El futuro digital es de todos

MinTIC

Ciclo 4a:

Desarrollo de aplicaciones web

UNIVERSIDAD NACIONAL DE COLOMBIA

MisiónTIC2022

VERSIÓN 1.0

Unidad de educación continua y permanente
Facultad de Ingeniería

Unidad Camilo Torres
Calle 44 # 45-67
Bloque B5 piso 1

(57) + 316 5000
uec_ibog@unal.edu.co

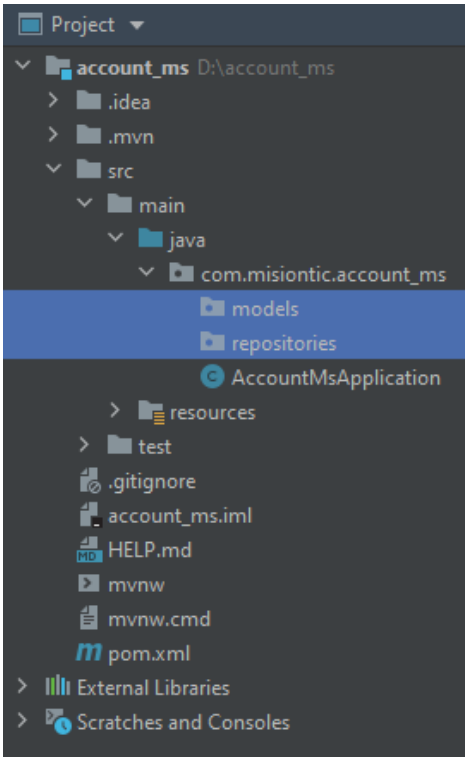
Actividad Práctica

(Microservicio AccountMS - Parte 2)

Una vez inicializado el proyecto de Spring Boot, se deben desarrollar las funcionalidades del microservicio *account_ms*, el cual expondrá una *API REST* para el manejo de las transacciones y de las cuentas bancarias del sistema de software planteado. De esta forma, el objetivo de esta guía es implementar las cuatro funcionalidades con las que contará el microservicio: una para crear una cuenta bancaria, otra para consultar la información de la cuenta de un usuario, otra para realizar una transacción y finalmente una para consultar todas las transacciones en las que un usuario se ha visto involucrado.

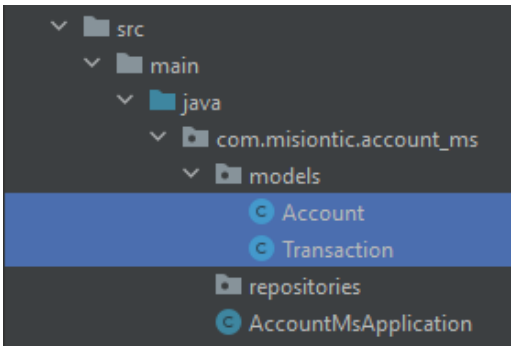
Modelos y Repositorios

En Spring Boot, cada *Modelo* es conformado por dos elementos: un modelo en el que se define la entidad asociada y sus atributos, y un repositorio, que define el mapeo de los datos almacenados en la Base de Datos. Por esta razón, en la carpeta *src/main/java/com.misiontic.account_ms* se deben crear las carpetas (o "*Packages*", según la nomenclatura de *Java*) *models* y *repositories*. Al hacerlo, la estructura del proyecto será la siguiente:



Luego, dentro de la carpeta *models* se deben crear dos clases de *Java* (archivos *.java*), uno para el modelo

Account, que representará a la cuenta bancaria de un usuario, y el otro para el modelo *Transaction*, que representará las transacciones entre cuentas bancarias. Al hacerlo, la estructura del proyecto será la siguiente:



Una vez creadas ambas clases, se debe definir cada entidad del microservicio. Para ello, en el archivo *models/Account* se tendrá el siguiente código:

```
package com.misiontic.account_ms.models;

import org.springframework.data.annotation.Id;
import java.util.Date;

public class Account {
    @Id
    private String username;
    private Integer balance;
    private Date lastChange;

    public Account(String username, Integer balance, Date lastChange) {
        this.username = username;
        this.balance = balance;
        this.lastChange = lastChange;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public Integer getBalance() {
        return balance;
    }
```

```
    public void setBalance(Integer balance) {
        this.balance = balance;
    }

    public Date getLastChange() {
        return lastChange;
    }

    public void setLastChange(Date lastChange) {
        this.lastChange = lastChange;
    }
}
```

Y en el archivo *models/Transaction* se tendrá el siguiente código:

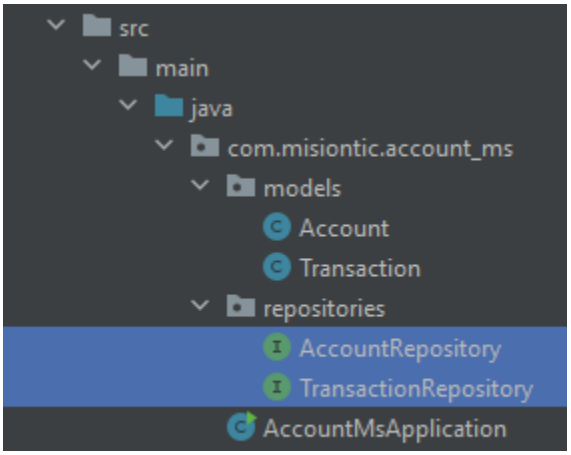
```
package com.misiontic.account_ms.models;

import org.springframework.data.annotation.Id;
import java.util.Date;

public class Transaction {
    @Id
    private String id;
    private String usernameOrigin;
    private String usernameDestiny;
    private Integer value;
    private Date date;

    public Transaction(String id, String usernameOrigin, String usernameDestiny, Integer
value, Date date) {
        this.id = id;
        this.usernameOrigin = usernameOrigin;
        this.usernameDestiny = usernameDestiny;
        this.value = value;
        this.date = date;
    }

    public String getId() {
        return id;
    }

    public String getUsernameOrigin() {
        return usernameOrigin;
    }

    public void setUsernameOrigin(String usernameOrigin) {
```

```
        this.usernameOrigin = usernameOrigin;
    }

    public String getUsernameDestiny() {
        return usernameDestiny;
    }

    public void setUsernameDestiny(String usernameDestiny) {
        this.usernameDestiny = usernameDestiny;
    }

    public Integer getValue() {
        return value;
    }

    public void setValue(Integer value) {
        this.value = value;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }
}
```

En ambos códigos se sigue un proceso similar: se crea una clase con un constructor, cuyos atributos representan las propiedades de la entidad, y cuyos métodos son los *setters* y *getters* de dichos atributos. Lo único fuera de lo común es que, en ambos casos, el atributo *username* e *id* utiliza el decorador *@Id* para indicarle a Spring Boot que dicho atributo será el identificador de la entidad.

Ahora, es necesario crear un repositorio por cada modelo, pues este no solo se encargará de indicarle a Spring Boot y a MongoDB cómo mapear cada atributo del modelo en la base de datos, sino que también permitirá realizar consultas a la base de datos. Cada repositorio se crea utilizando una interfaz, lo que significa que no es necesario definir la implementación exacta del repositorio, sino su esquema, pues Spring Boot tomará dicho esquema e implementará las funciones del repositorio.

Teniendo en cuenta lo anterior, dentro de la carpeta *repositories* se deben crear se deben crear dos interfaces de *Java* (archivos *.java*), una para el repositorio *AccountRepository*, y la otra para el modelo *TransactionRepository*. Al hacerlo, la estructura del proyecto debe ser la siguiente:



Una vez creados ambos repositorios, se debe definir cada uno de ellos. Para ello, en el archivo *repositories/AccountRepository* se tendrá el siguiente código:

```
package com.misiontic.account_ms.repositories;

import com.misiontic.account_ms.models.Account;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface AccountRepository extends MongoRepository <Account, String> { }
```

Y en el archivo *repositories/TransactionRepository* se tendrá el siguiente código:

```
package com.misiontic.account_ms.repositories;

import com.misiontic.account_ms.models.Transaction;
import org.springframework.data.mongodb.repository.MongoRepository;
import java.util.List;

public interface TransactionRepository extends MongoRepository<Transaction, String> {
    List<Transaction> findByUsernameOrigin (String usernameOrigin);
    List<Transaction> findByUsernameDestiny (String usernameDestiny);
}
```

En ambos casos, los repositorios extienden de la clase *MongoRepository*, utilizando la sentencia *<Modelo, TipoID>*, lo cual le indica a Spring Boot el modelo asociado al repositorio y el tipo de dato del *id* de dicho modelo.

Por otro lado, *TransactionRepository* contiene el encabezado de dos métodos: *findByUsernameOrigin* y

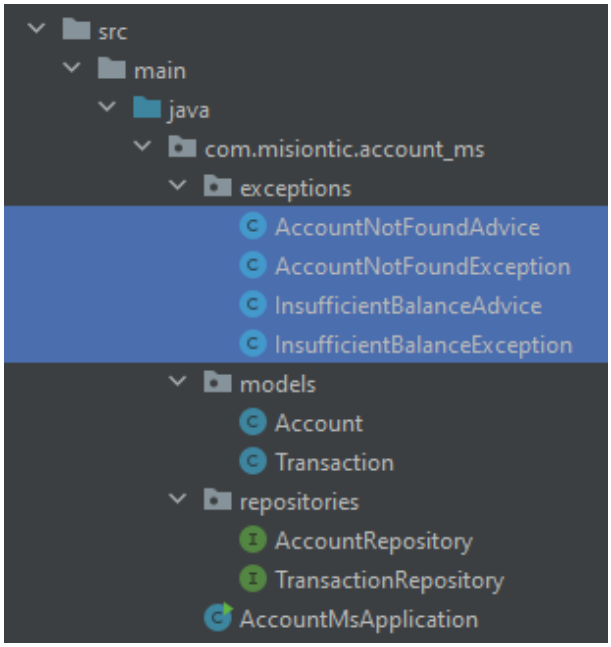
findByUsernameDestiny. Estos le indican a Spring Boot que debe implementar dos métodos, uno que se encargue de buscar una transacción utilizando el *username* del usuario que envía el dinero, y el otro que haga lo mismo utilizando el *username* del usuario que recibe el dinero. Se debe tener en cuenta que los nombres de los métodos y sus parámetros son importantes, por lo tanto, si se cambian Spring Boot no implementará adecuadamente dichas funciones.

Manejo de Excepciones

Generalmente, un microservicio suele recibir peticiones erróneas, lo cual produce errores dentro de su ejecución. Dichos errores se controlan en Spring Boot por medio de *excepciones*, las cuales se encargan de detener la ejecución de la petición dentro del microservicio, y de retornar una respuesta que indique las razones por las cuales la petición es inválida.

Java por defecto provee algunas excepciones, sin embargo, no es recomendable utilizarlas pues estas no proveen información detallada al usuario sobre el tipo de error ocurrido. Por ello, dentro de Spring Boot el correcto manejo de una *excepción* consta de dos partes, la primera es la *excepción*, la cual es lanzada donde ocurre el error, y la segunda es un *advice*, el cual captura la excepción y ofrece una vista adecuada. Dentro de dicho *advice* se suelen utilizar herramientas como los códigos de estados de las peticiones HTTP, y los mensajes personalizados que especifican la causa del error.

En este caso se crearán dos excepciones personalizadas, una llamada *AccountNotFound* para indicar que una cuenta no fue encontrada cuando se intentó realizar una transacción, y otra excepción llamada *InsufficientBalance*, para indicar que el saldo de una cuenta no es suficiente para realizar una transacción. Para implementar las excepciones se debe crear la carpeta (o *Package*) *exceptions*, y dentro de esta se deben crear las clases *AccountNotFoundAdvice*, *AccountNotFoundException*, *InsufficientBalanceAdvice*, y *InsufficientBalanceException*. Al hacerlo, la estructura del proyecto debe ser la siguiente:



El código de cada *Exception*/*Advice* es el siguiente:

- *AccountNotFoundAdvice*:

```
package com.misiontic.account_ms.exceptions;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

@ControllerAdvice
@ResponseBody

public class AccountNotFoundAdvice {
    @ResponseBody
    @ExceptionHandler(AccountNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    String EntityNotFoundAdvice(AccountNotFoundException ex){
        return ex.getMessage();
    }
}
```

- *AccountNotFoundException*:

```
package com.misiontic.account_ms.exceptions;

public class AccountNotFoundException extends RuntimeException {
    public AccountNotFoundException(String message) {
        super(message);
    }
}
```

- *InsufficientBalanceAdvice*:

```
package com.misiontic.account_ms.exceptions;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;


@ControllerAdvice
@ResponseBody
public class InsufficientBalanceAdvice {
    @ResponseBody
    @ExceptionHandler(InsufficientBalanceException.class)
    @ResponseStatus(HttpStatus.CONFLICT)
    String InsufficientBalanceAdvice(InsufficientBalanceException ex){
        return ex.getMessage();
    }
}
```

- *InsufficientBalanceException*:

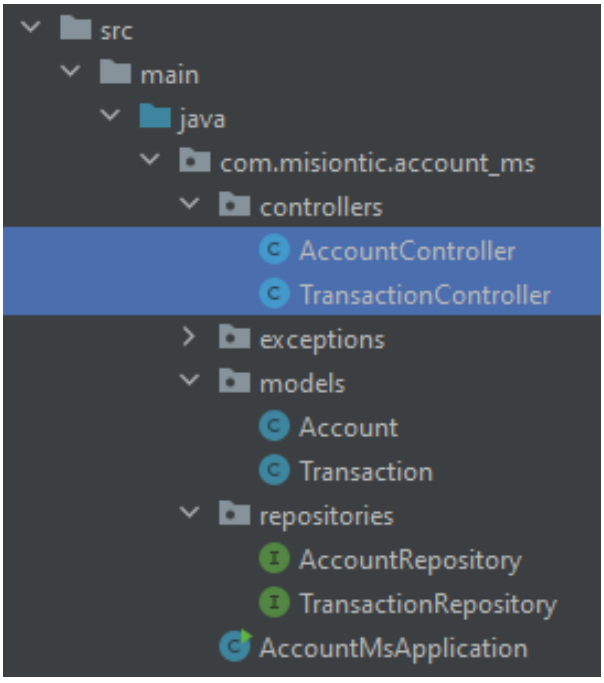
```
package com.misiontic.account_ms.exceptions;

public class InsufficientBalanceException extends RuntimeException {

    public InsufficientBalanceException(String message) {
        super(message);
    }
}
```

Controladores

En este punto ya se tienen dos modelos y sus respectivos repositorios, los cuales permiten realizar una serie de operaciones con la base de datos (buscar, crear, actualizar y eliminar), y se tiene además un sistema de manejo de errores personalizados. Una vez se tienen estos recursos, se deben implementar los controladores, los cuales permitirán el acceso del usuario a las funcionalidades del microservicio.

Generalmente, un controlador está asociado a una entidad (o modelo), por ello, en este caso se desarrollarán los dos controladores, *AccountController* y *TransactionController*. Cada uno de estos será una clase cuyos métodos cumplirán dos funciones, la primera es indicarle a Spring Boot por medio de decoradores (como *@GetMapping* o *@PostMapping*) cómo debe crear la vista asociada, y la segunda especificar cómo responderá dicha vista a una petición HTTP.

Esto significa que por cada funcionalidad que se desee crear, se debe implementar en un controlador, una función que especifique el comportamiento de la vista que se asociará con esta. Por ello, para implementar las tres funcionalidades indicadas anteriormente, se debe crear la carpeta (o *Package*) *controllers*, y dentro de esta se deben crear las clases *AccountController*, y *TransactionController*. Al hacerlo, la estructura del proyecto debe ser la siguiente:



En el controlador *AccountController* se van a implementar las funcionalidades para crear y consultar la información de la cuenta de un usuario. Para ello, en este archivo se tendrá el siguiente código:

```
package com.misiontic.account_ms.controllers;

import com.misiontic.account_ms.exceptions.AccountNotFoundException;
import com.misiontic.account_ms.models.Account;
import com.misiontic.account_ms.repositories.AccountRepository;
import org.springframework.web.bind.annotation.*;

@RestController
public class AccountController {

    private final AccountRepository accountRepository;

    public AccountController(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }

    @GetMapping("/accounts/{username}")
    Account getAccount(@PathVariable String username){
        return accountRepository.findById(username)
                .orElseThrow(() -> new AccountNotFoundException("No se encontro una cuenta
con el username: " + username));
    }

    @PostMapping("/accounts")
    Account newAccount(@RequestBody Account account){
        return accountRepository.save(account);
    }
}
```

Por otro lado, en el controlador *TransactionController* se van a implementar las dos funcionalidades restantes, una para realizar una transacción y la otra para consultar todas las transacciones en las que un usuario se ha visto involucrado. Para ello, en este archivo se debe utilizar el siguiente código:

```
package com.misiontic.account_ms.controllers;

import com.misiontic.account_ms.exceptions.AccountNotFoundException;
import com.misiontic.account_ms.exceptions.InsufficientBalanceException;
import com.misiontic.account_ms.models.Account;
import com.misiontic.account_ms.models.Transaction;
import com.misiontic.account_ms.repositories.AccountRepository;
import com.misiontic.account_ms.repositories.TransactionRepository;
import org.springframework.web.bind.annotation.*;
import java.util.Date;
import java.util.List;
import java.util.stream.Collectors;
```

```
import java.util.stream.Stream;

@RestController
public class TransactionController {

    private final AccountRepository accountRepository;
    private final TransactionRepository transactionRepository;

    public TransactionController(AccountRepository accountRepository,
TransactionRepository transactionRepository) {
        this.accountRepository = accountRepository;
        this.transactionRepository = transactionRepository;
    }

    @PostMapping("/transactions")
    Transaction newTransaction(@RequestBody Transaction transaction){
        Account accountOrigin =
accountRepository.findById(transaction.getUsernameOrigin()).orElse(null);
        Account accountDestinity=
accountRepository.findById(transaction.getUsernameDestiny()).orElse(null);

        if (accountOrigin == null)
            throw new AccountNotFoundException("No se encontro una cuenta con el username:
" + transaction.getUsernameOrigin());

        if (accountDestinity == null)
            throw new AccountNotFoundException("No se encontro una cuenta con el username:
" + transaction.getUsernameDestiny());

        if(accountOrigin.getBalance() < transaction.getValue())
            throw new InsufficientBalanceException("Saldo Insuficiente");

        accountOrigin.setBalance(accountOrigin.getBalance() - transaction.getValue());
        accountOrigin.setLastChange(new Date());
        accountRepository.save(accountOrigin);

        accountDestinity.setBalance(accountDestinity.getBalance() +
transaction.getValue());
        accountDestinity.setLastChange(new Date());
        accountRepository.save(accountDestinity);

        transaction.setDate(new Date());
        return transactionRepository.save(transaction);
    }

    @GetMapping("/transactions/{username}")
    List<Transaction> userTransaction(@PathVariable String username){
        Account userAccount = accountRepository.findById(username).orElse(null);
        if (userAccount == null)
```

```
            throw new AccountNotFoundException("No se encontro una cuenta con el username:
" + username);

        List<Transaction> transactionsOrigin =
transactionRepository.findByUsernameOrigin(username);
        List<Transaction> transactionsDestinity =
transactionRepository.findByUsernameDestiny(username);

        List<Transaction> transactions = Stream.concat(transactionsOrigin.stream(),
transactionsDestinity.stream()).collect(Collectors.toList());

        return transactions;
    }
}
```

Para comprender el código dentro de ambos controladores es necesario hacer una revisión detallada de este, sin embargo, algunos detalles que se deben tener en cuenta para ello son los siguientes:

- **Inyección de dependencias:** dentro de los constructores de ambos controladores se reciben instancias de los repositorios como parámetros. Es natural pensar que en algún momento se deben instanciar dichos repositorios, pero no es así, pues una de las tareas que realiza Spring Boot es identificar a la clase como un controlador (gracias al decorador *@RestController*) y crear las instancias de los repositorios que les pasará a los constructores de cada controlador., reduciendo de esta forma el trabajo del desarrollador. Al proceso que realiza Spring Boot se le conoce como *inyección de dependencias*.

- **Path:** como parámetro de los decoradores *@GetMapping* y *@PostMapping*, se tiene un *path* o ruta. Dicho path junto al tipo de decorador (*GET* o *POST*), indican cómo el usuario puede acceder a la vista generada por Spring Boot. Por ejemplo, si un usuario desea acceder a la función implementada por un método cuyo decorador sea `@GetMapping(/accounts/{username})`, entonces dicho usuario debe acceder a la URL *url_microservicio/accounts/{username}*, a través del método *GET*. A continuación, se explica el significado del texto entre corchetes.

- **Parámetros de una petición HTTP:** una vista puede recibir parámetros indicados por el usuario final. Algunos de los tipos de parámetros que puede recibir la vista son *Parámetros de Ruta* y *Parámetros de Body*, los primeros como su nombre lo indica se envían a través de las rutas, en el decorador estos se definen usando corchetes (*{ }*); los segundos se envían a través del cuerpo de la petición, pero no es necesario especificarlos en el decorador. En ambos casos los parámetros son capturados por el método con ayuda de otro decorador.

- **Desarrollo finalizado:** en este punto, el desarrollo ha terminado. Sin embargo, es posible que surjan dudas sobre cómo se conecta la aplicación con los controladores y los demás elementos creados en esta guía, si anteriormente solo se ha integrado la clase *AccountMsApplication* a dicha aplicación. Esto es posible gracias a Spring Boot, pues este se encarga de revisar todos y cada uno de los

archivos que se encuentran en la carpeta *src/main/java*, y con ayuda de los decoradores es capaz de identificar los distintos componentes del modelo MVC, e integrarlo; reduciendo así el trabajo del desarrollador.

Ejecución del Microservicio

Una vez se ha desarrollado el microservicio, se puede comprobar que no hay ningún error al ejecutarlo. Para ello se utiliza el comando *mvnw spring-boot:run* (al igual que en la clase pasada):

```
D:\account_ms>mvnw spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< com.misiontic:account_ms >----------------------
[INFO] Building account_ms 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
```

Si no se muestra ningún error significa que el desarrollo se ha realizado correctamente. Por otro lado, se debe tener en cuenta que será posible acceder a las funcionalidades del microservicio por medio de la URL *http://localhost:8080*. En la próxima sesión, una vez realizado el despliegue, se probarán cada una de las funcionalidades del microservicio.

Nota: de ser necesario, en el material de la clase se encuentra un archivo llamado *C4a.AP.05. account_ms.rar*, con el desarrollo del componente realizado en esta guía.