Hello! The goal of this video is to share with you a practical example of the TDD Outside In approach by doing a Code Kata.
The Code Kata is called Greeting Service Kata and it is available on GitHub, you will find the link in the description of this video.

The overall idea of TDD Outside In approach is to start implementing a new feature from the outermost part of our application by describing the behaviour we want to expect through an automated test.

So, let's get started and give a look at the requirements of the Code Kata. [try to read the requirements to provide context?]

The first feature we are going to work on is about *Greeting a User*, so, if we have to try to follow the TDD Outside In approach, the first step is to write a test that will document this feature from the outermost part of the application.
In our case, the most external part of the application can be an HTTP client. We expect to run an HTTP request against a particular endpoint and to receive back the greeting message as part of the HTTP response.

[here I will show how the test might look like]

Run the test, and it should be RED, as expected, the first step is to see a red bar.
Now, no matter what, our mission is to make this test pass quickly. And, we are allowed to do anything is needed to achieve a GREEN bar.

For practical reasons I already prepared few code snippets as a support for this demostration. Let's start with running the web server first. Run the test, and here we are, now we see we are getting a different error. It says the endpoint we are trying to hit does not exist.

The simplest move we can think of, in order to make the test to pass, is to make the endpoint available and return the message we expect to receive.

Run the test… And here we are! Green bar… Time to proceed further with the next point of the feature we are building.

Greeting a User with a customized message!

Again, now that we've understood what the flow is, let's try to describe the new behaviour we want to see in our application by writing a test first. Always starting from the outermost part of the application.

[here I will show how the test might look like]

Run the test first, here we go, we got a RED bar, as expected. Our mission now is to try to find the simplest way to go in a GREEN bar, at this point we are not interested in doing it the right way. Just do the minimal change that give us a GREEN bar.

The thing that come into my mind is to put a conditional on the query parameter, so that if it is set, our code will return the expected message. Run the test now and here we got a GREEN bar now.

That was the simplest possible change to make the test pass. We have stubbed the implementation, let's see if we can work to replace this constant with a variable. Run the test again, and here we go, GREEN bar.

From the point of view of the first requirement, we can assume that the first feature is now complete. But what about the second requirement? As we can see, right now the only way to access the feature is through an HTTP layer. No way to access it through a command line.

One viable way I see to make this feature available through a command line is to proceed with a refactor and trying to find a way to separate the way we use to handle the HTTP request and response from the part of the code that holds the logic that generates the greeting message. So, how to do that?

I would proceed with small steps that can help me to extract the logic part from the HTTP part.

[show and explain the refactor steps needed to extract the logic from the presentation layer]

Let's try to see how the tests looks like now that we've extracted the logic from the HTTP part. If we compare the HTTP tests with the GreetingService module tests, we can notice that they are basically describing the same thing. What can we tell about the HTTP tests? What is the behaviour we really expect to escribe here?

Before answer this question, I would like to see how the code responsible to handle the HTTP request and response, changed, after our refactor.

As we can see, the only responsibility that this module has, it to get a query parameter and to pass it down to the GreetingService module. This is the thing I would like to describe through the HTTP tests.

[Add the tests to describe the collaboration between the HTTP and the GreetingService module]

And here we go! Requirements are full-filled and the first feature is complete.

Now the things are going to get more complicated, I think, let's give a look at the second feature.

[move the attention to the second feature and talk a bit about it]

Greeting a User by choosing a random greeting message from a set of messages.

How could we describe it? Let's try to write a test for that feature. As always we will write a test starting from the outermost part of the application. That's the HTTP layer, in this case.

[copy and paste the existing test and then test starting from the expectation… Explain that we are not going to modify the existing test, but just creating a new one]

Here it is, now run the tests and see what happens… Great! Already GREEN, but we know this is a new feature. The why this test is already GREEN is because, by coincidence, one of the random greeting messages that we expect to receive is exactly the same of the first feature.

In situation like this, the process I generally tend to follow is to find a way to make the test of the first feature fails, by writing the code that is needed to accomodate the new feature. And then, the RED bar at that point will be an indicator that tells us the feature is not in place anymore, and we are allowed to remove the old test, while the test of the second feature should be GREEN.

[TBD. Try to reason about how to end up with a set of predefined messages in the greeting service module.]

Let's look where we do expect to make some code changes in order to implement this feature. From the point of view of the HTTP, seems nothing has to be done. As we already discussed before, the role of the HTTP is to handle the request and response from and to the GreetingService module. We are good with the state of this part of the code.

I am more interested in looking at the GreetingService module. If something has to change, probably is here. I imagine that this module - to make it easy - can return a message randomly picked from a predefined list.

Let's write a test for it.

[write the test for the GreetingService module]

And, as expected, if we run the test, at some point we should see that the first test should start to fail, together with the feature test we wrote before. This is the right moment to delete the outdated test from our test suite. Let's remove these failing tests, run the test suite again, and, here we go, we are in a GREEN bar, again.

At this point, I would not proceed with any refactor right now. I am quite comfortable with the current solution. Instead, I would like to move on, and trying to look for the next feature.

This time we are asked to greet the user with a different greeting message, based on the time of the day. I am quite curious to see how the development of this feature will affect our design, or, in a opposite way, how the design should change in order to accomodate this new requirement.

*Greeting a User by choosing a greeting message from a predefined set of messages, **based on the time of the day**.*

[Take some time to read and explain the feature, look in the code and see which parts will - eventually - change.]

For this feature, we are asked to change the greeting message based on a particular hour of the day. So, we should choose a greeting message from a different list of predefined message, based on the hour of the day.

One of the thing that comes to my mind is that we need to figure out how to retrieve the current hour of the day. Then, based on that, decide which greeting message to show.

I would start with the strategy to split up the feature into three different scenarios, one for each time interval, we have:

- **Morning (from 7 AM to 11 AM)**
- The rest of the day (from 12 PM to 8 PM)
- Night (from 9 PM to 6 AM)

For the purpose of this video, I will implement only the first time interval, then I will leave you - as an exercise - the change to complete the other two time intervals.

As always, for a new scenario, I will start by writing a test from the outermost part of the application. So, in this case we want to return a particular message only when the hour of the day is in the morning interval.

[Write the test and let the audience notice that we need some sort of mechanism to help us to retrieve the current hour of the day. We should not rely on the date time of the system. This will not allows us to have **reliable** tests.]

Let’s consider to inject a new collaborator, called **HourOfTheDay** that we will use to get the current hour of the day. Looking at the current design of our application, the client (the piece of code that should use this collaborator) should be the GreetingService module. What I imagine, is that the GreetingService module will ask the current hour to this new collaborator, and based on that, will pick a particular greeting message.

Said that, take a look at the code of the GreetingService and see how it should change in order to accept its new collaborator.

I would start with a test for the GreetingService so that we can clearly see how the collaboration between the GreetingService and its collaborator should behaves.

[Write the test for the GreetingService module that accepts the HourOfTheDay as argument in the greet function. I will start with the hour ***anyMorningHour***]

Let’s run the test, and as expected RED bar. What’s the minimum amount of code that is needed to get a GREEN bar?

[Make the test pass, and then move on with the other hour of the morning until we remove the duplication]

I am going to try to remove all this duplication. The idea is to introduce a collaborator that can be programmed based on the hour of the day we expect to get. A way to achieve this goal is to use macros, in Elixir.

[Extract a programmable HourOfTheDay collaborator]

Good, we are now done with the Morning interval. And, now the test from the outermost part should be GREEN, also, and not flacky anymore!

Another approach to end up with a programmable object would have been to use any mocking library out there (*one thing that I strongly advise to do, instead of reinventing the wheel*). This was just a demostrantion, so, I didn’t want to introduce any external library.
If you are curious you can check one of the branch of this code on GitHub where I used a mock library instead of use the macros. Look at the description of this video, you will also find another solution with a different design.

[At this point I could consider to end the episode and let the audience to try to complete it. An example: What about the real HourOfTheDay collaborator, that return the hour from the system, and should be used when we start the application?]

At this point I would like to invite you to try to complete this Code Kata, try to follow the same process to add the support for the remaining time intervals, as we saw before: the *Rest of the day* and the *Night*. If you are familiar with Elixir, you can just clone this code from GitHub, switch to the correct branch and continue from there.
Or, you can try to repeat it from scratch, trying to use the programming language you are more familiar with.

The Goal of this video was to give you an idea and a practical example about how the TDD Outside In approach could looks like in a real life application.

Hope you enjoyed this video! And I wish you an happy deliberate practice! Bye bye :)

[Motivate why we are extracting the collaborator for the list of messages. *Consider that we will end up by having a private function*.]