Thông thường: Các thành phần tạo và duy trì vòng đời các bean mà nó phụ thuộc.
- DI: Ứng dụng DI dựa trên 1 thực thể độc lập (container) để tạo và duy trì tất cả các thành phần, và "tiêm" chúng vào các thành phần khác cần nó (thông qua hàm dựng hoặc hàm set).



Dependency Injection - DI

- Thông thường

```
public class ProductService {
    private InventoryService inventoryService;

    public ProductService() {
        inventoryService = new InventoryService();
    }
}
```

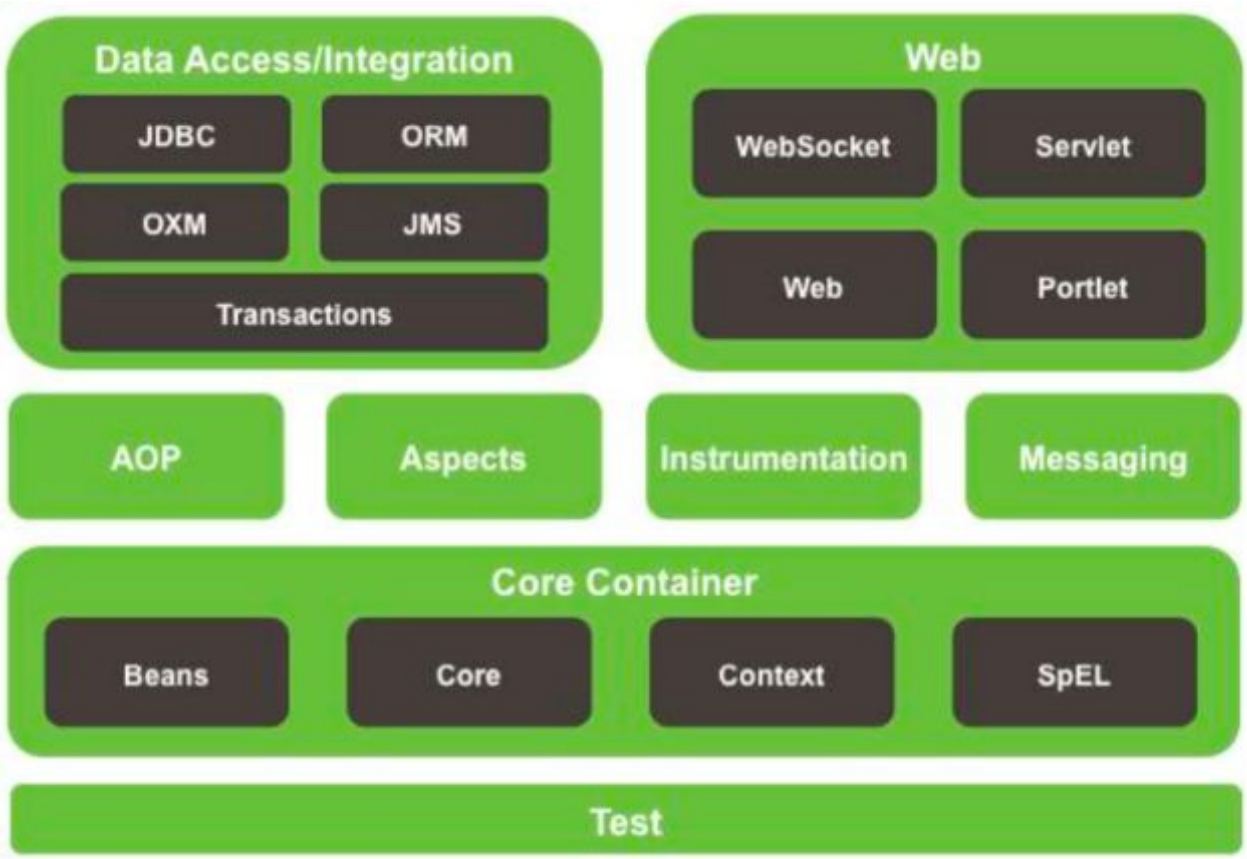
- Inversion of Control

```
public class ProductService {
    private InventoryService inventoryService;

    public ProductService(inventoryService InventoryService) {
        this.inventoryService = inventoryService;
    }
}
```

Lợi ích của DI

- Khi viết các ứng dụng Java phức tạp, các thành phần càng độc lập càng tốt, nhằm làm tăng khả năng tái sử dụng và bảo trì, cũng như khả năng kiểm thử độc lập.
- DI nhằm giúp kết nối các thành phần ứng dụng Java nhưng giữ cho chúng độc lập với nhau.
- Trong ví dụ trên, lớp InventoryService được thực thi độc lập và cung cấp cho lớp ProductService khi lớp này khởi tạo. Quá trình này được điều khiển bởi Spring.
- Các đối tượng sẽ được gắn vào nhau thông qua hàm Constructor hoặc hàm Set.

Spring modules



Kết nối các thành phần ứng dụng Spring

- Trước đây: Sử dụng file XML để mô tả các thành phần và mối quan hệ giữa chúng.

```
<bean id="inventoryService"
      class="com.example.InventoryService" />

<bean id="productService"
      class="com.example.ProductService" />
  <constructor-arg ref="inventoryService" />
</bean>
```

- Gần đây: Sử dụng Java annotation.

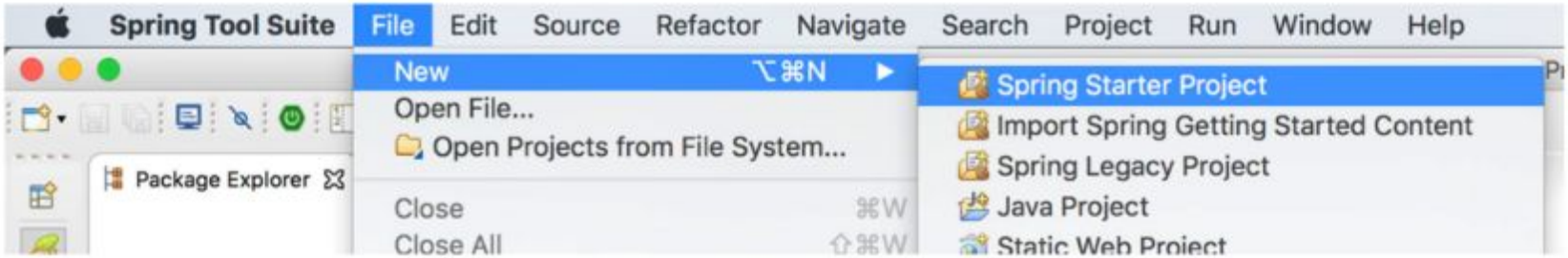
```
@Configuration
public class ServiceConfiguration {
  @Bean
  public InventoryService inventoryService() {
    return new InventoryService();
  }

  @Bean
  public ProductService productService() {
    return new ProductService(inventoryService());
  }
}
```

Cấu hình tự động (Auto Configuration)

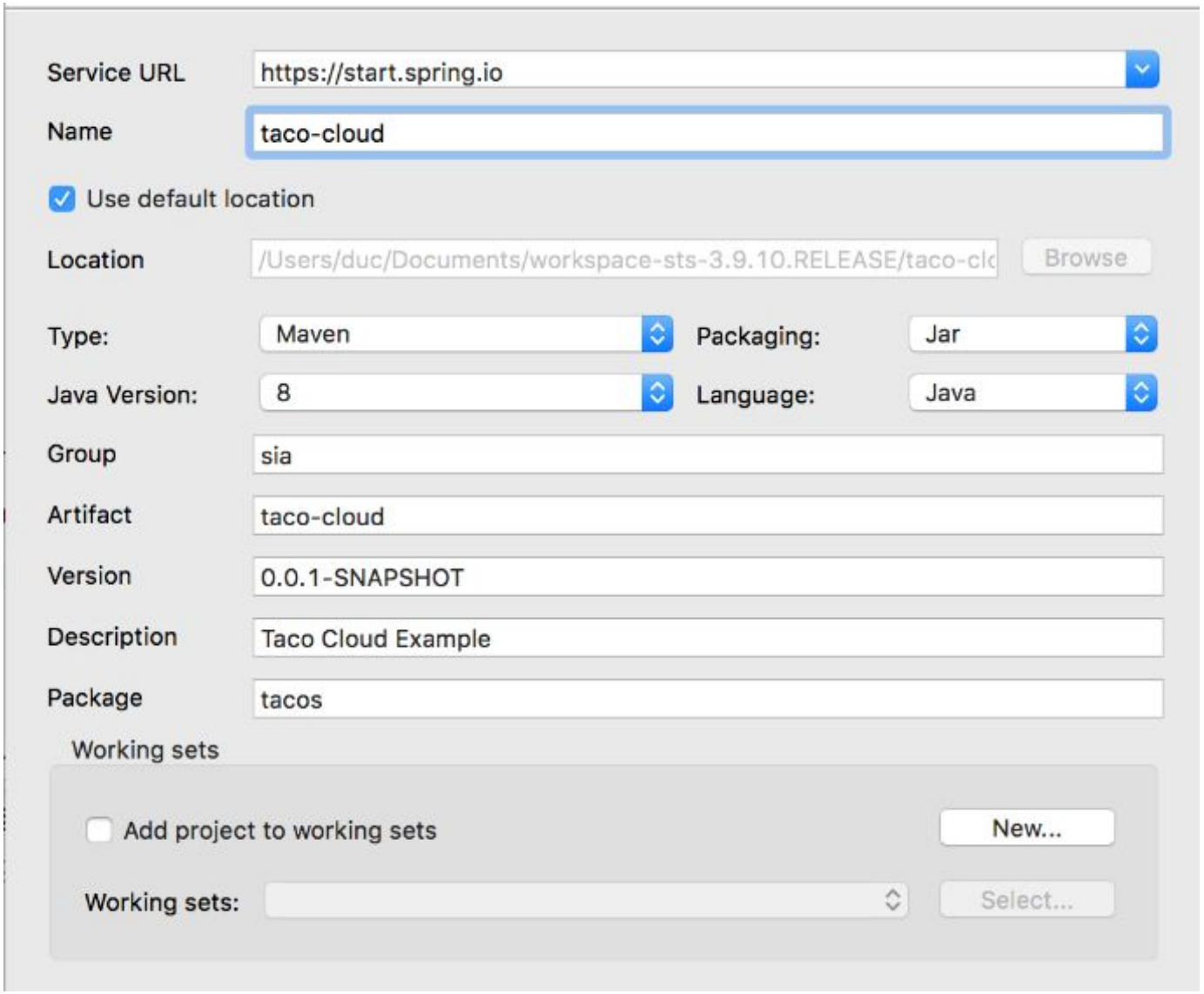
- Dựa trên các kỹ thuật auto wiring và component scanning.
  - Component scanning: Tự động phát hiện và tạo các beans.
  - Auto wiring: Tự động gắn (tiêm) các bean vào các bean khác mà nó phụ thuộc.
- Spring Boot: Kỹ thuật mới nhất. Là một phần mở rộng của Spring Framework, với kỹ thuật nổi bật nhất là tự động cấu hình.
  - Tự động phán đoán các thành phần cần được cấu hình và gắn kết, dựa trên các đối tượng trong classpath, biến môi trường ...
  - Giúp làm giảm bớt các mã cấu hình (với XML hay Java).
  - Hiện nay: Spring và Spring Boot luôn đi kèm với nhau. Thông thường sử dụng Spring Boot kết hợp Java annotation.

Khởi tạo ứng dụng Spring

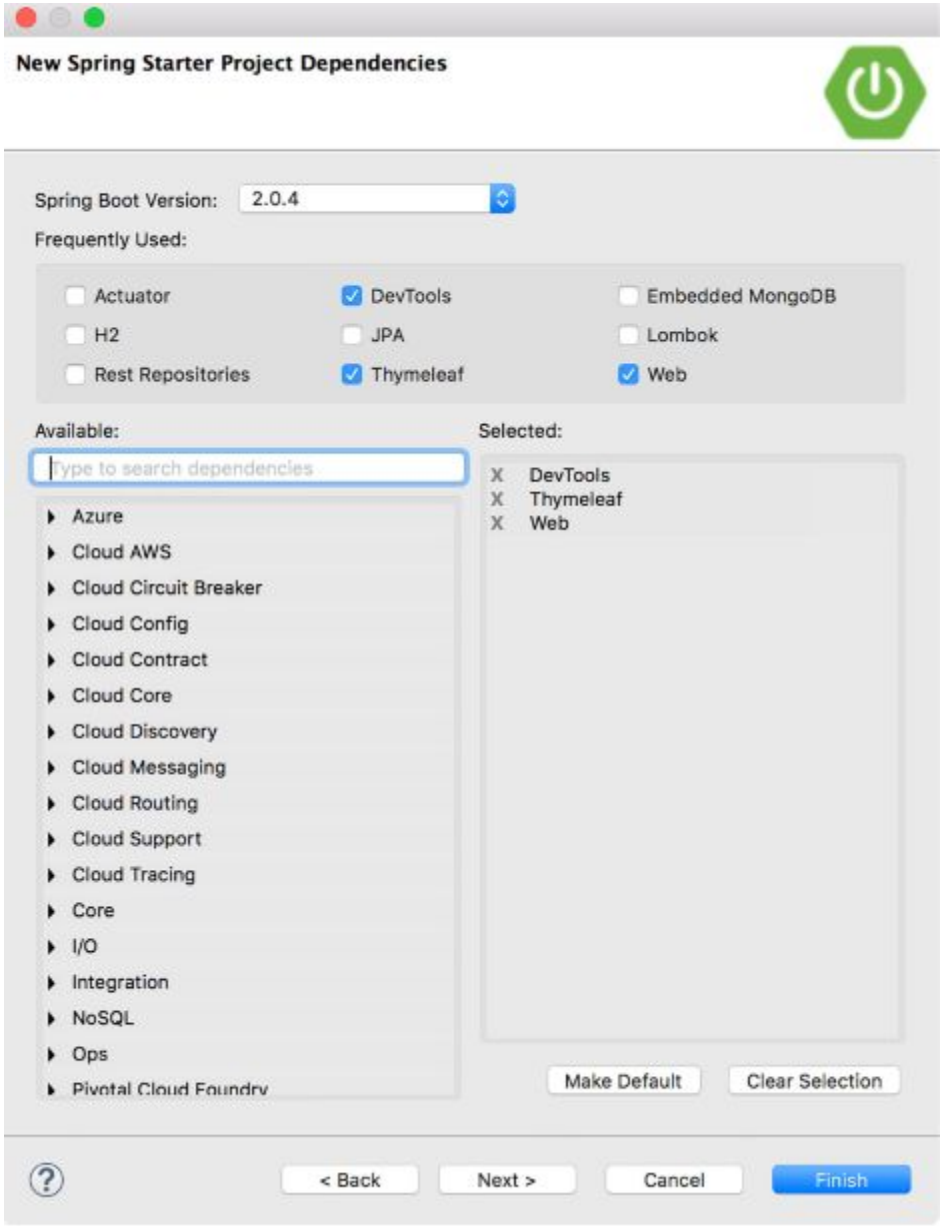
- Mục tiêu:
  - Xây dựng một ứng dụng đơn giản sử dụng Spring/Spring Boot và các thư viện, nền tảng liên quan.
  - Sử dụng Spring Initializr để khởi tạo ứng dụng.
  - Sử dụng công cụ Spring Tool Suite để quản lý phát triển ứng dụng.
- B1: Tạo project mới



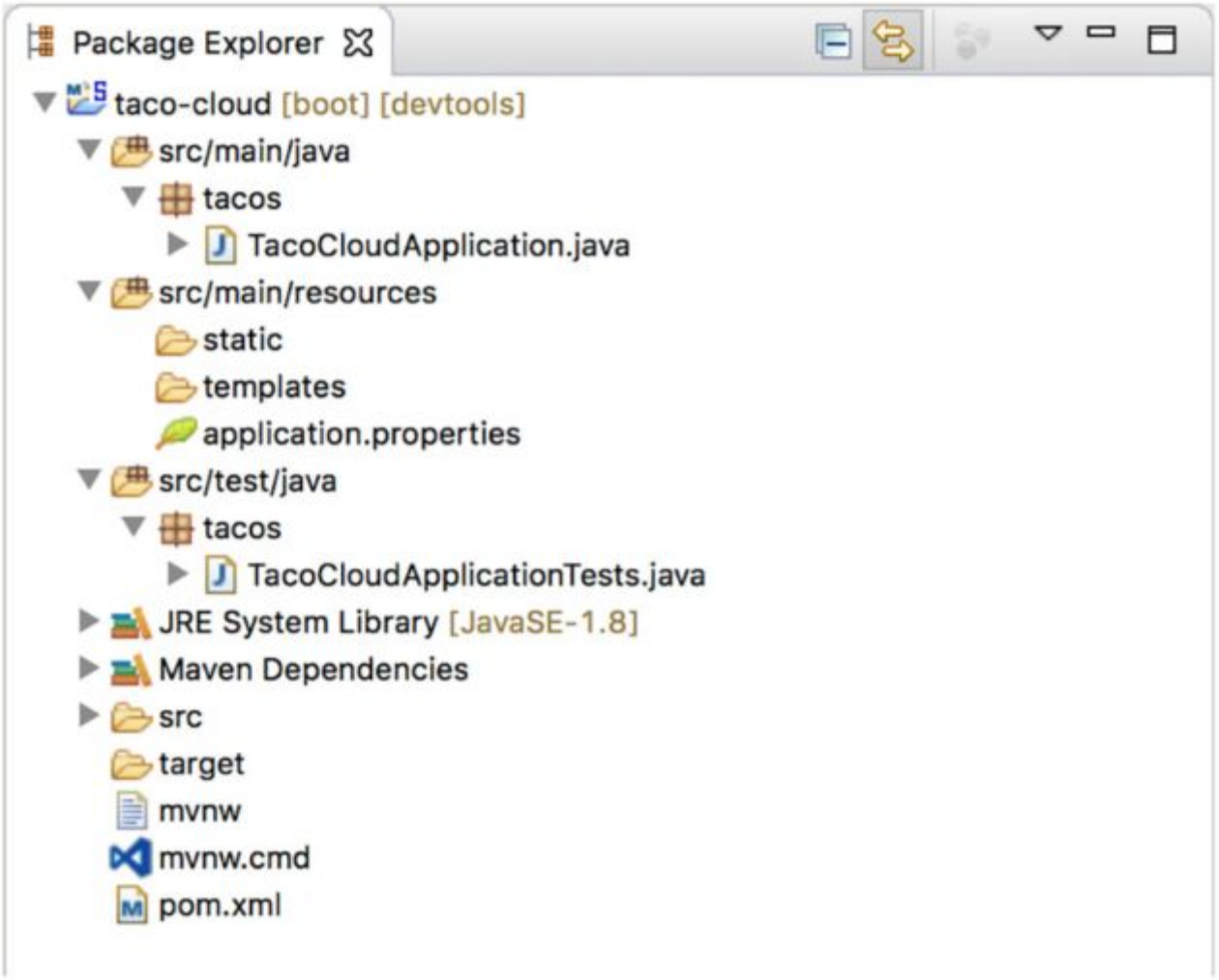
Nhập thông tin ứng dụng



Chọn các thư viện



Cấu trúc ứng dụng Spring



Cấu trúc ứng dụng

- src/main/java: Chứa các file mã nguồn Java chính của ứng dụng.
- TacoCloudApplication: SpringBoot main class dùng để khởi tạo ứng dụng.
- src/main/java: Chứa các file tài nguyên của ứng dụng.
- static: Chứa các nội dung tĩnh (hình ảnh, file code Js, ...)
- templates: Chứa các file giao diện dùng để tạo view cho trình duyệt.
- pom.xml: Đặc tả biên dịch của Maven (đọc thêm về Maven).

Lớp khởi tạo ứng dụng

```
package tacos;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class TacoCloudApplication {

  public static void main(String[] args) {
    SpringApplication.run(TacoCloudApplication.class, args);
  }

}
```

Spring Boot application

Runs the application

SpringBootApplication

`@SpringBootApplication` is a composite application that combines three other annotations:

- `@SpringBootConfiguration`—Designates this class as a configuration class. Although there's not much configuration in the class yet, you can add Java-based Spring Framework configuration to this class if you need to. This annotation is, in fact, a specialized form of the `@Configuration` annotation.
- `@EnableAutoConfiguration`—Enables Spring Boot automatic configuration. We'll talk more about autoconfiguration later. For now, know that this annotation tells Spring Boot to automatically configure any components that it thinks you'll need.
- `@ComponentScan`—Enables component scanning. This lets you declare other classes with annotations like `@Component`, `@Controller`, `@Service`, and others, to have Spring automatically discover them and register them as components in the Spring application context.

Hoàn thiện ứng dụng Spring

- SpringMVC: Một module chính của Spring, với một Controller ở trung tâm để xử lý các request/response.
- Controller đơn giản xử lý các request tới đường dẫn gốc (/) và chuyển request tới homepage view:

```
package tacos;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller                                  <-- The controller
public class HomeController {

  @GetMapping("/")                           <-- Handles requests for the root path /
  public String home() {
    return "home";                           <-- Returns the view name
  }

}
```

Tạo lớp HomeController

- Click chuột phải tacos, chọn New -> Class. Đặt tên là HomeController:

```
package tacos;
Import org.springframework.stereotype.Controller;
import
    org.springframework.web.bind.annotation.GetMapping
    ;
@Controller
public class HomeController {
    @GetMapping("/")
    public String home() {
     return "home";
    }
}
```

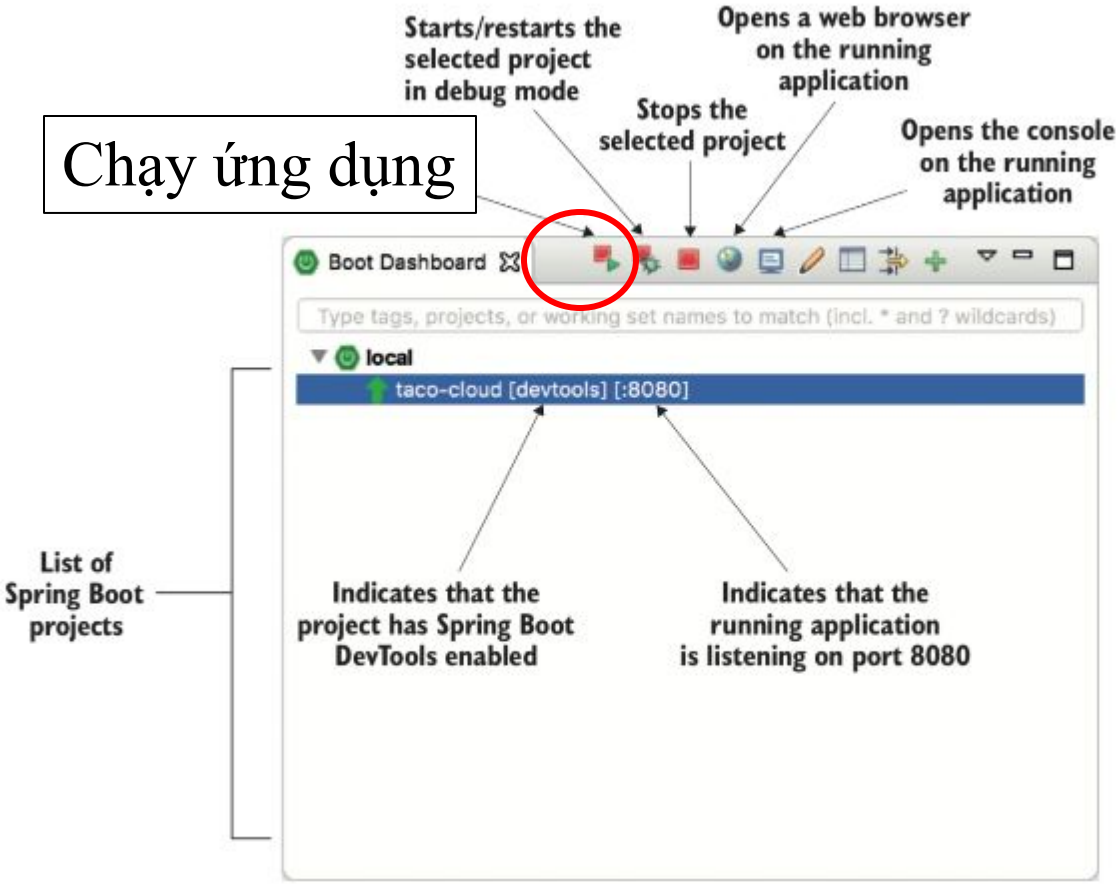
Tạo view

- View home.html dùng để hiển thị thông tin trang chủ (sau khi được điều hướng từ Controller).
- Sử dụng template Thymeleaf.
- Click chuột phải template, chọn New -> File. Đặt tên là home.html:

```
<!DOCTYPE html>
<html xmlns=http://www.w3.org/1999/xhtml
        xmlns:th="http://www.thymeleaf.org">
   <head>
    <title>Taco Cloud</title>
   </head>
   <body>
        <h1>Welcome to...</h1>
        <img th:src="@{/images/TacoCloud.png}"/>
        <h3><a href="/design"> Design your Taco</a></h3>
   </body>
</html>
```

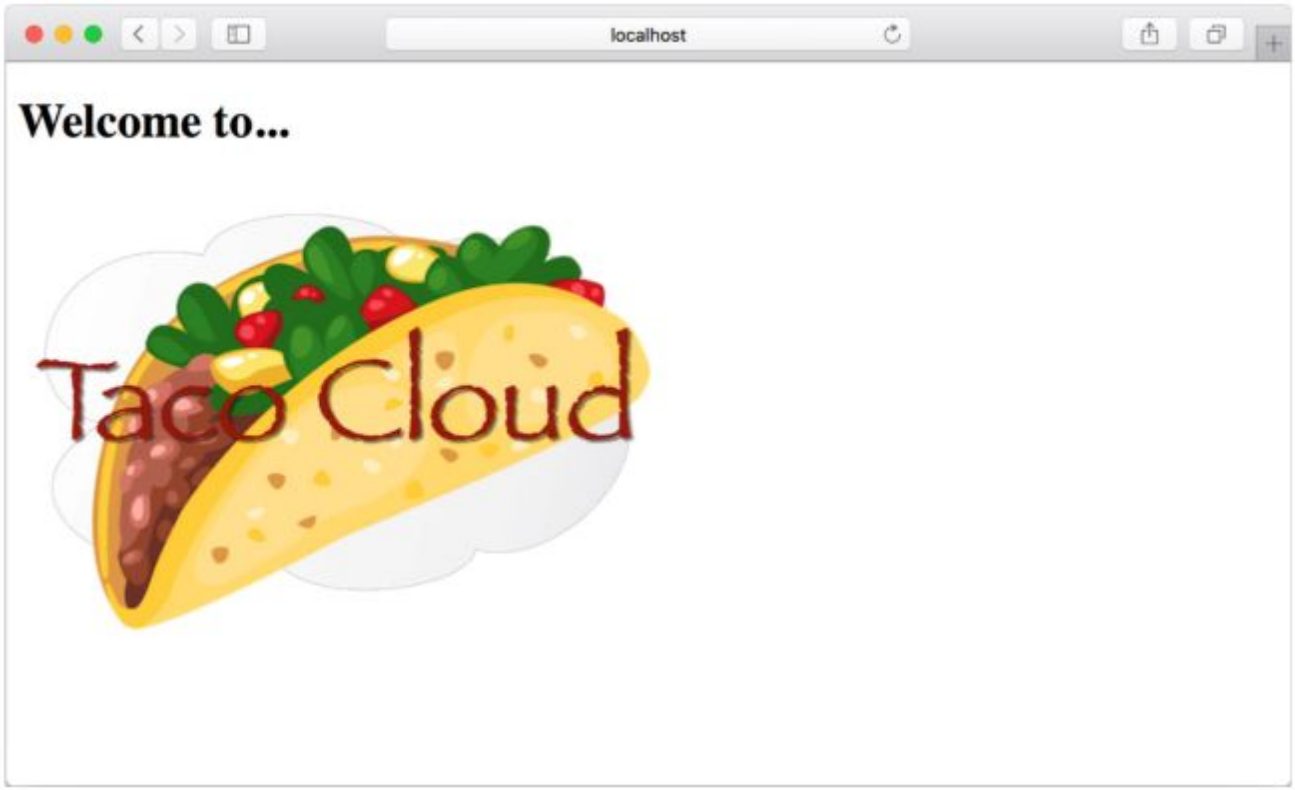
Dịch và chạy ứng dụng

- Sử dụng cửa sổ SpringBoot Dashboard (nằm góc dưới phải màn hình).
- Nếu cửa sổ chưa bật, chọn nút  trên thanh công cụ.



Kết quả khi chạy ứng dụng

- Chạy thành công, bật trình duyệt gõ: localhost:8080
- Lưu ý: Do Spring Boot được triển khai mặc định cùng Tomcat ở cổng 8080 nên khi chạy ứng dụng phải ngắt các Web server khác (VD Glassfish, hoặc Tomcat bản ngoài) đang chạy cổng 8080 đi.



Cải tiến ứng dụng Web Taco Cloud

- Mục tiêu:
  - Hệ thống liệt kê các lựa chọn thành phần (lấy từ CSDL) trên 1 trang web
  - Cho phép khách hàng xem và chọn thành phần của bánh
- Thành phần ứng dụng:
  - Lớp thực thể Ingredient định nghĩa các thành phần của bánh
  - Lớp Spring MVC Taco Design Controller tiếp nhận yêu cầu (request), trích xuất các tham số (thành phần) và chuyển đến view.
  - Trang view design.html hiển thị danh sách thành phần để hiển thị trên trang Web.

Cải tiến ứng dụng Web Taco Cloud

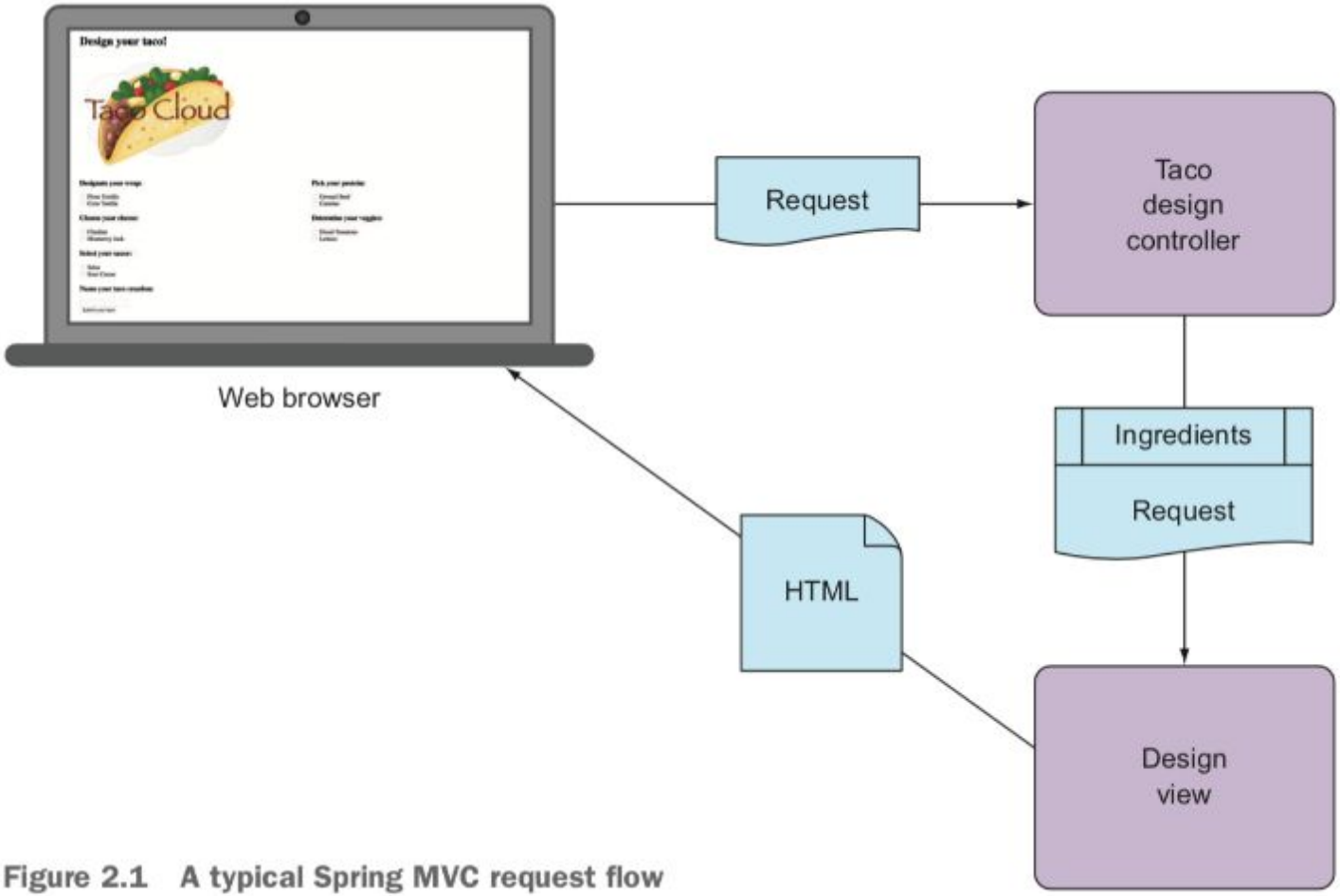


Figure 2.1 A typical Spring MVC request flow

Lớp Ingredient

- Click chuột phải tacos, chọn New -> Class. Đặt tên là Ingredient:

```
package tacos;

import lombok.Data;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
public class Ingredient {
    private final String id;
    private final String name;
    private final Type type;

    public static enum Type {
     WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
    }
}
```

Thư viện Lombok

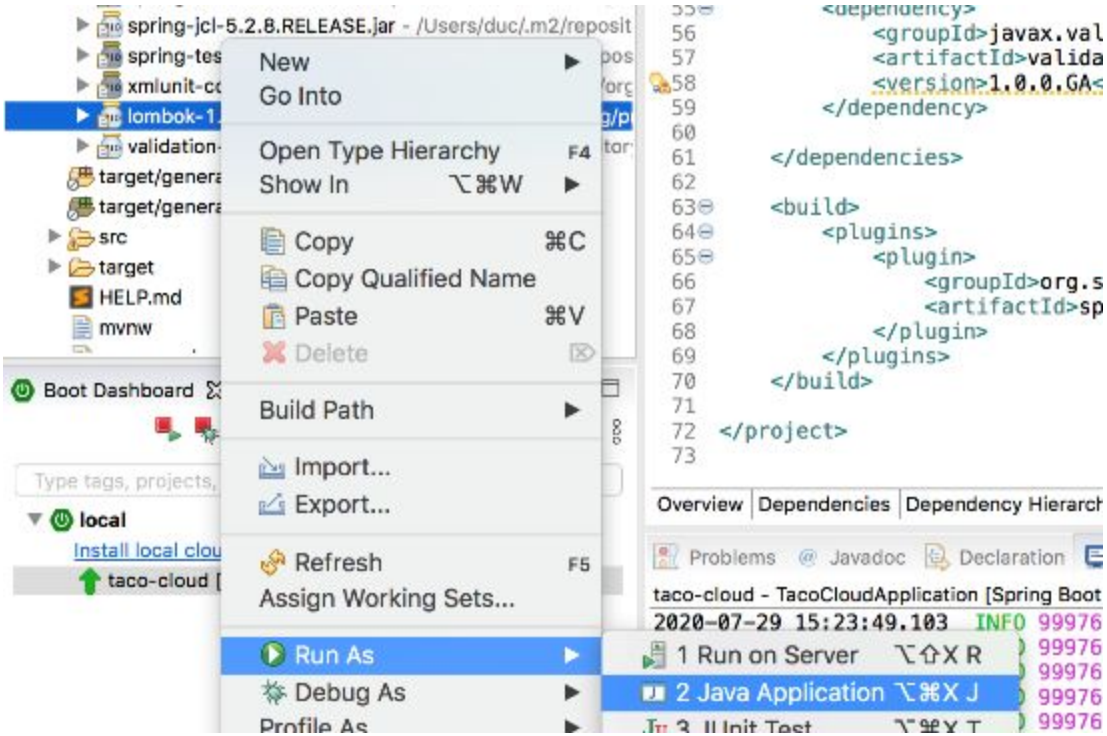
- Tự động tạo ra các hàm Constructor và Get/Set khi chạy
- Sử dụng Lombok để giảm bớt code Java
- Lombok là thư viện ngoài, do vậy phải bổ sung dependency vào file pom.xml

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
```

Thư viện Lombok

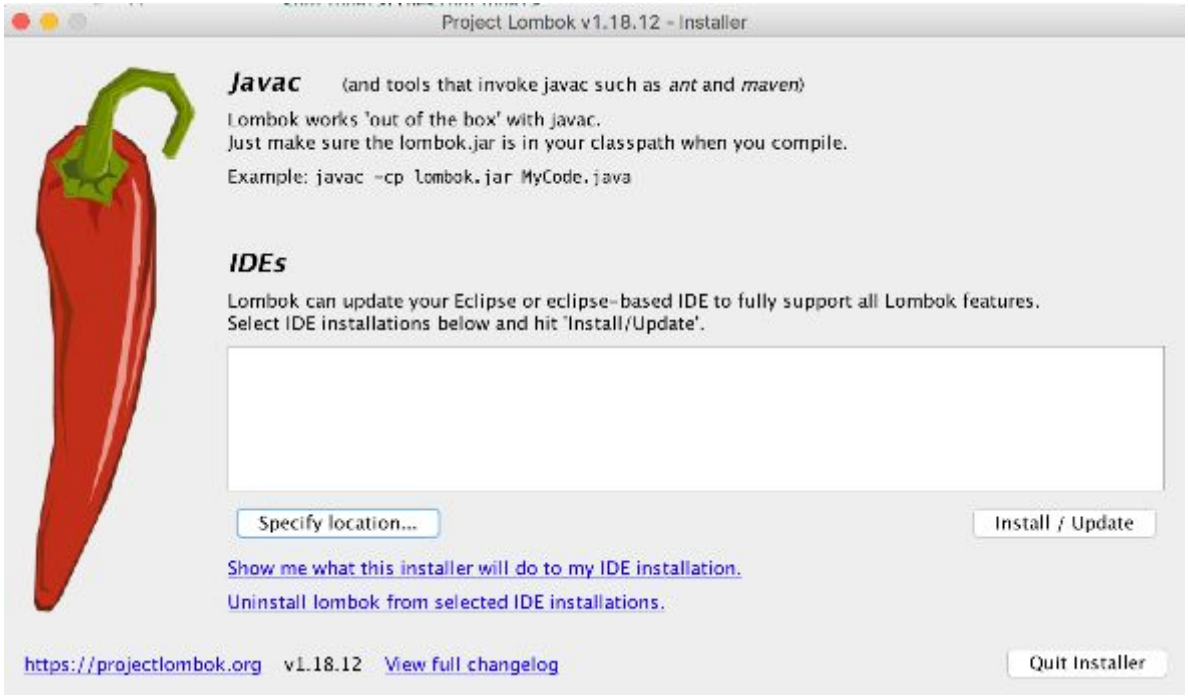
- Cần bổ sung Lombok thành extension của IDE

B1: Mở rộng Maven Dependencies trong cửa sổ Project Explorer, click chuột phải lombok.jar, chọn Run As -> Java Application:



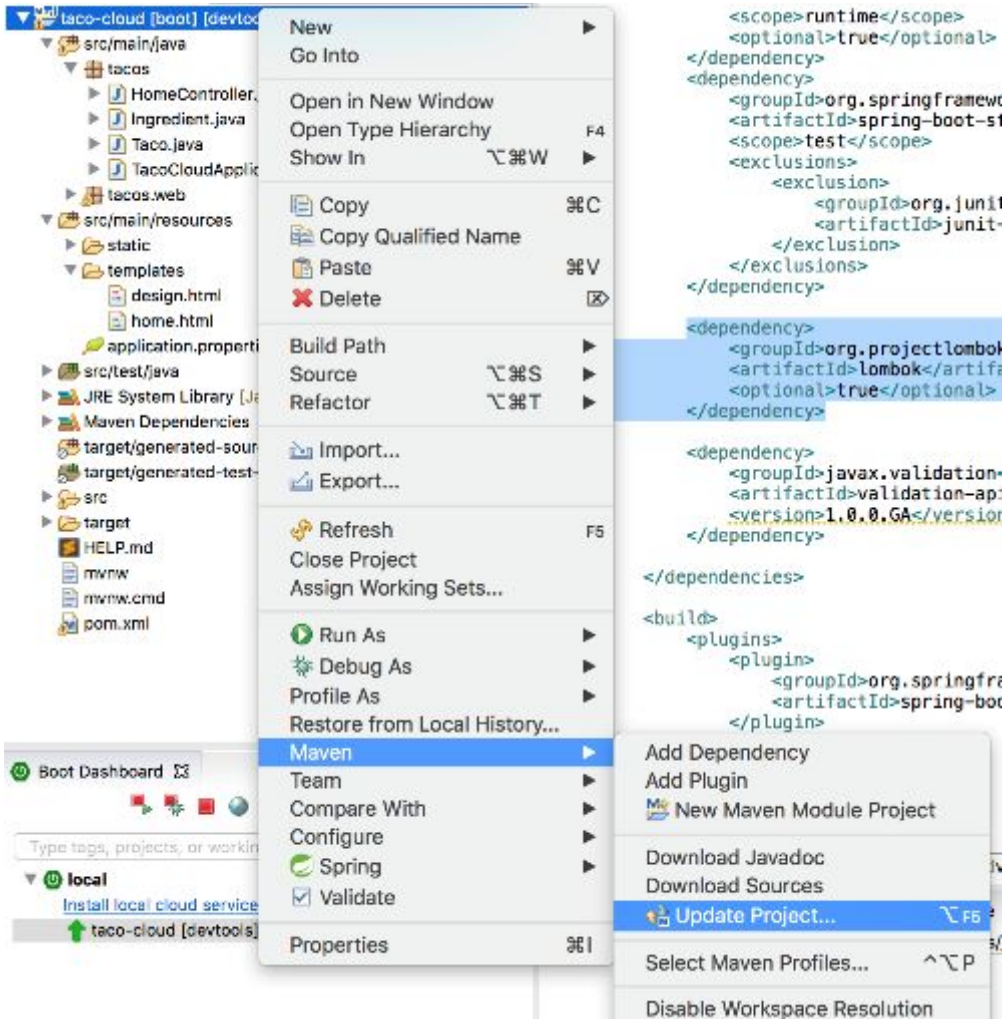
Thư viện Lombok

B2: Trong cửa sổ hiện ra, chọn Specify Location, chọn đường dẫn tới nơi cài đặt Spring Tool Suite -> Content -> Eclipse -> SpringToolSuite4.ini. Click Install -> Quick Installer. Khởi động lại Spring Tool Suite.



Thư viện Lombok

B3: Update Maven Dependencies: Click chuột phải và tên Project, chọn Maven -> Update Project. Bấm OK.



Lớp DesignTacoController

- Click chuột phải tacos, chọn New -> Class. Đặt tên là DesignTacoController, package là tacos.web:

```
package tacos.web;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.Errors;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import
    org.springframework.web.bind.annotation.RequestMapping;
import lombok.extern.slf4j.Slf4j;
import tacos.Taco;
import tacos.Ingredient;
import tacos.Ingredient.Type;
```

Lớp DesignTacoController (tiếp)

```
@Slf4j
@Controller
@RequestMapping("/design")
public class DesignTacoController {
 @ModelAttribute
 public void addIngredientsToModel(Model model) {
      List<Ingredient> ingredients = Arrays.asList(new
   Ingredient("FLTO", "Flour Tortilla", Type.WRAP),
            new Ingredient("COTO", "Corn Tortilla", Type.WRAP), new
   Ingredient("GRBF", "Ground Beef", Type.PROTEIN),
            new Ingredient("CARN", "Carnitas", Type.PROTEIN),
            new Ingredient("TMTO", "Diced Tomatoes", Type.VEGGIES),
   new Ingredient("LETC", "Lettuce", Type.VEGGIES),
            new Ingredient("CHED", "Cheddar", Type.CHEESE), new
   Ingredient("JACK", "Monterrey Jack", Type.CHEESE),
            new Ingredient("SLSA", "Salsa", Type.SAUCE), new
   Ingredient("SRCR", "Sour Cream", Type.SAUCE));
```

Lớp DesignTacoController (tiếp)

```
    Type[] types = Ingredient.Type.values();
     for (Type type : types) {
         model.addAttribute(type.toString().toLowerCase(),
    filterByType(ingredients, type));
     }
    }

    @GetMapping
    public String showDesignForm(Model model) {
     model.addAttribute(”taco", new Taco());
     return "design";
    }

    private List<Ingredient> filterByType(List<Ingredient>
    ingredients, Type type) {
     List<Ingredient> ingrList = new ArrayList<Ingredient>();
     for (Ingredient ingredient: ingredients) {
         if (ingredient.getType().equals(type))
    ingrList.add(ingredient);
     }
     return ingrList;

    }
}
```

Lớp DesignTacoController

- @Controller:
  - Đánh dấu lớp Controller, sẽ được tìm với component scan
- @RequestMapping:
  - Ấn định đường dẫn cho các request mà lớp sẽ xử lý
- @GetMapping:
  - Đánh dấu phương thức sẽ xử lý các GET request (có thể dùng @RequestMapping (method=RequestMethod.GET)
  - Các request sử dụng method khác sẽ được đánh dấu bằng @PostMapping, @PutMapping ...
  - Nếu muốn ấn định đường dẫn phụ thì bổ sung thêm cho phương thức (nếu không ấn định thì chính là đường dẫn của lớp)

Lớp DesignTacoController

- Phương thức showDesignForm():
  - Sẽ được gọi khi có Get Request gửi đến
  - Danh sách các thành phần được lọc theo loại
  - Danh sách theo loại này sau đó được thêm vào như 1 thuộc tính (attribute) của đối tượng Model.
  - Model là đối tượng chuyển giao dữ liệu giữa controller và view
  - Phía view sẽ có thể quét và lấy đối tượng đã được thêm vào như thuộc tính của Model
  - Phương thức trả về kết quả là tên của view sẽ tiếp nhập dữ liệu từ model

Trang view design.html

- Click chuột phải template, chọn New -> File. Đặt tên là design.html:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:th="http://www.thymeleaf.org">
<head>
<title>Taco Cloud</title>
<link rel="stylesheet" th:href="@{/styles.css}" />
</head>
<body>
 <h1>Design your taco!</h1>
 <img th:src="@{/images/TacoCloud.png}" />
 <form method="POST" th:object="${taco}">
  <div class="grid">
   <div class="ingredient-group" id="wraps">
    <h3>Designate your wrap:</h3>
    <div th:each="ingredient : ${wrap}">
     <input name="ingredients" type="checkbox"
      th:value="${ingredient.id}" /> <span th:text="${ingredient.name}">
INGREDIENT</span><br />
    </div>
   </div>
```

Trang view design.html (tiếp)

```
<div class="ingredient-group" id="proteins">
    <h3>Pick your protein:</h3>
    <div th:each="ingredient : ${protein}">
     <input name="ingredients" type="checkbox"
      th:value="${ingredient.id}" /> <span
th:text="${ingredient.name}">INGREDIENT</span><br />
    </div>
   </div>
   <div class="ingredient-group" id="cheeses">
    <h3>Choose your cheese:</h3>
    <div th:each="ingredient : ${cheese}">
     <input name="ingredients" type="checkbox"
      th:value="${ingredient.id}" /> <span
      th:text="${ingredient.name}">INGREDIENT</span><br />
    </div>
   </div>
   <div class="ingredient-group" id="veggies">
    <h3>Determine your veggies:</h3>
    <div th:each="ingredient : ${veggies}">
     <input name="ingredients" type="checkbox"
      th:value="${ingredient.id}" /> <span
th:text="${ingredient.name}">INGREDIENT</span><br />
    </div>
```

Trang view design.html (tiếp)

```
</div>
   <div class="ingredient-group" id="sauces">
    <h3>Select your sauce:</h3>
    <div th:each="ingredient : ${sauce}">
     <input name="ingredients" type="checkbox"
      th:value="${ingredient.id}" /> <span
th:text="${ingredient.name}">INGREDIENT</span><br />
    </div>
   </div>
  </div>
  <div>
   <h3>Name your taco creation:</h3>
   <input type="text" th:field="*{name}" /> <br />
   <button>Submit your taco</button>
  </div>
 </form>
</body>
</html>
```

Spring Views

- Spring hỗ trợ 1 số loại thư viện view: JSP, Thymeleaf ...
- Thymeleaf chỉ sử dụng file HTML để hiển thị dữ liệu nên có nhiều ưu điểm hơn so với các view cũ như JSP
- Thymeleaf có thể truy cập tới các đối tượng dữ liệu đã được lưu vào trong request dưới dạng các attribute
- Thymeleaf truy cập và hiển thị dữ liệu từ các đối tượng Java beans tương tự cú pháp của Expression Language
- Khi Spring chuyển các request từ controller sang view sẽ copy các attribute từ lớp Model sang request để các view trong Thymeleaf có thể truy cập được

Thymeleaf

- Thymeleaf templates bao gồm các thẻ HTML cùng với một số thuộc tính bổ sung giúp hiển thị dữ liệu từ các Java beans được lưu trong attribute của request.
- Thuộc tính bổ sung của Thymeleaf bắt đầu bằng chứ `th:` . Với cách tiếp cận này, Thymeleaf chỉ sử dụng các thẻ HTML cơ bản chứ không cần bổ sung thẻ mới như JSP.
- VD: Để truyền dữ liệu từ biến `message` (chẳng hạn la một attribute “message” được lưu trong request) ra thẻ <p> của HTML, thẻ sau sẽ được thêm vào Thymeleaf template:

```
<p th:text="${message}”></p>
```

Thymeleaf

- VD: Để truyền dữ liệu từ biến `message` (chẳng hạn la một attribute “message” được lưu trong request) ra thẻ <p> của HTML, thẻ sau sẽ được thêm vào Thymeleaf template:

```
<p th:text="${message}”></p>
```

- Khi template được chuyển đổi sang HTML, phần thân của thẻ <p> sẽ được thay thế bởi giá trị của thuộc tính “message” trong request và thuộc tính th: không còn nữa.

```
<p>This is a message</p>
```

Thymeleaf Expression

- Để lấy các thông tin từ trong Model, sử dụng Thymeleaf Expression:
  - `${…}`: Giá trị của 1 biến

```
model.addAttribute(“user”,user);
<p><span th:text="${user.firstName}"></span>.</p>
```

  - `*{…}`: Giá trị của 1 biến trong phạm vi lựa chọn (từ đối tượng th:object)

```
<div th:object="${session.user}">
    <p>Name: <span th:text="*{firstName}"></span>.</p>
</div>
```

  - `#{…}`: Lấy trong file .properties
  - `@{…}`: URL Expression

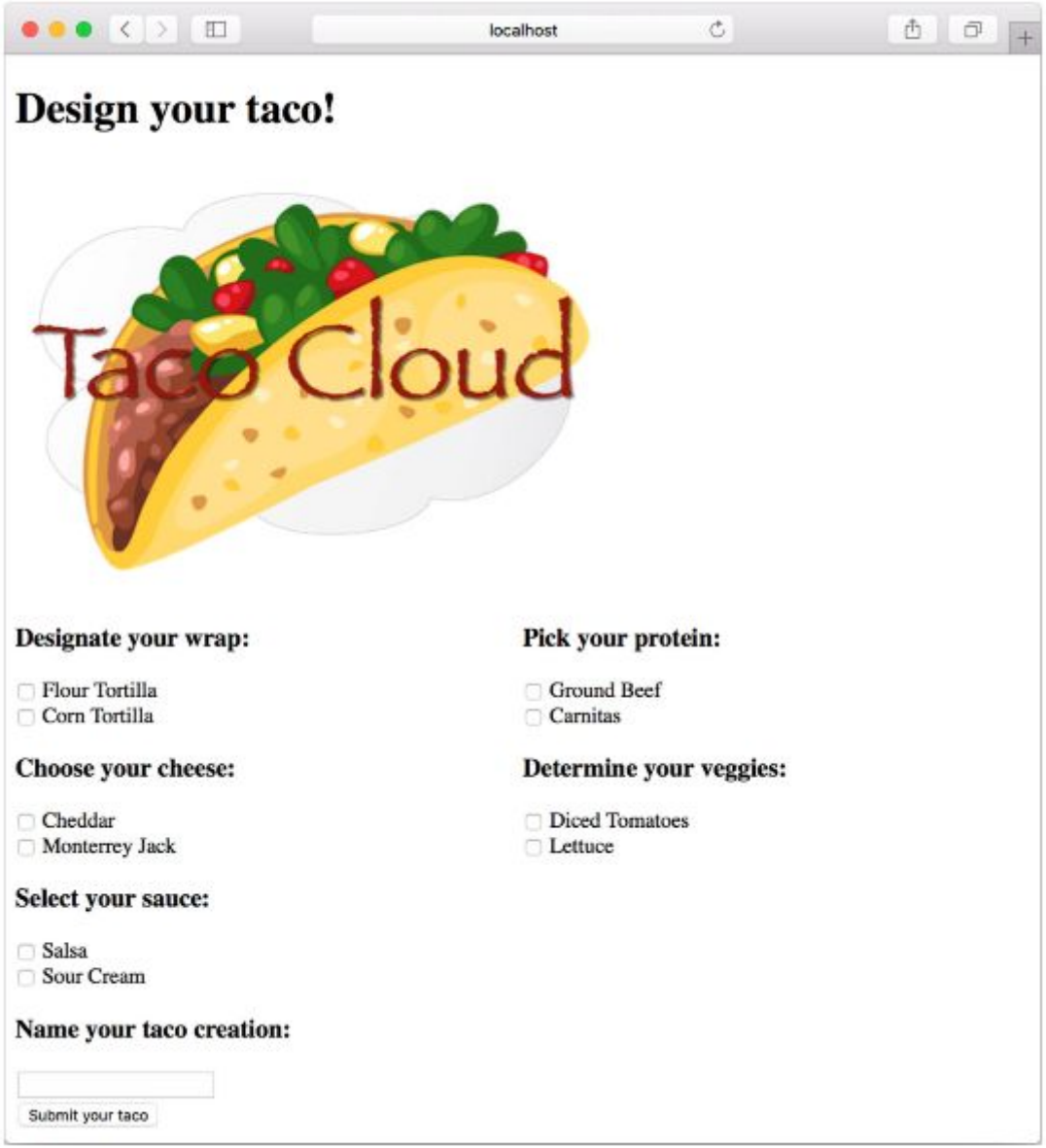
Thymeleaf

- Thymeleaf còn cung cấp thuộc tính th: each của thẻ để duyệt qua 1 tập hợp và hiển thị giá trị các phần tử của tập hợp sang HTML
- VD: Để hiển thị danh sách các thành phần “wrap” sang HTML, dùng đoạn mã sau:

```
<h3>Designate your wrap:</h3>
<div th:each="ingredient : ${wrap}">
    <input name="ingredients" type="checkbox”
    th:value=”${ingredient.id}” />
    <span th:text="${ingredient.name}">INGREDIENT</span><br/>
</div>
```

- Thuộc tính th:each sẽ lặp lại tag trên mỗi thành phần của danh sách wrap và hiển thị ra dạng HTML.

Kết quả giao diện trang chọn thành phần



Xử lý form chọn thành phần

- Form:
  - Method: POST
  - Action: Không có

  -> Form sẽ được submit tới cùng URL với GET request trước đó
- Viết thêm phương thức xử lý POST request trong TacoDesignController:

```
@PostMapping
public String processDesign(Taco taco) {
  // Save the taco design...
  // We'll do this later
  log.info("Processing design: " + taco);
  return "redirect:/orders/current";
}
```

Xử lý form chọn thành phần

- Khi form được submit, các trường của form sẽ được gán cho các thuộc tính của đối tượng taco. Đối tượng này sau đó sẽ được chuyển thành tham số của phương thức processDesign()
- Hiện tại phương thức processDesign() chưa thực hiện xử lý gì trên tham số đối tượng taco (phần tiếp theo sẽ tiến hành xử lý lưu thông tin đối tượng taco vào CSDL)
- Phương thức processDesign() cũng trả về 1 giá trị String là tên của view sẽ được chuyển đến. Tuy nhiên, giá trị trả về này có tiền tố redirect: cho biết đây là 1 view chuyển hướng (cụ thể là chuyển đến đường dẫn */orders/current*)

Lớp Taco

- Là lớp thực thể lưu thiết kế bánh
- Click chuột phải tacos, chọn New -> Class. Đặt tên là Taco (lưu ý thư viện Lombok tiếp tục được sử dụng để tự động tạo hàm constructor và get/set):

```
Package tacos;

import java.util.List;
import lombok.Data;

@Data
public class Taco {
  private String name;
  private List<String> ingredients;
}
```

Lớp OrderController

```
package tacos.web;

import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.Errors;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import lombok.extern.slf4j.Slf4j;
import tacos.Order;

 @Slf4j
 @Controller
 @RequestMapping("/orders")
 public class OrderController {
    @GetMapping("/current")
    public String orderForm(Model model) {
     model.addAttribute("order", new Order());
     return "orderForm";
    }
 }
```

Lớp OrderController

- Là lớp Controller tạo form đặt hàng tại đường dẫn `/orders/current`
- Lưu ý chú giải `@RequestMapping("/orders")` mức class kết hợp với `@GetMapping("/current")` của phương thức sẽ ấn định phương thức orderForm() sẽ được gọi để xử lý các request đến URL `/orders/current`
- Phương thức orderForm() chỉ thực hiện tạo một đối tượng Order rỗng vào lưu vào Model (sẽ bổ sung thêm xử lý ở phần sau), và trả về tên view là orderForm.

Lớp Order

- Là lớp thực thể đại diện cho đơn đặt hàng
- Click chuột phải tacos, chọn New -> Class. Đặt tên là Order:

```
package tacos;

import javax.validation.constraints.Digits;
import javax.validation.constraints.Pattern;
import org.hibernate.validator.constraints.CreditCardNumber;
import org.hibernate.validator.constraints.NotBlank;
import lombok.Data;

@Data
public class Order {
   private String name;
   private String street;
   private String city;
   private String state;
   private String zip;
   private String ccNumber;
   private String ccExpiration;
   private String ccCVV;
}
```

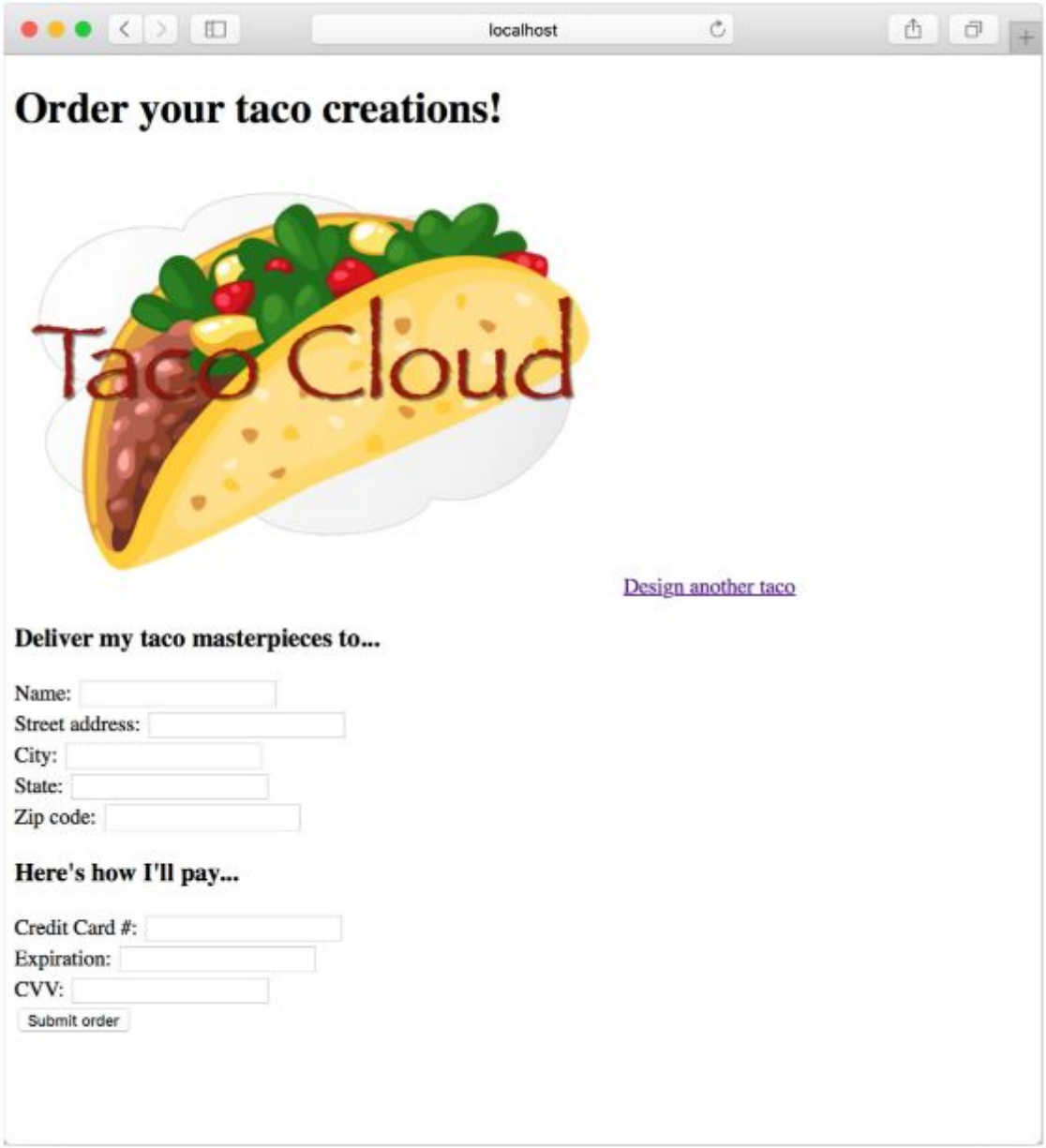
Trang view orderForm.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Taco Cloud</title>
    <link rel="stylesheet" th:href="@{/styles.css}" />
  </head>
<body>
    <form method="POST" th:action="@{/orders}"
   th:object="${order}">
      <h1>Order your taco creations!</h1>
      <img th:src="@{/images/TacoCloud.png}"/>
      <a th:href="@{/design}" id="another">Design another
   taco</a><br/>
      <div th:if="${#fields.hasErrors()}">
        <span class="validationError">
        Please correct the problems below and resubmit.
        </span>
</div>
```

Trang view orderForm.html

```
        <h3>Deliver my taco masterpieces to...</h3>
        <label for="name">Name: </label>
        <input type="text" th:field="*{name}"/><br/>
        <label for="street">Street address: </label>
        <input type="text" th:field="*{street}"/><br/>
        <label for="city">City: </label>
        <input type="text" th:field="*{city}"/><br/>
        <label for="state">State: </label>
        <input type="text" th:field="*{state}"/><br/>
        <label for="zip">Zip code: </label>
        <input type="text" th:field="*{zip}"/><br/>
        <h3>Here's how I'll pay...</h3>
        <label for="ccNumber">Credit Card #: </label>
        <input type="text" th:field="*{ccNumber}"/><br/>
        <label for="ccExpiration">Expiration: </label>
        <input type="text" th:field="*{ccExpiration}"/><br/>
        <label for="ccCVV">CVV: </label>
        <input type="text" th:field="*{ccCVV}"/><br/>
        <input type="submit" value="Submit order"/>
    </form>
</body>
</html>
```

Kết qủa giao diện trang đặt hàng



Xử lý form đặt hàng

- Form:
  - Method: POST
  - Action: /orders

- Viết thêm phương thức xử lý POST request trong OrderController (phương thức này tạm thời chưa xử ý gì và chỉ ghi thông tin vào log):

```
@PostMapping
    public String processOrder(Order order) {
        log.info("Order submitted: " + order);
        return "redirect:/";
}
```

Data Validation

- Spring hỗ trợ Java Bean's Validation API, nhờ đó có thể dễ dàng đưa vào các thao tác kiểm tra dữ liệu nhập mà không cần phải viết mã cho các hoạt động kiểm tra.
- Các bước để thực hiện data validation:
  - Khai báo các quy định về kiểm tra dữ liệu nhập trong các lớp thực thể. VD các lớp Taco, lớp Order
  - Ấn định việc các hoạt động data validation sẽ được thực hiện trong các phương thức của controller. VD các phương thức processDesign(), processOrder()
  - Cập nhật các views để đưa vào các kiểm tra và thông báo lỗi

Quy tắc kiểm tra dữ liệu

- Lớp Taco

```
package tacos;

import java.util.List;

import javax.validation.constraints.NotNull;

import javax.validation.constraints.Size;

import lombok.Data;

@Data

public class Taco {

  @NotNull

  @Size(min=5, message="Name must be at least 5 characters long")

  private String name;

  @Size(min=1, message="You must choose at least 1 ingredient")

  private List<String> ingredients;

}
```

Quy tắc kiểm tra dữ liệu

- Lớp Order

```
package tacos;

import javax.validation.constraints.Digits;

import javax.validation.constraints.Pattern;

import org.hibernate.validator.constraints.CreditCardNumber;

import javax.validation.constraints.NotBlank;

import lombok.Data;

@Data

public class Order {

  @NotBlank(message="Name is required")

  private String name;

  @NotBlank(message="Street is required")

  private String street;
```

Quy tắc kiểm tra dữ liệu

- Lớp Order

```
  @NotBlank(message="City is required")
  private String city;
  @NotBlank(message="State is required")
  private String state;
  @NotBlank(message="Zip code is required")
  private String zip;
  @CreditCardNumber(message="Not a valid credit card number")
  private String ccNumber;
  @Pattern(regexp="^(0[1-9]|1[0-2])([\\/])([1-9][0-9])$",
           message="Must be formatted MM/YY")
  private String ccExpiration;
  @Digits(integer=3, fraction=0, message="Invalid CVV")
  private String ccCVV;
}
```

Thực hiện validate ở các phương thức xử lý form

- Phương thức processDesign():

```
@PostMapping
public String processDesign(@Valid Taco taco, Errors errors) {
  if (errors.hasErrors()) {
    return "design";
  }
  // Save the taco design...
  // We'll do this in later
  log.info("Processing design: " + taco);
  return "redirect:/orders/current";
}
```

Thực hiện validate ở các phương thức xử lý form

- Phương thức processOrder():

```
@PostMapping
public String processOrder(@Valid Order order, Errors errors) {
    if (errors.hasErrors()) {
    return "orderForm";
    }
    log.info("Order submitted: " + order);
    return "redirect:/";
}
```

Kiểm tra và hiển thị lỗi tại view

- Thymeleaf cung cấp khả năng kiểm tra và hiển thị lỗi qua thuộc tính fields và th:errors
- VD muốn hiển thị lỗi nhập liệu cho trường Credit Card Number, bổ sung thẻ với các thuộc tính if và errors:

```
<label for="ccNumber">Credit Card #: </label>
  <input type="text" th:field="*{ccNumber}"/>
  <span class="validationError"
    th:if="${#fields.hasErrors('ccNumber')}"
    th:errors="*{ccNumber}">CC Num Error</span>
```