## Links for reference

|  |  |
| --- | --- |
| Junit user guide | <https://junit.org/junit5/docs/current/user-guide/#writing-tests> |
| Mockito ref links | <https://site.mockito.org/>  <https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html> |
|  |  |
|  |  |
| Spring framework guru repos for reference | <https://github.com/springframeworkguru/testing-junit5-mockito/tree/mvn-deps> |
| Misc lib | JsonAssert – assertions for json  JsonPath – x path for json <https://github.com/json-path/JsonPath>  <https://github.com/json-path/JsonPath/blob/master/json-path-assert/src/test/java/com/jayway/jsonassert/JsonAssertTest.java> |
| Sample codes of spring framework guru | <https://github.com/manideep-vv/Forked-Junit-Mockito-springRestCodes/tree/jsonPath> |

Spring junits

Annotations

|  |  |
| --- | --- |
| @RunWith(SpringRunner.class) |  |
| @ContextConfiguration(classes =BaseConfig.class, or any class which is annotated with stereo type annotation, no need of config class) | To create the beans and put into context |
| @SpringJUnitConfig(classes = {InnerClassconfigDemo.InnerClassConfig.class}) | @SpringJUnitConfig= @RunWith(SpringRunner.class)+ @ContextConfiguration(classes =BaseConfig.class) |
| @Autowired | To fetch and inject the bean from context |
| @MockBean  EmployeeController empController; | Means the mock bean will be created & it will be kept in the spring container context |
| @Configuration | U can write a separate @Configuration class and u can declare all the beans in that |
| @ActiveProfiles(“dev”) – this should be placed on top of test class |  |
|  |  |
|  |  |

Spring Boot Test annotations

|  |  |
| --- | --- |
| @SpringBootTest(webEnvironment =<option>) -> will enable the spring context +  @RunWith(SpringRunner.class) in junit 4 or  @ExtendWith(SpringExtension.class) in junit 5 – now check if this is already included or not  @ExtendWith(SpringExtension.class) Ⓐ  @ContextConfiguration(classes = {SpringDataConfiguration.class}) Ⓑ  public class HelloWorldSpringDataJPATest {  Ⓑ- means create the container with the beans defined in this config class | This @SpringBootTest will scan entire project and creates all beans it will bring full context and this can be bit heavy for complex project and best alternate is to start only related controllers using @WebMvcTest(PersonController.class)  @ContextConfiguration is needed to tell which class is annotated with @SpringBootApplication  This extension is used to integrate the Spring test context with the JUnit 5 Jupiter test.  @ExtendWith(SpringExtension.class) anno primary use is to enable the Spring TestContext Framework within your JUnit 5 tests.  we should tell with which beans it should create the container right?  Here the spring test context/container is created using the beans defined in SpringDataConfiguration class.  either in above class we should define the beans or or that must be a class anno with @SpringBootApplication |
| @JsonTest | Creates a spring boot configured json env for jackson |
| Spring boot test slices | This brings us light weight configurations which doesn’t enable complete defined auto configuration |
| @RestClientTest | Creates some auto configurations for rest modules |
| @WebMvcTest(PersonController.class) //most recommended way | To start only that particular controller and it is light weight this is like a slice and light weight |
|  |  |
|  |  |
|  |  |

@Configuration

U can define all beans related to our test class in a separate configuration class

@Configuration  
@ComponentScan("sfg.springframework.samples.petclinic.dummyComponents")  
public class MyConfig {  
  
 @Bean  
 public TCSVendorImpl getVendorBean(){  
 return new TCSVendorImpl();  
 }  
  
}

@ContextConfiguration / @SpringJUnitConfig

|  |  |
| --- | --- |
| junit 4 | junit 5  @ExtendWith(SpringExtension.class) – is used to integrate with spring test context with the JUnit 5 Jupiter test. |
| @RunWith(SpringRunner.class)  @ContextConfiguration(classes= EmployeeController.class) | @SpringJunitConfig == @ExtendWith(SpringExtension.class)+@ContextConfiguration()  So this @SpringJunitConfig is the combination of 2 annotations  Sometimes if container is not getting started means u might be using @Test anno from junit4 , so switch to junit5 @Test it will perfectly work & container will also start  @SpringJUnitConfig(classes = {InnerClassconfigDemo.InnerClassConfig.class}) |

This annotation will create the spring container with the provided beans, Once spring container is started, you can happily autowire the application context and u can autowire Environment class as below

@ContextConfiguration(classes = {TCSVendorImpl.class, IBMVendorImpl.class}) this means spring container will be created only with these 2 classes/beans

Actual code for junit 4

@RunWith(SpringRunner.class)  
@ContextConfiguration(classes = {TCSVendorImpl.class, IBMVendorImpl.class})

// in above we can give any spring annotated class, These are not config classes, these are just stereo type , we can give @Configuration classes also

//This is mandatory else container will be failed to start as beans will not be there in that context, so we have to tell some spring bean location

public class ProjectExecutionTest {  
  
 @Autowired  
 TCSVendorImpl **vendor**;  
  
 @Autowired  
 *ApplicationContext* **applicationContext**;  
  
 @Test  
 public void testVendorName(){  
 System.***out***.println("num of beans in context is "+**applicationContext**.getBeanDefinitionCount());  
 System.***out***.println("bean names are --> ");  
 Arrays.*stream*(**applicationContext**.getBeanDefinitionNames()).forEach(System.***out***::println);  
 System.***out***.println(**vendor**.getCompanyName());  
 }  
}

|  |  |
| --- | --- |
| @Component public interface *SoftwareMemVendor* {  public String getCompanyName(); }  @Primary public class TCSVendorImpl implements *SoftwareMemVendor*{  @PostConstruct  public void init(){  System.***out***.println("creating a bean called "+this.getClass());  }  @Override  public String getCompanyName() {  System.***out***.println("executing service clas");  return "Tata consultancy Services";  } } | public class IBMVendorImpl implements *SoftwareMemVendor*{  @PostConstruct  public void init(){  System.***out***.println("creating a bean called "+this.getClass());  }  @Override  public String getCompanyName() {  System.***out***.println("executing service clas");  return "IBM the world first";  } } |
|  |  |

In junit 5 there is a single annotation

This will load only those specific @configuration classes this is not like mocking it will start the spring container

@SpringJUnitConfig(classes (RestTemplateConfigMock.class, MongoClientConfig.class, LogExecutionTimeConfig.class, RestTemplateConfig.class))

Configure beans in inner class

Note:- the inner class must be a static class

Sometimes we many want only 2-3 beans, but in source code in 1 bean definition file we might have more bean definition, incase if we load that file, then all un-necessary beans

Also will be created,

to stop that we can write our bean definitions in either separate file or we can define those beans in static inner class

@RunWith(SpringRunner.class)  
@SpringJUnitConfig(classes = {InnerClassconfigDemo.InnerClassConfig.class})  
public class InnerClassconfigDemo {  
  
 @Configuration  
 static class InnerClassConfig{ // This must be a static class else this bean will not be created  
  
 @Bean //Because of this bean will be created  
 public TCSVendorImpl applicationName(){  
 System.***out***.println("created inner bean");  
 return new TCSVendorImpl();  
 }  
 }  
 @Autowired //above created bean will be autowired  
 TCSVendorImpl **vendor**;  
  
 @Test  
 public void testVendorName(){  
 System.***out***.println("our app name is "+**vendor**.getCompanyName());  
 }  
}

Using comp scanning

In this also we can use @ComponentScan() on top of @Configuration classes or on top of @Service class or on top of any stereo type classes

Note:- we can write the @ ComponentScan anno anywhere means either on @Configuration class or any service class

@Configuration  
@ComponentScan("sfg.springframework.samples.petclinic.dummyComponents") //Here also we can write   
public class MyConfig {  
  
}

|  |  |
| --- | --- |
| @ComponentScan("sfg.springframework.samples.petclinic.dummyComponents")  // we can write comp scan on this service class also which is not @Configuration class  public class TCSVendorImpl implements *SoftwareMemVendor*{  @PostConstruct  public void init(){  System.***out***.println("creating a bean called "+this.getClass());  }  @Override  public String getCompanyName() {  System.***out***.println("executing service clas");  return "Tata consultancy Services";  } } | @SpringJUnitConfig(TCSVendorImpl.class) public class CompScanDemo {   @Autowired  *ApplicationContext* **applicationContext**;   @Test  void name() {  System.***out***.println("Are comp scanned "+**applicationContext**.containsBean("cabService"));  } } |

Note:- don’t write this @COmponentScan on top of test class along with @SpringJunitConfig(config.class) the annotation wont work

Note:-This @ComponentScan must always be used in conjuction with @Configuration then it will work for sure

**This is a wrong procedure**

@SpringJUnitConfig(classes ={ProfilesDemoTest.StaticClassConfig.class})  
@ComponentScan("sfg.springframework.samples.petclinic.dummyComponents")  
public class ProfilesDemoTest {

### **@ActiveProfiles**

@ActiveProfiles("prod")  
 Means app will be started with prod profile , only normal beans and prod environment specific beans will be created

And other env ( dev,sit,qa)specific beans will not be created

|  |  |
| --- | --- |
| @ActiveProfiles("prod") @SpringJUnitConfig(classes ={ProfilesDemoTest.StaticClassConfig.class}) public class ProfilesDemoTest {  @Autowired  EmployeeService **employeeService**;  }  //Since the below bean is dev env related the below bean will not be created  @Component @Profile("dev") public class CabService {  @PostConstruct  public void init() {  System.***out***.println("cab service comp class has been created");  } } | @Component @Profile("prod") public class EmployeeService {  @PostConstruct  public void init() {  System.***out***.println("employee service comp class has been created");  }  public int getNumberOfEmployees(){  return 2;  } } |
|  |  |

### **Load property files or provide properties at runtime for junit**

@TestPropertySource(locations="classpath: config/application-test.yml", properties = {"ccs.mongo.connection.enabled-true", "spring.data.mongodb.authMechanism-PLAIN",

"ccs.mtls.resttemplate.enabled-true", "ccs.mtls.resttemplate. interceptors-com.wellsfargo.ccs.filter.LogInterceptor"})

Important points about loading property files

@TestPropertySource(locations="classpath: config/application-test.yml",

1. For the above as it is loading application-test.yml from classpath first it will check if that is already present in src/**test**/resources or not (priority will always be given for src/test/resources location only)
2. If it is present it will load else it will search from src/**main**/resources so make sure from which location you are loading
3. If same file is present in both the locations then it will load all the properties from src/test/resources only

//This properties test file is present in both locations & still that file will be loaded from src/test/resources folder only

@TestPropertySource(locations = "classpath:application-test1.properties")  
@SpringJUnitConfig(MyConfig.class)  
public class LoadPropertiesTest {  
  
 @Autowired  
 *ApplicationContext* **context**;  
  
 @Value("${companyName}")  
 String **companyName**;  
  
 @Test  
 void name() {  
 System.***out***.println("Loaded the value "+**companyName**);  
 }  
}

Spring MVC Test

Important note:- while testing controllers, we don’t need to start the container especially when we are using standalone setup, but If u want u can start no issue

Classes

MockHTTPSErvletRequest – which is a mock of HTTPServletRequest

MockHTTPSErvletResponse

• Standalone Setup 🡪 Very light weight - Ideal for unit tests • Tests one controller at a time • Allows for testing of controller requests and responses

• WebAppContext Setup • Loads larger context of Spring Configuration • Tests many controllers - per configuration • Allows for testing of application config

|  |  |
| --- | --- |
| Major classes |  |
| MockMVC | we can even autowire that anno but to be more specific use standalone setup |
| *MockMvcRequestBuilders* | *MockMvcRequestBuilders*.*get*("/vets.html")  **MockMvc mvc**= MockMvcBuilders.*standaloneSetup*(**ownerController**).build();//Light weight it tests only the specified controller at 1 time  No other spring controllers will be tested |
| *MockMvcResultMatchers* | *MockMvcResultMatchers*.*model*() |

Spring MVC Test uses a “fluent” API via several static imports • MockMvcRequestBuilders.\* - Builds request • MockMvcResultMatchers.\* - Create assertions against response • MockMvcBuilders.\* - Configure and build an instance of MockMvc

Testing single controller at a time

#### **Way-1 –Without starting spring container**

Actually we don’t need to start container to test controllers, but still if u want to start the context u can start

@ExtendWith(MockitoExtension.class)  
class VetControllerTest {  
  
 @Mock  
 *ClinicService* **clinicService**;

@InjectMocks  
 VetController **vetController**;

MockMvc **mockMvc**;  
 @BeforeEach  
 void setUp() {  
 **mockMvc**=MockMvcBuilders.*standaloneSetup*(**vetController**).build();  
 }  
 @Test  
 void showVetList() throws Exception {  
 BDDMockito.*given*(**clinicService**.findVets()).willReturn(Collections.*emptyList*());  
 **mockMvc**.perform(*MockMvcRequestBuilders*.*get*("/vets.html"))  
 .andExpect(*MockMvcResultMatchers*.*model*().attributeExists("vets"))  
 .andExpect(*MockMvcResultMatchers*.*status*().is2xxSuccessful())  
 .andExpect(*MockMvcResultMatchers*.*view*().name("vets/vetList"));  
 }

#### Way-2 starting the container

Actually we don’t need to start container to test controllers, but still if u want to start the context u can start

@SpringJUnitConfig(MyConfig.class)*// to create and start spring container*@ExtendWith(MockitoExtension.class) // I don’t know why we still need @EXtendWith as @SpringJunitConfig()=@ExtendWith(SpringExtension.class)+@ContextConfig  
class VetControllerTest {  
  
 @Mock   
 *ClinicService* **clinicService**;  
 @Autowired  
 *ApplicationContext* **applicationContext**;  
 @InjectMocks  
 VetController **vetController**;   
 MockMvc **mockMvc**;  
 @BeforeEach  
 void setUp() {  
 **mockMvc**=MockMvcBuilders.*standaloneSetup*(**vetController**).build();  
 }  
 @Test  
 void showVetList() throws Exception {  
 System.***out***.println("does context contains vetcontroller bean -->"+**applicationContext**.containsBean("vetController"));  
 BDDMockito.*given*(**clinicService**.findVets()).willReturn(Collections.*emptyList*());  
 **mockMvc**.perform(*MockMvcRequestBuilders*.*get*("/vets.html"))  
 .andExpect(*MockMvcResultMatchers*.*model*().attributeExists("vets"))  
 .andExpect(*MockMvcResultMatchers*.*status*().is2xxSuccessful())  
 .andExpect(*MockMvcResultMatchers*.*view*().name("vets/vetList"));  
 }

Sending data in @RequestBody

Note: - u can send object data in “GETMapping” method also

U should use **content()** to send data for @RequestBody

|  |  |
| --- | --- |
| @PostMapping("orders") @ResponseStatus(*HttpStatus*.***CREATED***) public BeerOrderDto placeOrder(  @PathVariable("customerId") UUID *customerId*, @RequestBody BeerOrderDto *beerOrderDto*){  System.***out***.println("controller is hit");  return **beerOrderService**.placeOrder(*customerId*, *beerOrderDto*); } | @Test void placeOrder() throws Exception {  String customerId = UUID.*randomUUID*().toString();  BeerOrderDto dto= new BeerOrderDto();  dto.setCustomerId(UUID.*randomUUID*()); dto.setCustomerRef("123");   **mockMvc**.perform(*MockMvcRequestBuilders*.*post*("/api/v1/customers/{customerId}/orders",customerId)  .**contentType**(MediaType.***APPLICATION\_JSON***)  .**content**(new ObjectMapper().writeValueAsString(dto))  ).andExpect(*MockMvcResultMatchers*.*status*().isCreated()); } |

|  |  |
| --- | --- |
| @RequestMapping(value = { "/vets1.html"}) public String showVetListWithParams(Person *person*,*Map*<String, Object> *model*) {  System.***out***.println("first name is "+*person*.getFirstName());  Vets vets = new Vets();  vets.getVetList().addAll(this.**clinicService**.findVets());  *model*.put("vets", vets);  return "vets/vetList"; } | @Test public void testWithParams() throws Exception {  **mockMvc**.perform(*MockMvcRequestBuilders*.*get*("/vets1.html")  .param("firstName","manideep")  .param("lastName","VV") //Here these 2 params will be binded to person in controller method  ) .andExpect(*MockMvcResultMatchers*.*status*().is2xxSuccessful())  .andExpect(*MockMvcResultMatchers*.*view*().name("vets/vetList"))  ; } |
| @RequestMapping(value = "/owners", method = *RequestMethod*.***GET***) public String processFindForm(Owner *owner*, *BindingResult result*, *Map*<String, Object> *model*) {   *// allow parameterless GET request for /owners to return all records* if (*owner*.getLastName() == null) {  *owner*.setLastName(""); *// empty string signifies broadest possible search* }   *// find owners by last name  Collection*<Owner> results = this.**clinicService**.findOwnerByLastName(*owner*.getLastName());  if (results.isEmpty()) {  *// no owners found  result*.rejectValue("lastName", "notFound", "not found");  return "owners/findOwners";  } else if (results.size() == 1) {  *// 1 owner found  owner* = results.iterator().next();  return "redirect:/owners/" + *owner*.getId();  } else {  *// multiple owners found  model*.put("selections", results);  return "owners/ownersList";  } } | @ParameterizedTest @ValueSource(ints = {0,1,2}) void processFindForm(int *owners*)throws Exception {  String lastName="vv";  Owner owner = new Owner();  owner.setAddress("HYD");  owner.setId(23);  if(*owners*==0){  BDDMockito.*given*(**clinicService**.findOwnerByLastName(lastName)).willReturn(Arrays.*asList*());  **mockMvc**.perform(*MockMvcRequestBuilders*.*get*("/owners")  .param("lastName", lastName))  .andExpect(*MockMvcResultMatchers*.*view*().name("owners/findOwners"));  }else if (*owners* == 1) {  BDDMockito.*given*(**clinicService**.findOwnerByLastName(lastName)).willReturn(Arrays.*asList*(owner));  **mockMvc**.perform(*MockMvcRequestBuilders*.*get*("/owners")  .param("lastName", lastName))  .andExpect(*MockMvcResultMatchers*.*view*().name("redirect:/owners/23"));  }else if(*owners*==2){  BDDMockito.*given*(**clinicService**.findOwnerByLastName(lastName)).willReturn(Arrays.*asList*(owner,new Owner()));  **mockMvc**.perform(*MockMvcRequestBuilders*.*get*("/owners")  .param("lastName", lastName))  .andExpect(*MockMvcResultMatchers*.*model*().attributeExists("selections"))  .andExpect(*MockMvcResultMatchers*.*view*().name("owners/ownersList"));   } } |

Asserting the validation errors

|  |  |
| --- | --- |
| @RequestMapping(value = "/owners/new", method = *RequestMethod*.***POST***) public String processCreationForm(@Valid Owner *owner*, *BindingResult result*) {  if (*result*.hasErrors()) {  System.***out***.println("form validation happenned");  return ***VIEWS\_OWNER\_CREATE\_OR\_UPDATE\_FORM***;  } else {  this.**clinicService**.saveOwner(*owner*);  return "redirect:/owners/" + *owner*.getId();  } } | @Test void processCreationForm() throws Exception {  **mockMvc**.perform(*MockMvcRequestBuilders*.*post*("/owners/new")  .param("firstName","Manideep")  .param("lastName","vv")  .param("city","nellore"))  .andExpect(*MockMvcResultMatchers*.*view*().name("owners/createOrUpdateOwnerForm"))  *//this means the method param named owner has errors* .andExpect(*MockMvcResultMatchers*.*model*().attributeHasErrors("owner"))  .andExpect(*MockMvcResultMatchers*.*model*().attributeHasFieldErrors("owner","address"))  *//This means the owner variable has a field named telephone which is having errors* .andExpect(*MockMvcResultMatchers*.*model*().attributeHasFieldErrors("owner","telephone")); } |

Sending data in path params

|  |  |
| --- | --- |
| @RequestMapping(value = "/owners/{ownerId}/edit", method = *RequestMethod*.***POST***) public String processUpdateOwnerForm(@Valid Owner *owner*, *BindingResult result*, @PathVariable("ownerId") int *ownerId*) {  if (*result*.hasErrors()) {  return ***VIEWS\_OWNER\_CREATE\_OR\_UPDATE\_FORM***;  } else {  *owner*.setId(*ownerId*);  this.**clinicService**.saveOwner(*owner*);  return "redirect:/owners/{ownerId}";  } } | @Test void processUpdateOwnerFormValid()throws Exception {  **mockMvc**.perform(*MockMvcRequestBuilders*.*post*("/owners/{ownerId}/edit",2)  .param("firstName","Manideep")  .param("lastName","vv")  .param("address","kavali")  .param("city","Hyd")  .param("telephone","98499")  )  .andExpect(*MockMvcResultMatchers*.*view*().name("redirect:/owners/{ownerId}"))  .andExpect(*MockMvcResultMatchers*.*status*().is3xxRedirection()); } |
|  |  |

Testing REST controller with JSONPath

Make sure to add spring-boot-starter-test so that it will bring all required dependencies else u will face some or other issues



Here also 2 ways are there, testing the rest controller

1. By starting the container-– this the preferred way so that we don’t need to add the custom converters by using @WebMvcTest(Contrller.class)
2. Without starting the container



To get the json output as a single string then use .andReturn() method

// Perform GET request with MockMvc

MvcResult result = mockMvc.perform(get("/api/resource/{id}", 1L))

.andExpect(status().isOk())

.andReturn();

// Parse the JSON response

String jsonString = result.getResponse().getContentAsString();

MyObject actualObject = objectMapper.readValue(jsonString, MyObject.class);

// Assert the response object fields

assertEquals(expectedObject.getId(), actualObject.getId());

assertEquals(expectedObject.getName(), actualObject.getName());

|  |  |
| --- | --- |
| @GetMapping(path = "/getPersonByName",produces = { "application/json" }) public ResponseEntity<Person> getPerson(){  Person p= new Person();p.setFirstName("mani");p.setLastName("vv");  return ResponseEntity.*of*(Optional.*of*(p)); } | @Test void getPerson() throws Exception {  *MvcResult* mvcResult = **mockMvc**.perform(*MockMvcRequestBuilders*.*get*("/api/person/getPersonByName"))  .andExpect(*MockMvcResultMatchers*.*status*().isOk())  .andReturn();  System.***out***.println(mvcResult.getResponse().getContentAsString());  } |
| //Here we are sending data in path parameters  @GetMapping(path = "/getPersonByName/{firstName}",produces = { "application/json" }) public ResponseEntity<Person> getPerson(@PathVariable String *firstName*){  Person person=new Person();  person.setFirstName(*firstName*);  return ResponseEntity.*of*(Optional.*of*(person)); } | @Test void sendingDataInPathParams() throws Exception {  String firstName="santu";  *MvcResult* mvcResult = **mockMvc**.perform(*MockMvcRequestBuilders*.*get*("/api/person/getPersonByName/{firstName}",firstName)  )  .andExpect(*MockMvcResultMatchers*.*status*().isOk())  .andExpect(*MockMvcResultMatchers*.*content*().contentType(MediaType.***APPLICATION\_JSON\_UTF8***))  .andExpect(*MockMvcResultMatchers*.*jsonPath*("$.firstName", Is.*is*(firstName)))  .andReturn();  System.***out***.println(mvcResult.getResponse().getContentAsString());  } |
| //Send data as object  // Here when in add @RequestBody then the test case will fail  @PostMapping(path = "/acceptPerson", produces = { "application/json" }) public ResponseEntity<Person> dataAsObject(Person *person*){  return ResponseEntity.*of*(Optional.*of*(*person*)); } | @Test void sendDataAsObject() throws Exception {  String firstName="santu"; String lastName="vv";  *MvcResult* mvcResult = **mockMvc**.perform(*MockMvcRequestBuilders*.*post*("/api/person/acceptPerson")  .param("firstName",firstName)  .param("lastName",lastName))  .andExpect(*MockMvcResultMatchers*.*status*().isOk())  .andExpect(*MockMvcResultMatchers*.*content*().contentType(MediaType.***APPLICATION\_JSON\_UTF8***))  .andExpect(*MockMvcResultMatchers*.*jsonPath*("$.firstName", Is.*is*(firstName)))  .andExpect(*MockMvcResultMatchers*.*jsonPath*("$.lastName",Is.*is*(lastName)))  .andReturn();  System.***out***.println(mvcResult.getResponse().getContentAsString());  } |
| @GetMapping(path = "/acceptPerson", produces = { "application/json" }) public ResponseEntity<Person> dataAsObject(Person *person*){  *person*.setBirthday(LocalDate.*now*().atStartOfDay());;  return ResponseEntity.*of*(Optional.*of*(*person*)); } | //Here we are asserting date values with help of date serialization techniques  @Test void sendDataAsObject() throws Exception {  String firstName="santu"; String lastName="vv";  *MvcResult* mvcResult = **mockMvc**.perform(*MockMvcRequestBuilders*.*get*("/api/person/acceptPerson")  .param("firstName",firstName)  .param("lastName",lastName))  .andExpect(*MockMvcResultMatchers*.*status*().isOk())  .andExpect(*MockMvcResultMatchers*.*content*().contentType(MediaType.***APPLICATION\_JSON\_UTF8***))  .andExpect(*MockMvcResultMatchers*.*jsonPath*("$.firstName", Is.*is*(firstName)))  .andExpect(*MockMvcResultMatchers*.*jsonPath*("$.lastName",Is.*is*(lastName)))  .andExpect(*MockMvcResultMatchers*.*jsonPath*("$.birthday",Is.*is*("2024-04-01T00:00:00")))  .andReturn();  System.***out***.println("controller output is "+mvcResult.getResponse().getContentAsString()); } |

#### Asserting date values

By default LocalDateTime when Rest controller is returning this it will not be serialized correctly, it will be serialized as below

{"id":null,"version":null,"createdDate":null,"lastModifiedDate":null,"firstName":"santu","lastName":null,"birthday":{"year":2024,"monthValue":4,"dayOfMonth":1,"hour":14,"minute":6,"second":58,"nano":842859900,"dayOfWeek":"MONDAY","dayOfYear":92,"month":"APRIL","chronology":{"id":"ISO","calendarType":"iso8601"}}}

|  |  |
| --- | --- |
| @GetMapping(path = "/acceptPerson", produces = { "application/json" }) public ResponseEntity<Person> dataAsObject(Person *person*){ //   *person*.setBirthday(LocalDateTime.*now*());;  return ResponseEntity.*of*(Optional.*of*(*person*)); } | public class Person extends BaseEntity {   protected String **firstName**;   protected String **lastName**;   protected LocalDateTime **birthday**;  } |

To serialize the date values correct we have to instruct mappers and set below serialization features

@BeforeEach  
 void setUp() {  
 **mockMvc**= MockMvcBuilders.*standaloneSetup*(**personController**)  
 .setMessageConverters(getMessageConverterAccToSir())  
*// .setMessageConverters(getMessageConverterAccToMe())* .build();  
  
 }

Here both way the code will work

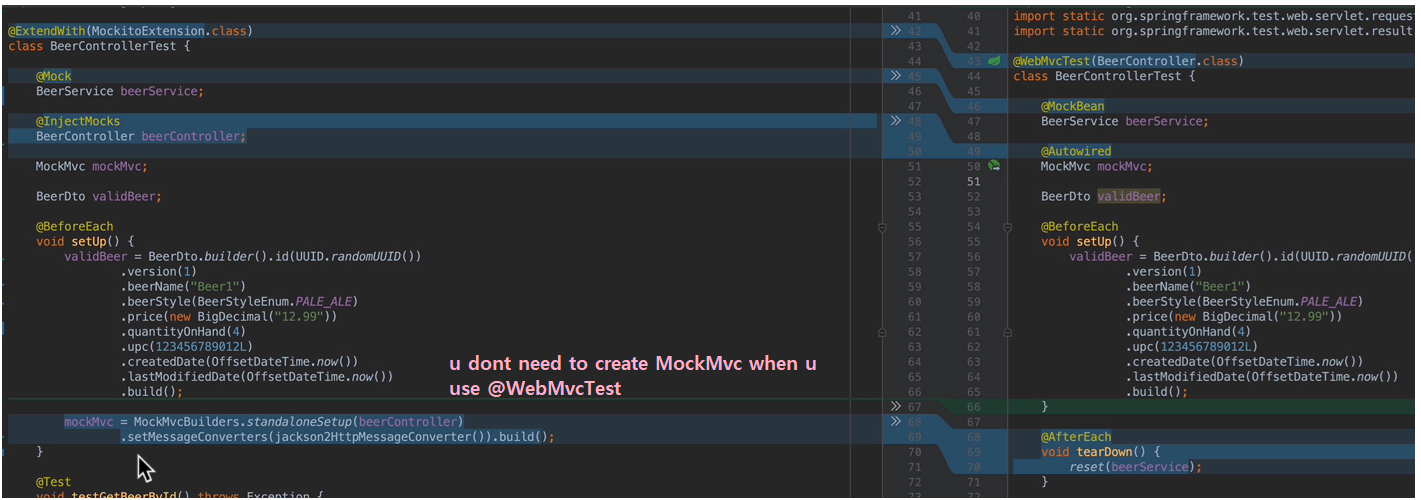
|  |  |
| --- | --- |
|  |  |
| //This code is acc to me  public MappingJackson2HttpMessageConverter getMessageConverterAccToMe(){  ObjectMapper mapper=new ObjectMapper();  *DateFormat* df= new SimpleDateFormat("YYYY-MM-dd");  mapper.setDateFormat(df);  mapper.registerModule(new JavaTimeModule());  mapper.setSerializationInclusion(JsonInclude.*Include*.***NON\_NULL***);  return new MappingJackson2HttpMessageConverter(mapper); } | This is acc to sir  public MappingJackson2HttpMessageConverter getMessageConverterAccToSir(){  ObjectMapper mapper=new ObjectMapper();  *//These 3 statements has to be un-commented as per john code, but these are working* mapper.configure(*SerializationFeature*.***WRITE\_DATES\_AS\_TIMESTAMPS***,false);  mapper.configure(*SerializationFeature*.***WRITE\_DATE\_TIMESTAMPS\_AS\_NANOSECONDS***,true);  mapper.setSerializationInclusion(JsonInclude.*Include*.***NON\_NULL***);  mapper.registerModule(new JavaTimeModule());  return new MappingJackson2HttpMessageConverter(mapper); } |

#### Testing by starting the container

1. The main advantage to start the container is you don’t need above converters to be set to mockmvc and u can autowire MockMvc directly instead of we creating it
2. We don’t need to create the mockmvc instead u can autowire the mockmvc from container ,

this is not needed 🡪 **mockMvc**= MockMvcBuilders.*standaloneSetup*(**personController**)

1. But make sure to reset the mocks using Mockito.reset(mock1,…) using @AfterEach



@WebMvcTest(PersonController.class) //This will start spring container & it is light weight as it start with only few controller beans not all beans   
class PersonControllerTestWithContainer {  
  
 @Autowired  
 MockMvc **mockMvc**;  
  
 @Test  
 void sendingDataInPathParams() throws Exception {  
 String firstName="santu";  
 *MvcResult* mvcResult = **mockMvc**.perform(*MockMvcRequestBuilders*.*get*("/api/person/getPersonByName/{firstName}",firstName)  
 )  
 .andExpect(*MockMvcResultMatchers*.*status*().isOk())  
 .andExpect(*MockMvcResultMatchers*.*content*().contentType(MediaType.***APPLICATION\_JSON\_UTF8***))  
 .andExpect(*MockMvcResultMatchers*.*jsonPath*("$.firstName", Is.*is*(firstName)))  
 .andReturn();  
 System.***out***.println(mvcResult.getResponse().getContentAsString());  
 }

}