Matuarity Model

Leonard Richardson

- Level 0. Swamp of POX --> HTTP as medium of transport
- Level 1. Resources --> Use multiple URI's
- Level 2. HTTP verbs --> Use transport native properties
- Level 3. Hypermedia controls (HATEOAS) --> no a priori knowledge necessary to navigate a service

Basics of Rest

```
@RestController
@RequestMapping("/v1/customers")
public class CustomerRestController {
 @Autowired
 private CustomerRepository customerRepository;
 ....
}
```

- in essence

```
@Controller
@ResponseBody
public @interface RestController
```

About CustomerRepository

```
public interface CustomerRepository
                        extends JpaRepository<Customer, Long> {

 Collection<Customer> findByFirstNameIgnoreCase(@Param("fn") Stri

 Optional<Customer> findById(@Param("id") Long id);

 Collection<Customer> findByLastNameIgnoreCase(@Param("ln") Strin
}
```

- Remark: The Optional return type of findById

RequestMethod.OPTIONS

```
@RequestMapping(method = RequestMethod.OPTIONS)
 ResponseEntity<?> options() {
  return ResponseEntity
    .ok()
    .allow(HttpMethod.GET, HttpMethod.POST,
           HttpMethod.HEAD, HttpMethod.OPTIONS,
           HttpMethod.PUT, HttpMethod.DELETE)
           .build();
 }
```

@GetMapping's

```
@GetMapping
ResponseEntity<Collection<Customer>> getCollection() {
 return ResponseEntity.ok(this.customerRepository.findAll());
}

 @GetMapping(value = "/{id}")
ResponseEntity<Customer> get(@PathVariable Long id) {
 return this.customerRepository.findById(id).map(ResponseEntity:
  .orElseThrow(() -> new CustomerNotFoundException(id));
}
```

@PostMapping

```
@PostMapping
ResponseEntity<Customer> post(@RequestBody Customer c) {

 Customer customer = this.customerRepository.save(new Customer(c
  .getFirstName(), c.getLastName()));

 URI uri = MvcUriComponentsBuilder.fromController(getClass()).pa
  .buildAndExpand(customer.getId()).toUri();
 return ResponseEntity.created(uri).body(customer);
}
```

@DeleteMapping

```
@DeleteMapping(value = "/{id}")
 ResponseEntity<?> delete(@PathVariable Long id) {
  return this.customerRepository.findById(id).map(c -> {
   customerRepository.delete(c);
   return ResponseEntity.noContent().build();
  }).orElseThrow(() -> new CustomerNotFoundException(id));
 }
```

@RequestMapping HEAD

```
@RequestMapping(value = "/{id}", method = RequestMethod.HEAD)
ResponseEntity<?> head(@PathVariable Long id) {
 return this.customerRepository.findById(id)
  .map(exists -> ResponseEntity.noContent().build())
  .orElseThrow(() -> new CustomerNotFoundException(id));
}
```

@PutMapping

```
@PutMapping(value = "/{id}")
 ResponseEntity<Customer> put(@PathVariable Long id,
                              @RequestBody Customer c) {
  return this.customerRepository
   .findById(id)
   .map(
  c -> {Customer customer = this.customerRepository.save(
        new Customer(c .getId(), c.getFirstName(), c.getLastName(
  URI selfLink = URI.create(
     ServletUriComponentsBuilder.fromCurrentRequest().toUriString
  return ResponseEntity.created(selfLink).body(customer);
 }).orElseThrow(() -> new CustomerNotFoundException(id));

}
```

CustomerNotFoundException

```
public class CustomerNotFoundException extends RuntimeException {
 private final Long id;

 public CustomerNotFoundException(Long id) {
  super("customer-not-found-" + id);
  this.id = id;
 }
 public Long getCustomerId() {
  return id;
 }
}
```

Error Handling

- Inside CustomerRestController
- Several -> new CustomerNotFoundException(id)
- How to treat exceptions consistently

Scaling

- Scale comes from consistency
- Consistency is driven by automation
- Centralize error handling logic
- SpringMVC support

@ControllerAdvice

- introduces behaviour -> and responds to exceptions
- for any number of Controllers
- centralize exception handling
- error handling -> part of effective API

About Errors

- should be unique and concise in pointing to error
- errors should be as helpfull as possible
- HTTP status codes not enough

Common practice

- send back an error code
- some sort of representation of the error
- human readable error message
- de facto standard: application/vnd.error_content_type

Add Hateoas

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>
```

CustomerControllerAdvice

```
import org.springframework.hateoas.VndErrors;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import ...
@ControllerAdvice(annotations = RestController.class)
public class CustomerControllerAdvice {


 private final MediaType vndErrorMediaType = MediaType
  .parseMediaType("application/vnd.error");

...

}
```

@ExceptionHandler

```
@ExceptionHandler(CustomerNotFoundException.class)
ResponseEntity<VndErrors> notFndExc(CustomerNotFoundException e)
return this.error(e, HttpStatus.NOT_FOUND, e.getCustomerId() + ""
}

@ExceptionHandler(IllegalArgumentException.class)
ResponseEntity<VndErrors> assrtExc(IllegalArgumentException ex) {
return this.error(ex, HttpStatus.NOT_FOUND,
                    ex.getLocalizedMessage());
}
```

ResponseEntity

```
private <E extends Exception> ResponseEntity<VndErrors> error(
                E error, HttpStatus httpStatus, String logref) {
String msg = Optional.of(error.getMessage()).orElse(
error.getClass().getSimpleName());
HttpHeaders httpHeaders = new HttpHeaders();
httpHeaders.setContentType(this.vndErrorMediaType);
return new ResponseEntity<>(
      new VndErrors(logref, msg), httpHeaders,httpStatus);
}
```

Hypermedia

- A RestAPI will work fine
- If client knows which endpoint to call
- When to call endpoint
- With what HTTP-method to call

Human interactin with the web

- <link> elements in resources provide information that will lead to state changes
- Go to Amazon.com
  1. search for a product
  2. add it to basker
  3. checkout
- links tell where to go
- links only appear when relevant
- try to make this mechanism also available for applications

<link>

- Has 2 important attributes

  1. rel
  2. href

- most common application:

- <link rel="stylesheet" href="mycss.css">
- browser reads first rel -> what is this?
- href tells browser where to find it -> can live everywhere

<link rel="stylesheet" href="mycss.css">

- the resource can live everywhere
- browser -> no assumptions about location
- browser just follows href
- rel attribute determines relevance of linked resource
- browser(client) -> decoupled from location resource
- rel has role of contract -> when stable -> client will not break

HAL

- <link> is xml
- different standards for JSON
- a defacto standard is HAL
- Hypertext Application Language := HAL
- HAl is specialization of JSON -> application/hal+json

Customer resource


```
{
    "firstName": "Hans",
    "lastName": "S",
    "_links": {
        "self": {
            "href": "http://localhost:8080/v2/customers/1"
        },
        "profile-photo": {
            "href": "http://localhost:8080/customers/1/photo/"
        }
    }
}
```

Working with Hal

- Spring Hateoas sits on top of MVC
- It provides plumbing to consume and produce resources and their links
- Instead of consuming and producing type T (MVC) Hateoas consumes and produces Resource

Resource

- is an envelope object that contains a payload and a set of links
- T is frequently converted into Resource or Resources
- conversion is described in a ResourceAssembler

CustomerHypermediaRestController

```
@RestController
@RequestMapping(value = "/v2", produces = "application/hal+json")
public class CustomerHypermediaRestController {

 private final CustomerResourceAssembler cra;

 private final CustomerRepository crepo;
 @Autowired
 CustomerHypermediaRestController(CustomerResourceAssembler cra,
                                  CustomerRepository crepo) {
 ...
 }
```

@GetMapping root

```
@GetMapping
ResponseEntity<Resources<Object>> root() {
 Resources<Object> objects = new Resources<>(
                                  Collections.emptyList());
 URI uri = MvcUriComponentsBuilder
 .fromMethodCall(MvcUriComponentsBuilder.on(getClass())
    .getCollection()).build().toUri();
 Link link = new Link(uri.toString(), "customers");
 objects.add(link);
return ResponseEntity.ok(objects);
}
```

@GetMapping /customers

```
@GetMapping("/customers")
 ResponseEntity<Resources<Resource<Customer>>> getCollection() {
 List<Resource<Customer>> collect = this.customerRepository
    .findAll().stream().map(customerResourceAssembler::toResource
 .collect(Collectors.<Resource<Customer>>toList());
 Resources<Resource<Customer>> resources =
                 new Resources<>(collect);
 URI self = ServletUriComponentsBuilder.fromCurrentRequest()
                .build().toUri();
 resources.add(new Link(self.toString(), "self"));
return ResponseEntity.ok(resources);
}
```

@GetMapping /customers/{id}

```
@GetMapping(value = "/customers/{id}")
ResponseEntity<Resource<Customer>> get(@PathVariable Long id) {
 return this.customerRepository.findById(id)
  .map(c -> ResponseEntity.ok(
      this.customerResourceAssembler.toResource(c)))
  .orElseThrow(() -> new CustomerNotFoundException(id));
}
```

@PostMapping /customers

```
@PostMapping(value = "/customers")
ResponseEntity<Resource<Customer>> post(@RequestBody Customer c)
 Customer customer = this.customerRepository.save(
    new Customer(c.getFirstName(), c.getLastName()));
 URI uri = MvcUriComponentsBuilder.fromController(getClass())
 .path("/customers/{id}").buildAndExpand(customer.getId()).toUri(
 return ResponseEntity.created(uri).body(
 this.customerResourceAssembler.toResource(customer));
}
```

@PutMapping /customers/{id}

```
@PutMapping("/customers/{id}")
ResponseEntity<Resource<Customer>> put(@PathVariable Long id,
 @RequestBody Customer c) {
 Customer customer = this.customerRepository.save(
     new Customer(id, c.getFirstName(), c.getLastName()));
 Resource<Customer> customerResource =
     this.customerResourceAssembler.toResource(customer);
 URI selfLink = URI.create(ServletUriComponentsBuilder
                .fromCurrentRequest().toUriString());
 return ResponseEntity.created(selfLink).body(customerResource);
}
```

CustomerResourceAssembler

- Assembler doing the Hateoas magic

```
@Component
class CustomerResourceAssembler implements
 ResourceAssembler<Customer, Resource<Customer>> {
 @Override
 public Resource<Customer> toResource(Customer customer) {
  URI selfUri = MvcUriComponentsBuilder
   .fromMethodCall(
    MvcUriComponentsBuilder.on(CustomerHypermediaRestController.c
    .get(customer.getId())).buildAndExpand().toUri();

  customerResource.add(new Link(selfUri.toString(), "self"));
  return customerResource;
 }
}
```

Resume

- The hypermedia in the response enable client to navigate the server withou a priori knoweledge
- HAL is a popular option but there are more
- WADL (2009) Web Application Description Language -> broad scope of describing HTTP services
- ALPS -> more recently , defines possible state transitions and attributes of resources involved in the transition -> media-type agnostic (Part of Spring Rest)

Versioning

- The one certainty: everything will change
- Building with change in mind
- Find ways to change services without breaking clients

Semantic versioning

- service should make explicit the assumptions about what clients will be able to work with
- semantic versioning is one approach

MAJOR.MINOR.PATCH

- MAJOR -> should only change when API has breaking changes with the previous version
- MINOR -> API has evolved but older clients can still work with the newer version
- PATCH -> signals bug fixes to the existing functionality

About Semantic versioning

- Semantic versioning signals clients a change
- Clients are not always able to move to newer version
- You don't want to force clients -> endangers the ability to evolve clients/servers independently
- Consider hosting both version side by side
- Try to forward request to older endpoint to newer endpoints
- A client needs to say which version it wants to use

VersionedRestController

```
@RestController
@RequestMapping("/api")
public class VersionedRestController {

 public static final String V1_MEDIA_TYPE_VALUE
         = "application/vnd.myapp-v1+json";
 public static final String V2_MEDIA_TYPE_VALUE
         = "application/vnd.myapp-v2+json";
 private enum ApiVersion {
  v1, v2
 }

 public static class Greeting {
    ..
 }
```

Helper class Greeting

```
public static class Greeting {
  private String how;
  private String version;

  public Greeting(String how, ApiVersion version) {
   this.how = how;
   this.version = version.toString();
  }
  public String getHow() {
   return how;
  }
  public String getVersion() {
   return version;
  }
 }
```

A versioned API

```
@GetMapping(value = "/{version}/hi",
            produces = APPLICATION_JSON_VALUE)
Greeting greetWithPathVariable(@PathVariable ApiVersion version){
    return greet(version, "path-variable");
}

private Greeting greet(ApiVersion version, String how) {
    return new Greeting(how, version);
}
```

- Distinguis between /v1/hi and /v2/hi

A versioned API using customheader

```
@GetMapping(value = "/hi", produces = APPLICATION_JSON_VALUE)
Greeting greetWithHeader(@RequestHeader("X-API-Version")
                                                ApiVersion v) {
 return this.greet(v, "header");
}
```

- X-API-Version is a customheader

A versioned API using acceptHeader

```
@GetMapping(value = "/hi", produces = V1_MEDIA_TYPE_VALUE)
Greeting greetWithContentNegotiationV1() {
return this.greet(ApiVersion.v1, "content-negotiation");
}

@GetMapping(value = "/hi", produces = V2_MEDIA_TYPE_VALUE)
Greeting greetWithContentNegotiationV2() {
return this.greet(ApiVersion.v2, "content-negotiation");
}
```