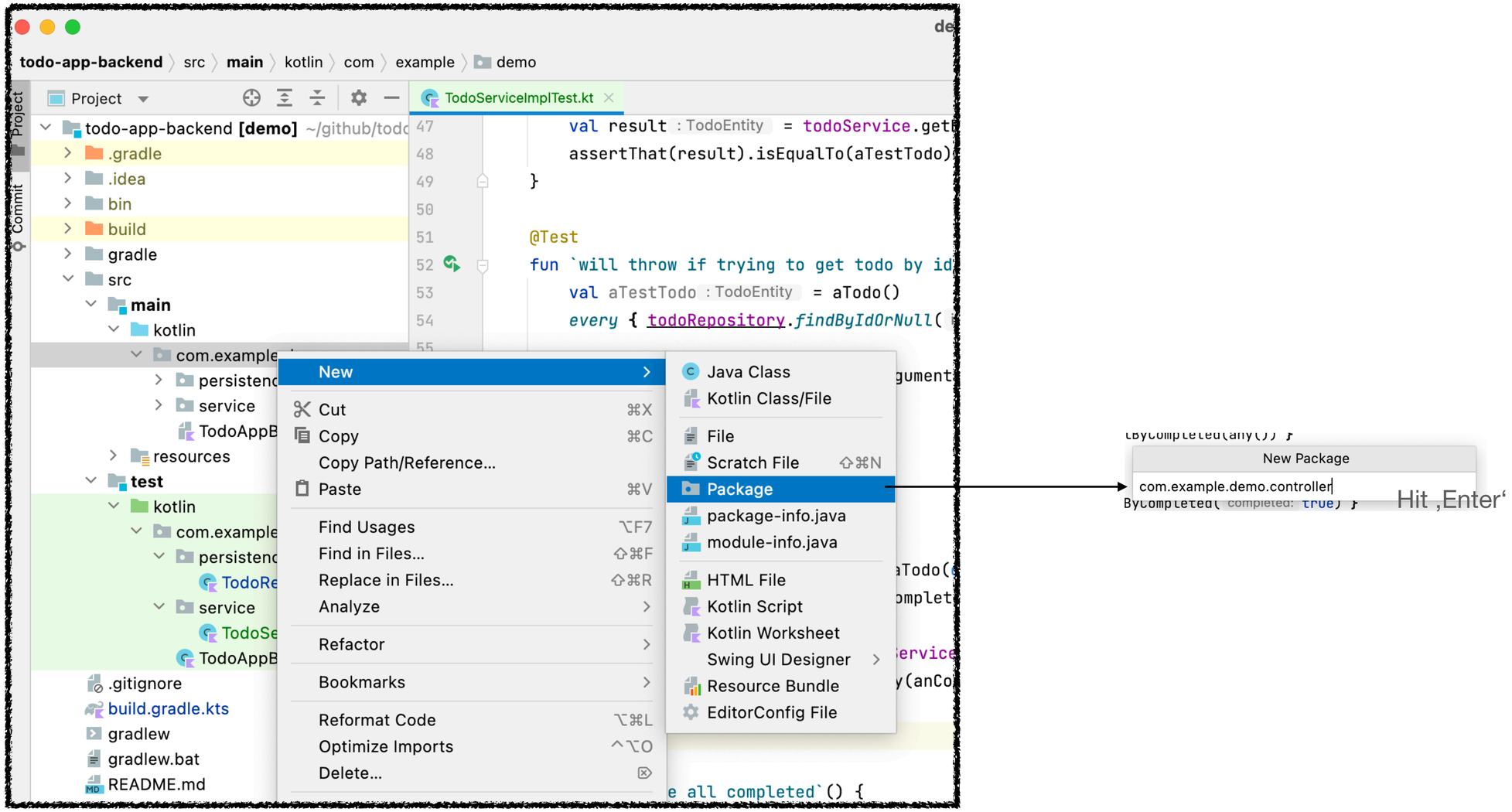
TODO App Backend

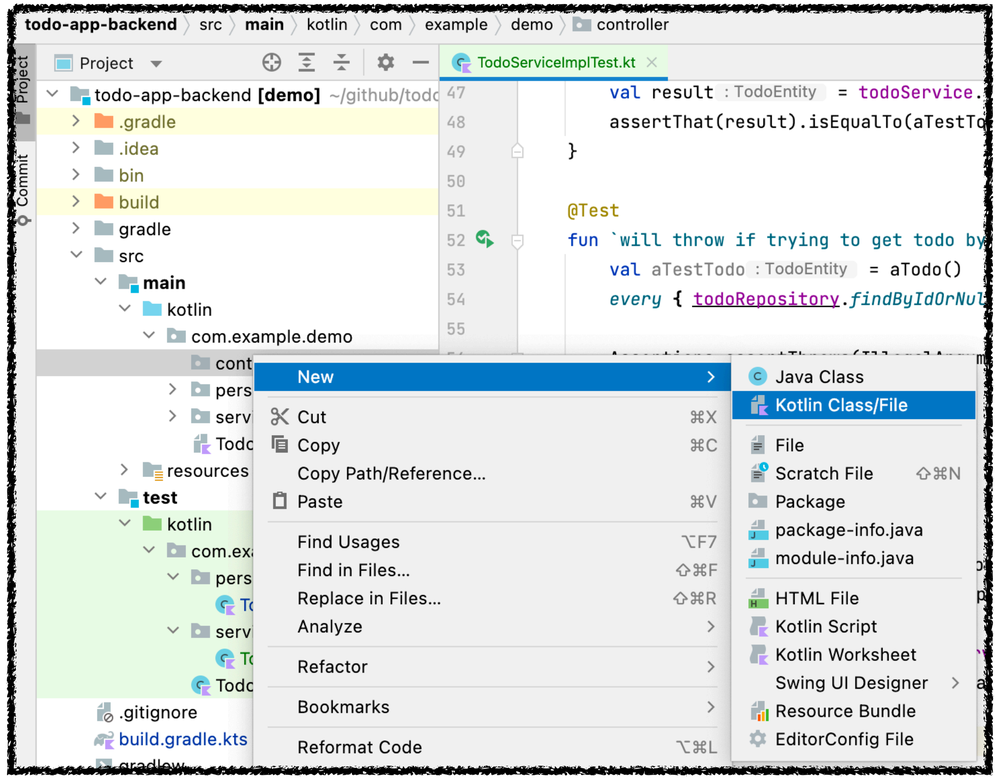
Create controller package

Exercise 5

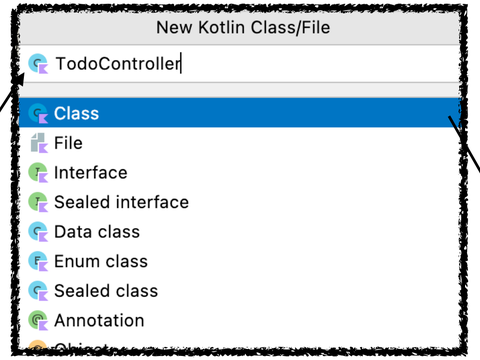


Create controller class

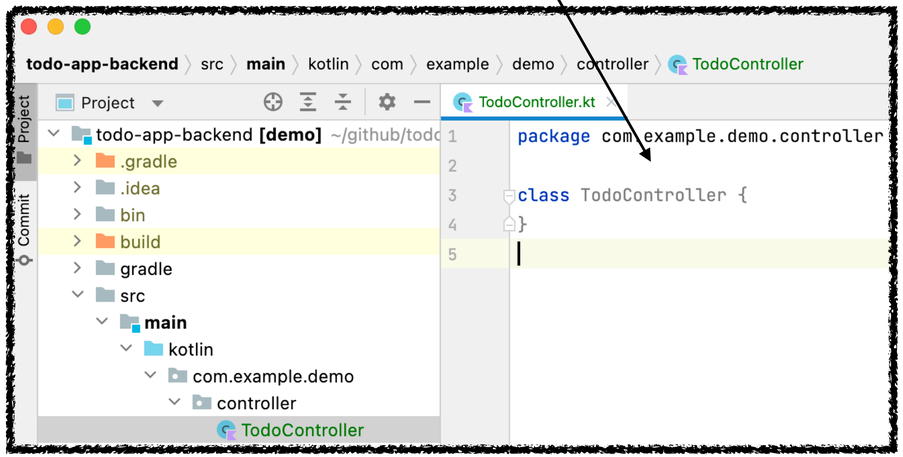
Exercise 5



Give it a name

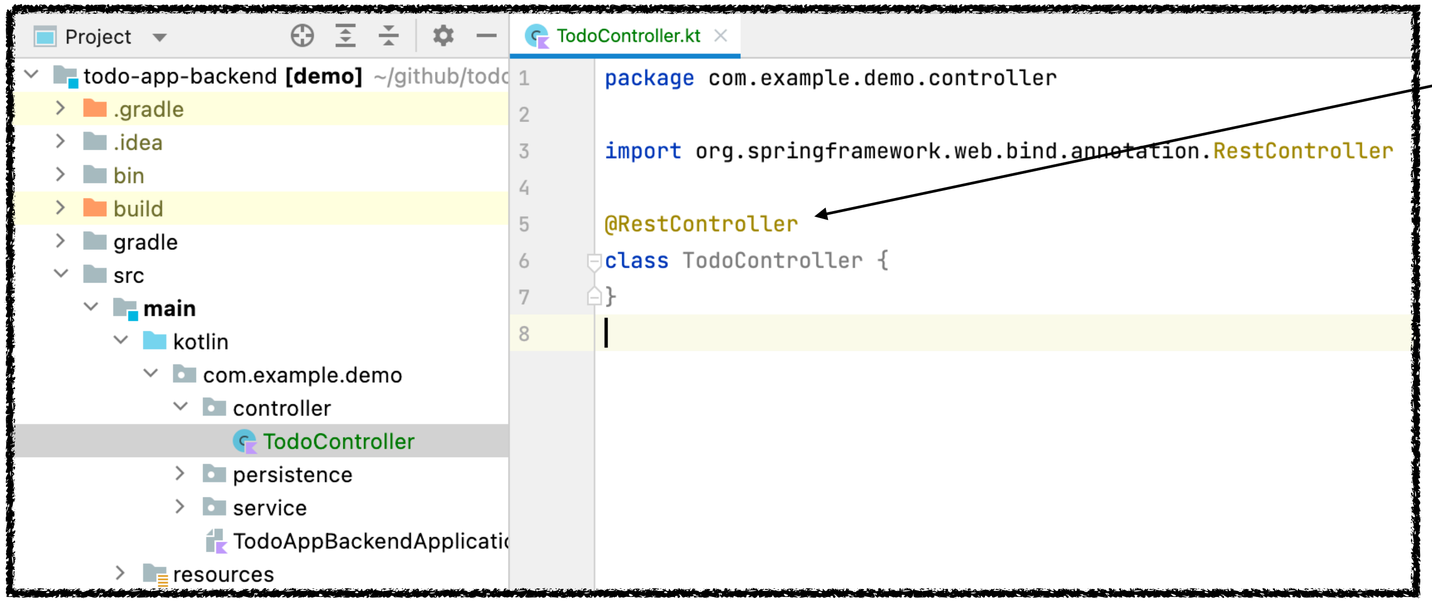


Empty class will be created



Make it a RestController Bean

Exercise 5



Annotate Class with @RestController
to make this class a rest controller bean which has, in comparison to @Service or @Component, a special behavior since it is a so called Request scoped bean.

-

It has different lifecycle as „normal“ beans and will be re-initialized on every request

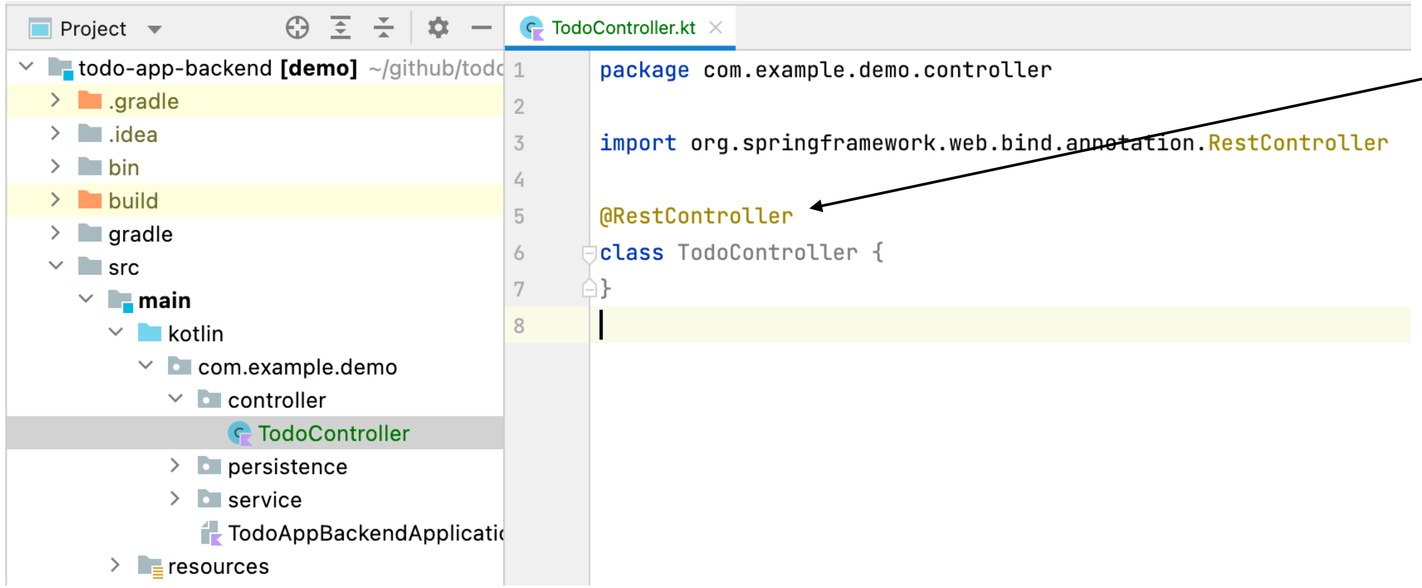
Remember: since this is now a bean it will automatically picked up by springs component scan on application start and added to the spring context.

-

without the bean annotation spring would ignore this class to be part of the spring config

Make it a RestController Bean

Exercise 5



Annotate Class with @RestController
to make this class a rest controller bean which has, in comparison to @Service or @Component, a special behavior since it is a so called Request scoped bean.
-
It has different lifecycle as „normal“ beans and will be re-initialized on every request

Remember: since this is now a bean it will automatically picked up by springs component scan on application start and added to the spring context.
-
without the bean annotation spring would ignore this class to be part of the spring config

Inject service(s) the controller wants to use

Exercise 5

```
@RestController
class TodoController(
    private val todoService: TodoService
) {

}
```

Use spring dependency injection mechanism to inject spring beans. In this case we are injecting our todoService.

-

Remember: the power of DI is that we don't need to care about the constructor of a bean we want to inject.

Prefix the paths of all the endpoints of a Controller

Exercise 5

```
@RestController
@RequestMapping("/todo")
class TodoController(
    private val todoService: TodoService
) {

}
```

We can prefix the paths of all endpoints that will later on be created in the controller class with a class level annotation

Add ‚create or update todo‘-Endpoint

Exercise 5

```
@RestController
@RequestMapping("/todo")
class TodoController(
    private val todoService: TodoService
) {
    @PutMapping
    fun putTodo(
        @RequestBody todoRequest: TodoRequest
    ) {
        todoService.createOrUpdate(todoRequest.toEntity())
    }
}

data class TodoRequest(
    val task: String,
    val completed: Boolean = false
) {
    fun toEntity() : TodoEntity = TodoEntity(
        task = task,
        completed = completed
    )
}
```

Since we defined no extra mapping on the controller method
our endpoint will be available under path „/todo“
with HTTP request method Put

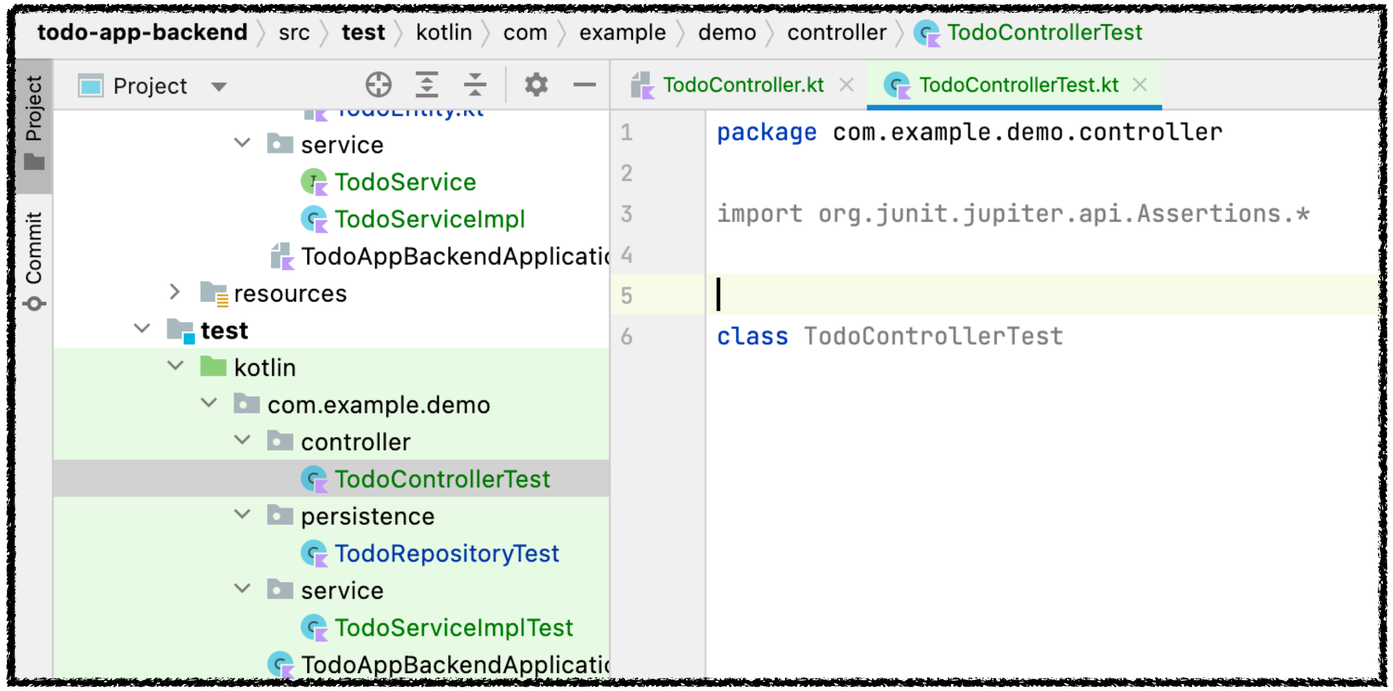
Since its a bad practice make the outside world (clients / consumers)
of our application rely on the format of our database entities,
we define an extra so called DTO (data transfer object) and make
It necessary to be passed as a request body as part of a request
To our endpoint.
-
Thereby we provide a so called anti corruption layer to don’t
Break clients if we would change our database schema in
the future.

Use the corresponding service function

Add a mapping function to conveniently convert
the request object to our internal entity

Test Endpoint - Add test class

Exercise 5



Create a Test class

Test the Endpoint - configure MockMvc

Exercise 5

```
@AutoConfigureMockMvc
@SpringBootTest
class TodoControllerTest(
    @Autowired private val mockMvc: MockMvc
) {


}
```

We will make this a so called mockMvc test
by adding corresponding annotation and inject
mockMvc to our test
-
MockMvc provides support for Spring MVC testing.
It encapsulates all web application beans
and makes them available for testing.

MockMvc provides an elegant and easy-to-use API
to call web endpoints and to inspect and assert
their response at the same time.

Because Spring prepares a fake web application context to mock
the HTTP requests and responses when we use mockMvc tests,
it may not support all the features of a full-blown Spring application.

But for our use case it fits very well.

Test the Endpoint - add test

Exercise 5

```
@AutoConfigureMockMvc
@SpringBootTest
class TodoControllerTest(
    @Autowired private val mockMvc: MockMvc
) {
    @Test
    fun `can put todo`() {
        mockMvc.put( urlTemplate: "/todo") {  this: MockHttpServletRequestDsl
            contentType = MediaType.APPLICATION_JSON
            content = TodoRequest(task = "test").asJsonString()
        }.andExpect {  this: MockMvcResultMatchersDsl
            status { isOk() }
        }
    }

    private fun TodoRequest.asJsonString() : String! =
        jacksonObjectMapper().writeValueAsString( value: this)
}
```

Call the path our endpoint we want to test is mapped to with http method ‚put‘

Define required request headers and request body

Expect status code 200(OK)

Use little helper function to convert Request object to json as string

Add delete Endpoint

Exercise 5

```
@RestController
@RequestMapping("/todo")
class TodoController(
    private val todoService: TodoService
) {
    @DeleteMapping("/{id}")
    fun deleteTodo(
        @PathVariable id: Int
    ) {
        todoService.removeById(id)
    }

    @PutMapping
    fun putTodo(
```

We can make partsof our endpoints path variable by putting curly braces around parts of the path.

Spring will automatically read the given value of this variable from the path and pass it as a parameter to our endpoint function when we annotate the variable With @PathVariable

-

Spring will care about the types for us automatically. If the path variable Would be something that can not be converted to an Integer (since we defined it needs to be of type Int), it would throw an Error. This is great because we don't have to take care about stuff like that on our own :)

Hint: Names of path placeholder in curly braces and name of path variable needs to match!

Execute corresponding service function with given id

Test delete Endpoint - add test

Exercise 5

```kotlin
@AutoConfigureMockMvc
@SpringBootTest
class TodoControllerTest(
    @Autowired private val mockMvc: MockMvc,
    @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 remove todo`() {
        val persistedTestData : (Mutable)List<TodoEntity!> = todoRepository.saveAll(testData)

        mockMvc.delete( urlTemplate: "/todo/${persistedTestData.last().id}")
            .andExpect { this: MockMvcResultMatchersDsl
                status { isOk() }
            }

        Assertions.assertThat(todoRepository.findAll().map { it.task })
            .containsExactly(
                persistedTestData[0].task,
                persistedTestData[1].task,
            )
    }
```

Inject repository bean to be able to
add test data and check db after test executions

Use helper function
to conveniently create
todos

Add test data to database

Call endpoint with http method ‚delete‘
and the id we want to delete as path variable
-
in this case we just say it should delete the last
todo of our test data

Check if only first 2 todos are still in
the database since we deleted the last one (the 3rd one)
-
We check specifically for the task value here since we
don’t want to care about object reference or id here

Test should work on clean DB every time

Exercise 5

Lets add this function to our Test Class
(it doesn't matter where it will replaced inside the class).
but usually it is a good practice to place it at the start of the
the test classes body

```
@BeforeEach
fun clearDatabase() {
    todoRepository.deleteAll()
}
```

Junit5 annotation to say this method should be
executed before every test

All the tests should not rely
on each other and should be able to run in isolation.
-
this can be archived if we just delete everything from the DB
before each test.

Add getAll Endpoint

Exercise 5

```
@RestController
@RequestMapping("/todo")
class TodoController(
    private val todoService: TodoService
) {
    @GetMapping("/all")
    fun getAll() : List<TodoResponse>  =
        todoService.getAll().toResponse()

    @DeleteMapping("/{id}")
```

```
47  data class TodoResponse(
48      val id: Int,
49      val task: String,
50      val completed: Boolean,
51  )
52  fun TodoEntity.toResponse() : TodoResponse  =
53      TodoResponse(id, task,completed)
54  fun List<TodoEntity>.toResponse() : List<TodoResponse>  =
55      map {it.toResponse()}
```

Really straight forward this time :)
/todo/all will just execute the getAll service method and returns whatever the service returns

-

no path variable or request body etc needed

But again don't rely on database entities on controller level. Database objects should never be visible for externals

-

To archive that behavior we add a dedicated todo response class and some handy converters to map list of todo entities to todo response objects (equivalent to what we did with the TodoRequest object in the former steps)

Test getAll Endpoint

Exercise 5

```
@Test
fun `can get all todo`() {
    val persistedTestData : (Mutable)List<TodoEntity!> = todoRepository.saveAll(testData)

    val responseBody: List<TodoResponse> = mockMvc.get( urlTemplate: "/todo/all")
        .andExpect { this: MockMvcResultMatchersDsl
            status { isOk() }
        }.andReturn().response.let { it: MockHttpServletResponse
            jacksonObjectMapper().readValue(it.contentAsString)
        }

    Assertions.assertThat(responseBody.size).isGreaterThanOrEqualTo(3)

}
```

Make sure we have some data in the DB

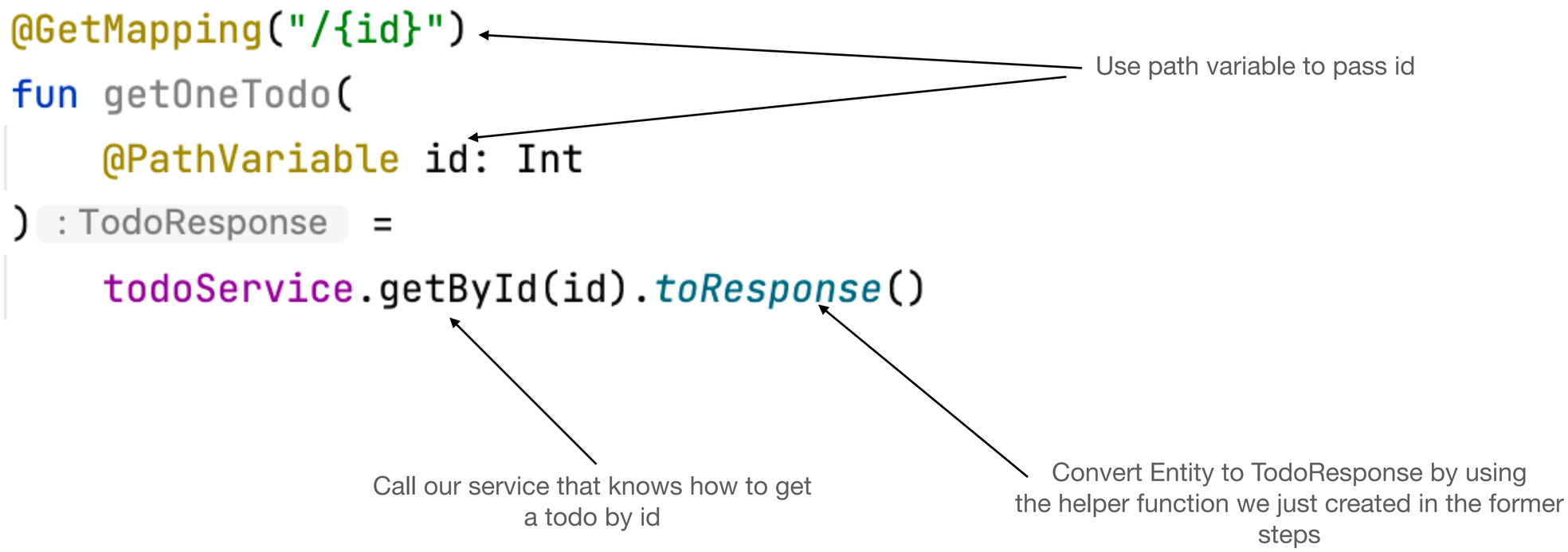
Call endpoint we want to test with http method get

Convert the response body (json as string) to a List<TodoResponse> and return this

We should have at least 3 entries in the DB

Add get One todo Endpoint

Exercise 5



Test get One todo Endpoint

Exercise 5

```
@Test
fun `can get a todo by id`() {
    val persistedTestData : (Mutable)List<TodoEntity!> = todoRepository.saveAll(testData)

    mockMvc.get( urlTemplate: "/todo/${persistedTestData[1].id}")
        .andExpect { this: MockMvcResultMatchersDsl
            status { isOk() }
            content { this: ContentResultMatchersDsl
                jsonPath( expression: "task", Matchers.containsString( substring: "cooking a meal"))
            }
        }
}
```

Add test data

MockMvc allows us to make assertions on the response body, headers etc in scope of the ‚content‘ lambda

Now that we have the content result MockMvc provides several handy helper function like jsonPath that allows us to parse the value of certain json paths

It wants 2 parameters - the json path itself and a Hamcrest Matcher

Repeat what we just learned

Exercise 5

Please add Controllers and corresponding tests:
1.) to get all incomplete tasks
2.) to get all completed tasks
3.) to remove all completed tasks