Ultimate Kotlin Presentation Notes

Christoph Pickl - 28.3.2018 - Austria, Linz

Project Setup

1. Browse **https://start.spring.io** and create new project (Gradle + Kotlin).
2. Update the following two files first:
    1. Change Gradle version in `gradle-wrapper.properties` to: **gradle-4.6-all.zip**
    2. Change Kotlin version in `build.gradle` to: **1.2.31**
3. Create a new IntelliJ project via **idea** command (or via the UI).
4. Run the application to verify everything is set up correctly.

Ping Pong

▷ Add a dependency to run web apps with Spring Boot.

```
compile('org.springframework.boot:spring-boot-starter-web')

// remove: "apply plugin: 'eclipse'"
```

▷ Write'n'run the test first.

```
@RunWith(SpringRunner::class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class PingTest {

    @Autowired
    private lateinit var rest: TestRestTemplate

    @Test
    fun `When GET ping as text Then pong text returned`() {
        val response = rest.exchange<String>(
            RequestEntity.get(URI.create("/ping"))
                .header("accept", "text/plain").build())

        assertThat(response.statusCode).isEqualTo(HttpStatus.OK)
        assertThat(response.body).isEqualTo("pong")
    }

}
```

▷ Implement the controller and re-run the test.

```
@RestController
@RequestMapping("/ping")
class PingController {

    @GetMapping(produces = ["text/plain"])
    fun pingPlain() = "pong"

}
```

▶ Write'n'run test for the JSON response.

```
@Test
fun `When GET ping accepting JSON Then JSON payload is returned`() {
    val response = rest.exchange(
            RequestEntity.get(URI.create("/ping"))
                    .header("accept", "application/json")
                    .build(),
            String::class.java
    )

    assertThat(response.body).isEqualTo("""{"message":"pong"}""")
}
```

▶ Extend the controller and re-run the test.

```
data class Pong(
        val message: String
)

@RestController
@RequestMapping("/ping")
class PingController {

    // ...

    @GetMapping(produces = ["application/json"])
    fun pingJson() = Pong("pong")

}
```

▶ Fix warning about missing `jackson-module-kotlin`.

```
2018-03-26 17:15:18.359  WARN 3771 --- [kground-preinit]
o.s.h.c.j.Jackson2ObjectMapperBuilder    : For Jackson Kotlin classes support
please add "com.fasterxml.jackson.module:jackson-module-kotlin" to the classpath

compile("com.fasterxml.jackson.module:jackson-module-kotlin")
```

▶ Fine-tune the application's log output in the `application.properties` file.

```
spring.main.banner-mode=off

logging.level.root=WARN
logging.level.XXXXXXXXX=TRACE
```

Accounts

▷ Add an account model first.

```
data class Account(
        val id: Long,
        val alias: String,
        val balance: Int,
        val type: AccountType
)
enum class AccountType {
    CURRENT,
    SAVING
}
```

GET /accounts

▷ Add the test first which checks for an empty list.

```
@RunWith(SpringRunner::class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class AccountControllerTest {
    @Autowired
    private lateinit var rest: TestRestTemplate
    @Test
    fun `GET accounts - Then return 200 and empty list`() {
        val response = rest.exchangeGet<List<Account>>("/accounts")
        assertThat(response.statusCode).isEqualTo(HttpStatus.OK)
        assertThat(response.body).isEmpty()
    }
    private inline fun <reified T : Any> TestRestTemplate.
        exchangeGet(url: String): ResponseEntity<T> =
            exchange(RequestEntity.get(URI.create(url)).build())
}
```

▷ Add accounts controller and re-run test.

```
@RestController
@RequestMapping("/accounts", produces = [MediaType.APPLICATION_JSON_VALUE])
class AccountController {
    @GetMapping
    fun getAccounts(): List<Account> = emptyList()
}
```

▷ Implement simple service and wire-up in controller.

```
@Service
class AccountService {
    val accountsById = mutableMapOf<Long, Account>()
    fun getAccounts(): List<Account> = accountsById.values.toList()
}
class AccountController(
    private val service: AccountService
) {
    @GetMapping
    fun getAccounts(): List<Account> = service.getAccounts()
    // ...
```

▶ Test for returning existing accounts.

1. Inject `AccountService` into test.
2. Add reusable test model as `Account.testInstance()`.
3. Extend the test.

```
@Test
fun `GET accounts - Given an existing account Then return that account`() {
    val savedAccount = saveAccount()
    val response = rest.exchangeGet<List<Account>>("/accounts")
    assertThat(response.body).containsExactly(savedAccount)
}

private fun saveAccount(): Account =
        Account.testInstance().also {
            service.accountsById[it.id] = it
        }

@Before
fun `reset datastore`() {
    service.accountsById.clear()
}
```

GET /account/{id}

▶ Write'n'run test first as usual.

```
@Test
fun `GET account - When GET non-existing account by its ID Then return 404`() {
    val response = rest.exchangeGet<Any>("/accounts/$notExistingId")

    assertThat(response.statusCode).isEqualTo(HttpStatus.NOT_FOUND)
}

@Test
fun `GET account - Given an existing account When GET that account by its ID
Then return that account`() {
    val savedAccount = saveAccount()

    val response = rest.exchangeGet<Account>("/accounts/${savedAccount.id}")

    assertThat(response.body).isEqualTo(savedAccount)
}
```

▶ Extend controller and add simple service method.

```
// ad controller:
@GetMapping(path = ["/{id}"])
fun getAccount(
        @PathVariable
        id: Long
): ResponseEntity<Account> {
    val found = service.getAccount(id)
    return if (found != null) ResponseEntity.ok(found)
    else ResponseEntity.notFound().build()
}

// ad service:
fun getAccount(id: Long): Account? = accountsById[id]
```

POST /account

- Test first as usual.

```
@Test
fun `POST account - Then return that account with new ID and persist it`() {
    val response = rest.exchange<Account>(
                  RequestEntity.post(URI.create("/accounts")).body(anyAccount))

    val expectedAccount = anyAccount.copy(id = response.body!!.id)
    assertThat(response.body).isEqualTo(expectedAccount)
    persistedAccountsContainsExactly(expectedAccount)
}

private fun persistedAccountsContainsExactly(account: Account) {
    assertThat(service.accountsById.values).containsExactly(account)
}
```

- Write implementation (controller + service).

```
// ad controller:
@PostMapping(consumes = [MediaType.APPLICATION_JSON_VALUE])
fun postAccount(
        @RequestBody
        account: Account
): Account = service.createAccount(account)

// ad service:
fun createAccount(account: Account): Account =
                                        account.also { accountsById[it.id] = it }
```

Persistence

▹ Add new required dependencies.

```
compile("org.springframework.boot:spring-boot-starter-data-jpa")
compile("com.h2database:h2:1.4.196")
```

Persistence Layer

▹ Implement JPA entity and repository.

```
@Entity
data class AccountJpa(
    @Id @GeneratedValue
    val id: Long,
    val alias: String,
    val balance: Int,
    val type: AccountTypeJpa
) {
    companion object
}

enum class AccountTypeJpa {
    CURRENT,
    SAVING
}

@Repository
interface AccountRepository : JpaRepository<AccountJpa, Long>
```

▹ Write'n'run the test.

```
@RunWith(SpringRunner::class)
@DataJpaTest
class AccountRepositoryTest {

    @Autowired
    private lateinit var em: TestEntityManager

    @Autowired
    private lateinit var repo: AccountRepository

    @Test
    fun `Given an account When find by ID Then return that account`() {
        val saved = em.persist(AccountJpa.unsavedTestInstance())

        val found = repo.findById(saved.id)

        assertThat(found).hasValue(saved)
    }
}
```

Service Layer

▷ Wire up service layer and add transformation logic.

```
@Service
class AccountService(
        private val repo: AccountRepository
) {
    fun getAccounts(): List<Account> = repo.findAll().map { it.toAccount() }
    fun getAccount(id: Long): Account? = repo.findById(id).unwrap()?.toAccount()
    fun createAccount(account: Account): Account =
                      repo.save(account.toAccountJpa().copy(id = 0)).toAccount()
}

fun <T> Optional<T>.unwrap(): T? = if (isPresent) get() else null

fun AccountJpa.toAccount() = Account(id, alias, balance, type.toAccountType())
fun Account.toAccountJpa() = AccountJpa(
                                    id, alias, balance, type.toAccountTypeJpa())
fun AccountTypeJpa.toAccountType() = when (this) {
    AccountTypeJpa.CURRENT -> AccountType.CURRENT
    AccountTypeJpa.SAVING -> AccountType.SAVING
}
fun AccountType.toAccountTypeJpa() = when (this) {
    AccountType.CURRENT -> AccountTypeJpa.CURRENT
    AccountType.SAVING -> AccountTypeJpa.SAVING
}
```

▷ Adapt the controller test.

```
@DirtiesContext(classMode = DirtiesContext.ClassMode.BEFORE_EACH_TEST_METHOD)
class AccountControllerTest {

    private fun persistedAccountsContainsExactly(account: Account) {
        assertThat(repo.findAll()).containsExactly(account.toAccountJpa())
    }

    private fun saveAccount(): Account =
        repo.save(Account.testInstance().copy(id = 0L).toAccountJpa())
            .toAccount()

    // delete `reset database`
}
```

▷ Fix the no-argument consructor error message. (**ATTENTION**: Rebuild the project in IDEA!)

```
Caused by: org.hibernate.InstantiationException: No default constructor for
entity:  : com.github.christophpickl.ultimatesteps.account.AccountJpa

classpath "org.jetbrains.kotlin:kotlin-noarg:${kotlinVersion}"
apply plugin: 'kotlin-jpa'
```

▷ Fix the MVC warning message.

```
2018-03-28 15:49:15.753  WARN 20368 --- [           main]
aWebConfiguration$JpaWebMvcConfiguration : spring.jpa.open-in-view is enabled by
default. Therefore, database queries may be performed during view rendering.
Explicitly configure spring.jpa.open-in-view to disable this warning

spring.jpa.open-in-view=false
```