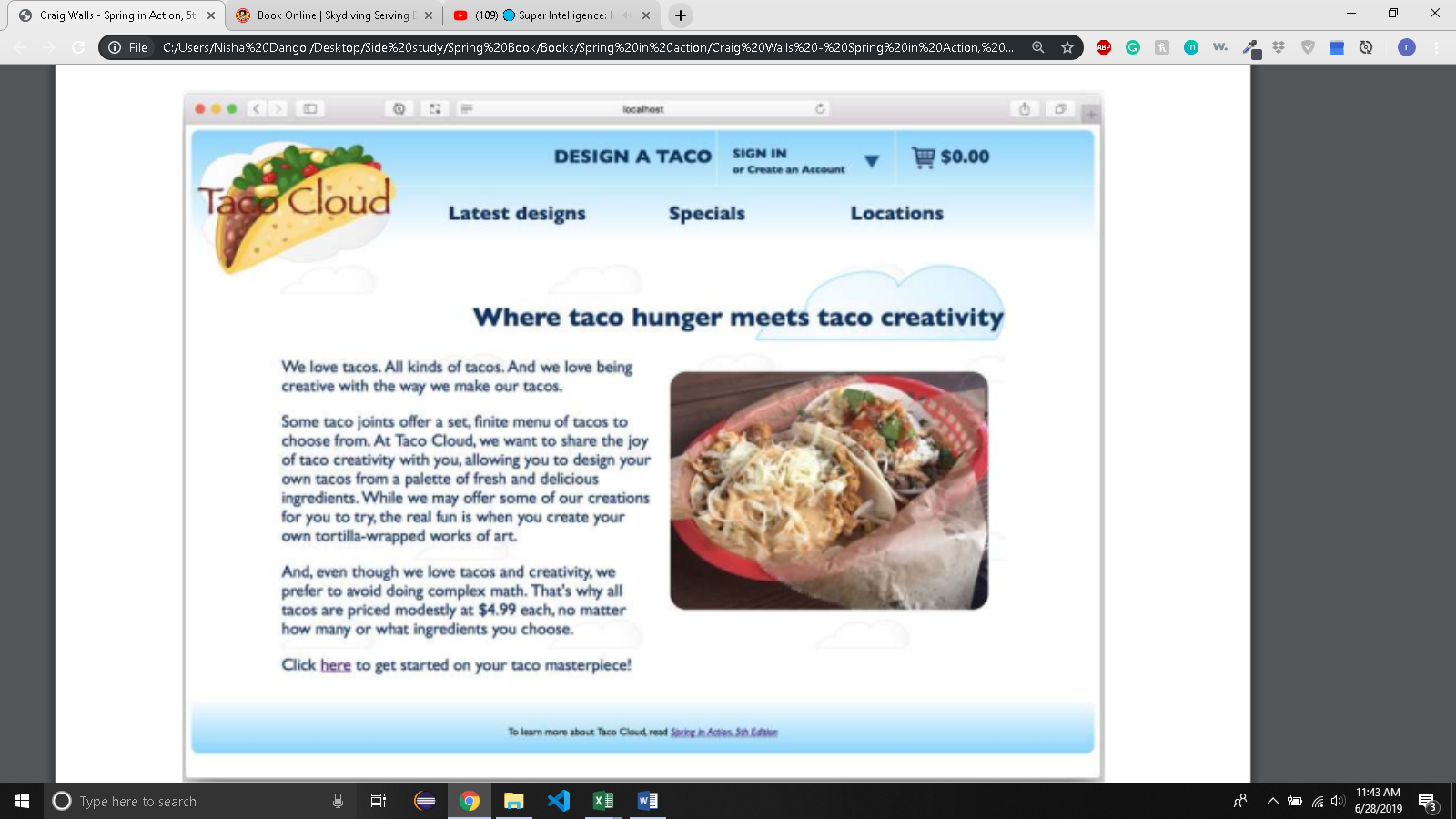
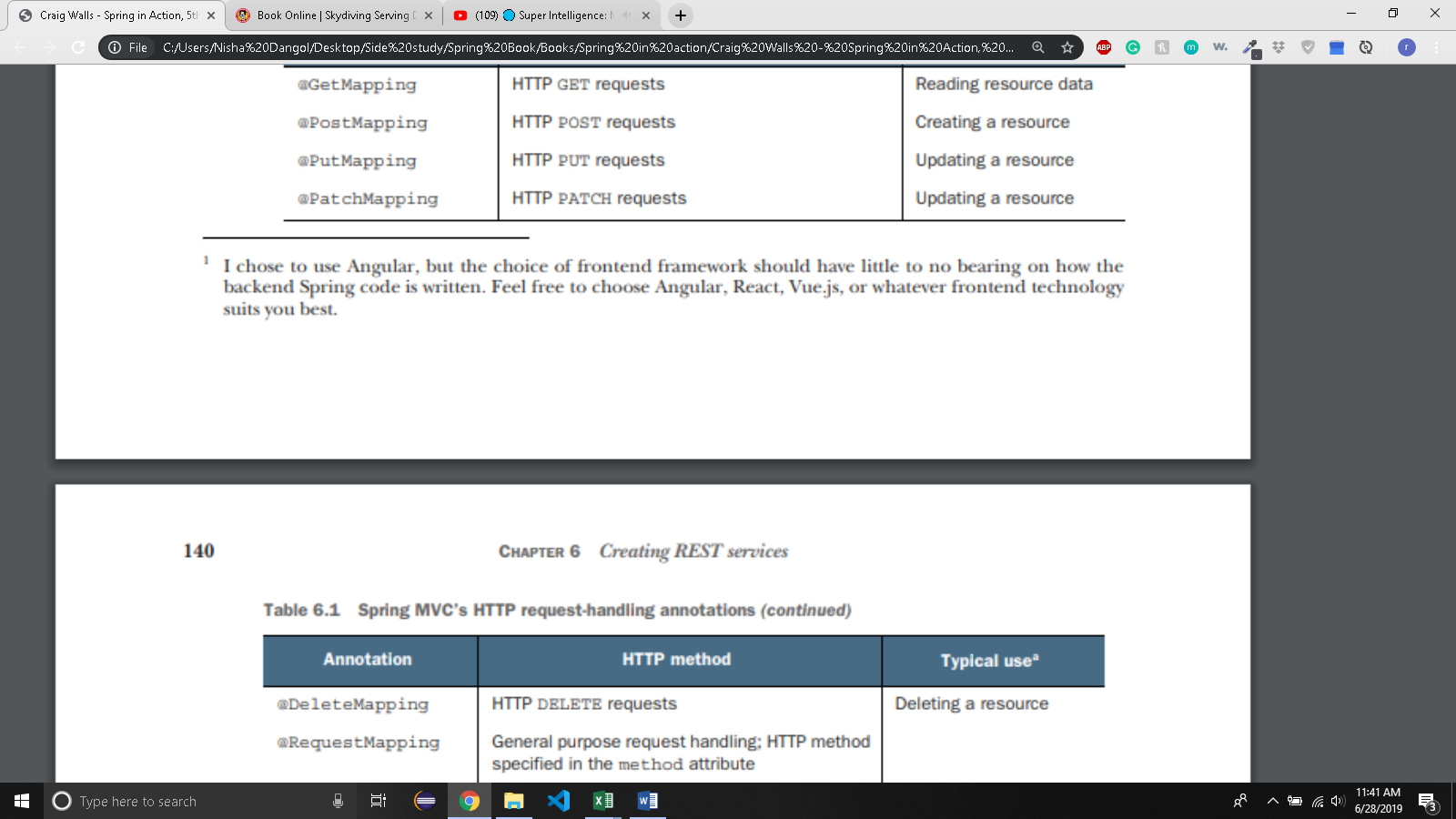
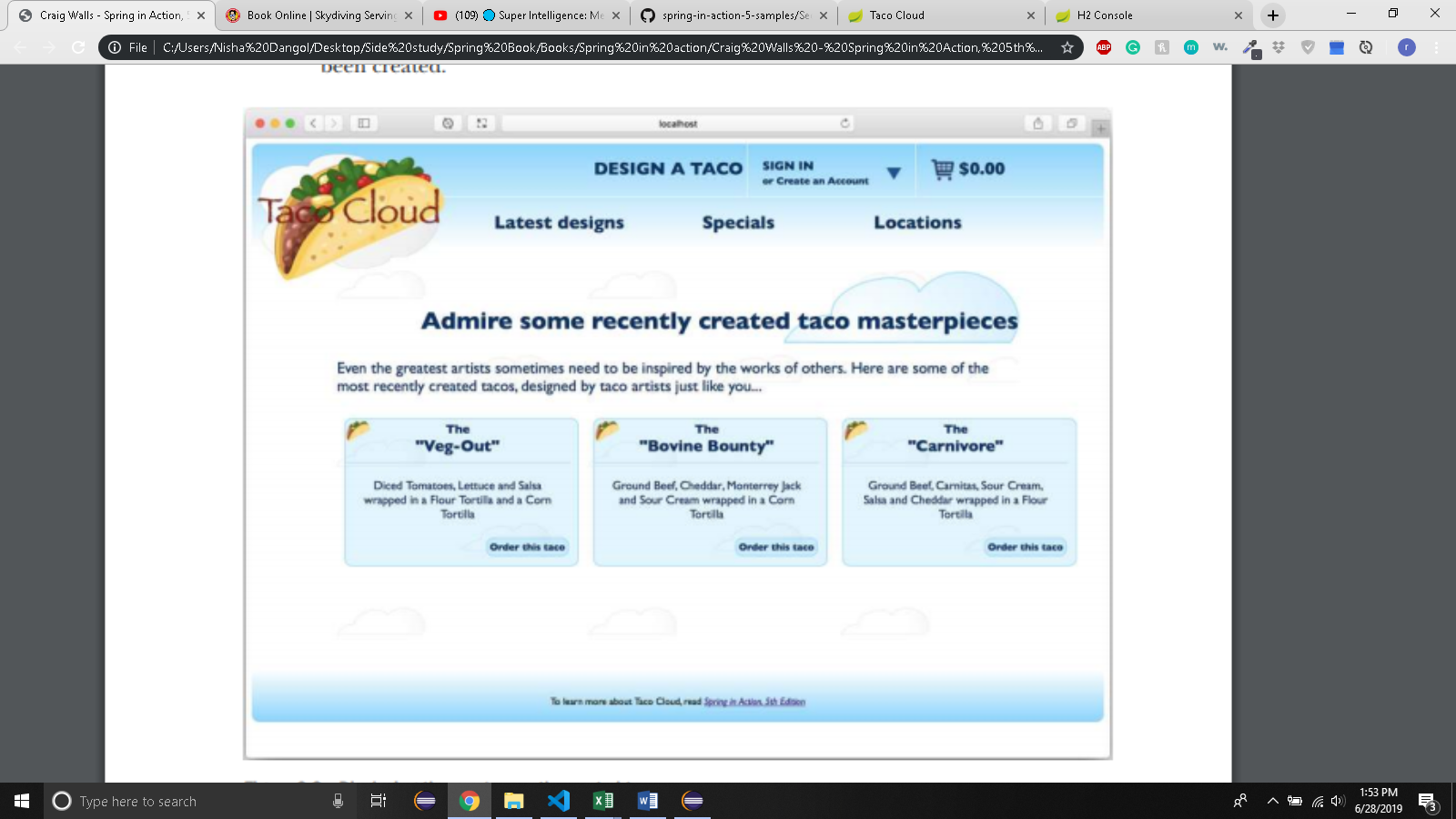
CHAPTER 6 – CREATING REST SERVICE

* Download node.js from nodejs.org/en/download
* Install the download file
* Install angular cli : npm install -g @angular/cli
* Go to eclipse project’s taco folder in command line and type: mvnw clean package
* Install python and add it in environment path at the end of installation.
  + This will let you use curl to format the json in readable form.
  + curl”localhost:8080/design/recent”|python -mjson.tool
* 6.1 WRITING RESTful controllers:
  + Server exposes an API through which all kinds of clients can interact with the backend functionality.
  + Our new webpage will look like this:



* + You need to create REST API that the Angular-based UI will communicate with to save and fetch taco data.
  + Spring MVC supports various annotations for HTTP requests:



* 6.1.1 RETRIEVING DATA FROM THE SERVER
  + Taco cloud needs to be able to display a list of most recently created tacos when the latest Designs link is clicked.
  + In the angular code, there is RecentTacosComponent that will display the most recently created tacos.
* import {Component, OnInit, Injectable} from '@angular/core';
* import{Http} from '@angular/http';
* import {HttpClient} from '@angular/common/http';
* @Component({
* selector: 'recent-tacos',
* templateUrl: 'recents.component.html',
* styleUrls:['./recents.component.css']
* })
* @Injectable()
* export class RecentTacosComponent implements OnInit{
* recentTacos: any;
* constructor(private httpClient: HttpClient){}
* ngOnInit(){
* this.httpClient.get('http://localhost:8080/design/recent')
* .subscribe(data => this.recentTacos = data);
* }
* }
  + - In ngOnInit() method, RecentTacosComponent uses the injected Http module to perform an HTTP GET request to <http://localhost:8080/design/recent>, expecting that the response will contain a list of taco designs, which will be placed in the recentTacos model variable
    - The view (in recents.component.html) will present that model data as html to be rendered in the browser.
    - The result will look something like this after 3 tacos have been created
* 
  + Now, let’s write the endpoint that hadles the GET requests for /design/recent and responds with a list of recently designed tacos.

package tacos.web.api;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.dao.EmptyResultDataAccessException;

import org.springframework.data.domain.PageRequest;

import org.springframework.data.domain.Sort;

import org.springframework.hateoas.EntityLinks;

import org.springframework.hateoas.Resource;

import org.springframework.hateoas.Resources;

import org.springframework.hateoas.mvc.ControllerLinkBuilder;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.CrossOrigin;

import org.springframework.web.bind.annotation.DeleteMapping;

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

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.ResponseStatus;

import org.springframework.web.bind.annotation.RestController;

import tacos.Taco;

import tacos.data.TacoRepository;

/\*Stereotype annotation for component scanning. Tells spring that

\* all handler methods should have their return value written directly

\* to the body of the response, rather than being carried in the model

\* to a view for rendering\*/

*@RestController*

/\*

\* This specifies that any of the handler methods will only accept requests if

\* the request's 'Accept header' includes "aplication/json". This limits your

\* API to produce only JSON results

\*/*@RequestMapping*(path = "/design", produces = "application/json")

/\* Allows angular client to consume your API \*/

*@CrossOrigin*(origins = "\*")

public class DesignTacoController {

private TacoRepository tacoRepo;

*@Autowired*

EntityLinks entityLinks;

public DesignTacoController(TacoRepository tacoRepo) {

this.tacoRepo = tacoRepo;

}

*@GetMapping*("/recent")

/\*

\*  PageRequest object specifies that you only want 0th page of 12 results

\* sorted in descending order by taco’s creation date.

\*/

public Resources<Resource<Taco>> recentTacos() {

PageRequest page = PageRequest.*of*(0, 12, Sort.*by*("createdAt").descending());

/\*

\*  The PageRequest is passed into the call to the findAll() method of

\* TacoRepository, and the content of that page of results is returned to the

\* client which will be used as model data to display to the user.

\*/

List<Taco> tacos = tacoRepo.findAll(page).getContent();

Resources<Resource<Taco>> recentResources = Resources.*wrap*(tacos);

recentResources.add(

ControllerLinkBuilder.*linkTo*(DesignTacoController.class)

.slash("recent")

.withRel("recents")

);

return recentResources;

}

/\*

\* Return a ResponseEntity<Taco>.

\*

\* Handles request for /design/{id}, where {id} is a placeholder. The actual

\* value is given the id parameter, which is mapped to the {id} placeholder

\* by @PathVariable

\*/

*@GetMapping*("/{id}")

public ResponseEntity<Taco> tacoById(*@PathVariable*("id") Long id) {

/\*

\* Optional because there may not be a Taco with given id. If the taco is found

\* you wrap the Taco object in ResponseEntity with HttpStatus of Ok

\*/

Optional<Taco> optTaco = tacoRepo.findById(id);

if(optTaco.isPresent()) {

return new ResponseEntity<>(optTaco.get(),*HttpStatus*.***OK***);

}

return new ResponseEntity<>(null,*HttpStatus*.***NOT\_FOUND***);

}

/\* • Will handle post requests for /design \*/

/\*

\* • The consumes attribute says that the method will only handle requests whose

\* Content-type matches application/json.

\*/

*@PostMapping*(consumes = "application/json")

/\*

\* • @ResponseStatus(HttpStatus.CREATED) tells the client that not only was the

\* request successful, but a resource was created as a result.

\*/

*@ResponseStatus*(*HttpStatus*.***CREATED***)

/\*

\* • @RequestBody indicates that the body of the the request should be converted

\* to a Taco object and bound to the parameter. Without this, Spring MVC would

\* assume that you want request parameters(either query parameter or form

\* parameters) to be bound to the Taco object. But @RequestBody ensures that

\* JSON in the request body is bound to the Taco object instead.

\*/

public Taco postTaco(*@RequestBody* Taco taco) {

/\*

\* • Once Taco object is received, it passes It to save() method on the

\* TacoRepository

\*/

return tacoRepo.save(taco);

}

/\*

\* @DeleteMapping is used to specify that the deleteOrder() method is reponsible

\* for handling DELETE requests for /orders/{orderId}

\*/

*@DeleteMapping*("/{orderId}")

*@ResponseStatus*(code = *HttpStatus*.***NO\_CONTENT***)

public void deleteOrder(*@PathVariable*("orderId") Long orderId) {

try {

tacoRepo.deleteById(orderId);

}

catch(EmptyResultDataAccessException e) {}

}

}

* + - In @RequestMapping, you can set produces to an array of String for multiple content types. Eg:

@RequestMappping(path = “/design”,produces = {“application/json”,”text/xml”})

* + - @CrossOrigin is used because we have angular portion of the application that will be running on a separate host and/or port from the API. This will prevent your angular client to consume the API. To solve this you write @CrossOrigin as it allows clients from any domain to consume the API
    - PageRequest object specifies that you only want 0th page of 12 results sorted in descending order by taco’s creation date.
    - The PageRequest is passed into the call to the findAll() method of TacoRepository, and the content of that page of results is returned to the client which will be used as model data to display to the user.
  + If you want an endpoint that fetches a single taco by its id, you can do the following:

/\*

\* Handles request for /design/{id}, where {id} is a placeholder. The actual

\* value is given the id parameter, which is mapped to the {id} placeholder

\* by @PathVariable

\*/

*@GetMapping*("/{id}")

public Taco tacoById(*@PathVariable*("id") Long id) {

/\*

\* Optional because there may not be a Taco with given id.If there is you get

\* the actual taco by .get() method

\*/

Optional<Taco> optTaco = tacoRepo.findById(id);

if(optTaco.isPresent()) {

return optTaco.get();

}

return null;

* + - If there is not taco with given id it returns null but http status code of 200(ok).}
    - The client is handed a response it can’t use but status code says ok.
    - You might want to set a status code of 404(not found)
  + You can do this by adding ResponseEntity<Taco> as follows;

/\*

\* Return a ResponseEntity<Taco>.

\*

\* Handles request for /design/{id}, where {id} is a placeholder. The actual

\* value is given the id parameter, which is mapped to the {id} placeholder

\* by @PathVariable

\*/

*@GetMapping*("/{id}")

public ResponseEntity<Taco> tacoById(*@PathVariable*("id") Long id) {

/\*

\* Optional because there may not be a Taco with given id. If the taco is found

\* you wrap the Taco object in ResponseEntity with HttpStatus of Ok

\*/

Optional<Taco> optTaco = tacoRepo.findById(id);

if(optTaco.isPresent()) {

return new ResponseEntity<>(optTaco.get(),*HttpStatus*.***OK***);

}

return new ResponseEntity<>(null,*HttpStatus*.***NOT\_FOUND***); }

* + You now have a start of the Taco cloud API for your angular client – any other client. For development testing, you may want to use command line utilities like curl or HTTPie to poke about the API

**curl localhost:8080/design/recent**

* 6.1.2 SENDING DATA TO THE SERVER:
  + To handle the client code (html) for taco form, an angular component named DesignComponenet(in a file named design.component.ts) is defined.
  + As it has form submission ,we use onSubmit() method like this:

onSubmit(){

this.httpClient.post(

‘http://localhost:8080/design’,

this.model,{

headers: new HttpHeaders().set(‘Content-type’,’application/json’),

}).subscribe(taco=>this.cart.addToCart(taco));

this.router.navigate([‘/cart’]);

}

)

}

* + - The post() method of httpClient is called instead of get(). This means that instead of getting data From the API, you are sending data to the API.
    - Specifically, you’re sending taco design which is held in the model variable, to the the API endpoint at /design with an HTTP POST request.
    - So, you need to add postTaco() method in DesignTacoController.

*@PostMapping*(consumes = "application/json")

*@ResponseStatus*(*HttpStatus*.***CREATED***)

public Taco postTaco(*@RequestBody* Taco taco) {

return tacoRepo.save(taco);

}

* + - * Will handle post requests for /design
      * The consumes attribute says that the method will only handle requests whose Content-type matches application/json.
      * @RequestBody indicates that the body of the the request should be converted to a Taco object and bound to the parameter. Without this, Spring MVC would assume that you want request parameters(either query parameter or form parameters) to be bound to the Taco object. But @RequestBody ensures that JSON in the request body is bound to the Taco object instead.
      * Once Taco object is received, it passes It to save() method on the TacoRepository
      * @ResponseStatus(HttpStatus.CREATED) tells the client that not only was the request successful, but a resource was created as a result.
* 6.1.3 UPDATING DATA ON THE SERVER
  + ALthought PUT is often used to update resource data, it’s actual the semantic opposite of GET.
  + Whereas get requests are for transferring data from the server to the client, PUT requests are for sending data from the client to the server.
  + In contrast, HTTP patch is to perform a patch or partial update of resource data.
  + Suppose, you want to be able to change the address on an order.
  + One way is to use PUT request:

@PutMapping(“/{orderId}”)

Public Order putOrder(@RequestBody Order order){

Return repo.save(order);

}

* + - This will work but it require client to submit the complete order data in the PUT request.
    - PUT means ‘put this data at this URL”, essentially replacing any data that’s already there.
  + If PUT does a wholesale replacement of the resource data, PATCH does the partial update.

@PatchMapping(path = “/{orderId}”, consumes = “application/json”)

Public Order patchOrder(@PathVariable(“orderId” Long orderId, @RequestBody Order patch){

Order order = repo.findById(orderId).get();

If(patch.getDeliveryName() != null){

Order.setDeliveryName(patch.getDeliveryame());

}

If(patch.getDeliveryStreet() != null){

Order.setDeliveryStreet(patch.getDeliveryStreet());

}

if (patch.getDeliveryCity() != null) {

order.setDeliveryCity(patch.getDeliveryCity());

}

if (patch.getDeliveryState() != null) { order.setDeliveryState(patch.getDeliveryState());

}

if (patch.getDeliveryZip() != null) {

order.setDeliveryZip(patch.getDeliveryState());

}

if (patch.getCcNumber() != null) {

order.setCcNumber(patch.getCcNumber());

}

if (patch.getCcExpiration() != null) {

order.setCcExpiration(patch.getCcExpiration());

}

if (patch.getCcCVV() != null) {

order.setCcCVV(patch.getCcCVV());

}

Return repo.save(order);

}

* + In case of putOrder() method, you accepted the complete data for an order and saved it.
  + In patchMapping(), instead of completely replacing the order with the new data sent it, it inspects each field of the incoming Order object and applies any non-null values to the existing order. This approach allows the client to only send the properties that should be changed and enables the server to retain the existing data for any properties not specified by the client.
* 6.1.4 DELETING DATA FROM THE SERVER
  + Lets say you want your API to allow for order resource to be deleted. The following controller method should do the trick.

*@DeleteMapping*("/{orderId}")

*@ResponseStatus*(code = *HttpStatus*.***NO\_CONTENT***)

public void deleteOrder(*@PathVariable*("orderId") Long orderId) {

try {

tacoRepo.deleteById(orderId);

}

catch(EmptyResultDataAccessException e) {}

}

* + EmptyResultDataExcpetion does nothing because if you try to delete an order that doesn’t exist, there is no change whatsoever.
  + @ReponseStatus will tell users that there is no content whether the order is there or not.
  + Responses to DELETE requests typically have no body and therefore should communicate an HTTP status code to let the client know not to expect any content.
* 6.2 ENABLING HYPERMEDIA:
  + Till now, the client is hard coded to know that it can obtain list of recently created tacos by issusing GET request for /design/recent. Similarly, it is hardcoded to know that it can append the ID of any taco in that list to /design to get the URL for that particular taco resource.
  + However, this is very brittle technique. If you change the URL scheme of API, the client won’t have any understanding of what to do. So, we use HATEOAS (Hypermedia as the Engine of Application State).
  + HATEOAS is a means of creating self-describing APIs wherein resources returned from an API contain links to related resources. This enables clients to navigate an API with minimal understanding of the APIs URL.
  + For example, suppose a client requests a list of recently designed tacos. In its raw form with no hyperlinks, the JSON will look like this

[

{ "id": 4,

"name": "Veg-Out",

"createdAt": "2018-01-31T20:15:53.219+0000",

"ingredients": [

{"id": "FLTO", "name": "Flour Tortilla", "type": "WRAP"},

{"id": "COTO", "name": "Corn Tortilla", "type": "WRAP"},

{"id": "TMTO", "name": "Diced Tomatoes", "type": "VEGGIES"},

{"id": "LETC", "name": "Lettuce", "type": "VEGGIES"},

{"id": "SLSA", "name": "Salsa", "type": "SAUCE"}

]

},

.. .

]

* + If the client wanted to perform some HTTP operation on the taco itself, it would need to know(via hardcoding) that it could append the value of the id property to a URL whose path is /design.
  + Similarly, it would also need to prefix that path with http:// or https:// and the hostname of the API.
  + If the API is enabled with hypermedia, the API will describe its own URLs, relieving the client of needing to be hardcoded with that knowledge.

{

"\_embedded": {

"tacoResourceList": [

{

"name": "Veg-Out",

"createdAt": "2018-01-31T20:15:53.219+0000",

"ingredients": [

{

"name": "Flour Tortilla", "type": "WRAP",

"\_links": {

"self": { "href": "http://localhost:8080/ingredients/FLTO" }

}

},

{

"name": "Corn Tortilla", "type": "WRAP",

"\_links": {

"self": { "href": "http://localhost:8080/ingredients/COTO" }

}

},

{

"name": "Diced Tomatoes", "type": "VEGGIES",

"\_links": {

"self": { "href": "http://localhost:8080/ingredients/TMTO" }

}

},

{

"name": "Lettuce", "type": "VEGGIES",

"\_links": {

"self": { "href": "http://localhost:8080/ingredients/LETC" }

}

},

{

"name": "Salsa", "type": "SAUCE",

"\_links": {

"self": { "href": "http://localhost:8080/ingredients/SLSA" }

}

}

],

"\_links": {

"self": { "href": "http://localhost:8080/design/4" }

}

},

...

]

},

"\_links": {

"recents": { "href": "http://localhost:8080/design/recent" }

}

}

* + This particular flavor of HATEOAS is known as HAL (Hypertext Application Language), a simple and commonly used format for embedding hyperlinks in JSON responses.
  + Each element in this new list of tacos includes a property named \_links that contains hyperlinks for the client to navigate the API.
  + Should a client application need to perform an HTTP request against a taco in the list, it doesn’t need to be developed with any knowledge of what the taco resource’s URL would look like. Instead it knows to ask for the self link which maps to http://localhost:8080 /design/4.
  + SPRING HATEOAS project brings hyperlink support to Spring. It offers a set of classes and resource assemblers that can be used to add links to resources before returning them from a Spring MVC controller.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-hateoas</artifactId>

</dependency>

* 6.2.1 Adding hyperlinks
  + Spring HATEOAS provides two primary types that represent hyperlinked resources: Resource and Resources.
  + The Resource type represents a single resource, whereas Resources is a collection of resources.
  + When returned from Spring MVC REST controller method, the links they carry will be included in the JSON(or XML) received by the client.

@GetMapping("/recent")

public Resources<Resource> recentTacos() {

PageRequest page = PageRequest.of(

0, 12, Sort.by("createdAt").descending());

List tacos = tacoRepo.findAll(page).getContent();

Resources<Resource> recentResources = Resources.wrap(tacos);

recentResources.add( new Link("http://localhost:8080/design/recent", "recents"));

return recentResources;

}

* + In this new version of recentTacos(), you use Resources.wrap() to wrap the list of tacos as an instance of Resources<Resource<Taco>>, which is ultimately returned from the method.
  + But before returning the Resources object, you add a link whose Relationship name is recents and whose URL is <http://localhost:8080/design/recent>. As a consequence, the following snippet of JSON is included in the resource returned from the API request:

"\_links": {

"recents": {

"href": <http://localhost:8080/design/recent>

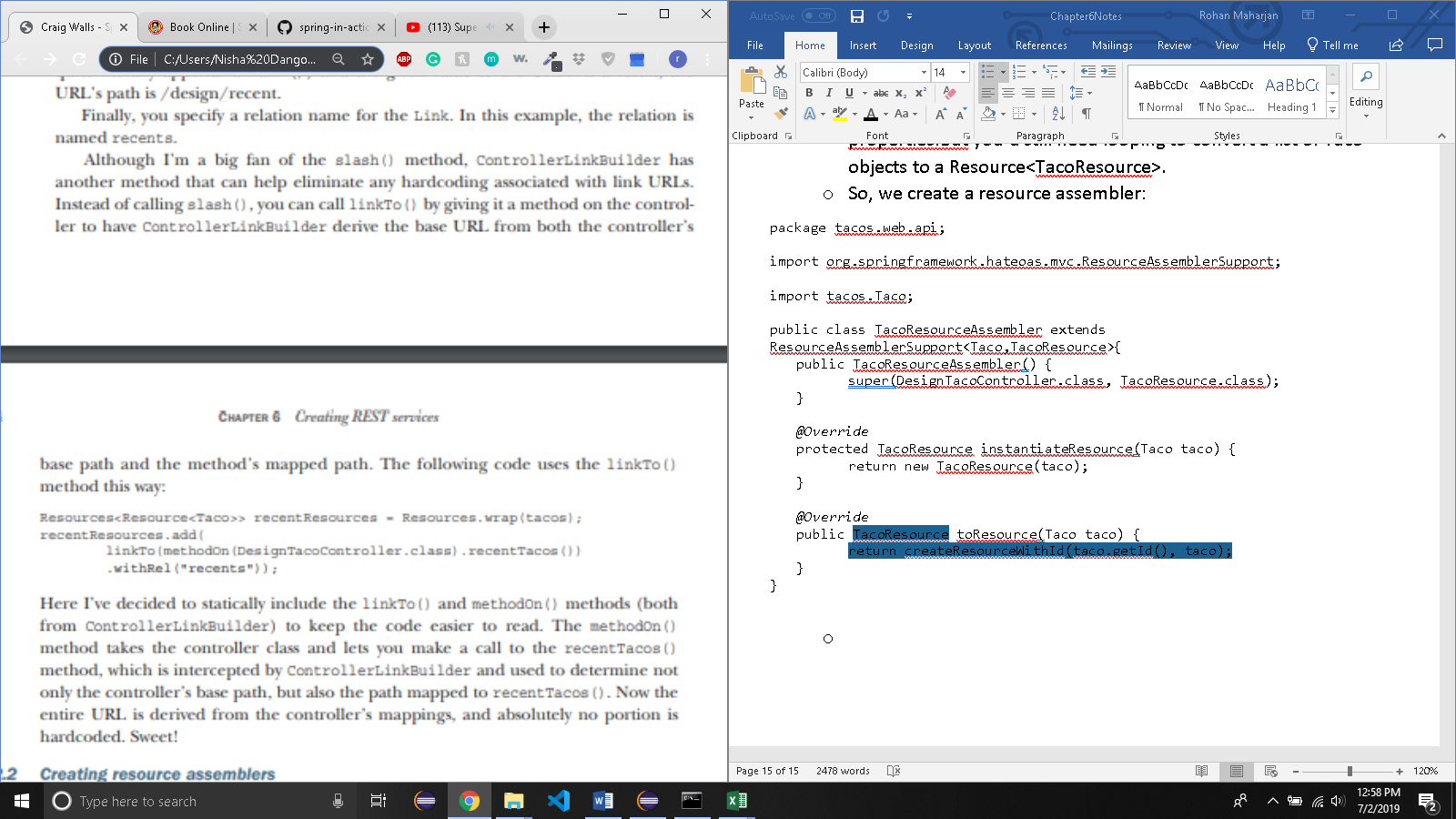
}

}

* + Hardcoding a URL like this is a very bad idea. We can use ControllerLinkBuilder to create links.
  + It is smart enough to know what the hostname is without you having to hardcode it.

Resources<Resource> recentResources = Resources.wrap(tacos); recentResources.add( ControllerLinkBuilder.linkTo(DesignTacoController.class) .slash("recent")

.withRel("recents"));

* + You don’t have to specify the /design path. Instead you ask for a link to DesignTacoController, whose base path is /design. ControllerLinkBuilder uses the controller’s base path as the foundation of the link object you’re creating.
  + Slash() method appends the slash(/) and the given value to the URL. As a result, the URL’s path is /design/recent.
  + Finally, you specify a relation name for the Link. In this example, the relation is named recents.
  + 
* 6.2.2 CREATING RESOURCE ASSEMBLERS
  + Now you need to add links to the taco resource contained within the list.
  + We could loop through each of the Resource<Taco> elements carried in the Resources object, adding a Link to each individually. But that’s a bit tedious and you’d need to repeat that looping code in the API wherever you return a list of taco resrouces.
  + Rather than let Resources.wrap() create a Resource object for each taco in the list, you’re going to define a utility class that converts Taco objects to a new TacoResource object.

package tacos.web.api;

import java.util.Date;

import java.util.List;

import org.springframework.hateoas.ResourceSupport;

import lombok.Getter;

import tacos.Ingredient;

import tacos.Taco;

public class TacoResource extends ResourceSupport {

@Getter private final String name;

@Getter private final Date createdAt;

@Getter private final List ingredients;

public TacoResource(Taco taco) {

this.name = taco.getName();

this.createdAt = taco.getCreatedAt();

this.ingredients = taco.getIngredients();

}

}

* + TacoResource isn’t that different from the Taco domain type. They both have name, createdAt, and ingredients properties. But TacoResource extends ResourceSupport to inherit a list of Link object and methods to manage the list of links
  + TacoResource doesn’t include the id property from Taco. That’s because there’s no need to expose any database specific IDs in the API. The resource’s self link will serve as the identifies for the resource from the perspective of any API client.
  + We could extend ResourceSupport in Taco class. But by creating a separate resource type i.e TacoResource, the Taco class isn’t unnecessarily cluttered with resource links for use cases where links aren’t needed. Also, we were able to leave the id property out so that it won’t be exposed in the API.
  + TacoResource has a single constructor that accepts a Taco and copies the pertinent properties from the Taco to its own properties.But you’d still need looping to convert a list of Taco objects to a Resource<TacoResource>.
  + So, we create a resource assembler:

package tacos.web.api;

import org.springframework.hateoas.mvc.ResourceAssemblerSupport;

import tacos.Taco;

public class TacoResourceAssembler extends ResourceAssemblerSupport<Taco,TacoResource>{

public TacoResourceAssembler() {

super(DesignTacoController.class, TacoResource.class);

}

*@Override*

protected TacoResource instantiateResource(Taco taco) {

return new TacoResource(taco);

}

*@Override*

public TacoResource toResource(Taco taco) {

return createResourceWithId(taco.getId(), taco);

}

}

* + TacoResourceAssembler has a default constructor that informs the superclass(ResourceAssemblerSupport) that it will be using DesignTacoController to determine the base path for any URLs in the links it creates when creating a TacoResource.
  + The instantiateResource() method is overridden to instantiate a TacoResource given a Taco. This method would be optional if TacoResource had a default constructor. However, TacoResource requires Taco as a parameter, so you’re required to override it.
  + Finally, the toResource() method is the only method that’s strictly mandatory when extending ResourceAssemblerSupport. Here, you are telling it to create a TacoResource object from a Taco, and to automatically give it a self link with the URL being derived from the Taco object’s id property.
  + Whereas instantiateResource() is intended to only instantiate a Resource object, toResource() is intended not only to create the Resource object, but also to populate it with links. Under the covers toResource() will call instantiateResource().
* Now, tweak the recentTacos() method to use TacoResourceAssembler:

*@GetMapping*("/recent")

public Resources<TacoResource> recentTacos() {

PageRequest page = PageRequest.*of*(0, 12, Sort.*by*("createdAt").descending());

List<Taco> tacos = tacoRepo.findAll(page).getContent();

List<TacoResource> tacoResources = new TacoResourceAssembler().toResources(tacos);

Resources<TacoResource> recentResources = new Resources<TacoResource>(tacoResources);

recentResources

.add(ControllerLinkBuilder.*linkTo*(DesignTacoController.class).slash("recent").withRel("recents"));

return recentResources;

}

* + After fetching the tacos from the repository, you pass the list of Taco objects to the toResources() method on a TacoResourceAssembler. This handy method cycles through all of the Taco objects, calling the toResource() method that you overrode in TacoResourceAssembler to create a list of TacoResource objects.
  + With that TacoResource list, you next create a Resources object and then populate it with the recents links as in the prior version of recentTacos()
  + At this point, a GET request to /design/recent will produce a list of tacos, each with a self link and a recents link on the list itself. But the ingredients will still be without a link. To address that, you’ll create a new resource assembler for ingredients:

package tacos.web.api;

import org.springframework.hateoas.mvc.ResourceAssemblerSupport;

import tacos.Ingredient;

public class IngredientResourceAssembler extends ResourceAssemblerSupport<Ingredient, IngredientResource>{

public IngredientResourceAssembler() {

super(IngredientController2.class, IngredientResource.class);

}

*@Override*

public IngredientResource toResource(Ingredient ingredient) {

return createResourceWithId(ingredient.getId(), ingredient);

}

*@Override*

protected IngredientResource instantiateResource(Ingredient ingredient) {

return new IngredientResource(ingredient);

}

}

* + As you can see, IngredientResourceAssembler is much like TacoResourceAssembler, but it works with Ingredient and IngredientResource objects instead of Taco and TacoResource objects.

package tacos.web.api;

import org.springframework.hateoas.ResourceSupport;

import lombok.Getter;

import tacos.Ingredient;

public class IngredientResource extends ResourceSupport{

*@Getter*

private String name;

*@Getter*

private String type;

public IngredientResource(Ingredient ingredient) {

this.name = ingredient.getName();

this.type = ingredient.getType();

}

}

* + As with TacoResource, IngredientResource extends ResourceSupport and copies pertinent properties from the domain type into its own set of properties (leaving out the id property). All that’s left is to make a slight change to TacoResource so that it carries IngredientResource objects instead of Ingredient objects:

package tacos.web.api;

import java.util.Date;

import java.util.List;

import org.springframework.hateoas.ResourceSupport;

import lombok.Getter;

import tacos.Ingredient;

import tacos.Taco;

public class TacoResource extends ResourceSupport{

private static final IngredientResourceAssembler ingredientAssembler = new IngredientResourceAssembler();

@Getter

private final String name;

@Getter

private final Date createdAt;

@Getter

private final List<IngredientResource> ingredients;

public TacoResource(Taco taco) {

this.name = taco.getTacoName();

this.createdAt = taco.getCreatedAt();

this.ingredients = ingredientAssembler.toResources(taco.getIngredients());

}

}

* + This new version of TacoResource creates a static final instance of IngredientResourceAssembler and uses its toResource() method to convert a given Taco object’s list of Ingredient into a list of IngredientResource. Your recent tacos list is now completely outfitted with hyperlinks, not only for itself (the recents link), but also for all of its taco entries and the ingredients of those tacos. The response should look a lot like the JSON
* 6.2.3 NAMING EMBEDDED RELATIONSHIP:
  + The @Relation annotation can help break the coupling between the JSON field name and the resource type class names as defined in Java. By annotating TacoResource with @Relation, you can specify how Spring HATEOAS should name the field in the resulting JSON:

@Relation(value="taco", collectionRelation="tacos")

public class TacoResource extends ResourceSupport {

...

}

* + Here you’ve specified that when a list of TacoResource objects is used in a Resources object, it should be named tacos. And although you’re not making use of it in our API, a single TacoResource object should be referred to in JSON as taco. As a result, the JSON returned from /design/recent will now look like this (no matter what refactoring you may or may not perform on TacoResource):

{

"\_embedded": {

"tacos": [

...

]

}

}

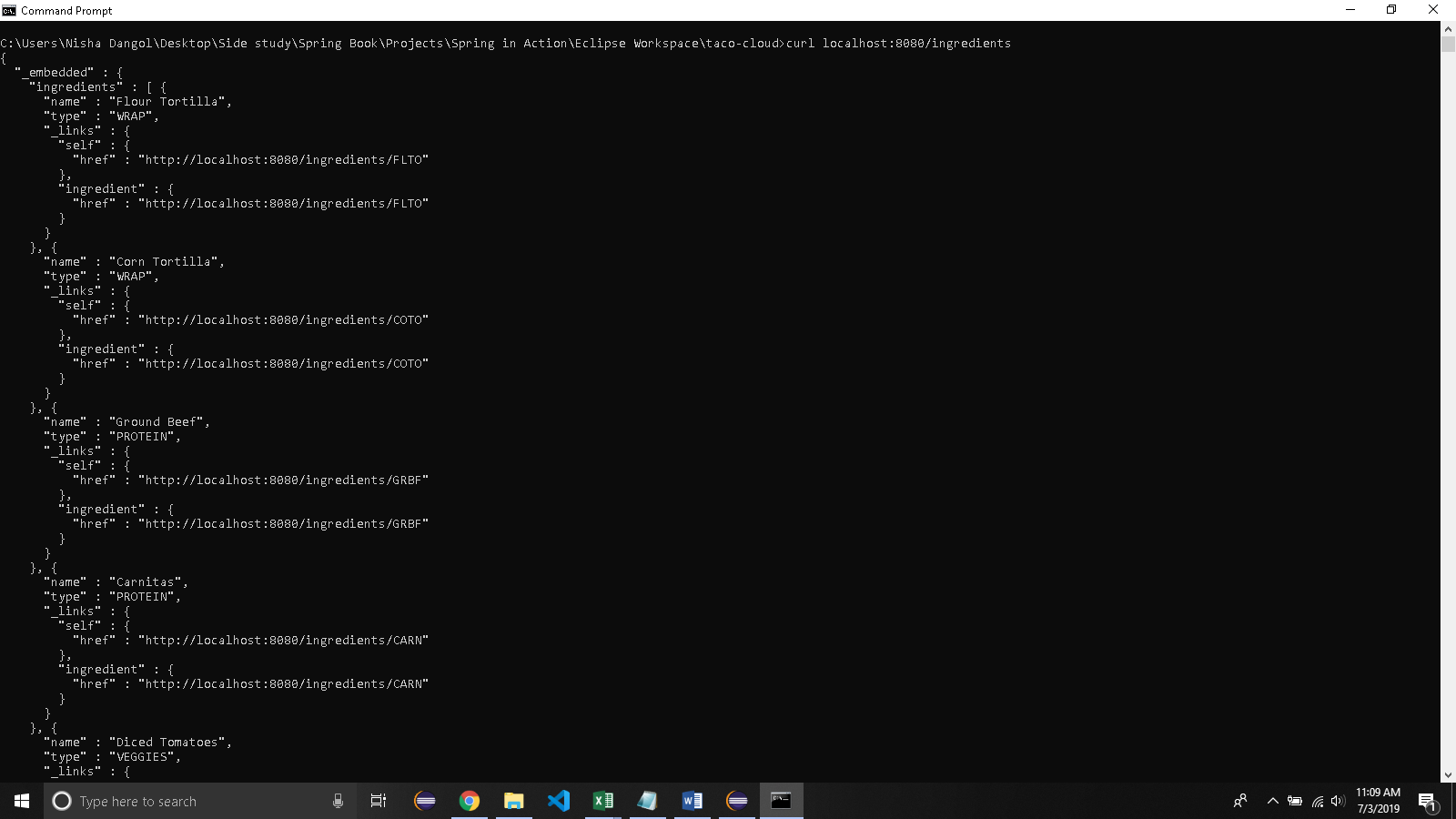
* + Let’s see how Spring Data REST can help you automatically create APIs based on the data repositories you created with Spring Data in chapter 3.
* 6.3 ENABLING DATA BACKED SERVICES:
  + Spring Data REST is another member of the Spring Data family that automatically creates REST APIs for repositories created by Spring Data
  + To start using Spring Data REST, you add the following dependency to your build:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-rest</artifactId>

</dependency>

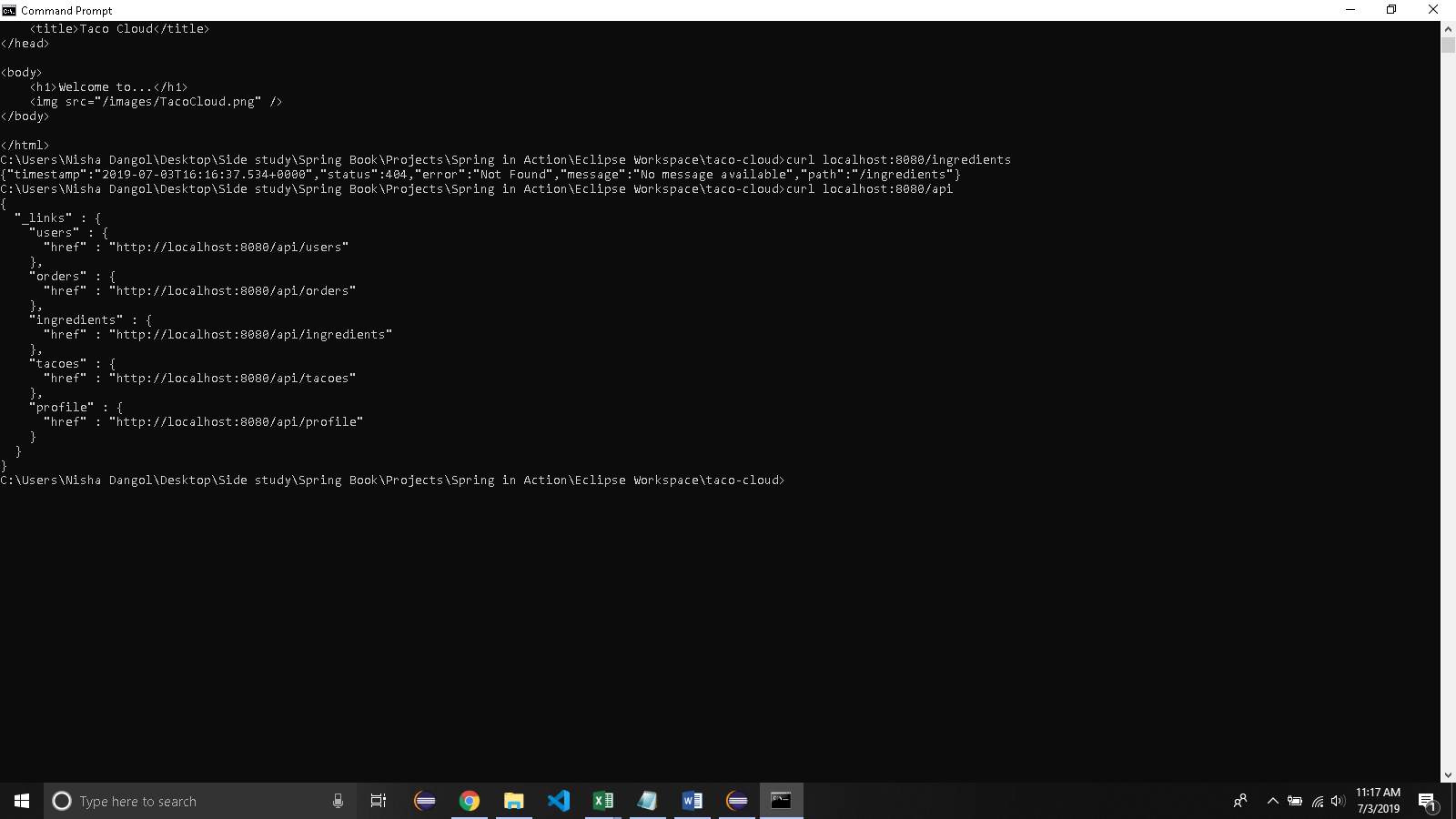
* + By simply having the Spring Data REST starter in the build, the application gets auto-configuration that enables automatic creation of a REST API for any repositories that were created by Spring Data (including Spring Data JPA, Spring Data Mongo, and so on).
  + remove any @RestController-annotated classes you’ve created up to this point before moving on.
  + Based on the set of repositories you’ve already defined for Taco Cloud, you should be able to perform GET requests for tacos, ingredients, orders, and users.
  + $ curl localhost:8080/ingredients
* 
  + you should know that Spring data rest also supports POST, PUT, and DELETE methods for the endpoints it creates.
  + you can POST to /ingredients to create a new ingredient and DELETE /ingredients/FLTO to remove flour tortillas from the menu.
  + One thing you might want to do is set a base path for the API so that its endpoints are distinct and don’t collide with any controllers you write. (In fact, if you don’t remove the IngredientsController you created earlier, it will interfere with the /ingredients endpoint provided by Spring Data REST.) To adjust the base path for the API, set the spring.data.rest.base-path property:

spring:

data:

rest:

base-path: /api

* 
  + When you write curl localhost:8080/api/tacos it gives an error because spring data makes tacoes instead of tacos.
* 6.3.1 ADJUSTING RESOURCE PATHS AND RELATION NAMES:
  + When creating endpoints for Spring Data repositories, Spring Data REST tries to pluralize the associated entity class. For the Ingredient entity, the endpoint is /ingredients. For the Order and User entities it’s /orders and /users. So far, so good.
  + As it turns out, Spring Data REST pluralized “taco” as “tacoes”, so to make a request for tacos
  + By adding a simple annotation to the Taco class, you can tweak both the relation name and that path:

@Data

@Entity

@RestResource(rel="tacos", path="tacos")

public class Taco {

...

}

* + let’s look at how you can sort the results from Spring Data REST endpoints.
* 6.3.2 PAGING AND SORTING
  + By default, requests to a collection resource such as /api/tacos will return up to 20 items per page from the first page. But you can adjust the page size and the page displayed by specifying the page and size parameters in your request
  + For example, to request the first page of tacos where the page size is 5, you can issue the following GET request (using curl):
    - $ curl "localhost:8080/api/tacos?size=5"
  + Assuming that there are more than five tacos to be seen, you can request the second page of tacos by adding the page parameter:
    - $ curl "localhost:8080/api/tacos?size=5&page=1"
  + You could use string manipulation to add those parameters to the URL, but HATEOAS comes to the rescue by offering links for the first, last, next, and previous pages in the response:

"\_links" : {

"first" : {

"href" : "http://localhost:8080/api/tacos?page=0&size=5"

},

"self" : {

"href" : "http://localhost:8080/api/tacos"

},

"next" : {

"href" : "http://localhost:8080/api/tacos?page=1&size=5"

},

"last" : {

"href" : "http://localhost:8080/api/tacos?page=2&size=5"

},

"profile" : {

"href" : "http://localhost:8080/api/profile/tacos"

},

"recents" : {

"href" : "http://localhost:8080/api/tacos/recent"

}

}

* + With these links, a client of the API need not keep track of what page it’s on and concatenate the parameters to the URL. Instead, it must simply know to look for one of these page navigation links by its name and follow it
  + The sort parameter lets you sort the resulting list by any property of the entity. For example, you need a way to fetch the 12 most recently created tacos for the UI to display. You can do that by specifying the following mix of paging and sorting parameters:
    - $ curl "localhost:8080/api/tacos?sort=createdAt,desc&page=0&size=12"
  + Here the sort parameter specifies that you should sort by the createdDate property and that it should be sorted in descending order (so that the newest tacos are first). The page and size parameters specify that you should see the first page of 12 tacos.
  + This is precisely what the UI needs in order to show the most recently created tacos. It’s approximately the same as the /design/recent endpoint you defined in DesignTacoController earlier in this chapter.
  + There’s a small problem, though. The UI code will need to be hardcoded to request the list of tacos with those parameters.
  + It would be great if the client could look up the URL from a list of links. And it would be even more awesome if the URL were more succinct, like the /design/recent endpoint you had before.
* 6.3.3. ADDING CUSTOM ENDPOINTS:
  + Spring Data REST is great at creating endpoints for performing CRUD operations against Spring Data repositories. But sometimes you need to break away from the default CRUD API and create an endpoint that gets to the core of the problem.
  + But when you write your own API controllers, their endpoints seem somewhat detached from the Spring Data REST endpoints in a couple of ways:
    - ϒ Your own controller endpoints aren’t mapped under Spring Data REST’s base path. You could force their mappings to be prefixed with whatever base path you want, including the Spring Data REST base path, but if the base path were to change, you’d need to edit the controller’s mappings to match.
    - ϒ Any endpoints you define in your own controllers won’t be automatically included as hyperlinks in the resources returned by Spring Data REST endpoints. This means that clients won’t be able to discover your custom endpoints with a relation name.
  + all mappings in a @RepositoryRestControllerannotated controller will have their path prefixed with the value of the spring.data .rest.base-path property (which you’ve configured as /api).
  + Rather than resurrect the DesignTacoController, which had several handler methods you won’t need, you’ll create a new controller that only contains the recentTacos() method.

package tacos.web.api;

import java.util.List;

import org.springframework.data.domain.PageRequest;

import org.springframework.data.domain.Sort;

import org.springframework.data.rest.webmvc.RepositoryRestController;

import org.springframework.hateoas.Resources;

import org.springframework.hateoas.mvc.ControllerLinkBuilder;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

import tacos.Taco;

import tacos.data.TacoRepository;

*@RepositoryRestController*

public class RecentTacosController {

private TacoRepository tacoRepo;

public RecentTacosController(TacoRepository tacoRepo) {

this.tacoRepo = tacoRepo;

}

*@GetMapping*(path = "/tacos/recent", produces = "application/hal+json")

public ResponseEntity<Resources<TacoResource>> recentTacos(){

PageRequest page = PageRequest.*of*(0,12,Sort.*by*("createdAt").descending());

List<Taco> tacos = tacoRepo.findAll(page).getContent();

List<TacoResource> tacoResources = new TacoResourceAssembler().toResources(tacos);

Resources<TacoResource> recentResources = new Resources<TacoResource>(tacoResources);

recentResources.add(ControllerLinkBuilder.*linkTo*(RecentTacosController.class).slash("recent").withRel("recents"));

return new ResponseEntity<>(recentResources,*HttpStatus*.***OK***);

}

}

* + Even though @GetMapping is mapped to the path /tacos/recent, the @RepositoryRestController annotation at the class level will ensure that it will be prefixed with Spring Data REST’s base path. As you’ve configured it, the recentTacos() method will handle GET requests for /api/tacos/recent.
  + One important thing to notice is that although @RepositoryRestController is named similarly to @RestController, it doesn’t carry the same semantics as @RestController. Specifically, it doesn’t ensure that values returned from handler methods are automatically written to the body of the response. Therefore you need to either annotate the method with @ResponseBody or return a ResponseEntity that wraps the response data. Here you chose to return a ResponseEntity
  + With RecentTacosController in play, requests for /api/tacos/recent will return up to 15 of the most recently created tacos, without the need for paging and sorting parameters in the URL. But it still doesn’t appear in the hyperlinks list when requesting /api/tacos. Let’s fix that.
* 6.3.4 ADDING CUSTOM HYPERLINKS TO SPRING DATA ENDPOINTS:
  + By declaring a resource processor bean, however, you can add links to the list of links that Spring Data REST automatically includes.
  + Spring Data HATEOAS offers ResourceProcessor, an interface for manipulating resources before they’re returned through the API.
  + For your purposes, you need an implementation of ResourceProcessor that adds a recents link to any resource of type PagedResources<Resource<Taco>> (the type returned for the /api/tacos endpoint).

package tacos.web.api;

import java.util.List;

import org.springframework.context.annotation.Bean;

import org.springframework.data.domain.PageRequest;

import org.springframework.data.domain.Sort;

import org.springframework.data.rest.webmvc.RepositoryRestController;

import org.springframework.hateoas.EntityLinks;

import org.springframework.hateoas.PagedResources;

import org.springframework.hateoas.Resource;

import org.springframework.hateoas.ResourceProcessor;

import org.springframework.hateoas.Resources;

import org.springframework.hateoas.mvc.ControllerLinkBuilder;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

import tacos.Taco;

import tacos.data.TacoRepository;

*@RepositoryRestController*

public class RecentTacosController {

private TacoRepository tacoRepo;

public RecentTacosController(TacoRepository tacoRepo) {

this.tacoRepo = tacoRepo;

}

*@GetMapping*(path = "/tacos/recent", produces = "application/hal+json")

public ResponseEntity<Resources<TacoResource>> recentTacos(){

PageRequest page = PageRequest.*of*(0,12,Sort.*by*("createdAt").descending());

List<Taco> tacos = tacoRepo.findAll(page).getContent();

List<TacoResource> tacoResources = new TacoResourceAssembler().toResources(tacos);

Resources<TacoResource> recentResources = new Resources<TacoResource>(tacoResources);

recentResources.add(ControllerLinkBuilder.*linkTo*(RecentTacosController.class).slash("recent").withRel("recents"));

return new ResponseEntity<>(recentResources,*HttpStatus*.***OK***);

}

*@Bean*

public ResourceProcessor<PagedResources<Resource<Taco>>> tacoProcessor(EntityLinks links){

return new ResourceProcessor<PagedResources<Resource<Taco>>>() {

*@Override*

public PagedResources<Resource<Taco>> process(PagedResources<Resource<Taco>> resource){

resource.add(links.linkFor(Taco.class)

.slash("recent")

.withRel("recents"));

return resource;

}

};

}

}

* + The ResourceProcessor shown in listing 6.8 is defined as an anonymous inner class and declared as a bean to be created in the Spring application context.
  + Spring HATEOAS will discover this bean (as well as any other beans of type ResourceProcessor) automatically and will apply them to the appropriate resources
  + In this case, if a PagedResources<Resource> is returned from a controller, it will receive a link for the most recently created tacos. This includes the response for requests for /api/tacos.