SPRING AND SPRING BOOT

Module 1 : REST(ful) Endpoints

Spring and SpringBoot

Spring Inversion of Control Container

- Spring uses Spring Core Inversion of control (IoC) container
- Spring boot allows you to configure how and when dependencies are provided to your application at runtime
- Inversion of control is often called Dependency Injection (DI)

Spring Initializr

- first step of a spring boot application

API Controls and JSON

API Controls

- examples : consumer Driven contract and Provider Driven contracts
- it is a formal agreement between a software provider and a consumer that abstractly communicates how to interact with each other

Eg :

Request
URL : / customers / {id}
HTTP Verb : GET
Body : None

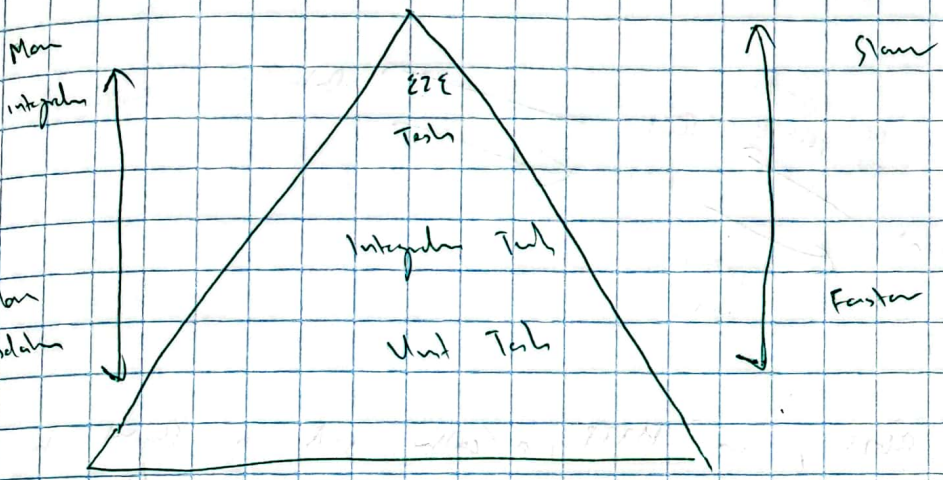
Response
HTTP status :
200 : OK if the user is authorized and can be successfully retrieved
401 : Unauthorized if the user is unauthenticated or unauthorized
404 : Not found if user is authorized but the data could not be found
Response Body Type : JSON
Example Response Body :

```
{
    "id" : 99
    "amount" : 123.45
}
```

Why are API contracts important?
- they communicate behaviour of a REST API

Testing First

Testing Pyramid



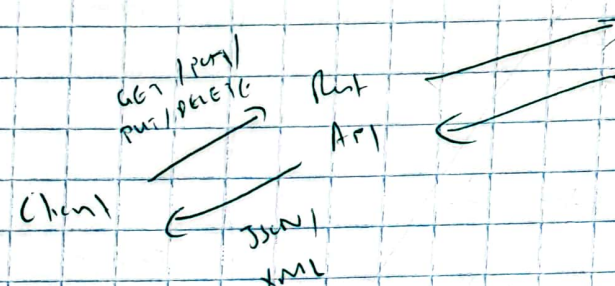
Red, green, refactor loop
- best time to refactor is red your working on during a TDD cycle
- this is called the Red, Green, Refactor development loop

1. Red: write a failing test for desired functionality
2. Green: implement the simplest thing that can work to make the test pass
3. Refactor: look for opportunities to simplify, reduce duplication, or otherwise improve the code without changing any behaviour - to refactor
4. Repeat!

Implementing GET

REST, CRUD and HTTP

- Rest : Representational State Transfer
- In a REST system data object are called Resource Representation



- In REST, an HTTP, a caller sent a Request to a URI, web receive request, route it to a request handle, handle create a response which is sent back to the caller

Component of Request and Response :

Request :
- method (verb)
- URI (endpoint)
- Body

Response :
- status code
- Body

- For CREATE : an HTTP method POST
- For READ : an HTTP method GET
- For UPDATE : an HTTP method PUT
- For DELETE : an HTTP method DELETE

RESTful (CRUD) operations :

Operation	API Endpoint	Http Method	Response Status
Create	/comments	POST	201
Read	/comments/{id}	GET	200
Update	/comments/{id}	PUT	200
Delete	/comments/{id}	DELETE	200

Curl Command Example

Read Comment with id of 123 request :

Request :
Method : GET
URL : http://comments.com/comments/123
Body : (empty)

Response :
Status Code : 200
Body :
{
"id" : 123,
"comment" : "Hi."
}

Spring Annotations and Component Scan
- Spring manages and instantiates objects → called spring beans

Spring Web Controllers
- in Spring Web, Request are handled by controllers.

@RestController
class CommentController {
}

- controllers get injected into Spring web, which routes API requests (handled by controller) to correct method

- a controller method can be a default handler method, to be called when a request that no method knows how to handle is received.

Eg: Post request handler method:

```
public ResponseEntity findById ( long requestId ) { }
```

- to tell spring to route request to this method only on GET requests. Use:

```
@ GetMapping ( " /courses / { requestId } " )
public ResponseEntity findById ( long requestId ) { }
```

- Spring to know how to get value of requestId parameter use:

```
@ GetMapping ( " / courses / { requestId } " )
public ResponseEntity findById ( @ PathVariable long requestId ) { }
```

- REST says we must a response code of 200.

```
@ RestController
class CourseResourceController {

  @ GetMapping ( " / courses / { requestId } " )
  public ~~Course~~ ResponseEntity<Course> findById (@PathVariable long requestId)
  {
    Course course = /* code to return course */ ;
    return ResponseEntity.ok ( course );
  }
}
```

Lab: Implementing GET

Understand K Test:

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
- this will start our spring boot application and make it available for our test to perform request to do it

@Autowired
TestRestTemplate restTemplate;
- asked Spring to inject a test helper that'll allow us to make HTTP request to K locally running application
- @Autowired is best used in tests

ResponseEntity<String> response = restTemplate.getForEntity("/contacts/79",
String.class);
- use restTemplate to make an HTTP GET request to our application endpoint /contacts/79

This test fails: we haven't intended Spring web how to handle GET contacts/79, so externally responds that endpoint is NOT-FOUND

Add K GET endpoint

@RestController
@RequestMapping("/contacts")
class ContactController {

@GetMapping("/{requestId}")
public ResponseEntity<String> findById() {
return ResponseEntity.ok("{}");
}

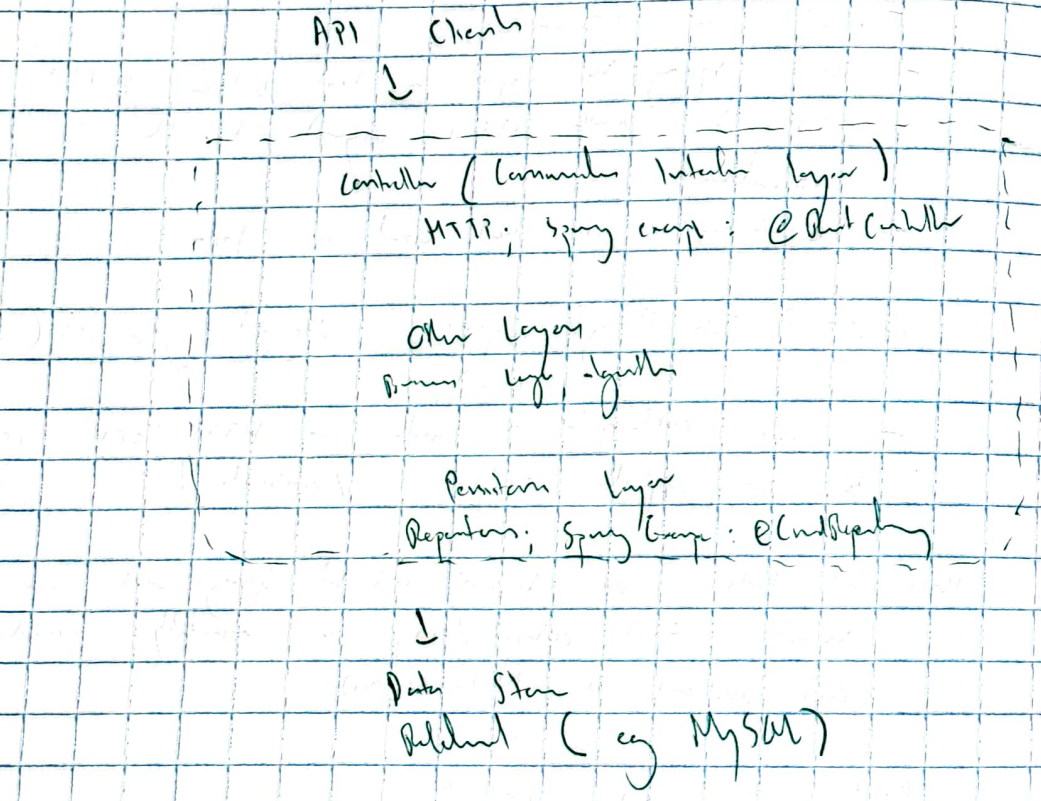
}

Repositories and Spring Data

Controller - Repository Architecture

- The repository is the interface between the application and the database and provide a common abstraction for any database

Layered Architecture :



Choosing a Database

- we'll use an embedded, in-memory database

DataBase Types:

Embedded, In-memory

REST API - Client

↓

Java Project
Controller
Repository
In-Memory DB

Persistent

REST API CLIENT

↓

Java Project
controller
Repository

↓

External
DB

#1 Auto Configuration

- because we added Spring Data depending C and a specific data provider, H2), Spring boot will automatically configure your application to communicate with H2

#2 Spring Data CrudRepository

The following is a complete implementation of all CRUD operations by extending CrudRepository :

```
interface CabinetRepository extends CrudRepository<Cabinet, Long> {}
```

Developing a Secure API

Implementing POST

Few questions we need to answer:

1. Who specifies the ID - client or server?
2. In API request, how do we represent the object to be created
3. By which HTTP method should we use in request?
4. What does API send as response?

Idempotence and HTTP

- An ~~code~~ idempotent operation is defined as one which, if performed more than once, results in the same outcome
- For each method, the HTTP standard specifies whether it is idempotent or not. GET, PUT and DELETE are idempotent whereas POST and PATCH are not

The POST request and response

The request: The POST method allows a Body, so we'll use the Body to send a JSON representation of the object:

Request:
- Method: POST
- URL: /cashcards/
- Body:

```
{
    amount: 123.45
}
```

- No ID in request as we allow the server to create the ID, thus the data contract for the POST operation is different than that of the Create operation.

The response :

- by returning 201 CREATED status, the API is specifically communicating that data was added
- the header should contain the URI of created resource

Response :

- Status Code : 201 CREATED
- Header : Location : /context/v3

Spring Web Convenience Method,

- to create above response we'll use ResponseEntity.created(uri).build()

Eg : return ResponseEntity.created(uriOfNewUser).build();

v/s

return ResponseEntity.status(HttpStatus.CREATED).header(HttpHeaders.LOCATION, uriOfNewUser.toASCIIString())
.build();

Lab : Implementing POST

Test :

```
@Test
void shouldCreateANewUser() {

    User newUser = new User(null, 25.00);
    ResponseEntity<Void> createResponse = restTemplate.postForEntity
                                ("/users", newUser, responseType: Void.class);
    assertThat(createResponse.getStatusCode()).isEqualTo(HttpStatus.CREATED);

    URI locationOfNewUser = createResponse.getHeaders().getLocation();
    ResponseEntity<String> getResponse = restTemplate.getForEntity(
                                locationOfNewUser, String.class);
    assertThat(getResponse.getStatusCode()).isEqualTo(HttpStatus.OK);

    DocumentContext documentContext = JsonPath.parse(getResponse.getBody());
    Number id = documentContext.read("$.id");
    Double amount = documentContext.read("$.amount");
```

```
assertThat(id).isNotNull();
assertThat(amount).isEqualTo(25.00);
```

Controller:

```
@PostMapping
private ResponseEntity<Void> createCashCard(@RequestBody CashCard
        newCashCardRequest, UriComponentsBuilder ucb) {

CashCard savedCashCard = cashCardRepository.save(newCashCardRequest);
URI locationOfNewCashCard = ucb
        .path("cashcards/{id}")
        .buildAndExpand(savedCashCard.id())
        .toUri();
return ResponseEntity.created(locationOfNewCashCard).build();
    }
}
```

Returning a list with GET

Requesting a list of cash cards

- CrudRepository has a findAll method that we can use to easily fetch all the cash cards in the database

```
@GetMapping()
private ResponseEntity<Iterable<CashCard>> findAll() {
        return ResponseEntity.ok(cashCardRepository.findAll());
    }
```

Consider:

- How do I return only cash cards that user owns?
- Should API return unlimited number of results or return them in "chunks" (known as pagination)
- Should cash cards be returned in a particular order?

Pagination and Sorting

- use PagingAndSortingRepository
- Pagination in Spring" to specify the page length and page index

Spring Data Pagination API

- Spring Data provides the PageRequest and Sort classes for pagination

Eg: A query to get page 2 with page size 10, sorting by amount in desc order

```
Page<CreditCard> page2 = creditCardRepository.findAll(PageRequest.of(
                1,
                10,
                Sort.by(new Sort.Order(Sort.Direction.DESC, "amount")));
```

The URI

Get second page: /creditcards?page=1
... when length is 3: /creditcards?page=1&size=3
... sorted by amount also: /creditcards?page=1&size=3&sort=amount
... in descending order: /creditcards?page=1&size=3&sort=amount,desc

The Java Code

Complete implementation of the controller method for our new "get a page of credit cards" endpoint:

```
@GetMapping
public ResponseEntity<List<CreditCard>> findAll(Pageable pageable) {
    Page<CreditCard> page = creditCardRepository.findAll(
        PageRequest.of(
            pageable.getPageNumber(),
            pageable.getPageSize(),
            pageable.getSortOr(Sort.by(Sort.Direction.DESC, "amount"))));
    return ResponseEntity.ok(page.getContent());
}
```

Lab: Refining a list with GET

@DirtiesContext
- fixes problem by telling Spring to start with a clean state, as if these other tests hadn't been run

Lab: Simple Spring Security

Add the following to build.gradle:
implementation 'org.springframework.boot:spring-boot-starter-security'

Authentication: act of a principal proving its identity to the system. An authenticated session is created when a user gets authenticated. A common mechanism: a session token is generated and placed in a cookie

Authorization: Spring Security implements authorization in the filter chain

Rounding out CRUD

How PUT and POST think about it:
① If you need the server to return the URI of the created resource, use POST

- POST creates a sub-resource (child resource) under (after) or within the request URI
- PUT creates or replaces a resource at a specific request URI

Summary:

HTTP	Operation	Definition of Resource URI	What does it do?	Response Status Code	Response Body
POST	CREATE	Server generates and returns URI	Creates a sub-resource	201 Created	Resource created

PUT	CREATE	Client supplies the URI	Creates a resource at request URI	201 Created	resource created
PUT	update	Client supplies URI	Replaces the resource	204 NO CONTENT	(empty)
PATCH	update	Client supplies URI	Partial update: modify field. included in request or existing record	200 OK	updated Resource

Complete Cat :