Rest Template – based on Servlet API, as Spring MVC

Spring WebClient – based on HTTP/Reactive Streams , as Spring WebFlux

Spring RestTemplate

* Blocking
* Based on Java Servlet API
* Used one thread per request
* Will block during request

Spring WebClient

* Is asynchronous, non-blocking
* Based on Spring Reactive Framework
* Requests are ‘queued’ and processed when available
* Uses Mono and Flux types
* Allows for functional / declarative programming

Which to use

* Performance will be similar until a heavier load
* RestTemplate has been in Spring for a long time and is very mature
* RestTemplate has been moved to ‘maintenance’ mode by Spring – no features will be added
* WebClient is being actively developed by Spring
* At some point in the future RestTemplate will be deprecated – likely YEARS in the future

Doc generated by OpenAPI Spec:

<https://sfg-beer-works.github.io/brewery-api/#tag/Beer-Service>

<https://github.com/sfg-beer-works/sfg-restful-brewery>

Assignment:

* You be the developer! Your assignment is to create a client to use the Beer API
* Pretend client is part of a larger application
* Create new project using Spring Initializr
  + Java 11+, Maven, Jar
  + Reactive Web (WebFlux), Lombok
    - JUnit 5 (automatically included with Spring Boot Test starter)
* Create POJO model from API spec, use Project Lombok
* Create Interface for beer client, add method for each API method
  + Return Mono of POJO Type, or ClientResponse if no object is returned
  + Use Mono for List - API returns a page, not a stream (to be covered in upcoming lesson)
* Create an implementation of Interface - Use IntelliJ, accept stub method implementation

Add external validation

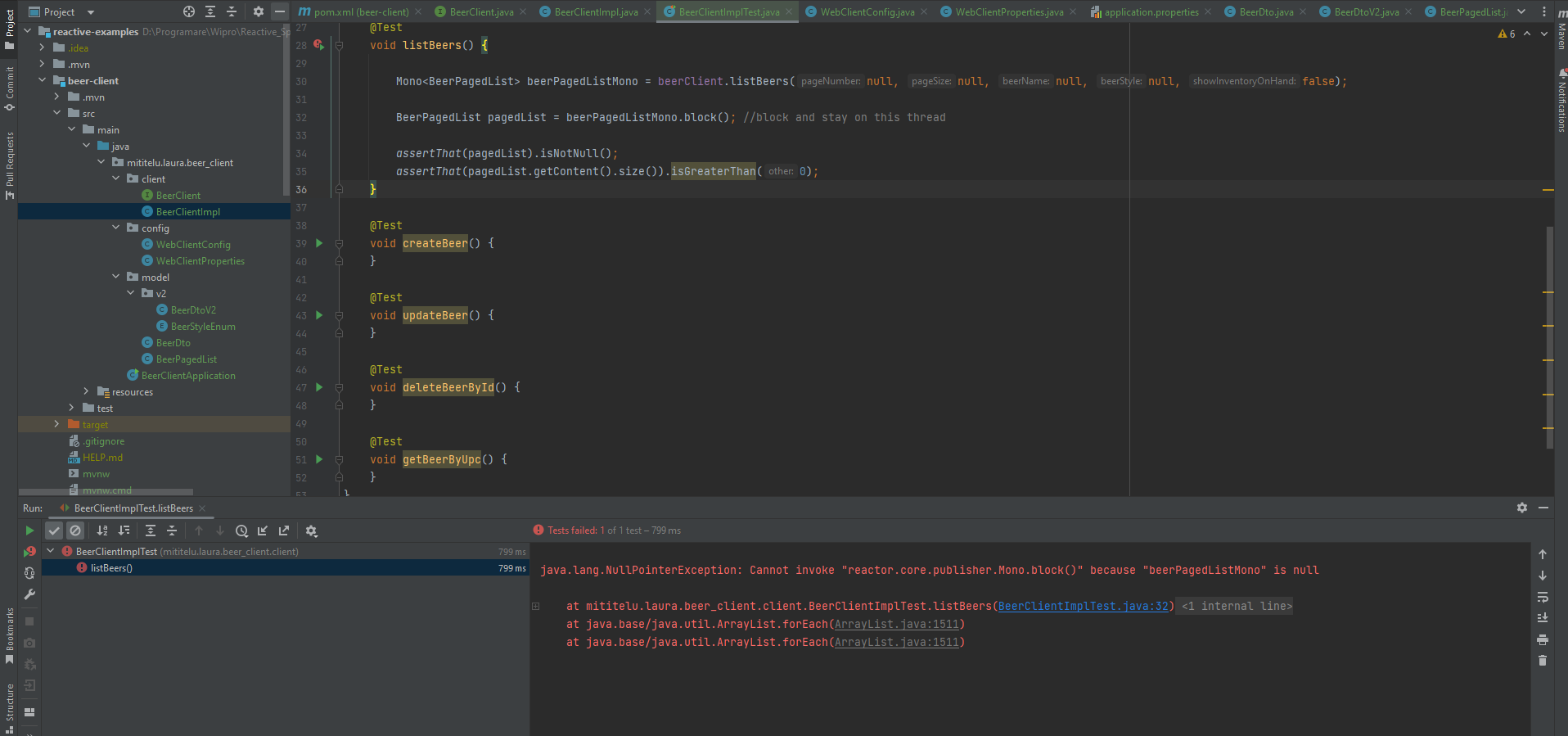
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-validation</artifactId>  
</dependency>

Spring does no longer include it by default

Added for pageable:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>

If not implemented:



@Override  
public Mono<BeerPagedList> listBeers(Integer pageNumber, Integer pageSize, String beerName, String beerStyle, Boolean showInventoryOnHand) {  
 return webClient.get()  
 .uri(uriBuilder -> uriBuilder.path(WebClientProperties.*BEER\_V1\_PATH*)  
 .queryParamIfPresent("pageNumber", Optional.*ofNullable*(pageNumber))  
 .queryParamIfPresent("pageSize", Optional.*ofNullable*(pageSize))  
 .queryParamIfPresent("beerName", Optional.*ofNullable*(beerName))  
 .queryParamIfPresent("beerStyle", Optional.*ofNullable*(beerStyle))  
 .queryParamIfPresent("showInventoryOnHand", Optional.*ofNullable*(showInventoryOnHand))  
 .build())  
 .retrieve()  
 .bodyToMono(BeerPagedList.class);  
}

Underneath, WebClient uses Netty – HHTP reactive, non-blocking

Logging

For logging we need to customize the Netty Connector

App properties add:

logging.level.reactor.netty.http.client=trace

@Bean  
public WebClient webclient(){  
 return WebClient.*builder*()  
 .baseUrl(WebClientProperties.*BASE\_URL*)  
 .clientConnector(new ReactorClientHttpConnector(HttpClient.*create*().wiretap("reactor.netty.client.HttpClient", LogLevel.*DEBUG*,  
 AdvancedByteBufFormat.*TEXTUAL*)))  
 .build();  
  
}

URL params

public static final String *BEER\_V1\_PATH\_GET\_BY\_ID* = "/api/v1/beer/{uuid}";

@Override  
public Mono<BeerDto> getBeerById(UUID id, Boolean showInventoryOnHand) {  
 return webClient.get()  
 .uri(uriBuilder -> uriBuilder.path(WebClientProperties.*BEER\_V1\_PATH\_GET\_BY\_ID*)  
 .queryParamIfPresent("showInventoryOnHand", Optional.*ofNullable*(showInventoryOnHand))  
 .build(id))  
 .retrieve()  
 .bodyToMono(BeerDto.class);  
}

POST

@Override  
public Mono<ResponseEntity<Void>> createBeer(BeerDto beerDto) {  
 return webClient.post()  
 .uri(uriBuilder -> uriBuilder.path(WebClientProperties.*BEER\_V1\_PATH*).build())  
 .body(BodyInserters.*fromValue*(beerDto))  
 .retrieve()  
 .toBodilessEntity();  
}

@Test  
void createBeer() {  
  
 BeerDto beerDto = BeerDto.*builder*()  
 .beerName("Ciuc")  
 .beerStyle("IPA")  
 .upc("25945652122")  
 .price(new BigDecimal("8.99"))  
 .build();  
  
 Mono<ResponseEntity<Void>> responseEntityMono = beerClient.createBeer(beerDto);  
 ResponseEntity response = responseEntityMono.block();  
 *assertThat*(response.getStatusCode()).isEqualTo(HttpStatus.*CREATED*);  
  
}

PUT

@Override  
public Mono<ResponseEntity<Void>> updateBeer(UUID id, BeerDto beerDto) {  
 return webClient.put()  
 .uri(uriBuilder -> uriBuilder.path(WebClientProperties.*BEER\_V1\_PATH\_GET\_BY\_ID*).build(id))  
 .body(BodyInserters.*fromValue*(beerDto))  
 .retrieve()  
 .toBodilessEntity();  
}

@Test  
void updateBeer() {  
  
 Mono<BeerPagedList> beerPagedListMono = beerClient.listBeers(null, null, null, null, null);  
 BeerPagedList pagedList = beerPagedListMono.block();  
 BeerDto beerDto = pagedList.getContent().get(0);  
  
 BeerDto updatedBeerDto = BeerDto.*builder*()  
 .beerName("Ursus")  
 .beerStyle(beerDto.getBeerStyle())  
 .price(beerDto.getPrice())  
 .upc(beerDto.getUpc())  
 .build();  
  
 Mono<ResponseEntity<Void>> responseEntityMono = beerClient.updateBeer(beerDto.getId(), updatedBeerDto);  
 ResponseEntity<Void> response = responseEntityMono.block();  
 *assertThat*(response.getStatusCode()).isEqualTo(HttpStatus.*NO\_CONTENT*);  
  
  
}

DELETE

@Test  
void deleteBeerById() {  
 Mono<BeerPagedList> beerPagedListMono = beerClient.listBeers(null, null, null, null, null);  
  
 BeerPagedList pagedList = beerPagedListMono.block();  
 BeerDto beerDto = pagedList.getContent().get(0);  
  
 Mono<ResponseEntity<Void>> responseEntityMono = beerClient.deleteBeerById(beerDto.getId());  
 ResponseEntity<Void> response = responseEntityMono.block();  
  
 *assertThat*(response.getStatusCode()).isEqualTo(HttpStatus.*NO\_CONTENT*);  
  
}

@Override  
public Mono<ResponseEntity<Void>> deleteBeerById(UUID id) {  
 return webClient.delete()  
 .uri(uriBuilder -> uriBuilder.path(WebClientProperties.*BEER\_V1\_PATH\_GET\_BY\_ID*).build(id))  
 .retrieve()  
 .toBodilessEntity();  
}

Exception Handling

@Test  
void deleteBeerByIdNotFound() {  
  
 Mono<ResponseEntity<Void>> responseEntityMono = beerClient.deleteBeerById(UUID.*randomUUID*());  
  
  
 Assertions.*assertThrows*(WebClientResponseException.NotFound.class, () -> {  
 ResponseEntity<Void> response = responseEntityMono.block();  
 *assertThat*(response.getStatusCode()).isEqualTo(HttpStatus.*NOT\_FOUND*);  
 });  
  
 //assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT\_FOUND); -> it will fail  
  
}

@Test  
void testDeleteBeerHandleException() {  
  
 Mono<ResponseEntity<Void>> responseEntityMono = beerClient.deleteBeerById(UUID.*randomUUID*());  
  
 ResponseEntity<Void> responseEntity = responseEntityMono  
 .onErrorResume(throwable ->{  
 if(throwable instanceof WebClientResponseException){  
 WebClientResponseException exception = (WebClientResponseException) throwable;  
 return Mono.*just*(ResponseEntity.*status*(exception.getStatusCode()).build());  
 }  
 else {  
 throw new RuntimeException(throwable);  
 }  
 })  
 .block();  
 *assertThat*(responseEntity.getStatusCode()).isEqualTo(HttpStatus.*NOT\_FOUND*);  
  
}

Reactive Threading

@Test  
void functionalTestGetBeerById(){  
 beerClient.listBeers(null, null, null, null, true)  
 .map(beerPagedList -> beerPagedList.getContent().get(0).getId())  
 .map(beerId -> beerClient.getBeerById(beerId, false))  
 .flatMap( mono -> mono)  
 .subscribe(beerDto ->{  
 System.*out*.println(beerDto.getBeerName());  
 *assertThat*(beerDto.getBeerName()).isEqualTo("foo");  
 });  
}

Test is completing so fast, we’re bypassing it!!

Sout and assertThat do not run

If I add Thread.*sleep*(5000);

It will execute but behave strangely – will not throw assert error, it will show the error as logging

@Test  
void functionalTestGetBeerById() throws InterruptedException {  
  
 AtomicReference<String> beerName = new AtomicReference<>();  
  
 beerClient.listBeers(null, null, null, null, true)  
 .map(beerPagedList -> beerPagedList.getContent().get(0).getId())  
 .map(beerId -> beerClient.getBeerById(beerId, false))  
 .flatMap( mono -> mono)  
 .subscribe(beerDto ->{  
 System.*out*.println(beerDto.getBeerName());  
 beerName.set(beerDto.getBeerName());  
  
 });  
  
 Thread.*sleep*(5000);  
 *assertThat*(beerName.get()).isEqualTo("foo");  
}

Thread.*sleep*(5000); -> not good

Execution will be kind of random

CountDownLatch

* For testing concurrent apps in multi-threaded env

@Test  
void functionalTestGetBeerById() throws InterruptedException {  
  
 AtomicReference<String> beerName = new AtomicReference<>();  
 CountDownLatch countDownLatch = new CountDownLatch(1);  
  
 beerClient.listBeers(null, null, null, null, true)  
 .map(beerPagedList -> beerPagedList.getContent().get(0).getId())  
 .map(beerId -> beerClient.getBeerById(beerId, false))  
 .flatMap( mono -> mono)  
 .subscribe(beerDto ->{  
 System.*out*.println(beerDto.getBeerName());  
 beerName.set(beerDto.getBeerName());  
 *assertThat*(beerDto.getBeerName()).isEqualTo("Mango Bobs");  
 countDownLatch.countDown();  
 });  
  
 countDownLatch.await(); //will stop and wait until that countdown latch has been incremented by a value of 1  
 //that is until countDownLatch.countDown(); get executed  
 *assertThat*(beerName.get()).isEqualTo("Mango Bobs");  
}

!! countDownLatch.countDown();

Should be the executed last in subscribe!!!

Only for testing!!

countDownLatch

<https://medium.com/swlh/understanding-reactors-flatmap-operator-a6a7e62d3e95>

<https://spring.io/blog/2019/12/13/flight-of-the-flux-3-hopping-threads-and-schedulers>