# jahe / spring-boot-cheatsheet.java

Last active 2 months ago • Report gist

Spring Boot Cheatsheet

spring-boot-cheatsheet.java

```
// Enable component-scanning and auto-configuration with @SpringBootApplication Annotation
// It combines @Configuration + @ComponentScan + @EnableAutoConfiguration
@SpringBootApplication
public class FooApplication {
  public static void main(String[] args) {
    // Bootstrap the application
    SpringApplication.run(FooApplication.class, args);
  }
}

// @Configuration:  Marks a class as a config class using Spring's Java based configuration
// @ComponentScan:  Enables component-scanning so that web controller classes can be
//                  automatically registered as beans in the Spring application context
// @EnableAutoConfiguration: Configures the application based on the dependencies

// Build and Run the application
gradle bootRun
// OR:
gradle build
gradle -jar build/libs/readinglist-0.0.1-SNAPSHOT.jar

// Testing classes in Spring Boot
@RunWith(SpringJUnit4ClassRunner.class)
// Load context via Spring Boot
@SpringApplicationConfiguration(classes = ReadinglistApplication.class)
@WebAppConfiguration
public class ReadinglistApplicationTests {
  // Test that the context successfully loads (the method can be empty -> the test will fail if the context cannot be loaded)
  @Test
  public void contextLoads() {
  }
}

// Make the test methods transactional (here I use Spock as my Test-Framework of choice)
// After each test a rollback is triggered so that the database is in its previous state again
@SpringBootTest
@Transactional
class MySpec extends Specification {

  @Autowired
  MyRepository myRepo

  def "Persist an entity"() {
    given:
    MyEntity entity = new MyEntity()

    when:
    myRepo.saveAndFlush(entity)

    then:
    myRepo.count() == 1
  }

  def "Persist another entity"() {
    given:
    MyEntity entity = new MyEntity()

    when:
    myRepo.saveAndFlush(entity)

    then:
    myRepo.count() == 1
  }
}

// application.properties is optional
// Configure the embedded tomcat server so listen on port 8081
server.port=8081

// List all libs with its version
gradle dependencies

// Inject the dependencies in the constructor function of a MVC Controller
// to show the dependent components of the class and to make the testing easier:
// The constructor can be called with an implementing mockup Repository for testing purposes
@Controller
@RequestMapping("/")
public class UserController {

  private UserRepository userRepository;

  @Autowired
  public UserController(UserRepository userRepository) {
    this.userRepository = userRepository;
  }
}

// Defining Condition that checks if the JdbcTemplate is available on the classpath
//
// Conditions are used by the auto-configuration mechanism of Spring Boot
// There are several configuration classes in the spring-boot-autoconfigure.jar
// which contribute to the configuration if specific conditions are met
public class JdbcTemplateCondition implements Condition {
  @Override
  public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
    try {
      context.getClassLoader().loadClass("org.springframework.jdbc.core.JdbcTemplate");
      return true;
    } catch (Exception e) {
      return false;
    }
  }
}

// Use a custom condition class to decide whether a Bean should be created or not
@Conditional(JdbcTemplateCondition.class)
public class MyService {
  ...
}

// Overriding Spring Boots auto-configuration for example the Spring Security configuration
// Therefore a specific Configuration class has to be in the classpath
// For Spring Security its the WebSecurityConfigurerAdapter.
// Spring then skips the Spring Security auto-configuration and uses the custom configuration instead.
// This class has to be extended and annotated with @Configuration so that it can be found
// by the component scan and registers it as a bean in the Spring application context.
// In addition there has to be a @EnableWebSecurity annotation for this class to enable Spring Security.

// The list with Auto Configuration classes
spring-boot-autoconfigure.jar -> spring.factories

// Generate report on application startup to the console about what configuration classes are being used
// With a VM parameter
-Ddebug

// OR in the application.properties
debug=true

// Integration test by loading Springs application context
// To to integration testing with Spring, all components of the application have to be configured and wired up.
// Instead of doing this by hand we can use Spring's SpringJUnit4ClassRunner.
// It helps load a Spring application context in JUnit-based application tests.
// This method with the @ContextConfiguration annotation doesn't apply extenal properites (application.properties) and logging
// @ContextConfiguration specifies how to load the application context: A configuraiton class is passed to it as a parameter
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=PlaylistConfiguration.class)
public class PlaylistServiceTests {

  @Autowired
  private PlaylistService playlistService;

  @Test
  public void testService() {
    Playlist playlist = playlistService.findByName("X-Mas Songs");
    assertEquals("X-Mas Songs", playlist.getName());
    assertEquals(12, playlist.countSongs());
  }
}

// Integration test by loading application context + external properties + logging
// Replace the @ContextConfiguration with @SpringApplicationConfiguration
// This loads the application just like the application context would be loaded by using SpringApplication
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes=PlaylistConfiguration.class)
public class PlaylistServiceTests {
  ...
}

// Test controller classes
//
// > Either by mocking the servlet container and without starting an application server
// > Or by starting the embedded servlet container (e.g. tomcat) and exercise tests in a real application server

// Test controller classes with Spring's Mock MVC framework
//
// First create a MockMvc Object with the MockMvcBuilders
// standaloneSetup()    - Builds a Mock MVC to serve one ore more manually created controllers
//                        so that the controller instances have to be instantiated manually.
//                        It is more like a unit test for very focused tests around a single controller.
// webAppContextSetup() - Builds a Mock MVC using a Spring application context which includes one ore more controllers
//                        using an instance of WebApplicationContext.
//                        Spring will load the controllers as well as their dependencies.
//                        Therefore the test class has to be annotated with @WebAppConfiguration
//                        to declare that the application context created by the SpringJUnit4ClassRunner
//                        should be an WebApplicationContext and not the basic non-web ApplicationContext.
//                        The webAppContextSetup() method takes an instance of the WebApplicationContext as a parameter.
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = PlaylistApplication.class)
@WebAppConfiguration
public class MockMvcWebTests {
  @Autowired
  private WebApplicationContext webContext;

  private MockMvc mockMvc;

  @Before
  public void setupMockMvc() {
    mockMvc = MockMvcBuilders
      .webAppContextSetup(webContext)
      .build();
  }

  @Test
  public void playlist() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.get("/playlist"))
      .andExpect(MockMvcResultMatchers.status().isOk())
      .andExpect(MockMvcResultMatchers.view().name("playlist"))
      .andExpect(MockMvcResultMatchers.model().attributeExists("songs"))
      .andExpect(MockMvcResultMatchers.model().attribute("songs",
      Matchers.is(Matchers.empty())));
  }
}

// The playlist() method can be rewritten with static imports
@Test
public void playlist() throws Exception {
  mockMvc.perform(get("/playlist"))
    .andExpect(status().isOk())
    .andExpect(view().name("playlist"))
    .andExpect(model().attributeExists("songs"))
    .andExpect(model().attribute("songs", is(empty())));
}

// Test method with HTTP POST request
@Test
public void postSong() throws Exception {
  mockMvc.perform(post("/playlist"))
    .contentType(MediaType.APPLICATION_FORM_URLENCODED)
    .param("interpret", "OutKast")
    .param("title", "Hey Ya!")
    .andExpect(status().is3xxRedirection())
    .andExpect(header().string("Location", "/playlist"));

  // Create expected song
  Song expectedSong = new Song();
  expectedSong.setId(1L);
  expectedSong.setInterpret("OutKast");
  expectedSong.setTitle("Hey Ya!");

  // Check if new song is in playlist
  mockMvc.perform(get("/playlist"))
    .andExpect(status().isOk())
    .andExpect(view().name("playlist"))
    .andExpect(model().attributeExists("songs"))
    .andExpect(model().attribute("songs", hasSize(1)))
    .andExpect(model().attribute("songs",
    contains(samePropertyValuesAs(expectedSong))));
}

// Testing with Spring Security
// First add the testCompile dependency
testCompile("org.springframework.security:spring-security-test")

// Apply the Spring Security configurer when creating the MockMvc instance
// SecurityMockMvcConfigurers.springSecurity() - returns a Mock MVC configurer that enables Spring Security for Mock MVC
@Before
public void setupMockMvc() {
  mockMvc = MockMvcBuilders
    .webAppContextSetup(webContext)
    .apply(springSecurity())
    .build();
}

// Test without being authenticated
@Test
public void unauthenticated() throws Exception() {
  mockMvc.perform(get("/"))
    .andExpect(status().is3xxRedirection())
    .andExpect(header().string("Location",
    "http://localhost/login"));
}

// There are two ways to use an authenticated user for the tests
// @WithMockUser - Loads the security with a UserDetails using the given username, password and authorization
// @WithUserDetails - Loads the security context by looking up a UserDetails object for the given username
// This UserDetails object is being used for the duration of the test method

// Bypassing the normal lookup of a UserDetails object and instead create one
@Test
```

```
@WithMockUser(
  username="clark",
  password="kent123",
  roles="USER"
)
public void authenticatedUser() throws Exception {
  ...
}

// Using a real user from a UserDetailsService
@Test
@WithUserDetails("clark")
public void authenticatedUser() throws Exception {
  PlaylistOwner expectedPlaylistOwner = new PlaylistOwner();
  expectedPlaylistOwner.setUsername("clark");
  expectedPlaylistOwner.setPassword("kent123");
  expectedPlaylistOwner.setFullname("Clark Kent");

  mockMvc.perform(get("/"))
    .andExpect(status().isOk())
    .andExpect(view().name("playlist"))
    .andExpect(model().attribute("owner",
      samePropertyValuesAs(expectedPlaylistOwner)))
    .andExpect(model().attribute("songs", hasSize(0)));
}

// Test with a real application server (embedded tomcat)
// @WebIntegrationTest declares that you not only want an application context
// but also to start an embedded servlet container
// You can use Spring's RestTemplate to perform HTTP requests against the application
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = PlaylistApplication.class)
@WebIntegrationTest
public class RealWebTest {

  @Test (expected=HttpClientErrorException.class)
  public void pageNotFound() {
    try {
      RestTemplate rest = new RestTemplate();
      // Perform GET request
      rest.getForObject("http://localhost:8080/ladida", String.class);
      fail("Should result in HTTP 404");
    } catch (HttpClientErrorException e) {
      assertEquals(HttpStatus.NOT_FOUND, e.getStatusCode());
      throw e;
    }
  }
}

// Start the server an a random port with "random=true" and inject actual port value
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = PlaylistApplication.class)
@WebIntegrationTest(randomPort=true)
public class RealWebTest {

  @Value("${local.server.port}")
  private int port;

  @Test (expected=HttpClientErrorException.class)
  public void pageNotFound() {
    ...
    rest.getForObject("http://localhost:{port}/ladida", String.class, port);
    ...
  }
}

// Test Frontend with Selenium
// First add Selenium as a testCompile dependency
testCompile("org.seleniumhq.selenium:selenium-java:2.52.0")
```

```
// Write a test class with a FirefoxDriver
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = PlaylistApplication.class)
@WebIntegrationTest(randomPort=true)
public class SeleniumWebTest {

  private static FirefoxDriver browser;

  @Value("${local.server.port}")
  private int port;

  @BeforeClass
  public static void openBrowser() {
    browser = new FirefoxDriver();
    browser.manage().timeouts()
      .implicitlyWait(10, TimeUnit.SECONDS);
  }

  @AfterClass
  public static void closeBrowser() {
    browser.quit();
  }

  @Test
  public void addSongToEmptyPlaylist() {
    String baseUrl = "http://localhost:" + port;

    browser.get(baseUrl);

    assertEquals("You have no songs in your playlist",
    browser.findElementByTagName("div").getText());

    browser.findElementByName("interpret")
      .sendKeys("OutKast");
    browser.findElementByName("title")
      .sendKeys("Hey Ya!");
    browser.findElementByTagName("form")
      .submit();

    WebElement dl = browser.findElementByCssSelector("dt.songHeadline");
    assertEquals("OutKast - Hey Ya!", dl.getText());

    WebElement dt = browser.findElementByCssSelector("dd.songTitle");
    assertEquals("Hey Ya!", dt.getText());
  }
}

// Execute Code on Startup (and refresh) of the application
@Component
public class MyListener implements ApplicationListener<ApplicationReadyEvent> {

  @Override
  public void onApplicationEvent(ApplicationReadyEvent event) {
    // doStuff();
  }
}

// Run Flyway migrations on In-Memory DB for an integration test (written in Groovy with Spock)
@SpringBootTest
@AutoConfigureTestDatabase
@ImportAutoConfiguration(FlywayAutoConfiguration.class)
@TestPropertySource(properties = [
        "flyway.enabled=true",
        "spring.jpa.hibernate.ddl-auto=none"
])
class MySpec extends Specification {
  def "foo"() {
    // do something...
  }
```

```

// Force a fresh version of the Spring context before each test method executes
@SpringBootTest
@DirtiesContext(classMode = ClassMode.BEFORE_EACH_TEST_METHOD)
class MySpec extends Specification {
  def "foo"() {
    // do something...
  }
}

// Application events are sent in the following order, as your application runs:
1. ApplicationStartedEvent is sent at the start of a run, but before any processing except the registration of listeners and initializers.
2. ApplicationEnvironmentPreparedEvent is sent when the Environment to be used in the context is known, but before the context is created.
3. ApplicationPreparedEvent is sent just before the refresh is started, but after bean definitions have been loaded.
4. ApplicationReadyEvent is sent after the refresh and any related callbacks have been processed to indicate the application is ready to se
5. ApplicationFailedEvent is sent if there is an exception on startup.

// Configure Loglevel from Lombok's @Slf4j Annotation via application.properties
@SpringBootApplication
@Slf4j
public class MyApp {
  public static void main(String[] args) {
    SpringApplication.run(MyApp.class, args);
    log.info("testing logging with lombok");
  }
}

// application.properties
logging.level.com.example.MyApp=WARN

// Recommended structuring of a Spring Boot application
com
 +- example
     +- myproject
         +- Application.java
         |
         +- domain // Entities + Repos
         |   +- Customer.java
         |   +- CustomerRepository.java
         |
         +- service
         |   +- CustomerService.java
         |
         +- web
             +- CustomerController.java
```