# 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.

```
@RestController
@RequestMapping("/users")
class UserController(
            private val userRepository: UserRepository,
            private val userResourceAssembler: UserResourceAssembler) {

    @GetMapping
    fun getUsers(pageable: Pageable,
pagedResourcesAssembler: PagedResourcesAssembler<User>): PagedResources<Resource<User>> {
        val users = userRepository.findAll(pageable)
        return pagedResourcesAssembler.toResource(users,this.userResourceAssembler)
    }

    @GetMapping("/{id}")
    fun getUser(id: Long): Resource<User>
    {
        return Resource(userRepository.findOne(id))
    }
}
```

A traditional Spring Mock MVC test to test this controller would be along these lines:

```
@RunWith(SpringRunner::class)
@WebAppConfiguration
@ContextConfiguration
      UserControllerTests
class {

    lateinit var mockMvc:
    MockMvc

    @Autowired
            val wac: WebApplicationContext?
    private =                              null

    @Before
    fun setup()
    {
            .mockMvc =
        thisMockMvcBuilders.webAppContextSetup(          this.wac).build()
```

```
}

@Test
fun testGetUsers()
{
    this.mockMvc.perform(get("/users")
            .accept(MediaType.APPLICATION_JSON))
            .andDo(print())
            .andExpect(status().isOk)
}

@EnableSpringDataWebSupport
@EnableWebMvc
@Configuration
      SpringConfig
class {

    @Bean
    fun userController(): UserController
    {
                UserController(userRepository(),
        return UserResourceAssembler())
    }
```

```

        @Bean
        fun userRepository(): UserRepository {
            val userRepository = Mockito.mock(UserRepository::class.java)
            given(userRepository.findAll(Matchers.any(Pageable::class.java)))
                    .willAnswer({ invocation ->
                        val pageable = invocation.arguments[0] as Pageable
                        PageImpl(
                                listOf(
                                        User(id = 1, fullName = "one", password = "one", email = "one@one.com"),
                                        User(id = 2, fullName = "two", password = "two", email = "two@two.com"))
                                , pageable, 10)
                    })
            return userRepository
        }
    }
}
```

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:

```
@RunWith(SpringRunner::class)
@WebMvcTest(UserController::class)
      UserControllerSliceTests
class {

    @Autowired
    lateinit var mockMvc: MockMvc

    @MockBean
    lateinit var userRepository: UserRepository

    @SpyBean
    lateinit var userResourceAssembler: UserResourceAssembler

    @Test
    fun testGetUsers() {

        this.mockMvc.perform(get("/users").param("page","0").param("size","1")
                .accept(MediaType.APPLICATION_JSON))
                .andDo(print())
                .andExpect(status().isOk)
    }

    @Before
    fun setUp(): Unit {
        given(userRepository.findAll(Matchers.any(Pageable::class.java)))
                .willAnswer({ invocation ->
                    val pageable = invocation.arguments[0] as Pageable
                    PageImpl(
                            listOf(
                                    User(id = 1, fullName = "one", password = "one", email = "one@one.com"),
                                    User(id = 2, fullName = "two", password = "two", email = "two@two.com"))
                            , pageable, 10)
                    })
    }
}
```

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:

```
@RunWith(SpringRunner::class)
@WebMvcTest(UserController::class)
      UserControllerSliceTests
class {

    @Autowired
    lateinit var mockMvc:
    MockMvc

    @Autowired
    lateinit var userRepository:
    UserRepository

    @Autowired
    lateinit var userResourceAssembler:
    UserResourceAssembler

    @Test
    fun testGetUsers() {

        this.mockMvc.perform(get("/users").param("page","0").param("size","1")
                .accept(MediaType.APPLICATION_JSON))
                .andDo(print())
                .andExpect(status().isOk)
    }

    @Before
    fun setUp(): Unit {
        given(userRepository.findAll(Matchers.any(Pageable::class.java)))
                .willAnswer({ invocation ->
                    val pageable = invocation.arguments[0] as Pageable
                    PageImpl(
                        listOf(
                            User(id = 1, fullName = "one", password = "one", email = "one@one.com"),
                                    User(id    , fullName       , password
                                    =        2=            "two"=          "two"
, email
=        "two@two.com"))
                            ,
                            pageable,  10)
                })
    }

    @TestConfiguration
         SpringConfig
    class {

        @Bean
        fun userResourceAssembler(): UserResourceAssembler
        {
            return UserResourceAssembler()
        }

        @Bean
        fun userRepository(): UserRepository
        {
            return mock(UserRepository::class.java)
        }
    }
```

```
}
```

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:

```
@RunWith(SpringRunner::class)
@WebMvcTest(UserController::class)
class UserControllerExplicitConfigTests {

    @Autowired
    lateinit var mockMvc: MockMvc

    @Autowired
    lateinit var userRepository: UserRepository

    @Test
    fun testGetUsers() {

        this.mockMvc.perform(get("/users").param("page","0").param("size","1"))
```

```
                .accept(MediaType.APPLICATION_JSON))
                .andDo(print())
                .andExpect(status().isOk)
    }

    @Before
    fun setUp(): Unit {
        given(userRepository.findAll(Matchers.any(Pageable::class.java)))
                .willAnswer({ invocation ->
                    val pageable = invocation.arguments[0] as Pageable
                    PageImpl(
                        listOf(
                            User(id = 1, fullName = "one", password = "one", email = "one@one.com"),
                            User(id = 2, fullName = "two", password = "two", email = "two@two.com"))
                        , pageable, 10)
                })
    }
```

```

                        (scanBasePackageClasses =
    @SpringBootApplicationarrayOf(UserController::
class))
    @EnableSpringDataWebSupport
          SpringConfig
    class {

        @Bean
        fun userResourceAssembler(): UserResourceAssembler
        {
            return UserResourceAssembler()
        }

        @Bean
        fun userRepository(): UserRepository
        {
            return mock(UserRepository::class.java)
        }
    }

}
```

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.