# TODO App Backend

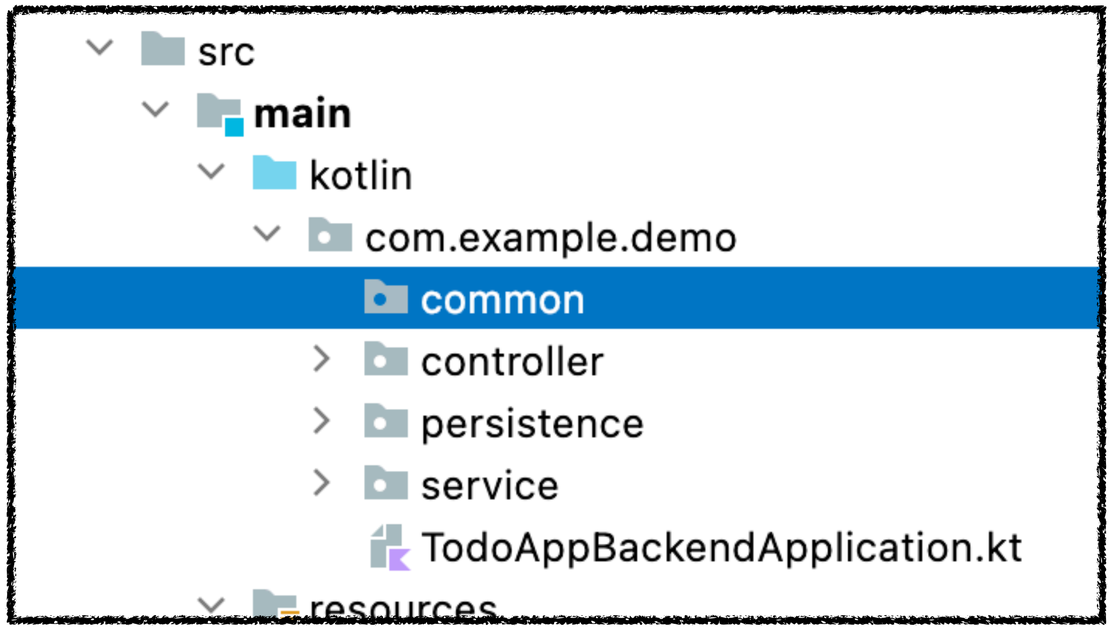
# Controller Error Handling

## Exercise 6

- Currently our REST-Controllers return objects directly, that’s ok since we rely on Spring-Boot default handling responses and errors
- But in a real world we often want to be more precise when returning errors to the user of our API, as well as returning certain status codes, as it can help the API client properly respond to issues

# Add Custom Exception

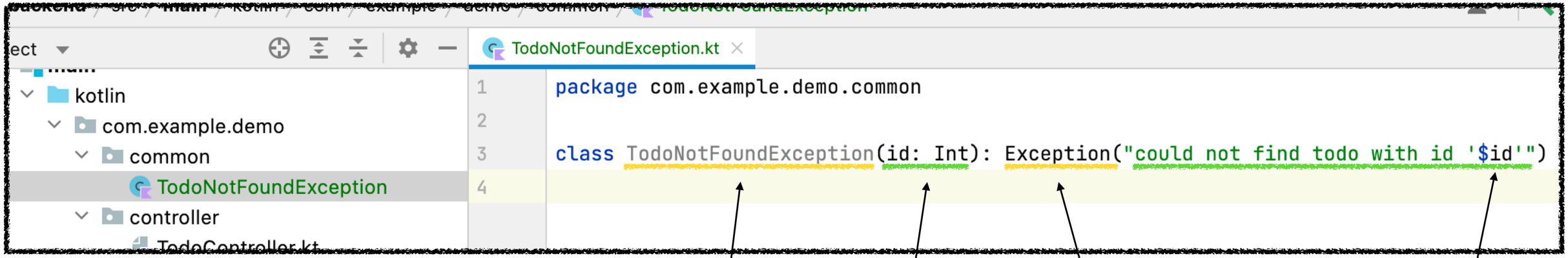
## Exercise 6




Create a new package called ‚common‘.
(you could give it any name of your choice)

# Add Custom Exception

## Exercise 6




Add a class called TodoNotFoundException
that requires an integer constructor parameter

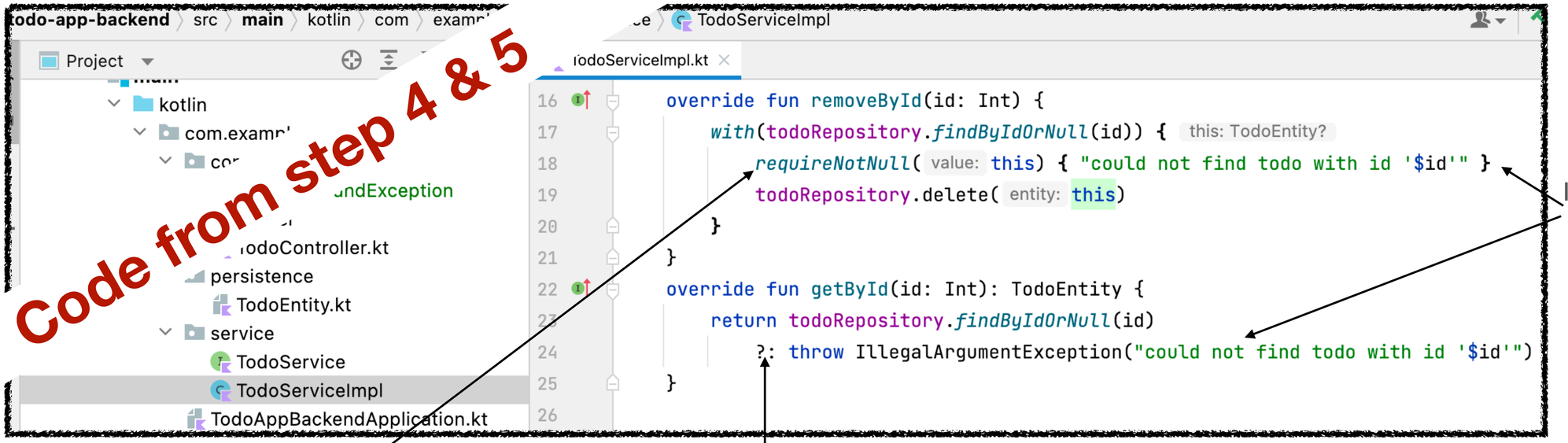
It will inherit Exception class
which is a so called ‚checked exception‘
and part of the Kotlin standard library

We pass a meaningful error message
including the id parameter of our
custom exception to ‚Exception‘

With this little trick we can now use our
own Exception inside of our code base and thereby
have the possibility to distinguish and handle
certain errors later on.
Our custom exception will internally behave like any
Other checked exception since its a child of ‚Exception‘.

# We used Build-in Exceptions so far

## Exercise 6




Code from step 4 & 5

In our current implementation we have 2 functions that will throw an Exception if a given Entity is null

A build-in Kotlin function to check if something is null. If checked object isn't null the object will be automatically casted to a non nullable object for us.

-

if required value is null it will throw an IllegalArgumentException

Here we say (by the use of a JPA method query) „please give me a todo by a given id, if id not exists just return null".
Afterwards we use Kotlin so called ‚Elvis operator' (?:).
If first operand isn't null (left hand side of Elvis), then it will be returned.
If it is null, then the second operand (right hand side of Elvis) will be returned/executed.
In our case we throw an IllegalArgumentException.

# Use Custom Exception / Refactor current code

## Exercise 6

Code from former Exercises

```
override fun removeById(id: Int) {
    with(todoRepository.findByIdOrNull(id)) {
        requireNotNull(this) { "could not find todo wit
        todoRepository.delete(this)
    }
}
override fun getById(id: Int): TodoEntity {
    return todoRepository.findByIdOrNull(id)
        ?: throw IllegalArgumentException("could not fi
}
```

Refactored version that uses custom exception

```
16  17  override fun removeById(id: Int) {
17  18      todoRepository.findByIdOrNull(id)?.run {
18  19          todoRepository.delete(this)
19  20      } ?: throw TodoNotFoundException(id)
20  21
21  22  }
22  23  override fun getById(id: Int): TodoEntity {
23  24      return todoRepository.findByIdOrNull(id)
24  25          ?: throw TodoNotFoundException(id)
25  26  }
```

Since we throw custom exception now we can let spring react on this certain exception =)

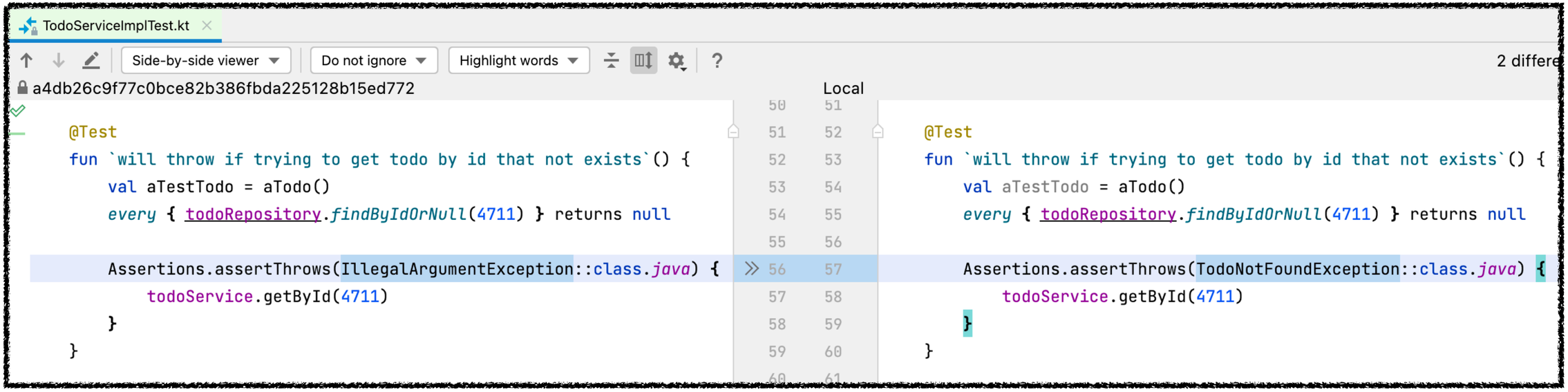
We use Elvis operator again to throw our custom exception if operant to the left of Elvis is null

# Adjust corresponding test since implementation has changed

## Exercise 6

Old

New




Since implementation of our TodoServiceImpl class changed and is now throwing a different exception in case of requesting unknown todo id we need to change the corresponding test as well

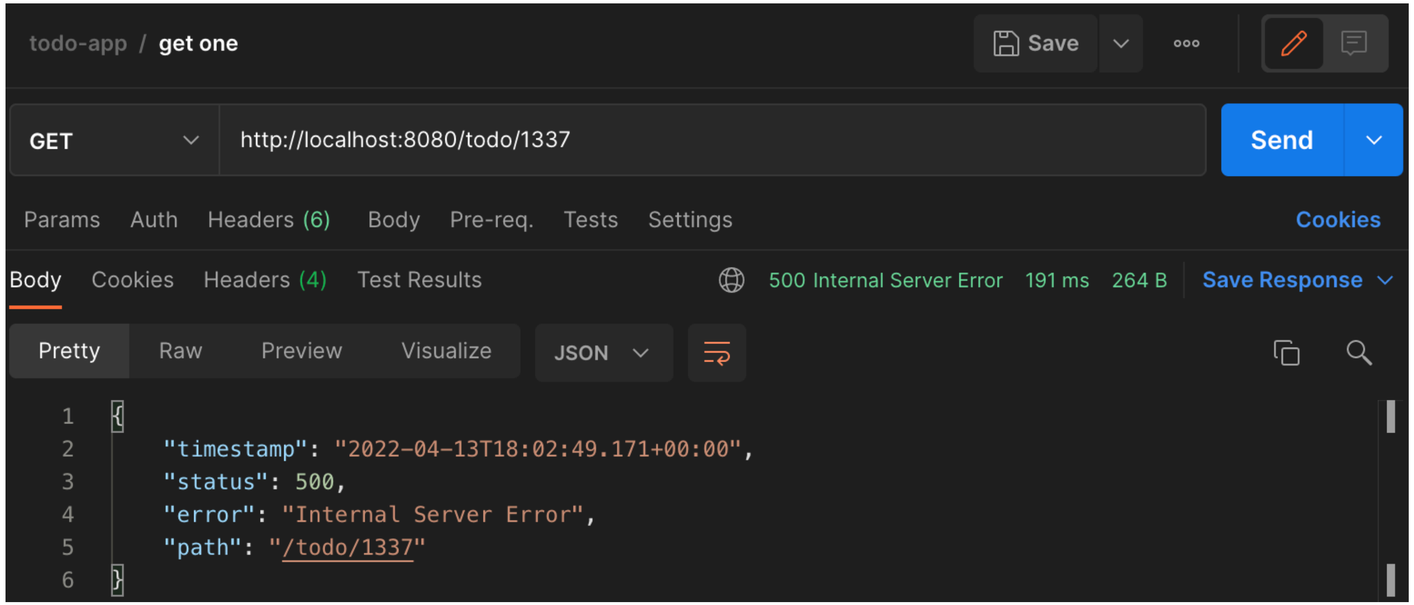
# Though no real error handling so far

## Exercise 6

When testing one of the controllers by hand
we see the controller will return an
status of 500 (Internal Server Error).
This is because we don't handle errors / exceptions
regarding our controller responses at all.
So far we rely on spring to handle exceptions
That can occur somewhere in our code base.
Spring will not distinguish exceptions and handle all as 500.

But that's in turn a really bad experience for the users of our API :(

But since we have a custom exception thrown in our code,
we can configure spring to return a certain http status code
whenever this exception gets thrown. =)



# Custom Response status (option1 - marking exception class)

## Exercise 6

TodoNotFoundException.kt

```kotlin
package com.example.demo.common

import org.springframework.http.HttpStatus
import org.springframework.web.bind.annotation.ResponseStatus

@ResponseStatus(code = HttpStatus.NOT_FOUND, reason = "Todo not found")
class TodoNotFoundException(id: Int): Exception("could not find todo with id '$id'")
```

Since we have a custom exception in place
we can just mark the exception class with
@ResponseStatus
-
From now on spring will, whenever this exception
is thrown, return a 404 (not found) status
-
Using this approach (marking the *Exception* class)
is super straight forward and let us easily archive
global handling of certain exceptions
for all controllers in our application.

If we handle the exceptions they also will be catched
by spring automatically instead of just bubbling up.

We have three ways to use *@ResponseStatus*
to convert an *Exception* to an HTTP response status:

- using @ExceptionHandler (I wouldn't recommend)
- using @ControllerAdvice
- marking the *Exception* class

# Prove Custom Response status works

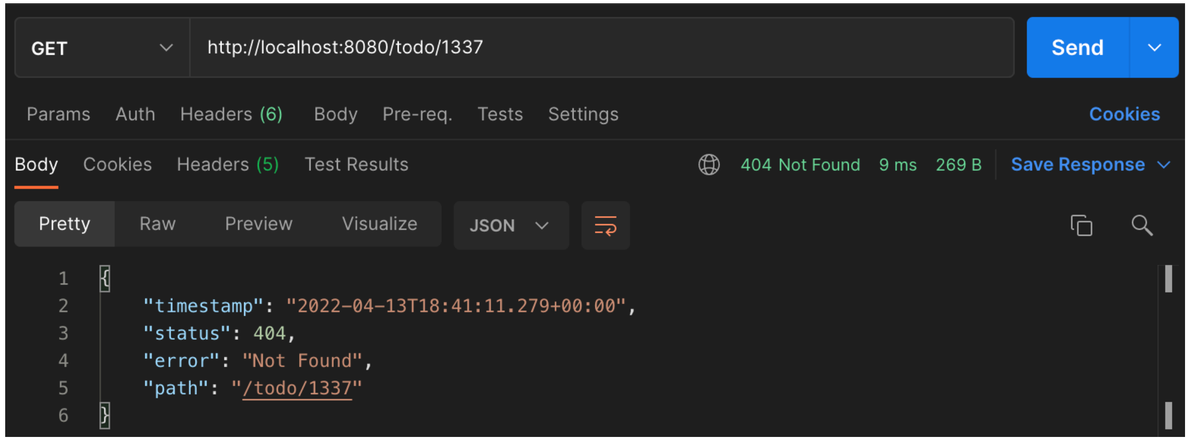
## Exercise 6

TodoControllerTest.kt

```kotlin
59      @Test
60      fun `will return 404 on unknown task id requested`() {
61          mockMvc.get( urlTemplate: "/todo/1337")
62              .andExpect { this: MockMvcResultMatchersDsl
63                  status { isNotFound() } // check if status is 404 (not found)
64              }
65      }
```

Yay it works. Whenever someone tries to get a todo by an id we not know we return a 404 instead of 500.

We can also double check by starting our application and call endpoint with any http client (e.g. Postman).




little drawback: we still use the standard error response from spring boot.
Lets have a look how we can get full control of what will be returned on the next slide…

# Custom Response status (option2 - controller advice)

## Exercise 6

TodoNotFoundException.kt

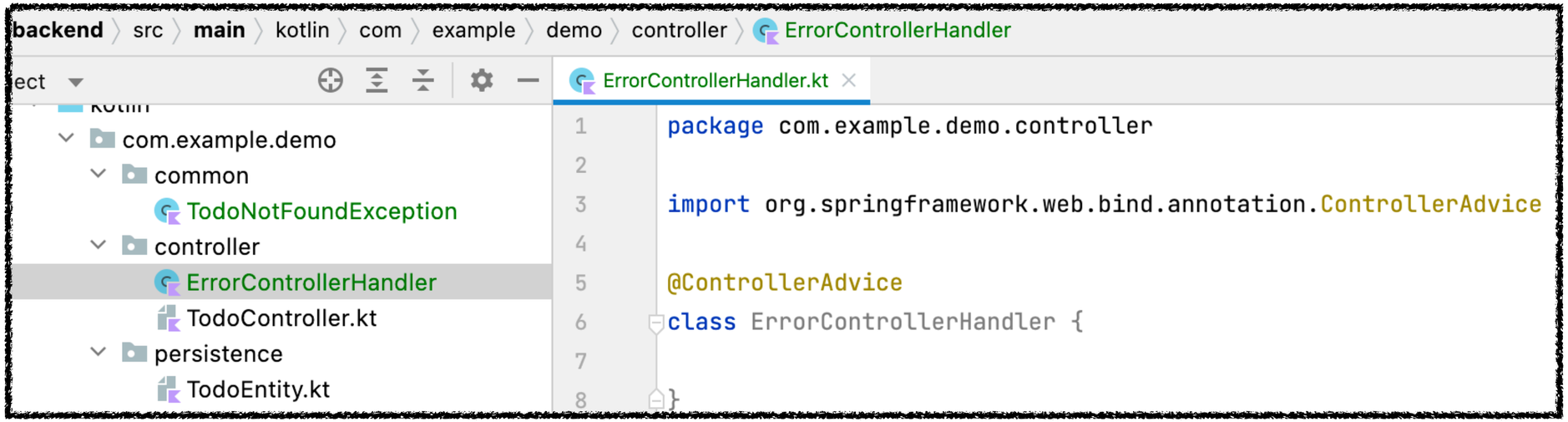
```kotlin
package com.example.demo.common

class TodoNotFoundException(id: Int): Exception("could not find todo with id '$id'")
```

First lets remove the @ResponseStatus annotation we added during option 1

# Custom Response status (option2 - controller advice)

## Exercise 6




Add a class annotated with @ControllerAdvice to our controller package.
(It could life in any package you want and can have any name you want.
Since it's annotated with @ControllerAdvice spring will recognize it as a bean
and thereby pick it up during component scan on application start.)

# Custom Response status (option2 - controller advice)

## Exercise 6

```
@ControllerAdvice
class ErrorControllerHandler {

    data class Error(
        val status: Int,
        val message: String?,
        val other: String
    )
}
```

We can add a custom Error object that will be returned instead of the default spring error response.

Therefore we introduce an inner class that is representing our custom error response here

# Custom Response status (option2 - controller advice)

## Exercise 6

```
@ControllerAdvice
class ErrorControllerHandler {

    data class Error(
        val status: Int,
        val message: String?,
        val other: String
    )

    @ExceptionHandler(TodoNotFoundException::class)
    fun handleTodoNotFoundException(ex: TodoNotFoundException) : ResponseEntity<Error> =
        ResponseEntity(
            Error(
                status = HttpStatus.NOT_FOUND.value(),
                message = ex.message,
                other = "look ma, this is custom stuff"
            ),
            HttpStatus.NOT_FOUND
        )
}
```

We are adding an exception handler to our controller advise class that is listening on TodoNotFoundExceptions. It will, whenever our TodoNotFoundException will be thrown, catch it automatically and return a so called ResponseEntity (object that comes with spring-boot) that wraps our error object as well as a certain http status code.

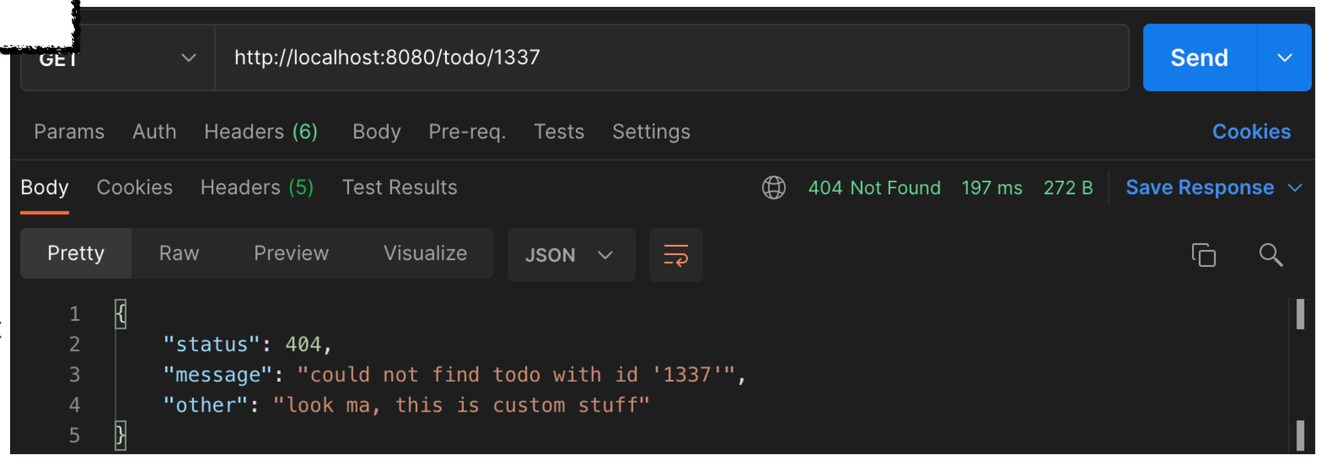
# Prove Controller advise works

## Exercise 6

TodoControllerTest.kt

```
59      @Test
60      fun `will return 404 on unknown task id requested`() {
61          mockMvc.get( urlTemplate: "/todo/1337")
62              .andExpect { this: MockMvcResultMatchersDsl
63                  status { isNotFound() } // check if status is 404 (not found)
64              }
65      }
```

Just rerun our test that checks if 404 is returned on unknown id. It should still be green.




Yay we are even getting a custom error response object =)

The *@ControllerAdvice* annotation allows us to consolidate our multiple *@ExceptionHandlers* into a single, global error handling component.
The actual mechanism is extremely simple but also very flexible:

- It gives us full control over the body of the response as well as the status code.
- It provides mapping of several exceptions to the same method, to be handled together.
- It makes good use of the newer RESTful *ResponseEntity* response.