SECTION 02 - ANGULAR SERVICE TESTING IN DEPTH

Index

- Inject mock version from HttpClient

```
import { HttpClientTestingModule } from "@angular/common/http/testing";

beforeEach(() => {
 TestBed.configureTestingModule({
   imports: [HttpClientTestingModule],
   providers: [CoursesService],
 });
});
```

- Retrieve mock data

```
import { HttpTestingController } from "@angular/common/http/testing";

httpTestingController = TestBed.inject<HttpTestingController>(
 HttpTestingController
);
```

- toBeTruthy

- Means that the service should not return *null* or *undefined*.
  `expect(courses).toBeTruthy('No courses returned');`

- toBe

- Used to compare the result against the expected one.
  `expect(courses.length).toBe(12, "incorrect number of courses");`

- expectOne("URL")

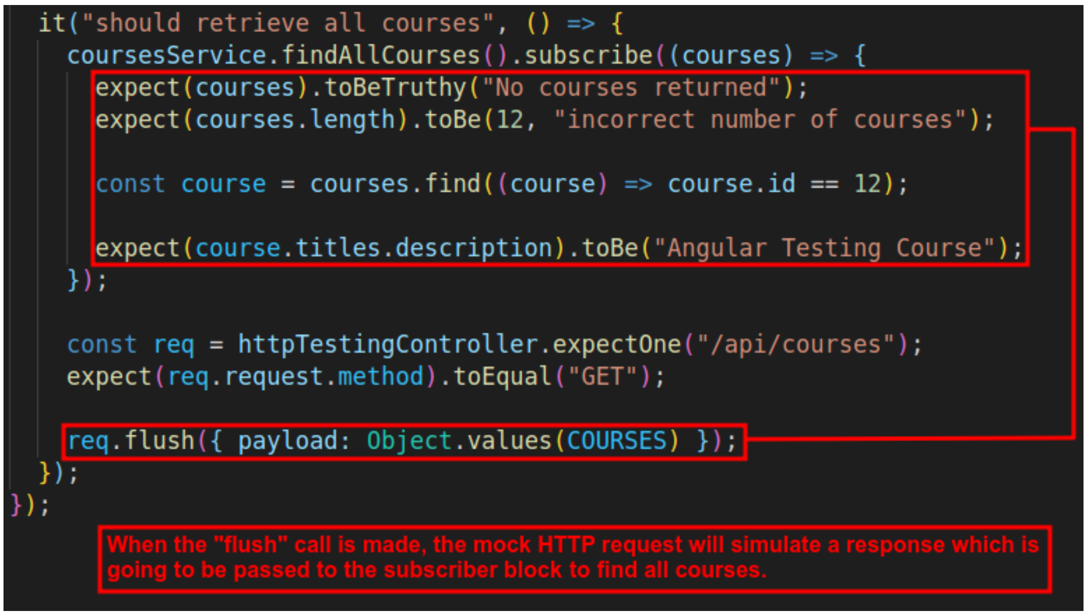
- Expect that a single request has been made which matches the given URL, and return its mock.

- On the mock returned we can assert for instance the type of request:
  `expect(req.request.method).toEqual("GET");`

- flush()

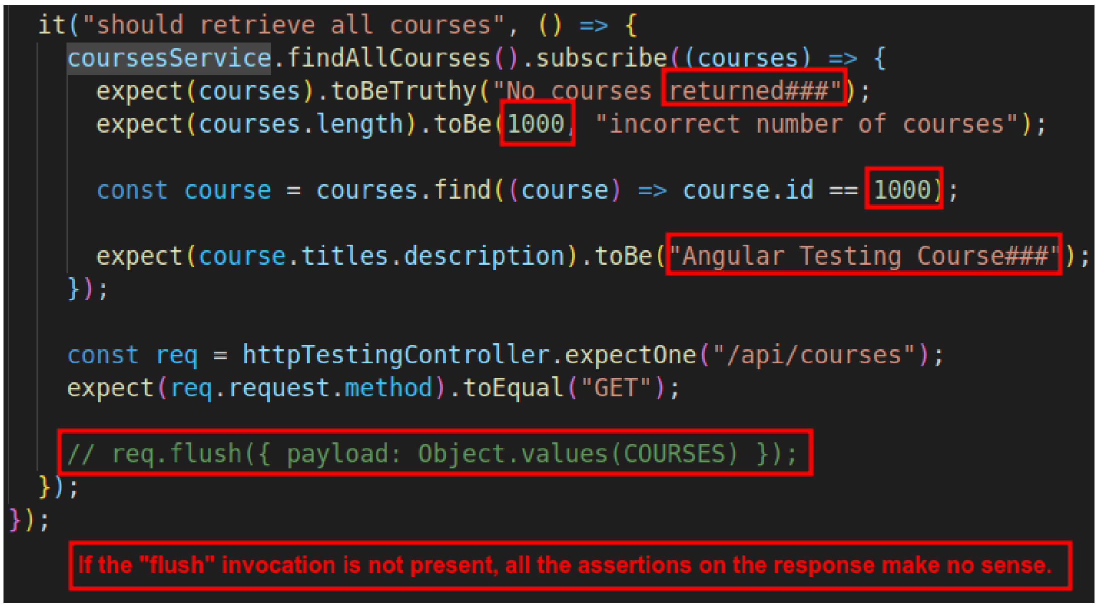
- When the *flush* call is made, **the mock HTTP request** will simulate a response which is going to be passed to the subscriber block to find all courses.

- So if I comment the *flush* invocation, no matter which values I expect, they all will be correct, so it will have no sense.



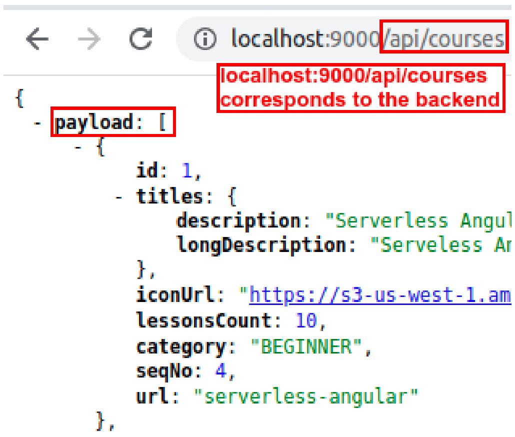
```
it("should retrieve all courses", () => {
  coursesService.findAllCourses().subscribe((courses) => {
    expect(courses).toBeTruthy("No courses returned");
    expect(courses.length).toBe(12, "incorrect number of courses");

    const course = courses.find((course) => course.id == 12);

    expect(course.titles.description).toBe("Angular Testing Course");
  });

  const req = httpTestingController.expectOne("/api/courses");
  expect(req.request.method).toEqual("GET");

  req.flush({ payload: Object.values(COURSES) });
 });
});
```

When the "flush" call is made, the mock HTTP request will simulate a response which is going to be passed to the subscriber block to find all courses.



```
 it("should retrieve all courses", () => {
  coursesService.findAllCourses().subscribe((courses) => {
    expect(courses).toBeTruthy("No courses returned###");
    expect(courses.length).toBe(1000, "incorrect number of courses");

    const course = courses.find((course) => course.id == 1000);

    expect(course.titles.description).toBe("Angular Testing Course###");
  });

  const req = httpTestingController.expectOne("/api/courses");
  expect(req.request.method).toEqual("GET");

  // req.flush({ payload: Object.values(COURSES) });
 });
});
```

If the "flush" invocation is not present, all the assertions on the response make no sense.



localhost:9000/api/courses

localhost:9000/api/courses corresponds to the backend

```
{
 - payload: [
    - {
          id: 1,
        - titles: {
              description: "Serverless Angul
              longDescription: "Serveless Ar
          },
          iconUrl: "https://s3-us-west-1.am
          lessonsCount: 10,
          category: "BEGINNER",
          seqNo: 4,
          url: "serverless-angular"
      },
```

Here is from where "payload" comes when it is used as a property of the object passed to the "flush" method

- Structure for testing a service method

```
describe("CoursesService", () => {
  let coursesService: CoursesService,
    httpTestingController: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [CoursesService],
    });

    coursesService = TestBed.inject<CoursesService>(CoursesService);
    httpTestingController = TestBed.inject<HttpTestingController>(
      HttpTestingController
    );
  });

  it("should retrieve all courses", () => {
    coursesService.findAllCourses().subscribe((courses) => {
      expect(courses).toBeTruthy("No courses returned");
      expect(courses.length).toBe(12, "incorrect number of courses");

      const course = courses.find((course) => course.id == 12);

      expect(course.titles.description).toBe("Angular Testing Course");
    });

    const req = httpTestingController.expectOne("/api/courses");
    expect(req.request.method).toEqual("GET");

    req.flush({ payload: Object.values(COURSES) });
  });
});
```

Create the service by dependency injection and inject to it its HttpClient dependency.

Inject **HttpTestingController** in order to simulate the call to the server.

By invoking ***expectedOne***("URL") we define a **Mock Http Request object** to assert that the request has been made only once and it returns the request so then by invoking ***flush*** on the request we pass test data to the mock request.

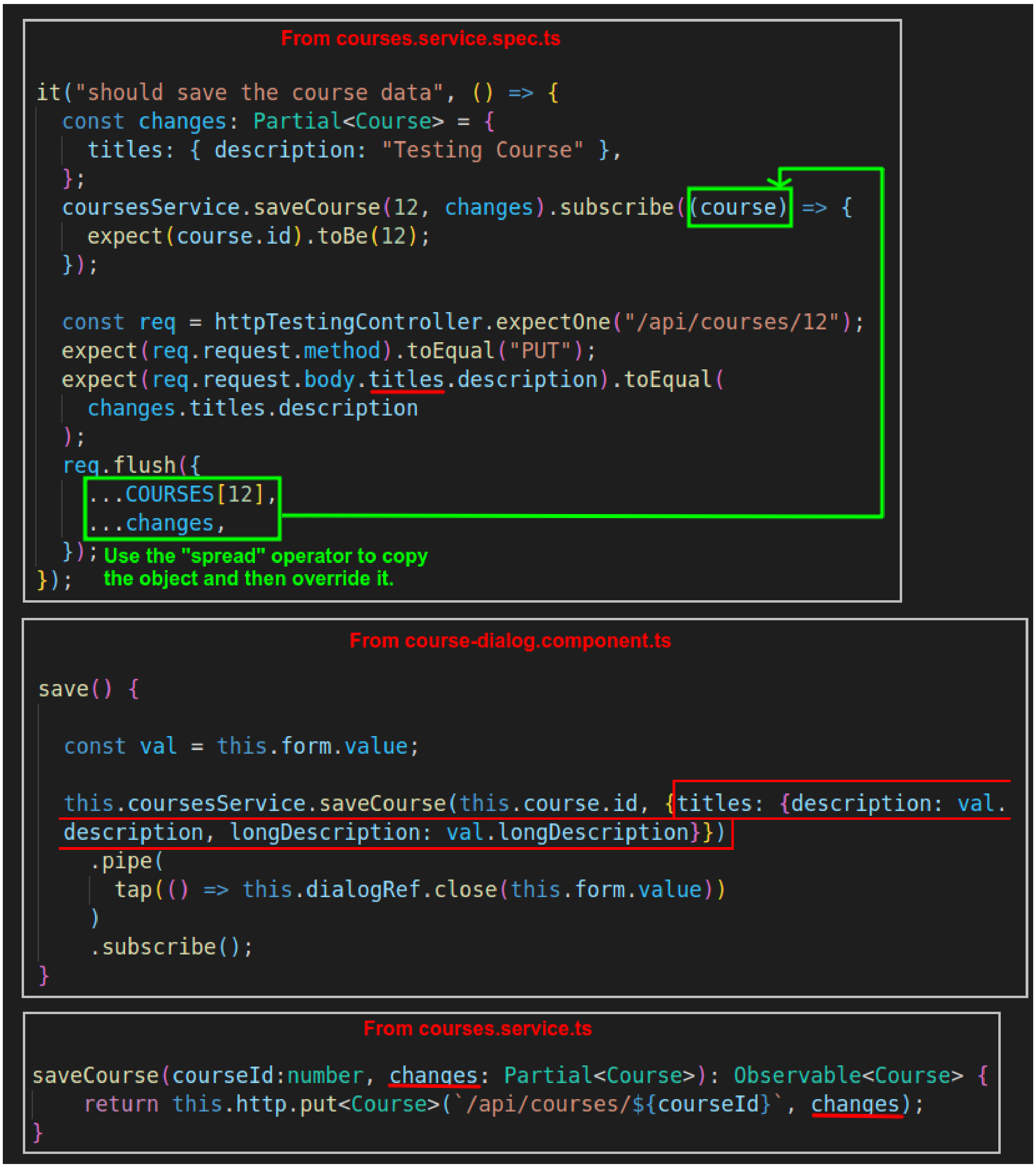
- verify()

- httpTestingController.**verify()** asserts that no other request was invoke other than the defined in *expectOne()* method.

- It's a good practice to put it in the afterEach method:

```
afterEach(() => {
 httpTestingController.verify();
})
```

- Testing "save" service method



From courses.service.spec.ts

```
it("should save the course data", () => {
  const changes: Partial<Course> = {
    titles: { description: "Testing Course" },
  };
  coursesService.saveCourse(12, changes).subscribe((course) => {
    expect(course.id).toBe(12);
  });

  const req = httpTestingController.expectOne("/api/courses/12");
  expect(req.request.method).toEqual("PUT");
  expect(req.request.body.titles.description).toEqual(
    changes.titles.description
  );
  req.flush({
    ...COURSES[12],
    ...changes,
  });
});
```

Use the "spread" operator to copy the object and then override it.

From course-dialog.component.ts

```
save() {

  const val = this.form.value;

  this.coursesService.saveCourse(this.course.id, {titles: {description: val.
  description, longDescription: val.longDescription}})
    .pipe(
      tap(() => this.dialogRef.close(this.form.value))
    )
    .subscribe();
}
```

From courses.service.ts

```
saveCourse(courseId:number, changes: Partial<Course>): Observable<Course> {
    return this.http.put<Course>(`/api/courses/${courseId}`, changes);
}
```

- Testing error handling

```
it("should give an error if save course fails", () => {
  const changes: Partial<Course> = {
    titles: { description: "Testing Course" },
  };

  coursesService.saveCourse(12, changes).subscribe(
    () => fail("the save course operation should have failed"),
    (error: HttpErrorResponse) => {
      expect(error.status).toBe(500);
    }
  );
  const req = httpTestingController.expectOne("/api/courses/12");
  expect(req.request.method).toEqual("PUT");

  req.flush("Save course failed", {
    status: 500,
    statusText: "Internal server error",
  });
});
```

- Testing a paginated request

```
it("should find a list of lessons", () => {
  coursesService.findLessons(12).subscribe((lessons) => {
    expect(lessons).toBeTruthy();
    expect(lessons.length).toBe(3);
  });
  const req = httpTestingController.expectOne(
    (req) => req.url == "/api/lessons"
  );
  expect(req.request.method).toEqual("GET");
  expect(req.request.params.get("courseId")).toEqual("12");
  expect(req.request.params.get("filter")).toEqual("");
  expect(req.request.params.get("sortOrder")).toEqual("asc");
  expect(req.request.params.get("pageNumber")).toEqual("0");
  expect(req.request.params.get("pageSize")).toEqual("3");

  req.flush({
    payload: findLessonsForCourse(12).slice(0, 3),
  });
});
```