public class PersonRepositoryImpl implements PersonRepository {  
  
 Person michael = new Person(1, "Michael", "Weston");  
 Person fiona = new Person(2, "Fiona", "Glenanne");  
 Person sam = new Person(3, "Sam", "Axe");  
 Person jesse = new Person(4, "Jesse", "Porter");  
  
 @Override  
 public Mono<Person> getById(Integer id) {  
 return Mono.*just*(michael);  
 }  
  
 @Override  
 public Flux<Person> findAll() {  
 return Flux.*just*(michael, fiona, sam, jesse);  
 }  
}

We do not want blocking.

We prefer to create a subscriber!

@Test  
void getByIdMapFunction(){  
 Mono<Person> personMono = personRepository.getById(1);  
 personMono.map(person -> {  
 System.*out*.println(person.toString());  
 return person.getFirstName();  
 }).subscribe(firstName ->  
 System.*out*.println("from map: "+ firstName));  
}

@Test  
void findPersonById(){  
 Flux<Person> personFlux = personRepository.findAll();  
  
 final Integer id = 3;  
  
 Mono<Person> personMono = personFlux.filter(person ->  
 person.getId() == id  
 ).next();  
  
 personMono.subscribe(person -> {  
 System.*out*.println(person.toString());  
 });  
  
}

@Test  
void findPersonByIdNotFound(){  
 Flux<Person> personFlux = personRepository.findAll();  
  
 final Integer id = 8;  
  
 Mono<Person> personMono = personFlux.filter(person ->  
 person.getId() == id  
 ).next(); //returns first element that matches or emtpy  
  
 personMono.subscribe(person -> {  
 System.*out*.println(person.toString());  
 }); //if not found nothing is being printed  
  
}

@Test  
void findPersonByIdNotFoundWithException(){  
 Flux<Person> personFlux = personRepository.findAll();  
  
 final Integer id = 8;  
  
 Mono<Person> personMono = personFlux.filter(person ->  
 person.getId() == id  
 ).single(); //returns first element or throws NoSuchElementException  
 // if empty source or IndexOutOfBoundsException if more than one el  
  
 personMono.subscribe(person -> {  
 System.*out*.println(person.toString());  
 }); //throws exception  
  
}

personMono  
 **.doOnError(throwable -> {  
 System.*out*.println("I went boom");  
 })** .subscribe(person -> {  
 System.*out*.println(person.toString());  
});

still throws the exception : Exceptions$ErrorCallbackNotImplemented

personMono  
 .doOnError(throwable -> {  
 System.*out*.println("I went boom");  
 })  
 **.onErrorReturn(Person.*builder*().build())** .subscribe(person -> {  
 System.*out*.println(person.toString());  
});

I went boom

Person(id=null, firstName=null, lastName=null)

Assignment

* Current Find By Id hard coded to return same mono
* Update Repository to return mono matching id
* Take data from find all, filter on id
* Return an empty mono if not found
* Write test to verify interactions

package mititelu.laura.reactiveexamples;  
  
import mititelu.laura.reactiveexamples.domain.Person;  
import reactor.core.publisher.Flux;  
import reactor.core.publisher.Mono;  
  
public class PersonRepositoryImpl implements PersonRepository {  
  
 Person michael = new Person(1, "Michael", "Weston");  
 Person fiona = new Person(2, "Fiona", "Glenanne");  
 Person sam = new Person(3, "Sam", "Axe");  
 Person jesse = new Person(4, "Jesse", "Porter");  
  
 @Override  
 public Mono<Person> getById(Integer id) {  
 return findAll().filter(person -> person.getId()==id).next();  
 }  
  
 @Override  
 public Flux<Person> findAll() {  
 return Flux.*just*(michael, fiona, sam, jesse);  
 }  
}

package mititelu.laura.reactiveexamples;  
  
import mititelu.laura.reactiveexamples.domain.Person;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import reactor.core.publisher.Flux;  
import reactor.core.publisher.Mono;  
  
import java.util.List;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
class PersonRepositoryImplTest {  
  
 PersonRepository personRepository;  
  
 @BeforeEach  
 void setUp() {  
 personRepository = new PersonRepositoryImpl();  
 }  
  
 @Test  
 void getById() {  
  
 //blocking  
 Mono<Person> personMono = personRepository.getById(1);  
  
 Person person = personMono.block();  
 System.*out*.println(person.toString());  
  
 }  
  
 @Test  
 void getByIdSubscribe(){  
 Mono<Person> personMono = personRepository.getById(1);  
 personMono.subscribe(person -> {  
 System.*out*.println(person.toString());  
 });  
 }  
  
 @Test  
 void getByIdMapFunction(){  
 Mono<Person> personMono = personRepository.getById(1);  
 personMono.map(person -> {  
 System.*out*.println(person.toString());  
 return person.getFirstName();  
 }).subscribe(firstName ->  
 System.*out*.println("from map: "+ firstName));  
 }  
  
 @Test  
 void fluxTestBlockFirst(){  
 Flux<Person> personFlux = personRepository.findAll();  
  
 Person person = personFlux.blockFirst(); //gives us back the first person  
 System.*out*.println(person.toString());  
  
 }  
  
 @Test  
 void testFluxSubscribe(){  
 Flux<Person> personFlux = personRepository.findAll();  
  
 personFlux.subscribe(person -> {  
 System.*out*.println(person.toString());  
 });  
  
 }  
  
 @Test  
 void testFluxToListMono(){  
 Flux<Person> personFlux = personRepository.findAll();  
 Mono<List<Person>> personListMono = personFlux.collectList();  
  
 personListMono.subscribe(list ->{  
 System.*out*.println("printing list");  
 list.forEach(person -> {  
 System.*out*.println(person.toString());  
 });  
 });  
 }  
  
 @Test  
 void findPersonById(){  
 Flux<Person> personFlux = personRepository.findAll();  
  
 final Integer id = 3;  
  
 Mono<Person> personMono = personFlux.filter(person ->  
 person.getId() == id  
 ).next();  
  
 personMono.subscribe(person -> {  
 System.*out*.println(person.toString());  
 });  
  
 }  
  
 @Test  
 void findPersonByIdNotFound(){  
 Flux<Person> personFlux = personRepository.findAll();  
  
 final Integer id = 8;  
  
 Mono<Person> personMono = personFlux.filter(person ->  
 person.getId() == id  
 ).next(); //returns first element that matches or emtpy  
  
 personMono.subscribe(person -> {  
 System.*out*.println(person.toString());  
 }); //if not found nothing is being printed  
  
 }  
  
 @Test  
 void findPersonByIdNotFoundWithException(){  
 Flux<Person> personFlux = personRepository.findAll();  
  
 final Integer id = 8;  
  
 Mono<Person> personMono = personFlux.filter(person ->  
 person.getId() == id  
 ).single(); //returns first element or throws NoSuchElementException  
 // if empty source or IndexOutOfBoundsException if more than one el  
  
 personMono  
 .doOnError(throwable -> {  
 System.*out*.println("I went boom");  
 })  
 .onErrorReturn(Person.*builder*().build())  
 .subscribe(person -> {  
 System.*out*.println(person.toString());  
 });  
  
 }  
  
 @Test  
 void testNewFindByIdFound(){  
 final Integer id = 3;  
 Mono<Person> personMono = personRepository.getById(id);  
  
 personMono.subscribe(person -> {  
 System.*out*.println(person.toString());  
 });  
 }  
  
 @Test  
 void testNewFindByIdNotFound(){  
 final Integer id = 99;  
 Mono<Person> personMono = personRepository.getById(id);  
 personMono.subscribe(person -> {  
 System.*out*.println(person.toString());  
 });  
  
 }  
  
}

Tool Step Verifier

it gives us some utilities to inspect the monos or model some fluxes, any reactive type to go ahead and see if they are behaving as we expect.

@Test  
void getByIdSubscribe(){  
 Mono<Person> personMono = personRepository.getById(1);  
  
 **StepVerifier.*create*(personMono).expectNextCount(1).verifyComplete();**  
  
 personMono.subscribe(person -> {  
 System.*out*.println(person.toString());  
 });  
}

**expectNextCount(1).verifyComplete(); -> expect 1 interaction from Mono**

@Test  
void testNewFindByIdNotFound(){  
 final Integer id = 99;  
 Mono<Person> personMono = personRepository.getById(id);  
  
 StepVerifier.*create*(personMono).verifyComplete();  
 // or StepVerifier.create(personMono).expectNextCount(0).verifyComplete();  
  
 personMono.subscribe(person -> {  
 System.*out*.println(person.toString());  
 });  
  
}

@Test  
void testFluxSubscribe(){  
 Flux<Person> personFlux = personRepository.findAll();  
  
 StepVerifier.*create*(personFlux).expectNextCount(4).verifyComplete();  
  
 personFlux.subscribe(person -> {  
 System.*out*.println(person.toString());  
 });  
  
}

USE STEP VERIFIER instead of sout!!

StepVerifier is part of Spring Boot Starter reactor test