

TODO APP Backend

Adding custom Queries to Repository

Exercise4

The screenshot shows an IDE interface with a project tree on the left and a code editor on the right. The project tree includes folders for .gradle, .idea, bin, build, gradle, src (with main, kotlin, com.example.demo, persistence), test (with kotlin, com.example.demo, persistence), and resources. The code editor window has 'TodoEntity.kt' selected in the tabs. The code defines a TodoEntity class with fields id, task, and completed, and an interface TodoRepository extending JpaRepository.

```
package com.example.demo.persistence
import org.springframework.data.jpa.repository.JpaRepository
import javax.persistence.*

@Table(name = "todo")
@Entity(name = "todo")
class TodoEntity(
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    var id: Int = 0,
    var task: String,
    var completed: Boolean,
)
interface TodoRepository: JpaRepository<TodoEntity, Int> {
    fun findAllByCompleted(completed: Boolean): List<TodoEntity>
    fun deleteAllByCompleted(completed: Boolean)
}
```

Sometimes it is necessary / more efficient to work with specific queries to archive certain use-cases. In our example app we want to be able to get all Todos that are completed/incompleted as well as delete all completed / incomplete.

Remember: Since we implement JpaRepository we automatically get convenient query abstractions like „save“, „findById“, „delete“ etc. Basically everything we need to archive CRUD

Define custom Queries that fulfill our special needs that can not be archived with pure CRUD

Test the custom Queries

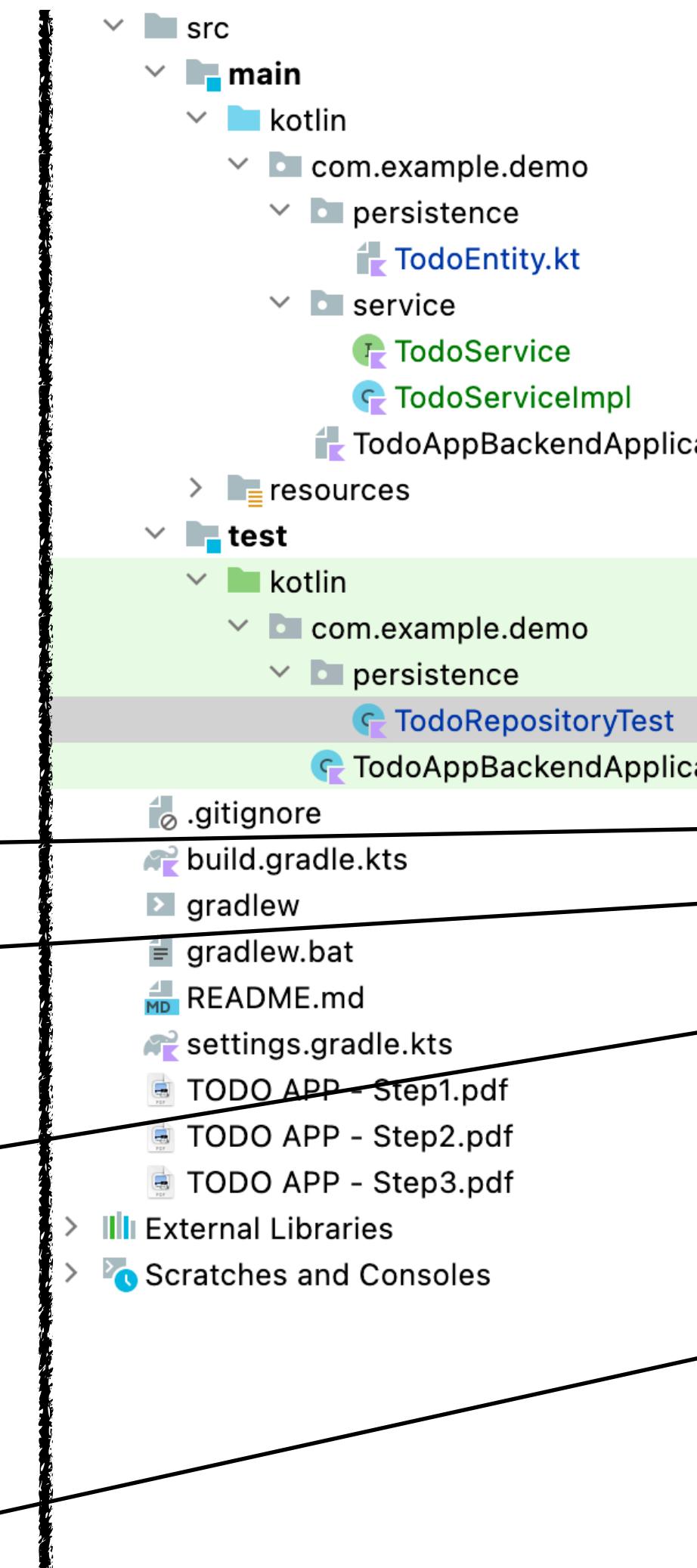
Exercise4

Add test data to the database

Use our custom query
and save the result in variable

Check if our custom query result
contains ONLY the one test data todo
that is ,completed'

Same for completed == false



```
class TodoRepositoryTest {
    @Autowired private val todoRepository: TodoRepository

    private fun aTodo(
        task: String = "a sample task",
        completed: Boolean = false
    ): TodoEntity = TodoEntity(
        task = task,
        completed = completed
    )

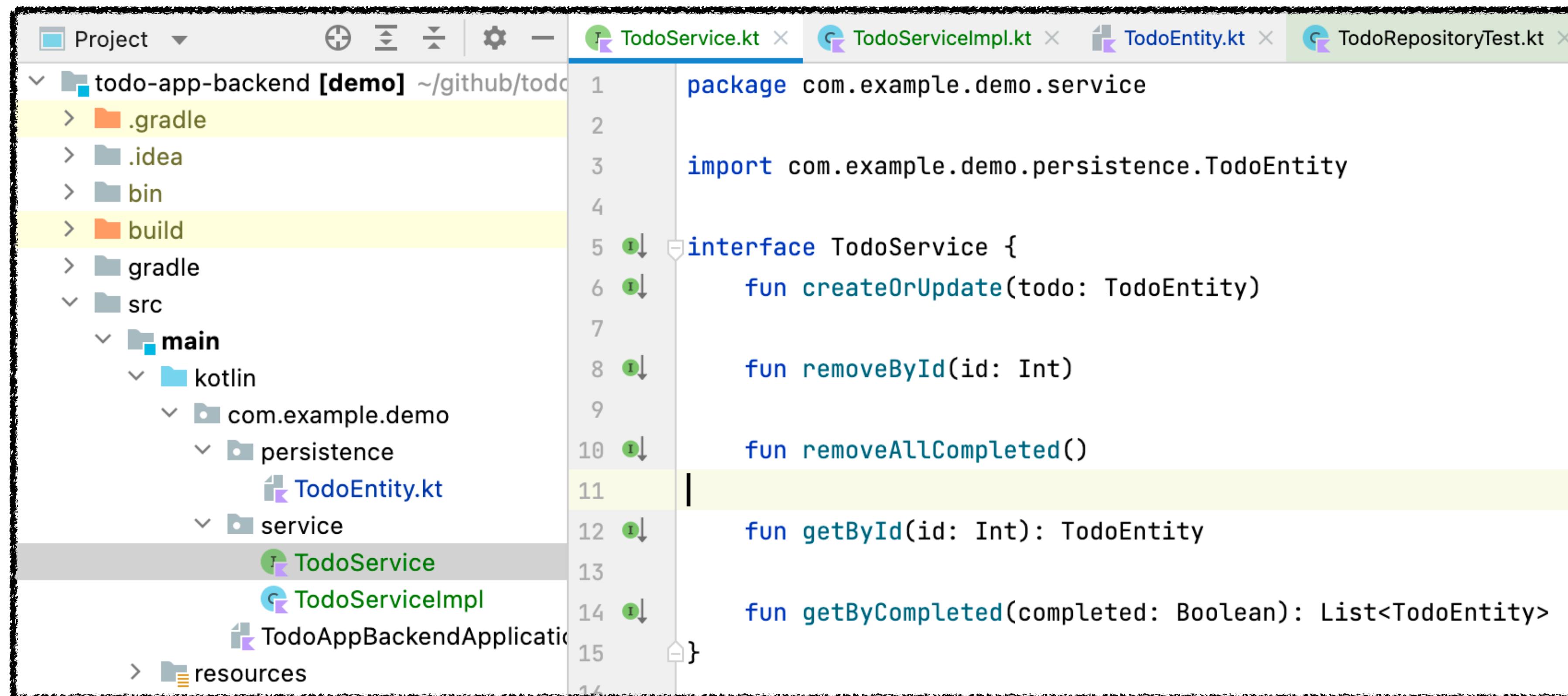
    private val testData: List<TodoEntity> = listOf(
        aTodo(task = "cleaning up", completed = true),
        aTodo(task = "cooking a meal"),
        aTodo(task = "coding kotlin")
    )

    @Test
    fun `can get all completed todos`() {
        val testData: (Mutable)List<TodoEntity!> = todoRepository.saveAll(testData)
        val completed: List<TodoEntity> = todoRepository.findAllByCompleted(completed: true)
        Assertions.assertThat(completed.map { it.task }).containsExactly("cleaning up")
    }

    @Test
    fun `can get all incomplete todos`() {
        val testData: (Mutable)List<TodoEntity!> = todoRepository.saveAll(testData)
        val completed: List<TodoEntity> = todoRepository.findAllByCompleted(completed: false)
        Assertions.assertThat(completed.map { it.task }).containsExactly(
            "cooking a meal",
        )
    }
}
```

Adding a Service Interface

Exercise4



The screenshot shows a Java IDE interface with the following details:

- Project Bar:** Shows "todo-app-backend [demo] ~/github/todo" as the current project.
- Toolbars:** Standard IDE toolbars for file operations.
- Editor Tabs:** Four tabs are open: "TodoService.kt" (selected), "TodoServiceImpl.kt", "TodoEntity.kt", and "TodoRepositoryTest.kt".
- File Structure (Left):** The project structure tree shows the following hierarchy:
 - todo-app-backend [demo]
 - .gradle
 - .idea
 - bin
 - build
 - gradle
 - src
 - main
 - kotlin
 - com.example.demo
 - persistence
 - TodoEntity.kt
 - service
 - TodoService
 - TodoServiceImpl
 - resources
- Code Editor (Right):** The "TodoService.kt" file contains the following Kotlin code:

```
1 package com.example.demo.service
2
3 import com.example.demo.persistence.TodoEntity
4
5 interface TodoService {
6     fun createOrUpdate(todo: TodoEntity)
7
8     fun removeById(id: Int)
9
10    fun removeAllCompleted()
11
12    fun getById(id: Int): TodoEntity
13
14    fun getByCompleted(completed: Boolean): List<TodoEntity>
15}
```

Adding an Interface is not necessarily needed but a great way to describe what our application can do and giving us the possibility to easily change the implementation in the future. Thereby we are adding the possibility to decouple our „technical“ implementation (ServiceImpl) from the business logic (Interface).

Add Class that wants to Implement the Interface

Exercise4

Class is annotated with `@Service` to let it be picked up as a Bean by Spring.
Also we use this `@Service` to make clear this class contains a service with business logic.

```
demo – TodoServiceImpl.kt [demo.main]
todo-app-backend > src > main > kotlin > com > example > demo > service > TodoServiceImpl
TodoService.kt < TodoServiceImpl.kt < TodoEntity.kt < README.md < .gitignore

Project Commit
todo-app-backend [demo] ~/github/todo
> .gradle
> .idea
> bin
> build
> gradle
src
  main
    kotlin
      com.example.demo
        persistence
          TodoEntity.kt
        service
          TodoService
          TodoServiceImpl
        TodoAppBackendApplication
  resources
  test

package com.example.demo.service
import org.springframework.stereotype.Service
@Service
class TodoServiceImpl: TodoService {
}

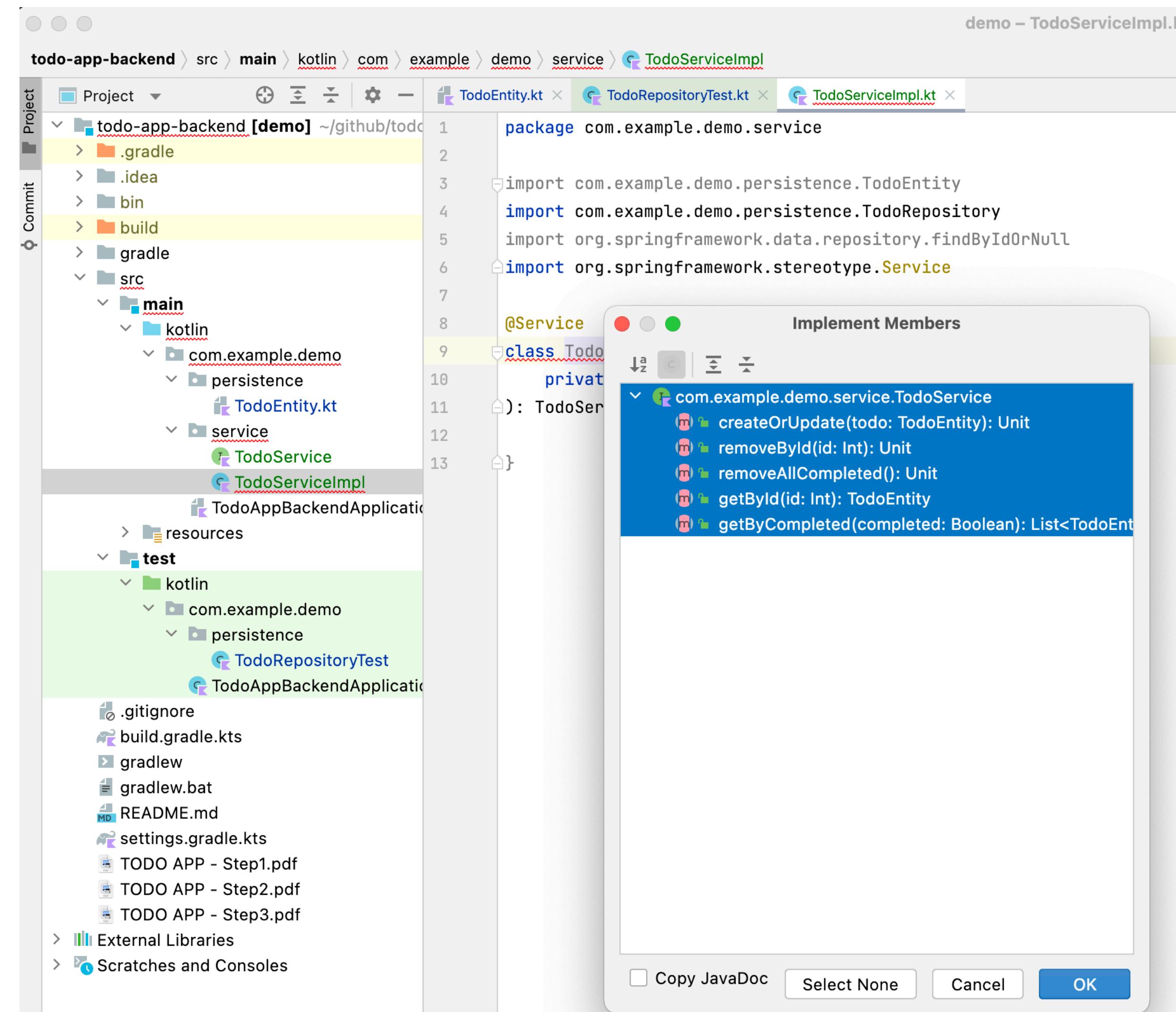
Class 'TodoServiceImpl' is not abstract and does not implement abstract member
public abstract fun add(todo: TodoEntity): Unit defined in com.example.demo.TodoService
Implement members ↗↔ More actions... ↗↔

@Service
public open class TodoServiceImpl
  : TodoService
  demo.main
```

IntelliJ is already complaining about that we should include all the methods
that are required by interface we want to implement :)

Let IntelliJ create all the methods that needs to be implemented

Exercise4



Select all and hit OK

All necessary methods auto-created for us :)

Exercise4

```
@Service
class TodoServiceImpl: TodoService {
    override fun createOrUpdate(todo: TodoEntity) {
        TODO(reason: "Not yet implemented")
    }

    override fun removeById(id: Int) {
        TODO(reason: "Not yet implemented")
    }

    override fun removeAllCompleted() {
        TODO(reason: "Not yet implemented")
    }

    override fun getById(id: Int): TodoEntity {
        TODO(reason: "Not yet implemented")
    }

    override fun getByCompleted(completed: Boolean): List<TodoEntity> {
        TODO(reason: "Not yet implemented")
    }
}
```

Inject the todoRepository Bean into our Service

Exercise4

```
@Service  
class TodoServiceImpl(  
    private val todoRepository: TodoRepository  
) : TodoService {  
  
    override fun createOrUpdate(todo: TodoEntity) {  
        TODO(reason: "Not yet implemented")  
    }  
  
    override fun removeById(id: Int) {  
        TODO(reason: "Not yet implemented")  
    }  
}
```

We use springs dependency injection mechanism here to be able to use the todoRepository in our service

Implement interface by using todoRepository

Exercise4

```
@Service
class TodoServiceImpl(
    private val todoRepository: TodoRepository
): TodoService {
    override fun createOrUpdate(todo: TodoEntity) {
        todoRepository.save(todo)
    }

    override fun removeById(id: Int) {
        with(todoRepository.findByIdOrNull(id)) { this: TodoEntity?
            requireNotNull(value: this) { "could not find todo with id '$id'" }
            todoRepository.delete(entity: this)
        }
    }

    override fun getById(id: Int): TodoEntity {
        return todoRepository.findByIdOrNull(id)
            ?: throw IllegalArgumentException("could not find todo with id '$id'")
    }

    override fun removeAllCompleted() {
        todoRepository.deleteAllByCompleted(completed: true)
    }

    override fun getByCompleted(completed: Boolean): List<TodoEntity> {
        return todoRepository.findAllByCompleted(completed)
    }
}
```

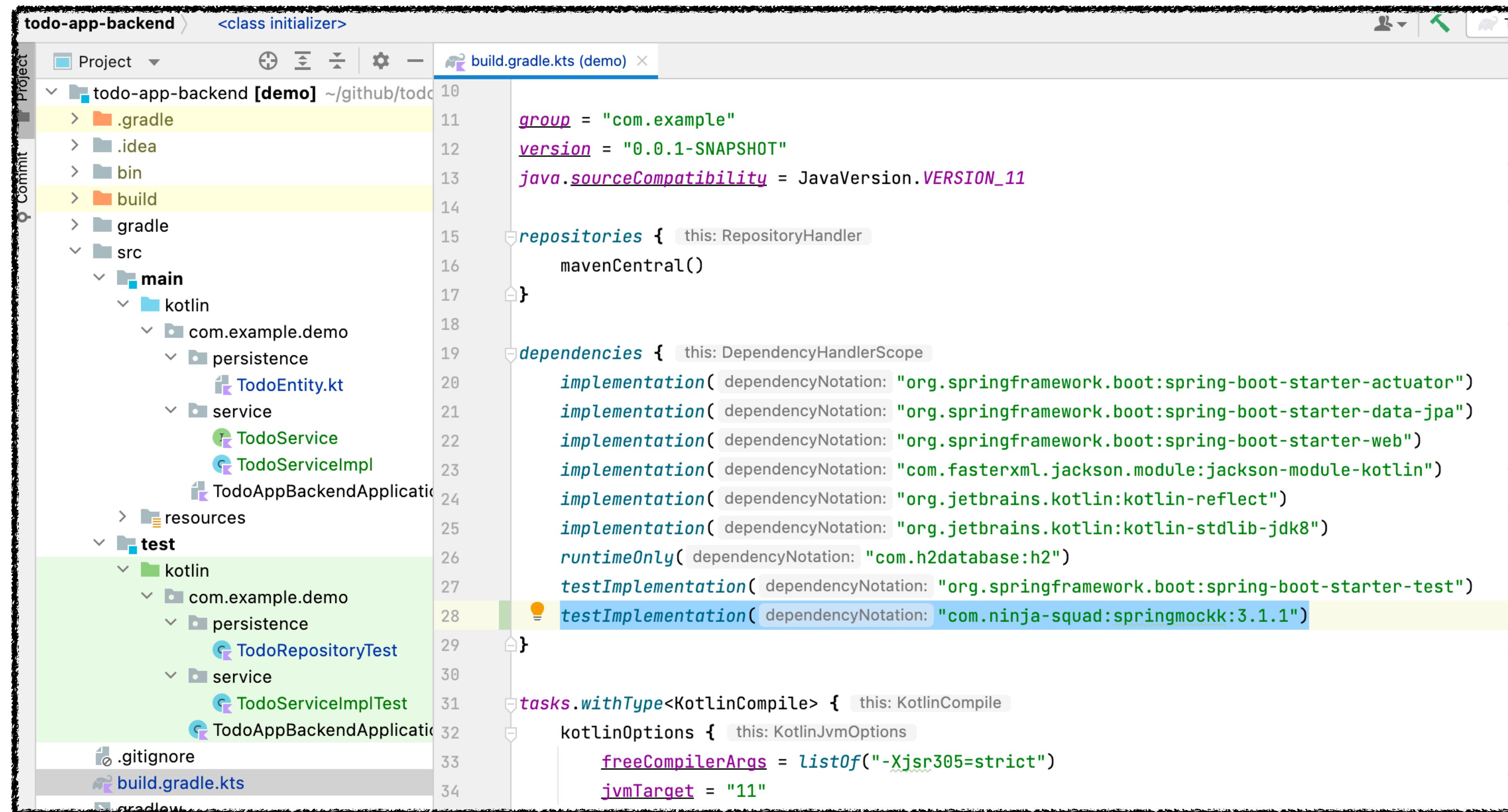
Using the so called ,Elvis operator' to do something as a fallback if the left hand side operation of the Elvis (?:) is `null`. In our case we just throw an IllegalStateException on null case.

Using ,with' - a so called scoping function (part of the Kotlin standard lib) here

Will throw IllegalStateException if todo could not be found in DB

Test the Service / add Spring-Mockk Dependency

Exercise4



The screenshot shows an IDE interface with a project named "todo-app-backend" open. The left sidebar displays the project structure, including .gradle, .idea, bin, build, gradle, and src folders. The src folder contains main and test subfolders. Under main/main, there are kotlin, com.example.demo, persistence, service, and resources subfolders. Under test/test, there are kotlin, com.example.demo, persistence, service, and resources subfolders. The main/kotlin/com.example.demo/persistence/TodoEntity.kt file is currently selected. The right pane shows the build.gradle.kts file with the following content:

```
10 group = "com.example"
11 version = "0.0.1-SNAPSHOT"
12 java.sourceCompatibility = JavaVersion.VERSION_11
13
14 repositories { this: RepositoryHandler
15     mavenCentral()
16 }
17
18 dependencies { this: DependencyHandlerScope
19     implementation(dependencyNotation: "org.springframework.boot:spring-boot-starter-actuator")
20     implementation(dependencyNotation: "org.springframework.boot:spring-boot-starter-data-jpa")
21     implementation(dependencyNotation: "org.springframework.boot:spring-boot-starter-web")
22     implementation(dependencyNotation: "com.fasterxml.jackson.module:jackson-module-kotlin")
23     implementation(dependencyNotation: "org.jetbrains.kotlin:kotlin-reflect")
24     implementation(dependencyNotation: "org.jetbrains.kotlin:kotlin-stdlib-jdk8")
25     runtimeOnly(dependencyNotation: "com.h2database:h2")
26     testImplementation(dependencyNotation: "org.springframework.boot:spring-boot-starter-test")
27
28     testImplementation(dependencyNotation: "com.ninja-squad:springmockk:3.1.1")
29 }
30
31 tasks.withType<KotlinCompile> { this: KotlinCompile
32     kotlinOptions { this: KotlinJvmOptions
33         freeCompilerArgs = listOf("-Xjsr305=strict")
34         jvmTarget = "11"
35     }
36 }
```

A line of code at line 28, `testImplementation("com.ninja-squad:springmockk:3.1.1")`, is highlighted with a yellow background and a small orange icon, indicating it is the current focus or has been modified.

Create Test Class for our Service Impl

Exercise4

```
todo-app-backend > src > test > kotlin > com > example > demo > service > TodoServiceImplTest.kt can get existing todo by id
Project Commit
todo-app-backend [demo] ~/github/todo-app-backend
  > .gradle
  > .idea
  > bin
  > build
  > gradle
  > src
    > main
      > kotlin
        > com.example.demo
          > persistence
            > TodoEntity.kt
          > service
            > TodoService
            > TodoServiceImpl
            > TodoAppBackendApplication
    > resources
  > test
    > kotlin
      > com.example.demo
        > persistence
          > TodoRepositoryTest
        > service
          > TodoServiceImplTest
          > TodoAppBackendApplication
10 import org.junit.jupiter.api.Assertions
11 import org.junit.jupiter.api.Test
12 import org.springframework.beans.factory.annotation.Autowired
13 import org.springframework.boot.test.context.SpringBootTest
14 import org.springframework.data.repository.findByIdOrNull
15
16
17 @SpringBootTest
18 class TodoServiceImplTest(
19     @Autowired private val todoService: TodoService
20 ) {
21
22     @MockKBean
23     private lateinit var todoRepository: TodoRepository
24
25     private fun aTodo(
26         task: String = "a sample task",
27         completed: Boolean = false
28     ): TodoEntity = TodoEntity(
29         task = task,
30         completed = completed
31     )
32 }
```

Annotate the class with `@SpringBootTest`
-> this will initialize the spring context on test start,
which allows us to use our beans with
spring DI (dependency injection)

Inject our subject under test

Let's Mock the repository bean.
This will create fake repository that we
can control per test to provoke all the
behaviors we want to test.

(this is what we added Mockk dependency for)

Again add a little helper function to create todos
in a convenient fashion

Add tests

Exercise4

```
33  @Test
34  fun `will create or update by calling repository save`() {
35      val aTestTodo : TodoEntity = aTodo()
36      every { todoRepository.save(aTestTodo) } returns aTestTodo
37
38      todoService.createOrUpdate(aTestTodo)
39      verify { todoRepository.save(aTestTodo) }
40  }
41
42  @Test
43  fun `can get existing todo by id`() {
44      val aTestTodo : TodoEntity = aTodo()
45      every { todoRepository.findByIdOrNull(id: 4711) } returns aTestTodo
46
47      val result : TodoEntity = todoService.getById(id: 4711)
48      assertThat(result).isEqualTo(aTestTodo)
49  }
50
51  @Test
52  fun `will throw if trying to get todo by id that not exists`() {
53      val aTestTodo : TodoEntity = aTodo()
54      every { todoRepository.findByIdOrNull(id: 4711) } returns null
55
56      Assertions.assertThrows {
57          todoService.getById(id: 4711)
58      }
59  }
```

Configure our mocked bean (the repository) to whenever it gets called return a test todo we have just created

Call our service function

Check if our service function has called the expected repository function (save) of our mocked bean and thereby tried to persist our test todo

Configure mocked bean (todoRepository) to return a test todo whenever a given id (4711) gets passed as parameter

Call our service function we want to test and save result

Check if result is exactly our test todo reference

Configure mock to return null on given id

Check if expected exception has been thrown when service function gets called with this given id

Add more tests

Exercise4

```
61 @Test
62 fun `can find all completed`() {
63     val anCompletedTodo : TodoEntity = aTodo(completed = true)
64     every { todoRepository.findAllByCompleted( completed: true) } returns listOf(anCompletedTodo)
65
66     val result : List<TodoEntity> = todoService.getByCompleted( completed: true)
67     assertThat(result).containsExactly(anCompletedTodo)
68 }
69
70 @Test
71 fun `can remove all completed`() {
72     justRun { todoRepository.deleteAllByCompleted(any()) }
73
74     todoService.removeAllCompleted()
75     verify { todoRepository.deleteAllByCompleted( completed: true) }
76 }
```

Configure mock

Call service function we want to test

Check result

Since we need to mock a void function that is not returning something, we can do like this

Call service function we want to test

Check if repository function has been called with expected parameter

Repeat what we just learned

Exercise4

Based on what we learned during this tutorial,
please add a ‚getAll‘ functionality to our service
by extending the interface, the serviceImpl and the tests. 😊

The getAll should return a List of TodoEntity.
Therefore you need to find out which build-in jpa repository
function this is and also prove this by a test.

Hint: the jpa repository will give us all possibilities we need to
archive the exercise. No need to write custom jpa query this time.