

Spring Boot Web Slice test – Sample

 javacodegeeks.com/2017/06/spring-boot-web-slice-test-sample.html

Spring Boot [introduced test slicing](#) a while back and it has taken me some time to get my head around it and explore some of its nuances.

Background

The main reason to use this feature is to reduce the boilerplate. Consider a controller that looks like this, just for variety written using [Kotlin](#).

```
01 @RestController
02 @RequestMapping("/users")
03 class UserController(
04     val userRepository:
05         private UserRepository,
06
07     val userResourceAssembler: UserResourceAssembler)
08     private {
09
10         @GetMapping
11         fun getUsers(pageable:
12             Pageable,
13
14             pagedResourcesAssembler: PagedResourcesAssembler<User>) :
15             PagedResources<Resource<User>> {
16
17             val users =
18                 userRepository.findAll(pageable)
19
20             return pagedResourcesAssembler.toResource(users, this
21                 .userResourceAssembler)
22         }
23     }
```

```

14      @GetMapping("/{id}")
15      fun getUser(id: Long): Resource<User>
16      {
17          return Resource(userRepository.findOne(id))
18      }

```

A traditional [Spring Mock MVC test](#) to test this controller would be along these lines:

```

01  @RunWith(SpringRunner::class)
02  @WebAppConfiguration
03  @ContextConfiguration
04      UserControllerTests
05      class {
06          lateinit var mockMvc:
07              MockMvc
08          @Autowired
09              val wac: WebApplicationContext?
10              private = null
11          @Before
12          fun setup()
13          {
14              .mockMvc =
15                  thisMockMvcBuilders.webAppContextSetup(
16                      this.wac).build()

```

```
14    }  
15  
16    @Test  
17    fun testGetUsers()  
18    {  
19        this.mockMvc.perform(get("/users"))  
20            .accept(MediaType.APPLICATION_JSON)  
21            .andDo(print())  
22            .andExpect(status().isOk)  
23    }  
24  
25    @EnableSpringDataWebSupport  
26    @EnableWebMvc  
27    @Configuration  
28    class SpringConfig  
29    {  
30        @Bean  
31        fun userController(): UserController  
32        {  
33            return UserController(userRepository(),  
34                return UserResourceAssembler())  
35        }  
36    }
```

```

33
34         @Bean
35         fun userRepository(): UserRepository
36         {
37             val userRepository =
38                 Mockito.mock(UserRepository::class.java)
39
40             given(userRepository.findAll(Matchers.any(Pageable::class.java)))
41
42                 .willAnswer({ invocation -
43                     >
44                         val pageable =
45                             invocation.arguments[0] as
46                             Pageable
47
48                         PageImpl(
49
50                             listOf(
51
52                                 User(id = 1, fullName = "one", password
53                                     = "one", email = "one@one.com"),
54
55                                 User(id = 2, fullName = "two", password
56                                     = "two", email = "two@two.com"))
57
58                             , pageable, 10)
59
60                 })
61
62             return userRepository
63
64         }
65     }
66 }

```

There is a lot of ceremony involved in setting up such a test – a web application context which understands a web environment is pulled in, a configuration which sets up the Spring MVC environment needs to be created and MockMvc which is handle to the testing framework needs to be set-up before each test.

Web Slice Test

A web slice test when compared to the previous test is far simpler and focuses on testing the controller and hides a lot of the boilerplate code:

```
01  @RunWith(SpringRunner::class)
02  @WebMvcTest(UserController::class)
03      UserControllerSliceTests
    class {
04
05      @Autowired
06      lateinit var mockMvc:
        MockMvc
07
08      @MockBean
09      lateinit var userRepository:
        UserRepository
10
11      @SpyBean
12      lateinit var userResourceAssembler:
        UserResourceAssembler
13
14      @Test
15      fun testGetUsers()
        {
```

```

16
17         this.mockMvc.perform(get("/users").param("page","0").param("size","1")
18
19             .accept(MediaType.APPLICATION_JSON))
20
21             .andDo(print())
22
23             .andExpect(status().isOk)
24     }
25
26     @Before
27     fun setUp(): Unit
28     {
29         given(userRepository.findAll(Matchers.any(Pageable::class.java)))
30
31             .willAnswer({ invocation -
32                 >
33
34                     val pageable =
35                         invocation.arguments[
36                             ] as
37                         0Pageable
38
39                     PageImpl(
40
41                         listOf(
42
43                             User(id
44                                 =
45                                 1, fullName
46                                 = "one", password
47                                 = "one", email
48                                 = "one@one.com"),
49
50                             User(id
51                                 =
52                                 2, fullName
53                                 = "two", password
54                                 = "two", email
55                                 = "two@two.com"))
56
57                     , pageable, 10)

```

```
33         })
```

```
34     }
```

```
35 }
```

It works by creating a Spring Application context but filtering out anything that is not relevant to the web layer and loading up only the controller which has been passed into the `@WebTest` annotation. Any dependency that the controller requires can be injected in as a mock.

Coming to some of the nuances, say if I wanted to inject one of the fields myself the way to do it is have the test use a custom Spring Configuration, for a test this is done by using a inner static class annotated with `@TestConfiguration` the following way:

```
01 @RunWith(SpringRunner::class)
```

```
02 @WebMvcTest(UserController::class)
```

```
03     UserControllerSliceTests  
    class {
```

```
04
```

```
05     @Autowired
```

```
06     lateinit var mockMvc:  
        MockMvc
```

```
07
```

```
08     @Autowired
```

```
09     lateinit var userRepository:  
        UserRepository
```

```
10
```

```
11     @Autowired
```

```
12     lateinit var userResourceAssembler:  
        UserResourceAssembler
```

```
13
```

```

14      @Test
15      fun testGetUsers()
16      {
17          this.mockMvc.perform(get("/users").param("page","0").param("size","1")
18              .accept(MediaType.APPLICATION_JSON))
19              .andDo(print())
20              .andExpect(status().isOk)
21      }
22
23      @Before
24      fun setUp(): Unit
25      {
26          given(userRepository.findAll(Matchers.any(Pageable::class.java)))
27              .willAnswer({ invocation -
28                  >
29                  val pageable = invocation.getArgument(0) as Pageable
30                  PageImpl(
31                      listOf(
32                          User(id = 1, fullName = "one", password = "one", email = "one@one.com"),

```



```

31         User(id      , fullName      , password
           =          2=             "two"=             "two"
           , email
           =             "two@two.com"))

```

```

32         ,
           pageable, 10)

```

```

33     })

```

```

34 }

```

```

35

```

```

36 @TestConfiguration

```

```

37     SpringConfig
    class {

```

```

38

```

```

39         @Bean

```

```

40         fun userResourceAssembler(): UserResourceAssembler
        {

```

```

41             return UserResourceAssembler()

```

```

42         }

```

```

43

```

```

44         @Bean

```

```

45         fun userRepository(): UserRepository
        {

```

```

46             return mock(UserRepository::class.java)

```

```

47         }

```

```

48     }

```

49

50 }

The beans from the “TestConfiguration” adds on to the configuration which the Slice tests depend on and don’t completely replace it.

On the other hand, if I wanted to override the loading of the main “@SpringBootApplication” annotated class then I can pass in a Spring Configuration class explicitly, but the catch is that I have to now take care of all of loading up the relevant Spring Boot features myself (enabling auto-configuration, appropriate scanning etc), so a way around it to explicitly annotate the configuration as a Spring Boot Application the following way:

```
01 @RunWith(SpringRunner::class)
02 @WebMvcTest(UserController::class)
03     UserControllerExplicitConfigTests
    class {
04
05         @Autowired
06         lateinit var mockMvc:
            MockMvc
07
08         @Autowired
09         lateinit var userRepository:
            UserRepository
10
11         @Test
12         fun testGetUsers()
            {
13
14             this.mockMvc.perform(get("/users").param("page","0").param("size","1"))
```

```

15         .accept (MediaType.APPLICATION_JSON) )
16         .andDo (print () )
17         .andExpect (status () .isOk)
18     }
19
20     @Before
21     fun setUp(): Unit
22     {
23         given (userRepository.findAll (Matchers.any (Pageable::class.java)))
24         .willAnswer ({ invocation -
25             >
26                 val pageable = invocation.arguments[0] as Pageable
27                 PageImpl (
28                     listOf (
29                         User (id = 1, fullName = "one", password = "one", email = "one@one.com"),
30                         User (id = 2, fullName = "two", password = "two", email = "two@two.com"))
31                 , pageable, 10)
32     })
33 }

```

```
32
33         (scanBasePackageClasses =
    @SpringBootApplicationarrayOf (UserController::
    class))
34     @EnableSpringDataWebSupport
35     SpringConfig
    class {
36
37         @Bean
38         fun userResourceAssembler(): UserResourceAssembler
    {
39             return UserResourceAssembler()
40         }
41
42         @Bean
43         fun userRepository(): UserRepository
    {
44             return mock (UserRepository::class.java)
45         }
46     }
47
48 }
```

The catch though is that now other tests may end up finding this inner configuration which is far from ideal!, so my learning has been to depend on bare minimum slice testing, and if needed extend it using `@TestConfiguration`.

I have a little more detailed code sample available at [my github repo](#) which has working examples to play with.

Reference: [Spring Boot Web Slice test – Sample](#) from our [JCG partner](#) Biju Kunjummen at the [all and sundry](#) blog.
