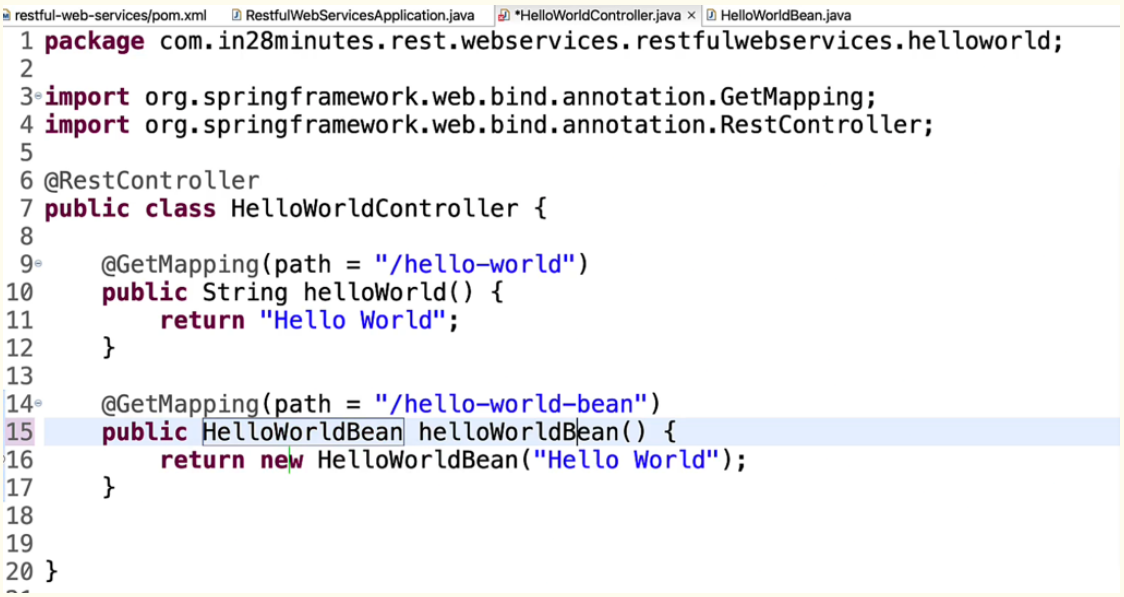
SPRING REST

1) Let's start with new project.

start.spring.io.

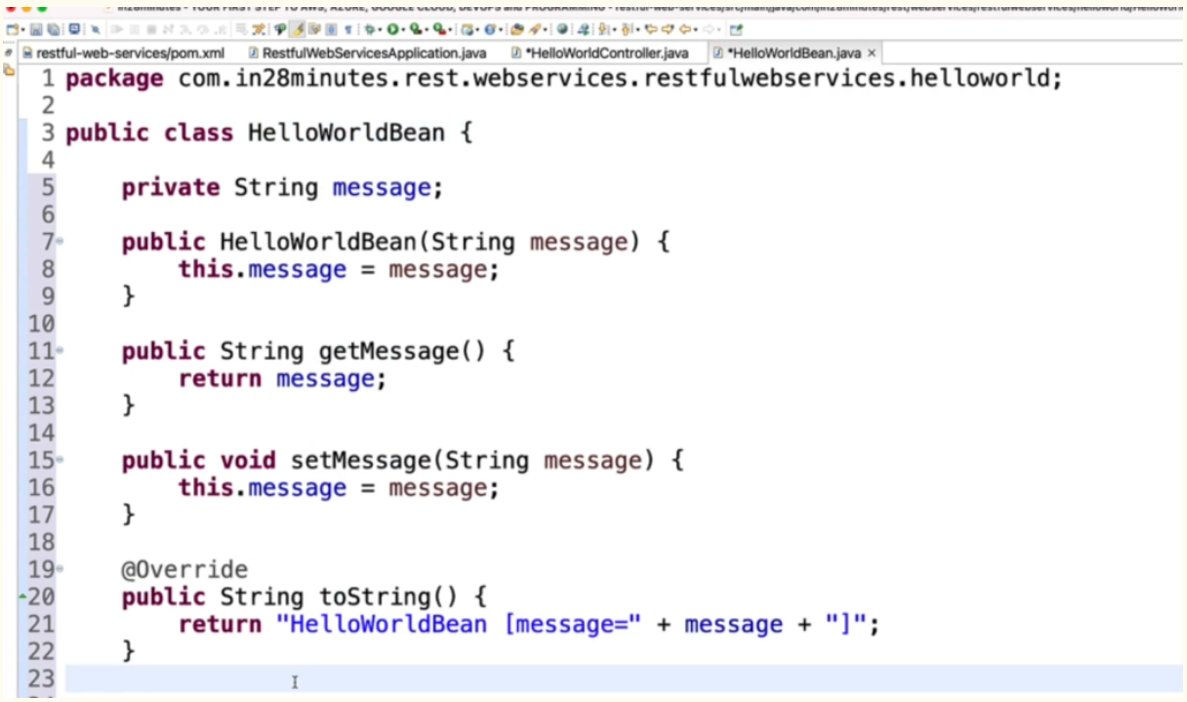
- Spring Web
- JPA
- H2
- devtools

2) New Class - HelloWorldController.java




```java
package com.in28minutes.rest.webservices.restfulwebservices.helloworld;

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

@RestController
public class HelloWorldController {

    @GetMapping(path = "/hello-world")
    public String helloWorld() {
        return "Hello World";
    }

    @GetMapping(path = "/hello-world-bean")
    public HelloWorldBean helloWorldBean() {
        return new HelloWorldBean("Hello World");
    }


}
```

Also, HelloWorldBean class,



```java
package com.in28minutes.rest.webservices.restfulwebservices.helloworld;

public class HelloWorldBean {

    private String message;

    public HelloWorldBean(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    @Override
    public String toString() {
        return "HelloWorldBean [message=" + message + "]";
    }
```

→ DYNAMIC URL in HelloWorldController.

```java
// Path Parameters
// /users/{id}/todos/{id}  => /users/2/todos/200
// /hello-world/path-variable/{name}
// /hello-world/path-variable/Ranga

@GetMapping(path = "/hello-world/path-variable/{name}")
public HelloWorldBean helloWorldPathVariable(@PathVariable String name) {
    return new HelloWorldBean(String.format("Hello World, %s", name));
}
```

→ Let's start with social media app, to start we make a class, let's name it User.java, in a package user.

```java
public class User {

    private Integer id;
    private String name;
    private LocalDate birthDate;
```

constructor, getter, setter, toString.

This User will be used by RESTAPI.

In order to play with my database, so as to create a user, delete a user, we create a DAO object. (Data Access object)

→ Make a new class UserDaoService.

Now, in this class let's make all of our CRUD functions.

To talk the DB, we use JPA/Hibernate, but for simplicity let's take

static ArrayList for now.

```java
@Component
public class UserDaoService {
    // JPA/Hibernate > Database
    // UserDaoService > Static List

    private static List<User> users = new ArrayList<>();

    static {
        users.add(new User(1,"Adam",LocalDate.now().minusYears(30)));
        users.add(new User(2,"Eve",LocalDate.now().minusYears(25)));
        users.add(new User(3,"Jim",LocalDate.now().minusYears(20)));
    }

    public List<User> findAll() {
        return users;
    }
```

→ Let's implement a GET request

GET /users

Make a new class UserResource

```java
@RestController
public class UserResource {

    private UserDaoService service;

    public UserResource(UserDaoService service) {
        this.service = service;
    }

    // GET /users
    @GetMapping("/users")
    public List<User> retrieveAllUsers() {
        return service.findAll();
    }
```

Test it in browser.

→ Go back to UserDaoService and implement →

```
//public User save(User user) {

public User findOne(int id) {
    Predicate<? super User> predicate = user -> user.getId().equals(id);
    return users.stream().filter(predicate).findFirst().get();
}
```

→ Now in UserResource,

```
// GET /users
@GetMapping("/users/{id}")
public User retrieveUser(@PathVariable int id) {
    return service.findOne(id);
}
```

→ Post request to add users

for POST, I need id, so we need to autoincrement id first.

Let's update our UserDaoService accordingly.

Let's have a variable for count.

```
private static List<User> users = new ArrayList<>();

private static int usersCount = 0;

static {
    users.add(new User(++usersCount,"Adam",LocalDate.now().minusYears(30)));
    users.add(new User(++usersCount,"Eve",LocalDate.now().minusYears(25)));
    users.add(new User(++usersCount,"Jim",LocalDate.now().minusYears(20)));
}
```

Save function to save the new user.

```
public User save(User user) {
    user.setId(++usersCount);
    users.add(user);
    return user;
}
```

All of this is temporary, later we will actually store everything in DB.

→ Le user Resource.

```
//POST /users
@PostMapping("/users")
public void createUser(@RequestBody User user) {
    service.save(user);

}
```

Use Postman & test the API.

Let's also return a status as 201, which shows user added correctly.

- **Important Response Statuses**
  - **200** — Success
  - **201** — Created
  - **204** — No Content
  - **401** — Unauthorized (when authorization fails)
  - **400** — Bad Request (such as validation error)
  - **404** — Resource Not Found
  - **500** — Server Error

Let's add this on POST method.

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@RequestBody User u
    service.save(user);

    return ResponseEntity.created(null).build();
}
```

↳ this will return 201

Also, let's return back the URI.

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@RequestBody User user) {
    User savedUser = service.save(user);
    // /users/4 => /users ,   user.getID
    URI location = ServletUriComponentsBuilder.fromCurrentRequest()
                    .path("/{id}")
                    .buildAndExpand(savedUser.getId())
                    .toUri();
    return ResponseEntity.created(location).build();
}
```

Now let's create an 404 page.

In UserDaoService, change findOne function, to return null if user not found.

```
public User findOne(int id) {
    Predicate<? super User> predicate = user -> user.getId().equals(id);
    return users.stream().filter(predicate).findFirst().orElse(null);
}
```

Response still 200, we need 404.

In UserResource.java, throw an exception,

```
@GetMapping("/users/{id}")
public User retrieveUser(@PathVariable int id) {
    User user = service.findOne(id);

    if(user==null)
        throw new UserNotFoundException("id:"+id);

    return user;
}
```

Make this Exception class and extend RuntimeException.

```java
@ResponseStatus(code = HttpStatus.NOT_FOUND)
public class UserNotFoundException extends RuntimeException {

    public UserNotFoundException(String message) {
        super(message);
    }

}
```

Try sending request to /users/1012
will receive 404 not found.

→ Now, let's create custom structure for an 404 request

A new class ErrorDetails in a package Exception.

```java
public class ErrorDetails {

    private LocalDate timestamp;
    private String message;
    private String details;
```

Getters only , constructor, toString();

Now, we need to generate our own Exception Handling class,

Customized Response Entity

UserResource.java | UserDaoService.java | UserNotFoundException.java | restful-web-services/pom.xml | ErrorDetails.java | *CustomizedResponseEntityExceptionHandler.jav

```
12 import com.in28minutes.rest.webservices.restfulwebservices.user.UserNotFoundException;
13
14
15 @ControllerAdvice
16 public class CustomizedResponseEntityExceptionHandler extends ResponseEntityExceptionHanc
17
18     @ExceptionHandler(Exception.class)
19     public final ResponseEntity<Object> handleAllExceptions(Exception ex, WebRequest requ
20         ErrorDetails errorDetails = new ErrorDetails(LocalDateTime.now(),
21                 ex.getMessage(), request.getDescription(false));
22
23         return new ResponseEntity(errorDetails, HttpStatus.INTERNAL_SERVER_ERROR);
24
25     }
26
27     @ExceptionHandler(UserNotFoundException.class)
28     public final ResponseEntity<Object> handleUserNotFoundException(Exception ex, WebRequ
29         ErrorDetails errorDetails = new ErrorDetails(LocalDateTime.now(),
30                 ex.getMessage(), request.getDescription(false));
31
32         return new ResponseEntity(errorDetails, HttpStatus.NOT_FOUND);
33
34     }
35
```

request

Now, send a GET request to /users/412 and it will return 404, through POSTMAN.

-> we need to make Response Entity as a generic type.

So, add <ErrorDetails> in front of all ResponseEntity.

```
ResponseEntity<ErrorDetails>(errorDetails, HttpStatus.NOT_FOUND);
```

↳ on all 4 places.

Now let's add a DELETE request.

In UserDaoService,

```
public void deleteById(int id) {
    Predicate<? super User> predicate = user -> user.getId().equals(id);
    users.removeIf(predicate);
}
```

In UserResource,

```
@DeleteMapping("/users/{id}")
public void deleteUser(@PathVariable int id) {
    service.deleteById(id);
}
```

Add validations →

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

In the postmapping, add @Valid

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@Valid @RequestBody User user) {
```

Now, add validations to User bean.

```java
@Size(min=2)
private String name;

@Past
private LocalDate birthDate;

public User(Integer id, String name, LocalDate birthDate) {
    super();
    this.id = id;
    this.name = name;
    this.birthDate = birthDate;
}
```

Send a request, let's customize this error handling to get more information. We need to override a method present in ResponseEntity.

```java
@Override
protected ResponseEntity<Object> handleMethodArgumentNotValid(
        MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatusCode statu

    ErrorDetails errorDetails = new ErrorDetails(LocalDateTime.now(),
            ex.getMessage(), request.getDescription(false));


    return new ResponseEntity(errorDetails, HttpStatus.BAD_REQUEST);
}
```

→ Send a wrong create user request, which will return a long error.

→ Let's generate a custom error.
In the method, add —

```
ErrorDetails errorDetails = new ErrorDetails(LocalDateTime.now(),
        "Total Errors:" + ex.getErrorCount() + " First Error:" + ex.getFieldError
```

()

Now our API CRUD is complete with exception handling & validation.

→ Let's make a documentation for the same using open API & swagger UI.

For that, we need to add a dependency -

```
<dependency>
        <groupId>org.springdoc</groupId>
        <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
        <version>2.0.0</version>
</dependency>
```

Maven re-build & open /swagger-ui/index.html

Then open api-docs.

→ Content Negotiation

- xml / json
- English / French / Spanish.

for this, we need to add a dependency.

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

After rebuild →
Now send a get request to /users with a header → Accept : application/xml

-> Moving ahead these things can cause problem with a little advance stuff, so for now lets comment out these 2 dependencies.

```
<!--
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.0.0-M4</version>
</dependency>

<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>

-->
```

#11 Internationalisation - i18n

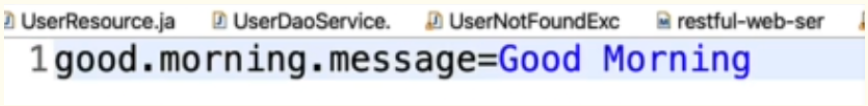
Go to Hello world Controller,

```
@GetMapping(path = "/hello-world-internationalized")
public String helloWorldInternationalized() {
    return "Hello World V2";
```

This is where we will apply i18n.

For this, let's make a new file in resources and name it

messages.properties -



} for English, later we can define for others, first lets start with English.

In HelloWorldController,

```
@RestController
public class HelloWorldController {

    private MessageSource messageSource;

    public HelloWorldController(MessageSource messageSource) {
        this.messageSource = messageSource;
    }
```

} Constructor injection.

Now, in the GetMapping

```
@GetMapping(path = "/hello-world-internationalized")
public String helloWorldInternationalized() {
    messageSource.get

    return "Hello

    //1:good.morni
```

```
getClass() : Class<?> - Object
getMessage(MessageSourceResolvable resolvable, Locale locale) : Strin
getMessage(String code, Object[] args, Locale locale) : String - Message
getMessage(String code, Object[] args, String defaultMessage, Locale lo
```

```
Try to resolve the message
message was found.
Parameters:
    code the message co
    'calculator.noRateSet'
    encouraged to base n
    package names, avoir
```

Choose the 3rd one.

```
@GetMapping(path = "/hello-world-internationalized")
public String helloWorldInternationalized() {

    Locale locale = LocaleContextHolder.getLocale();
    messageSource.getMessage("good.morning.message", null, "Default Message", locale
```

```
@GetMapping(path = "/hello-world-internationalized")
public String helloWorldInternationalized() {
    Locale locale = LocaleContextHolder.getLocale();
    return messageSource.getMessage("good.morning.message", null, "Default Messa
```

Now, lets add same for dutch -> in a new file, messages_nl.properties

```
UserResource.ja   restful-web-ser   HelloWorldContr   messages.proper
1 good.morning.message=Goedemoergen
```

In postman, we change header Accept-language : nl, it should return dutch.

Do same for french 'fr' & Deutsch 'de'.

VERSIONING

↳ v1.0.0 v1.0.1

v2.0.0

Variety of options for versioning

- URL
- Request parameter
- Header
- Media Type

Create a new class VersioningPersonController in a package versioning

```java
@RestController
public class VersioningPersonController {

    @GetMapping("/v1/person")
    public Person getFirstVersionOfPerson() {
        return new PersonV1("Bob Charlie");
    }

}
```

PersonV1

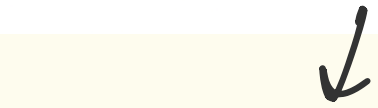
Create class 'PersonV1'
Create record 'PersonV1'
Fix project setup...

Create PersonV1 class in versioning1

```java
public class PersonV1 {
    private String name;

    public PersonV1(String name) {
        super();
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "PersonV1 [name=" + name + "]";
    }
```

Send a request. Now we want a different version of this,

```java
@GetMapping("/v2/person")
public PersonV2 getSecondVersionOfPerson() {
    return new PersonV2("Bob Charlie");
}
```



```java
2
3 public class PersonV2 {
4     private Name name;
5
6 }
```

→ generate getter, constructor & toString.

And then the name class

```java
public class Name {
    private String firstName;
    private String lastName;

    public Name(String firstName, String lastName) {
        super();
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    @Override
    public String toString() {
        return "Name [firstName=" + firstName + ", lastName=" + lastName + "]";
    }
```

Now change the getmapping method

```java
@GetMapping("/v2/person")
public PersonV2 getSecondVersionOfPerson() {
    return new PersonV2(new Name("Bob", "Charlie"));
}
```

Testing /v2/person.

This was URI versioning
Let's talk about request
parameter versioning.

Eg → localhost:8080/person?version=1
localhost:8080/person?version=2.

```
@GetMapping(path = "/person", params = "version=1")
public PersonV1 getFirstVersionOfPersonRequestParameter() {
    return new PersonV1("Bob Charlie");
}

@GetMapping(path = "/person", params = "version=2")
public PersonV2 getSecondVersionOfPersonRequestParameter() {
    return new PersonV2(new Name("Bob", "Charlie"));
}
```

↳ localhost:8080/person?version=2

Now lets see how to do this
using custom headers.

Next one is custom headers versioning.

```
@GetMapping(path = "/person/header", headers = "X-API-VERSION=1")
public PersonV1 getFirstVersionOfPersonRequestHeader() {
    return new PersonV1("Bob Charlie");
}
```

To test this, Let's move to postman

send request with a header

key → X-API-version

value → 1.

for v2.

```
@GetMapping(path = "/person/header", headers = "X-API-VERSION=2")
public PersonV2 getSecondVersionOfPersonRequestHeader() {
    return new PersonV2(new Name("Bob", "Charlie"));
}
```

Again test with Postman.

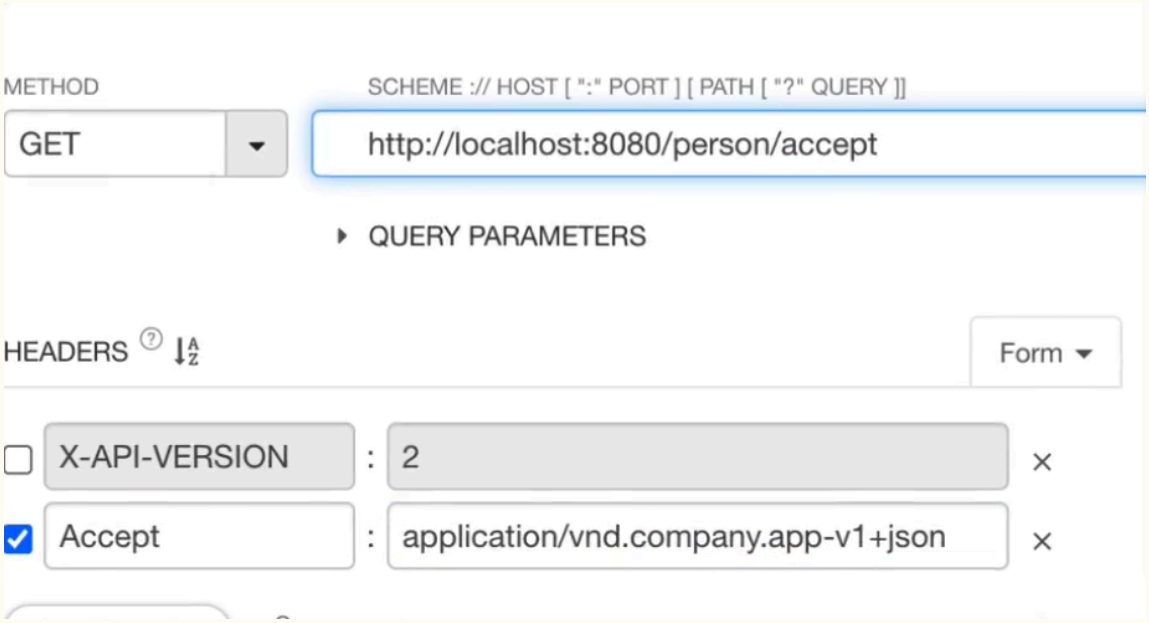
→ Last type of versioning is

Media type versioning a.k.a

accept header.

```java
@GetMapping(path = "/person/accept", produces = "application/vnd.company.app-v1+json")
public PersonV1 getFirstVersionOfPersonAcceptHeader() {
    return new PersonV1("Bob Charlie");
}
```

To test this, in postman



for V2 →

```java
@GetMapping(path = "/person/accept", produces = "application/vnd.company.app-v2+json"
public PersonV2 getSecondVersionOfPersonAcceptHeader() {
    return new PersonV2(new Name("Bob", "Charlie"));
}
```

Test in postman.

Companies Example using versioning.

URI Versioning - Twitter
- *http://localhost:8080/v1/person*
- *http://localhost:8080/v2/person*

Request Parameter versioning - Amazon
- *http://localhost:8080/person?version=1*
- *http://localhost:8080/person?version=2*

(Custom) headers versioning - Microsoft
- SAME-URL headers=[X-API-VERSION=1]
- SAME-URL headers=[X-API-VERSION=2]

Media type versioning - GitHub
- SAME-URL produces=application/vnd.company.app-v1+json
- SAME-URL produces=application/vnd.company.app-v2+json

HATEOAS

→ Hypermedia as the Engine of Application State

→ Implemented using HAL → JSON Hypertext Application Language.

- Let's implement it in our project.

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

Now in our link ${users/${id}}, along with our response we send a link to all users. i.e. /users.

Now change in UserResource file, modify the retrieve User method.

```
@GetMapping("/users/{id}")
public EntityModel<User> retrieveUser(@PathVariable int id) {
    User user = service.findOne(id);

    if(user==null)
        throw new UserNotFoundException("id:"+id);

    EntityModel<User> entityModel = EntityModel.of(user);

    WebMvcLinkBuilder link =  linkTo(methodOn(this.getClass()).retrieveAllUsers());
    entityModel.add(link.withRel("all-users"));

    return entityModel;
}
```

This is data.

This is the data structure.

STATIC FILTERING

- Serialization
  - is converting Java to JSON/XML

Let's start with customizing the REST API response.

→ @JsonProperty

Go to User.java

```
@Size(min=2, message = "Name shou
@JsonProperty("user_name")
private String name;

@Past(message = "Birth Date should
@JsonProperty("birth_date")
private LocalDate birthDate;
```

- Filtering → Returning only selected fields.

Two kinds of filtering -

- Static filtering
- Dynamic filtering.

Customize filtering for a bean for specific REST API.

Same filtering for a bean across different REST API

→ Create a new package filtering.
Inside this, create FilteringController.java

```
@RestController
public class FilteringController {

    public SomeBean filtering() {
        return new SomeBean("value1","value2", "value")
    }

}
```

@GetMapping("/filtering")

Make a POJO for Some Bean.

```
public class SomeBean {
    private String field1;
    private String field2;
    private String field3;
    public SomeBean(String f
```

So, the url returns everything for now, assume field2 is password & we don't want your client to return this field.

```
@JsonIgnore
private String field2;
```

and now if you reload, the field will be missing from the o/p.

Same is true if there are more than one bean. So for that, let's return a list of Some bean.

```java
@GetMapping("/filtering-list")
public List<SomeBean> filteringList() {
    return Arrays.asList(new SomeBean("value1","value2", "value3"),
            new SomeBean("value4","value5", "value6"));
}
```

now load this url & you will see that list is returned w/o field-2.

JPA / Hibernate

→ Start a new spring start project.
(Spring web, JPA, H2, JDBC)

→ In the console logs, there is a url for H2 DB. like

```
- Starting...
- Added connection conn0: url=jdbc:h2:mem:eae25c24-5ec2-439e-bc2c-61bae6a81a6b user=SA
```

→ To enable H2 DB, in application.prop.

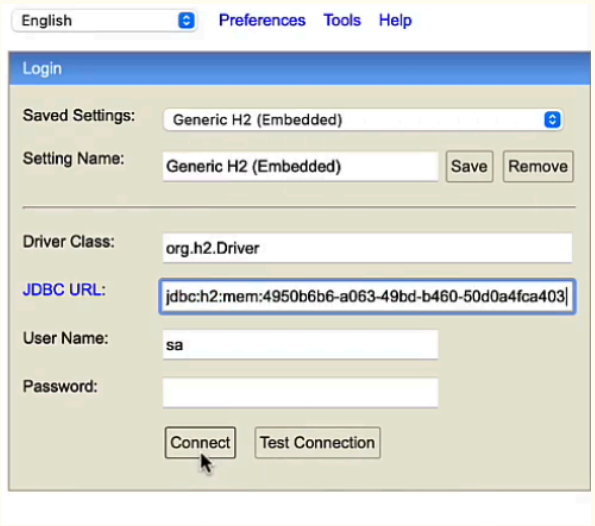
arnJpaAndHibernateApplication.java | application.properties ×

```
spring.h2.console.enabled=true
```

Restart the project.

→ Now go to localhost:8080/h2-console.

→ Now in JDBC url field, copy the url from console.



But this is a dynamic url and we will have to change it every time we start the server.

=> Creating a static url. See application.properties

```
1 spring.h2.console.enabled=true
2 spring.datasource.url=jdbc:h2:mem:testdb
```

Restart the app.

Paste the new url in /h2-console.

<- Now in resources folder, make another file schema.sql.

As, let's create the table.

```
1 create table course
2 (
3     id bigint not null,
4     name varchar(255) not null,
5     author varchar(255) not null,
6     primary key (id)
7
8 );
```

Refresh everything, a new table will be created.

→ Let's create an ORM to interact with the DB.

Create a class CourseJDBCRepository in a package called cource.jdbc.

↳ package inside package.

```
@Repository
public class CourseJdbcRepository {
    @autowire
    private JdbcTemplate springJdbcTemplate;

    private static String INSERT_QUERY =

            """
                insert into course (id, name, author)
                values(1, 'Learn AWS','in28minutes');

            """;

    public void insert() {
        springJdbcTemplate.update(INSERT_QUERY);
    }
}
```

→ DI.

we need to run this query at the start of the project.

That is why we have Command Line Runner.

helps in running at start.

```
@Component
public class CourseJdbcCommandLineRunner implements CommandLineRunner{

    @Autowired
    private CourseJdbcRepository repository;

    @Override
    public void run(String... args) throws Exception {
        repository.insert();

    }

}
```

Now, let's make a course class

```
private long id;
private String name;
private String author;
```

Implement all the methods - and a no arg constructor.

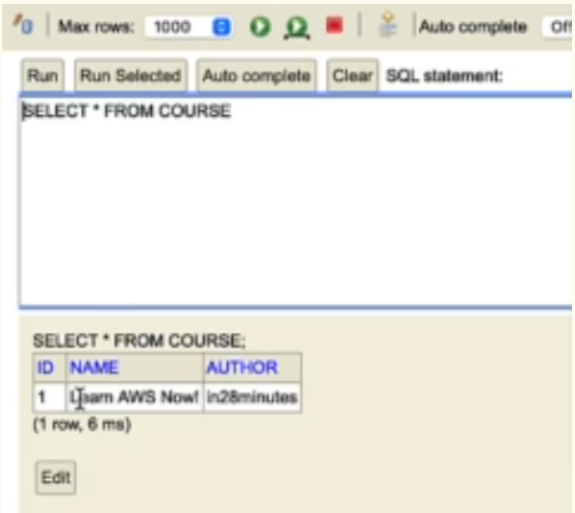
Now, we need to execute this data from our query, for that let's modify our jdbc template.

```
private static String INSERT_QUERY =

        """
            insert into course (id, name, author)
            values(?, ?,?);

        """;

public void insert(Course course) {
    springJdbcTemplate.update(INSERT_QUERY,
            course.getId(), course.getName(), course.getAuthor());
}
```

Now, in the command line runner, we need to modify ->

```
@Override
public void run(String... args) throws Exception {
    repository.insert(new Course(1, "Learn AWS Now!", "in28minutes"));

}
```

Connect to the /h2-console again, and check if the db got created.



Now, we have done one, let's add more of these.

```
@Override
public void run(String... args) throws Exception {
    repository.insert(new Course(1, "Learn AWS Now!", "in28minutes"));
    repository.insert(new Course(2, "Learn Azure Now!", "in28minutes"));
    repository.insert(new Course(3, "Learn DevOps Now!", "in28minutes"));

}
```

Restart the server & changes will be there in the H2-console.

Similarly, let's add a delete method.

```
private static String DELETE_QUERY =

        """
            delete from course
            where id = ?

        """;


public void insert(Course course) {
    springJdbcTemplate.update(INSERT_QUERY,
            course.getId(), course.getName(), course.getAuthor());
}

public void deleteById(long id) {
    springJdbcTemplate.update(INSERT_QUERY,id);
}
```

DELETE_QUERY

Add in command line runner →

```
repository.insert(new Course(
repository.deleteById(1);
```

Let's do a read as well,

```
private static String SELECT_QUERY =
        """
            select * from course
            where id = ?
        """;
```

```
public Course findById(long id) {
    //ResultSet -> Bean => Row Mapper =>
    return springJdbcTemplate.queryForObject(SELECT_QUERY,
            new BeanPropertyRowMapper<>(Course.class), id);
}
```

we need to return a course and map it to our Course Bean, by creating a new Row mapper using Reflection

↳ This takes exact names of variables and match it directly to the rows of our table.

Now, in our command line runner, let's print the courses.

```
System.out.println(repository.findById(2));
System.out.println(repository.findById(3));
```

With more data and operations, writing queries can get really really difficult, hence we use ORM for that purpose.

↓

Let's map our Course Bean directly to our DB table, using JPA.

```
@Entity
public class Course {

    @Id
    private long id;

    @Column(name="name")
    private String name;

    @Column(name="author")
    private String author;
}
```

→ @Entity

→ we can give name, if the db name & class name are different.

→ @Column

→ not needed as the name matches.

Now let's create a repository. For that, create a package first named jpa inside course. Then create a class named CourseJpaRepository

@Transactional

```
@Repository
public class CourseJpaRepository {

    @PersistenceContext
    private EntityManager entityManager;

    public void insert(Course course) {
        entityManager.merge(course);
    }
```

EntityManager uses Entity and inbuilt methods to interact.

Instead of autowiring, we can use PersistenceContext from JPA.

Also, let's implement findById & delete

```
public Course findById(long id) {
    return entityManager.find(Course.class, id);
}

public void deleteById(long id) {
    Course course = entityManager.find(Course.class, id);
    entityManager.remove(course);
}
```

In command line runner, lets autowire JPA repository.

```
@Autowired
private CourseJpaRepository repository;
```

To show all the queries being executed by JPA, in application properties.

```
spring.datasource.url=jdbc:
spring.jpa.show-sql=true
```

Now lets make use of Spring Data JPA and further reduce our overhead.

JDBC to Spring JDBC to JPA to Spring Data JPA

- **JDBC**
  - Write a lot of SQL queries! (*delete from todo where id=?*)
  - And write a lot of Java code
- **Spring JDBC**
  - Write a lot of SQL queries (*delete from todo where id=?*)
  - BUT lesser Java code
- **JPA**
  - Do NOT worry about queries
  - Just Map Entities to Tables!
- **Spring Data JPA**
  - Let's make JPA even more simple!
  - I will take care of everything!

Let's create another package Spring Data JPA and create a new class - Repository.

I

interface.

```
public interface CourseSpringDataJpaRepository extends JpaRepository<Course, Long>{

}
```

Also, we need to update the command line runner.

```
repository.save(new Course(1, "Learn AWS Jpa!"
repository.save(new Course(2, "Learn Azure Jpa
repository.save(new Course(3, "Learn DevOps Jp

repository.deleteById(1l);

System.out.println(repository.findById(2l));
System.out.println(repository.findById(3l));
```

Now, let's also find all the courses.

```
System.out.println(repository.findAll());
```

To calc:, how many entities are present, we can do,

```
System.out.println(repository.count());
```

Custom methods → We can also define custom methods.

Naming convention is findBy ——

- In the interface,

```
List<Course> findByAuthor(String author);
```

In cmdRunner →

```
System.out.println(repository.findByAuthor("in28minutes"));
System.out.println(repository.findByAuthor(""));
```

Also, let's implement findByName,

```
List<Course> findByName(String name);
```

In cmdRunner,

```
System.out.println(repository.findByName("Learn AWS Jpa!"));
System.out.println(repository.findByName("Learn DevOps Jpa!"));
```

Spring Data + REST

Now let's upgrade our UserResource.

First, let's transform our User class into an Entity.

* Change User class →

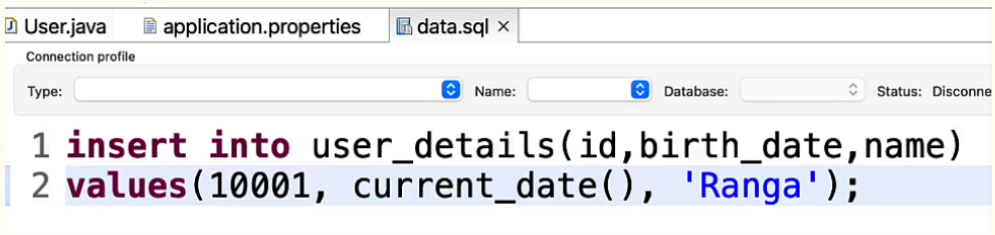
```
@Entity(name = "user_details")
public class User {

    @Id
    @GeneratedValue
    private Integer id;
```

Also, in app properties →

```
spring.datasource.url=jdbc:h2:mem:testdb
```

Now test if h2-console is working.

In resources, create a file data.sql.



Before testing, add this to application.properties.

```
spring.jpa.defer-datasource-initialization=true
```

Now, add more data -)

```
insert into user_details(id,birth_date,name)
values(10001, current_date(), 'Ranga');

insert into user_details(id,birth_date,name)
values(10002, current_date(), 'Ravi');

insert into user_details(id,birth_date,name)
values(10003, current_date(), 'Sathish');
```

Make a new package JPA & create following
Now create an interface for Data JPA -)

```
public interface UserRepository extends JpaRepository<User, Integer> {

}
```

Make a copy of RestController class UserResource, name it to UserJPAResource. Change format of all the mapping urls, add "/jpa"

Now autowire the repository.

```
private UserRepository repository;
```

Using Constructor injection, let's add into an constructor.

```
public UserJpaResource(UserDaoService service, UserRepository repository) {
    this.service = service;
    this.repository = repository;
}
```

Change the findAll,

```
@GetMapping("/jpa/users")
public List<User> retrieveAllUsers() {
    return repository.findAll();
}
```

Also, we need a constructor in User.java.

```
protected User() {

}
```

Now, test the url for read. After that, let's create other methods.

```
@GetMapping("/jpa/users/{id}")
public EntityModel<User> retrieveUser(@PathVariable int id) {
    Optional<User> user = repository.findById(id);

    if(user.isEmpty())
        throw new UserNotFoundException("id:"+id);

    EntityModel<User> entityModel = EntityModel.of(user.get());

    WebMvcLinkBuilder link =  linkTo(methodOn(this.getClass()).retrieve
    entityModel.add(link.withRel("all-users"));

    return entityModel;
}
```

```java
@DeleteMapping("/jpa/users/{id}")
public void deleteUser(@PathVariable int id) {
    repository.deleteById(id);
}
```

```java
@PostMapping("/jpa/users")
public ResponseEntity<User> createUser(@Valid @RequestBody User user) {

    User savedUser = repository.save(user);

    URI location = ServletUriComponentsBuilder.fromCurrentRequest()
                    .path("/{id}")
                    .buildAndExpand(savedUser.getId())
                    .toUri();

    return ResponseEntity.created(location).build();
}
```

Test all of them.

To clean up, now we can remove UserDaoService from our code.

#11 Now for our Social Media App, let's create Post Service as well.

- Inside user package, create a new class Post.

```java
@Entity
public class Post {

    @Id
    @GeneratedValue
    private Integer id;

    private String description;
```

Also have getters, setters, to String.

Now check if the Post is created in H2 console.

In the User Entity →

```java
@OneToMany(mappedBy = "user")
private List<Post> posts;
```

add @JsonIgnore

This is the field in Post that owns the relationship.

And, in Post Entity →

```java
@ManyToOne
private User user;
```

Add @JsonIgnore.

Let's also add Lazy, as when we get our post entity, we do not want to fetch user.

```
@ManyToOne(fetch = FetchType.LAZY)
@JsonIgnore
private User user;
```

In application.properties

```
spring.jpa.show-sql=true
```

Now restart and we can see all the hibernate on console.

In data.sql →

```
insert into post(id,description,user_id)
values(20001,"I want to learn AWS", 10001);

insert into post(id,description,user_id)
values(20002,"I want to learn DevOps", 10001);

insert into post(id,description,user_id)
values(20003,"I want to Get AWS Certified", 10002);

insert into post(id,description,user_id)
values(20004,"I want to learn Multi Cloud", 10002);
```

Move to User.java and make getters and setters for posts.

Now, in UserJPAResource, make a GET for posts.

```
@GetMapping("/jpa/users/{id}/posts")
public List<Post> retrievePostsForUser(@PathVariable int id) {
    Optional<User> user = repository.findById(id);

    if(user.isEmpty())
        throw new UserNotFoundException("id:"+id);

    return user.get().getPosts();

}
```

Now test it.

Now inside JPA package, copy UserRepository & rename to PostRepository.

```
import com.in28minutes.rest.webservices.restfulwebservices.user.Post;

public interface PostRepository extends JpaRepository<Post, Integer> {

}
```

Now let's add this in the RestController.

```
private PostRepository postRepository;

public UserJpaResource(UserRepository repository, PostRepository postRepository) {
    this.repository = repository;
    this.postRepository = postRepository;
}
```

Add the post method to create a post →

```
@GetMapping("/jpa/users/{id}/posts")
public List<Post> createPostForUser(@PathVariable int id, @Valid @RequestBody Post pos)
    Optional<User> user = repository.findById(id);

    if(user.isEmpty())
        throw new UserNotFoundException("id:"+id);

    return user.get().getPosts();

}
```

For validations, let's move to post & define →

```
@Size(min = 10)
private String description;
```

Move to posts.java and create getter and setter for user.

Change the method to as follows →

```
@PostMapping("/jpa/users/{id}/posts")
public ResponseEntity<Object> createPostForUser(@PathVariable int id, @Va
    Optional<User> user = repository.findById(id);

    if(user.isEmpty())
        throw new UserNotFoundException("id:"+id);

    post.setUser(user.get());

    Post savedPost = postRepository.save(post);

    URI location = ServletUriComponentsBuilder.fromCurrentRequest()
            .path("/{id}")
            .buildAndExpand(savedPost.getId())
            .toUri();

    return ResponseEntity.created(location).build();

}
```

Test in postman with a description as a body parameter.

```
docker run --detach --env MYSQL_ROOT_PASSWORD=dummypassword --env
MYSQL_USER=social-media-user --env MYSQL_PASSWORD=dummypassword --env
MYSQL_DATABASE=social-media-database --name mysql --publish 3306:3306 mysql:8-
oracle
```

[
Launch docker from terminal.

In app-properties -)

Comment out datasource url and add

following:

```
spring.datasource.url=jdbc:mysql://localhost:3306/social-media-database
spring.datasource.username=social-media-user
spring.datasource.password=dummypassword

spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

to create tables automatically on launch.

```
<!--<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency> -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

use mysql-connector-j

Now check in SQL workbench if you got the new db.

Let's setup some basic security →

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

In the logs we can see a password

and on localhost:8080 we can see

it asks for pswrd.

Test with username as user & pswrd.

→ Let's configure our own password.

in application.properties

```
spring.security.user.name=username
spring.security.user.password=password
```

Also, for post request we can add

auth header

Now let's have a custom filter chain

so as to setup our own authentication

system.

Create a new package security,

then a new class Spring Security Configuration

```java
@Configuration
public class SpringSecurityConfiguration {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

//      1) All requests should be authenticated
        http.authorizeHttpRequests(
                auth -> auth.anyRequest().authenticated()
                );
//      2) If a request is not authenticated, a web page is shown
        http.httpBasic(withDefaults());

//      3) CSRF -> POST, PUT
        http.csrf().disable()


        return http.build();
    }
```