Testing In SpringBoot

- 2 separate styles of testing:

  1. junit testing
  2. integration testing

- Integration test -> any test that need access to ApplicationContext

Add Dependencies

- Add in pom.xml
- Include devtools

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

- A project from start.spring.io already contains this

A Test project

```
@SpringBootTest
@RunWith(SpringRunner.class)
public class ApplicationContextTests {

    @Autowired
    private ApplicationContext applicationContext;

    @Test
    public void contextLoads() throws Throwable {
        assertThat("the application context should have loaded."
                   , applicationContext
                   , is(notNullValue()));
    }
}
```

Anatomy of Test

1. 2 annotations
    1. @RunWith -> tells Junit test runner strategy to use
        1. SpringRunner.class -> provides generic test support
    2. @SpringBootTest -> Marks class as a SpringBoot test
        1. provide support to scan for ApplicationContext
        2. default it looks for @SpringBootConfiguration

SpringBootConfiguration

- Remember:

@SpringBootConfiguration

```
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
  @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.c
  @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationEx
public @interface SpringBootApplication
```

mvn project stucture

```
.
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    ├── main ❶
    │   ├── java
    │   │   └── com
    │   │       └── example
    │   │           └── Application.java
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test ❷
        └── java
            └── com
                └── example
                    └── ApplicationTests.java
```

- sourceroots test & main should be identical to find @SpringBootConfiguration

Integration Testing

- Autoconfiguration can switch external dependencies for mocked dependencies

Test slices

- selective activation of slices of auto-configuration of distinct layers of the stack

Mocking

- Mock only selected components in ApplicationContext
- Enables testing collaboration components while still being able to mock out object at the boundary of the application

@MockBean

- instruct Spring to create a Mockito Mock in the ApplicationContext
- effectively mutes the definition of the original

@MockBean example1

```
@Service
public class AccountService {
 private final AccountRepository accountRepository;
 private final UserService us;
 @Autowired
 public AccountService(AccountRepository ar, UserService us) {
    this.accountRepository = ar;
    this.us = us;
 }
 public List<Account> getUserAccounts() {
   return Optional.ofNullable(us.getAuthenticatedUser())
        .map(u -> accountRepository
                    .findAccountsByUsername(u.getUsername()))
        .orElse(Collections.emptyList());
   }
}
```

@MockBean example2

```
@RunWith(SpringRunner.class)
public class AccountServiceTests {
 @MockBean private UserService us;
 @MockBean private AccountRepository ar;
 private AccountService ac;
 @Test
 public void getUserAccountsReturnsSingleAccount(){
 ac=new AccountService(ar,us);
 given(this.ar.findA...).willReturn(....);
 given(this.us.getA...).willReturn(....);
 List<Account> actual = ac.getUserAccounts();
 assertThat(actual).size().isEqualTo(1);
 }
}
```

Using Constructor Injection

```
@Autowired
public AccountService(AccountRepository ar, UserService us) {
   this.accountRepository = ar;
   this.us = us;
}
```

- you can use field injection but that would mask the ability to understand the preconditions in order to construct a valid object
- always use constructor injection instead of field injection

Remote HTTP call UserService

```
@Service
public class UserService {
private final String sh = "my-server";
private final RestTemplate restTemplate;

@Autowired public UserService(RestTemplate restTemplate) {
 this.sh = sh;
 this.restTemplate = restTemplate;
}
public User getAuthenticatedUser() {
 URI url = URI.create(String.format("http://%s/ua/v1/me", sh));
 RequestEntity<Void> request = get(url).header(...).build();
 return restTemplate.exchange(request, User.class).getBody();
}
}
```

- the @MockBean for UserService isolates AccountService

The ServletContainer in SpringBootTests

- In a majority of cases the full blown servlet envirionment is overkill during integration testing
- In a continuous delivery worl of microservices build time can be precious

Slices

- multiple testing annotations are available to that target a specific slice in your application

@WebMvcTest

- Testing individual MVC Controllers
- autoconfigures the necessary spring mvc infrastructure to test interaction with controllers

The AccountController

```
@RestController
@RequestMapping(path = "/v1")
public class AccountController {
 private AccountService accountService;
 @Autowired
 public AccountController(AccountService accountService) {
  this.accountService = accountService;
 }
 @RequestMapping(path = "/accounts")
 public ResponseEntity getUserAccounts() throws Exception {
  return Optional.ofNullable(accountService.getUserAccounts())
   .map(a -> new ResponseEntity<List<Account>>(a, HttpStatus.OK))
   .orElseThrow(() -> new Exception("Accounts don't exist"));
 }
}
```

A AccountController test

```
@RunWith(SpringRunner.class)
@WebMvcTest(AccountController.class)
public class AccountControllerTest {
 @Autowired private MockMvc mvc;
 @MockBean private AccountService accountService;
 @Test
 public void getUserAccountsShouldReturnAccounts(){
  String content = "[{\"username\": \"user\"," +
                   "\"accountNumber\": \"123456789\"}]";
  given(accountService.getUserAccounts()).willReturn(
   Collections.singletonList(new Account("user", "123456789")));
     mvc.perform(
        get("/v1/accounts").accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(content().json(content));
 }
```

Mock the MVC client

- @Autowired private MockMvc mvc; represents the client that calls our controller
- Inside the test: this.mvc.perform() represents the client issuing a request
- perform expects a request -> get("/v1/accounts").accept(MediaType.APPLICATION_JSO
- perform returns something of type ResultActions

ResultActions

```
public interface ResultActions {

    /**
     * Perform an expectation.
     *
     * <h4>Example</h4>
     * <pre class="code">
     * static imports: MockMvcRequestBuilders.*, MockMvcResultMatchers.*
     *
     * mockMvc.perform(get("/person/1"))
     *   .andExpect(status().isOk())
     *   .andExpect(content().contentType(MediaType.APPLICATION_JSON))
     *   .andExpect(jsonPath("$.person.name").value("Jason"));
     *
     * mockMvc.perform(post("/form"))
     *   .andExpect(status().isOk())
     *   .andExpect(redirectedUrl("/person/1"))
     *   .andExpect(model().size(1))
     *   .andExpect(model().attributeExists("person"))
     *   .andExpect(flash().attributeCount(1))
     *   .andExpect(flash().attribute("message", "success!"));
     * </pre>
     */
    ResultActions andExpect(ResultMatcher matcher) throws Exception;
```

@DataJpaTest

- only autconfiguration classes required for executing test for spring data JPA are activated under this slice
- also TestEntityManager is available:

  1. has extra utility methods
  2. allows to interact with datastore without needing a repository

A Repository Test

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class AccountRepositoryTest {
 @Autowired private AccountRepository ar;
 @Autowired private TestEntityManager em;
 @Test
 public void findUserAccountsShouldReturnAccounts() {
  this.em.persist(new Account("Hans"));
  List<Account> account = this.ar.findAccountsByUsername("Hans");
  assertThat(account.size(),is(1);
  Account actual = account.get(0);
  assertThat(actual.getUsername(),is("Hans");
 }
```

RestClientTest

- provides support for Spring Rest Client
- the UserService (next slide) uses a RestTemplate to call an external service

The UserService

```
@Service public class UserService {
 private final String sh="myapp";
 private final RestTemplate restTemplate;
 @Autowired public UserService(RestTemplate restTemplate){
  this.sh = sh;
  this.restTemplate = restTemplate;
 }
 public User getAuthenticatedUser() {
  URI url = URI.create(String.format("http://%s/ua/v1/me", sh));
  RequestEntity<Void> request = get(url).
   header(HttpHeaders.CONTENT_TYPE,APPLICATION_JSON_VALUE).build(
  return restTemplate.exchange(request, User.class).getBody();
 }
}
```

The UserServiceTest

```
@RunWith(SpringRunner.class)
@RestClientTest({ UserService.class })
@AutoConfigureWebClient(registerRestTemplate = true)
public class UserServiceTests {
 private String sh="myapp";
 @Autowired private UserService us;
 @Autowired private MockRestServiceServer server;
 @Test
 public void getAuthenticatedUserShouldReturnUser() {
  this.server.expect(
   requestTo(String.format("http://%s/..", sh))).andRespond(
   withSuccess(getResource("user.json"), APPLICATION_JSON));
  User user = us.getAuthenticatedUser();
  assertThat(user.getUsername()).isEqualTo("user");
 }
}
```

The Test explained

```
@RestClientTest({ UserService.class })
@AutoConfigureWebClient(registerRestTemplate = true)
```

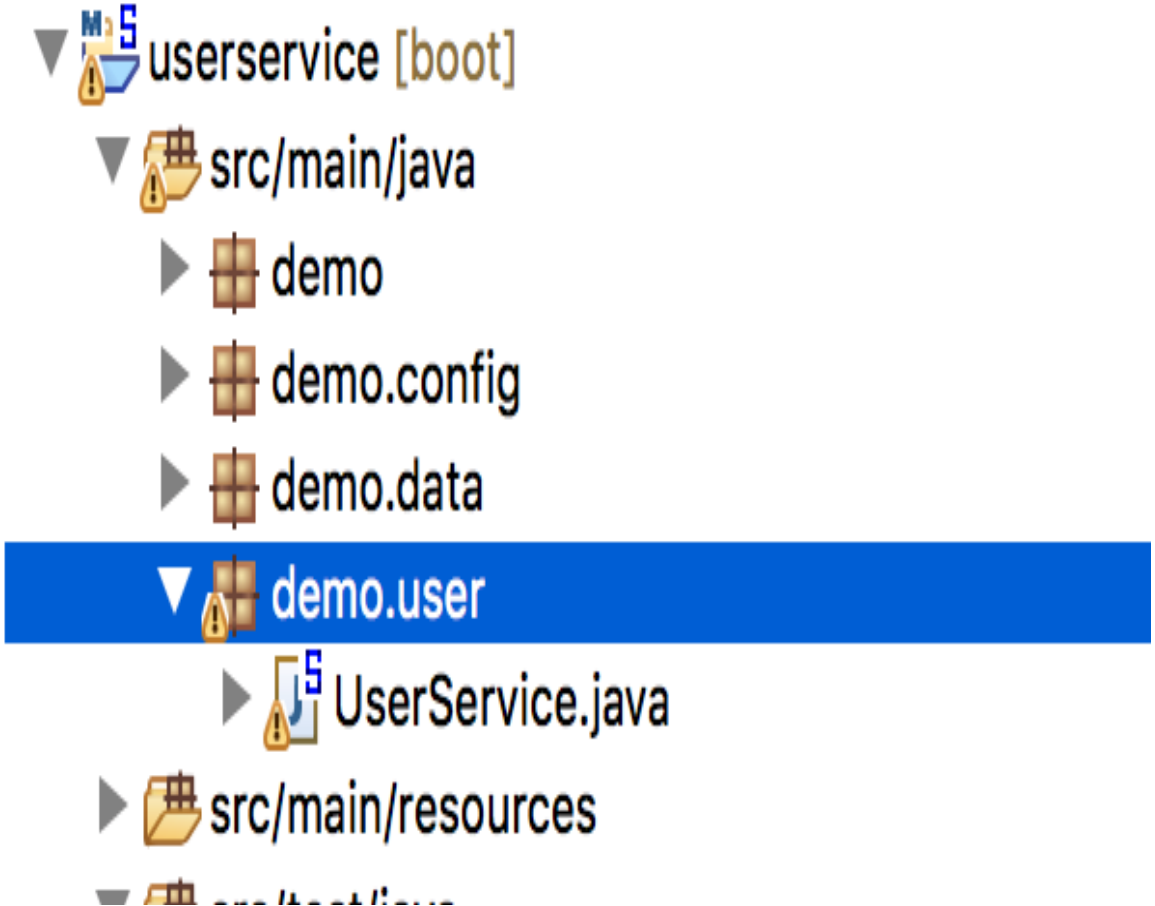
1. Isolate the UserService class for Rest interaction
2. Autoconfigurable also makes a MockRestServerService type available

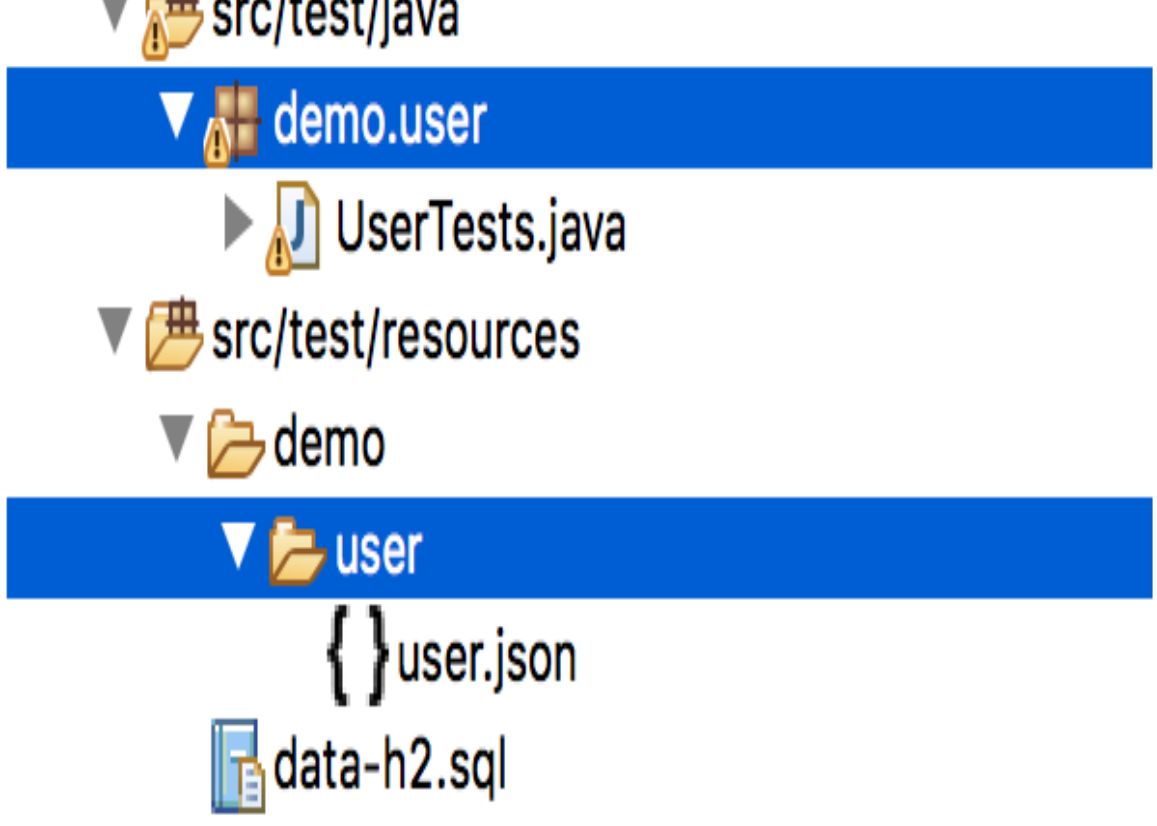
```
this.server.expect(
requestTo(String.format("http://%s/..", sh))).andRespond(
withSuccess(getResource("user.json"), APPLICATION_JSON));
```

1. The mockserver is told how to respond when a certain request comes in
2. The json response is read from file

The json response

- getResource("user.json") -> private utility method to get json expectation from file




src/test/java
demo.user
UserTests.java
src/test/resources
demo
user
user.json
data-h2.sql

The json user

user.json

```
1  {
2    "username": "user",
3    "firstName": "Rico",
4    "lastName": "Post",
5    "email": "rpost@example.com",
6    "createdAt": 12345,
7    "lastModified": 12346,
8    "id": 0
9  }
10
```

End to End Testing

- Focus on validating application businessfeatures
- From the user perspective

Testing distributed systems

- often involves shared state between different microservices
- design end to end test for data consistency
- when data is shared over microservices eventually consistency is what to achieve