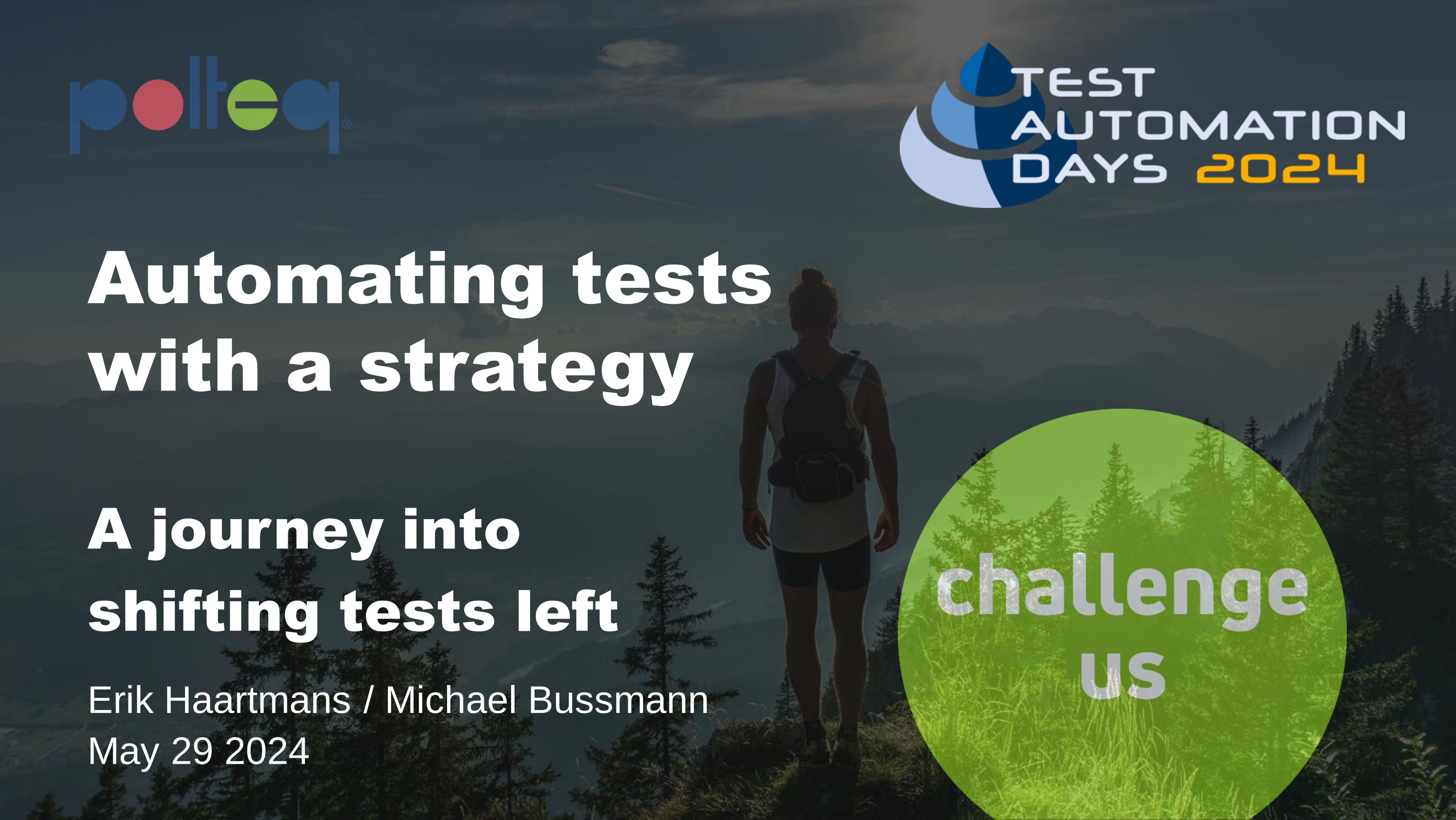
Automating tests with a strategy

A journey into shifting tests left

Erik Haartmans / Michael Bussmann

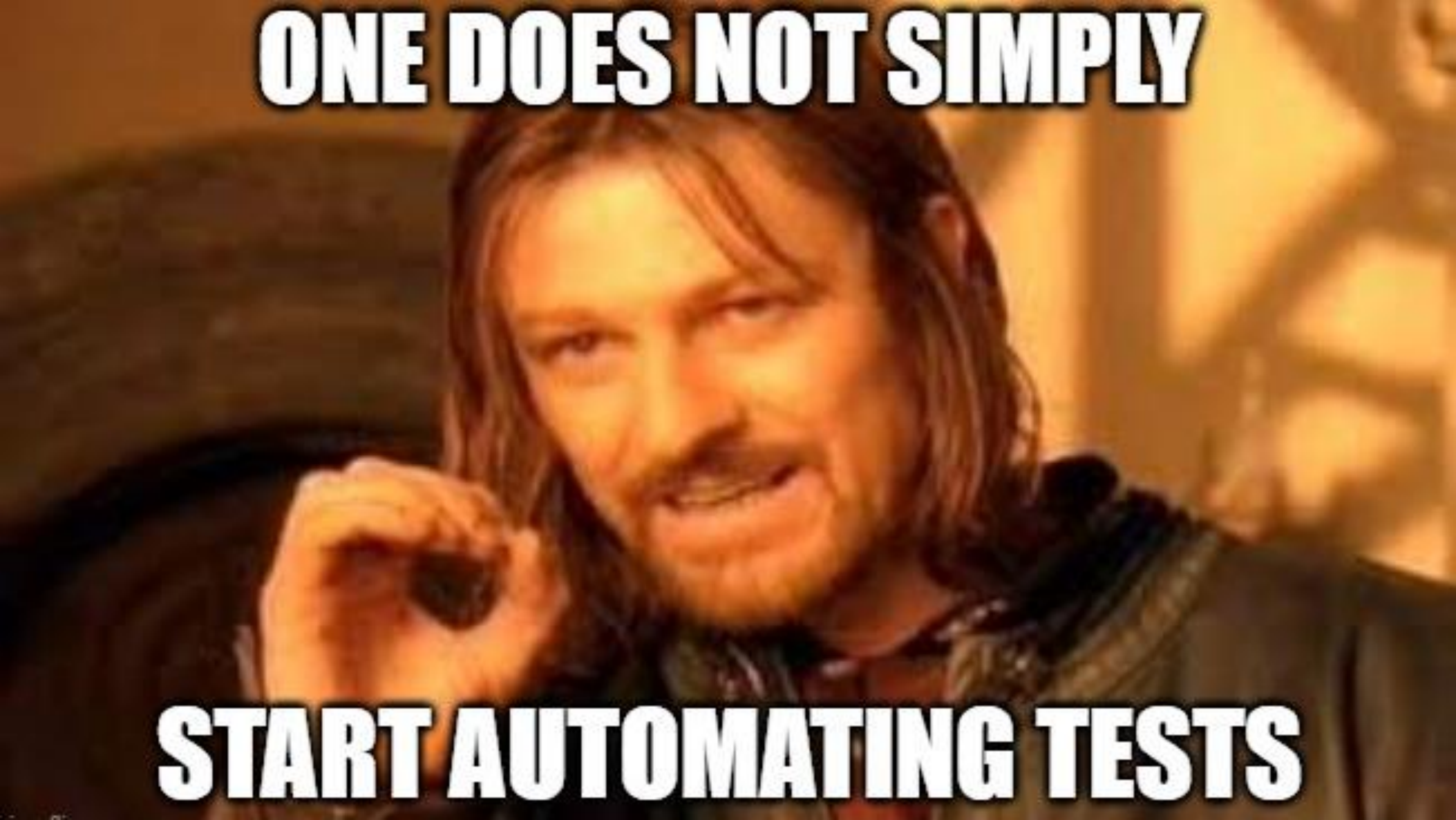
May 29 2024

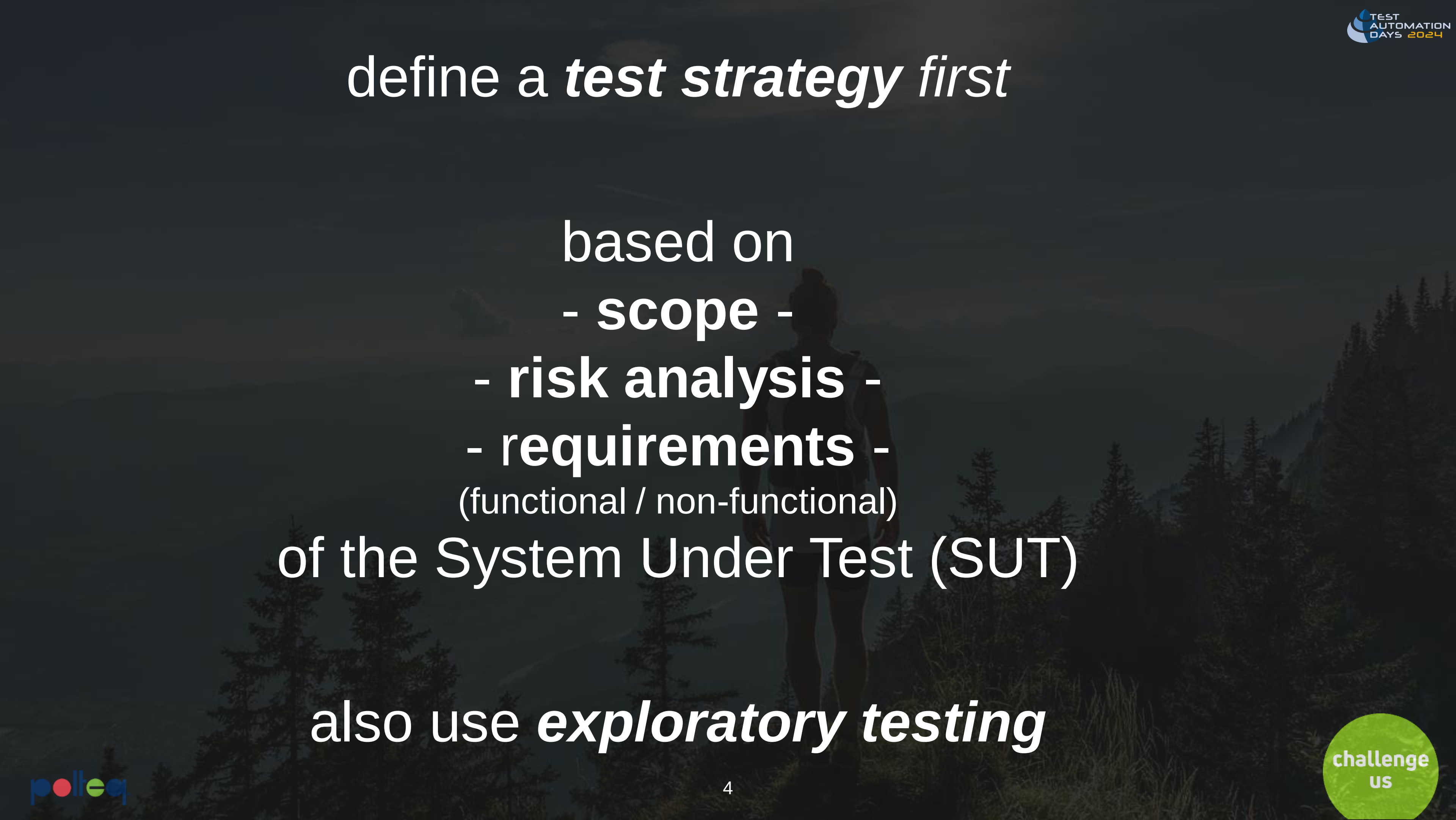
challenge us



YOUR AUTOMATION ENGINEERS WERE SO PREOCCUPIED WITH WHETHER THEY COULD...

... THEY DIDN'T STOP TO THINK IF THEY SHOULD


ONE DOES NOT SIMPLY
START AUTOMATING TESTS



define a ***test strategy*** *first*

based on

- **scope** -

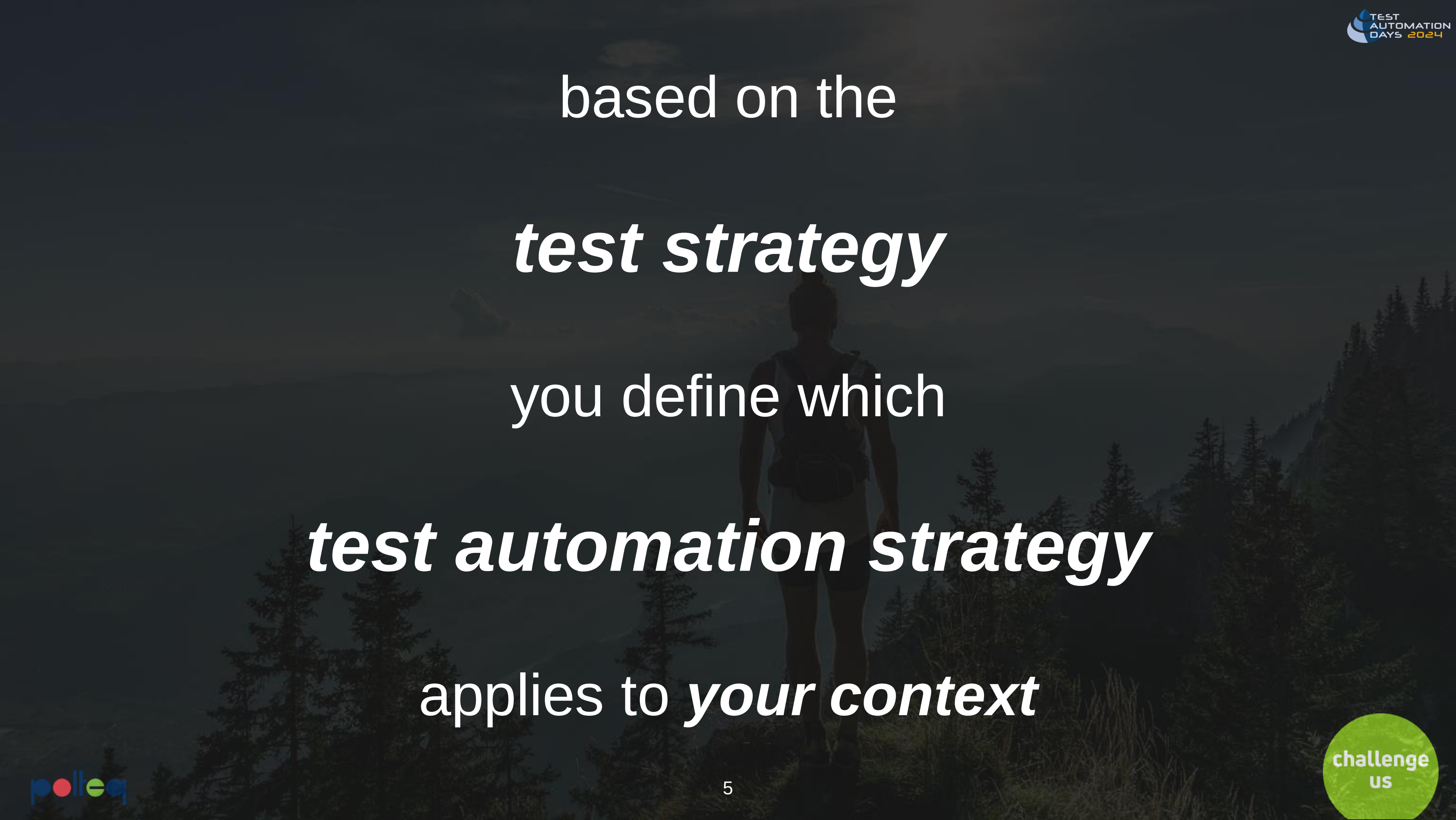
- **risk analysis** -

- **requirements** -

(functional / non-functional)

of the System Under Test (SUT)

also use ***exploratory testing***



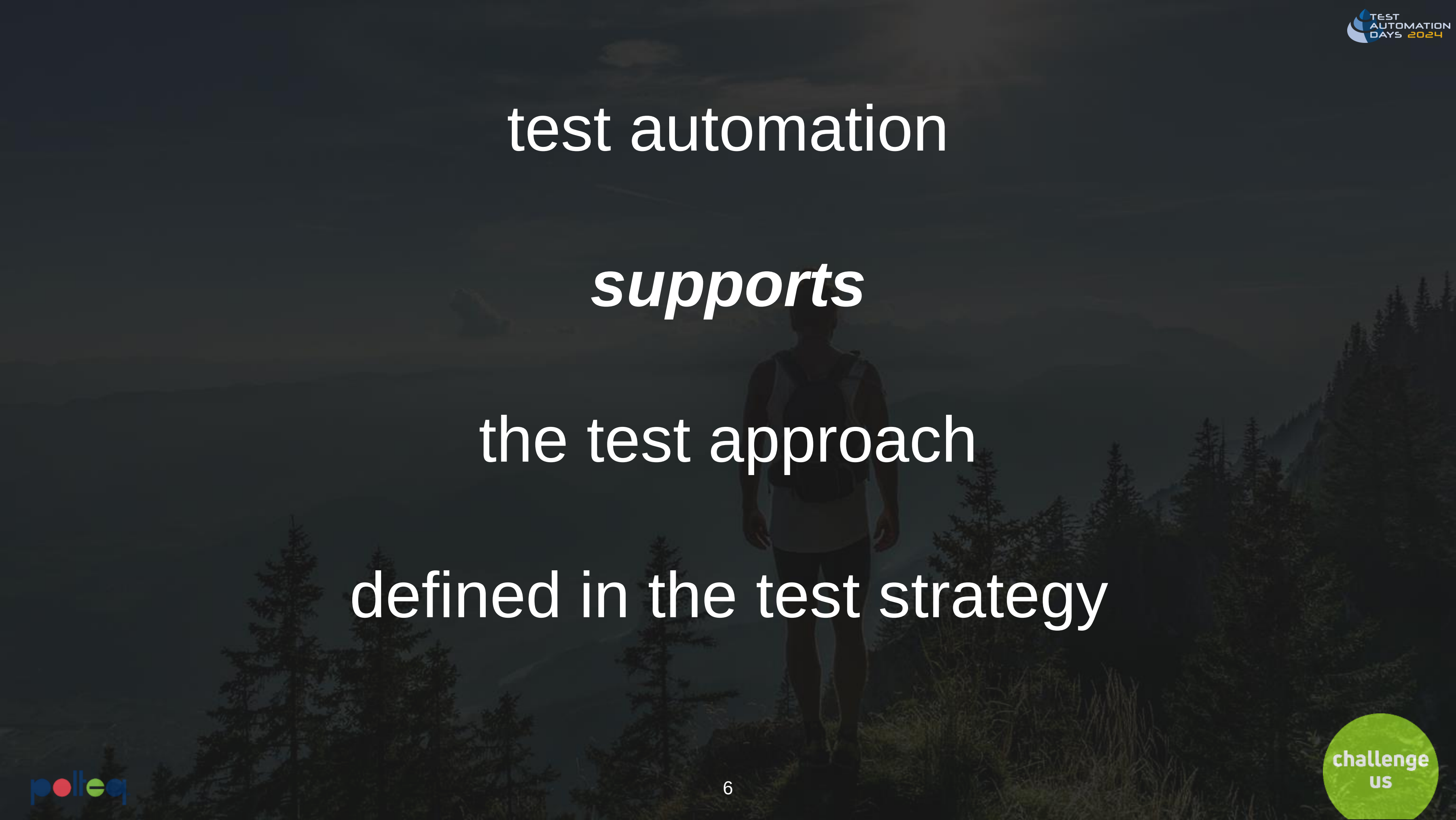
based on the

test strategy

you define which

test automation strategy

applies to ***your context***



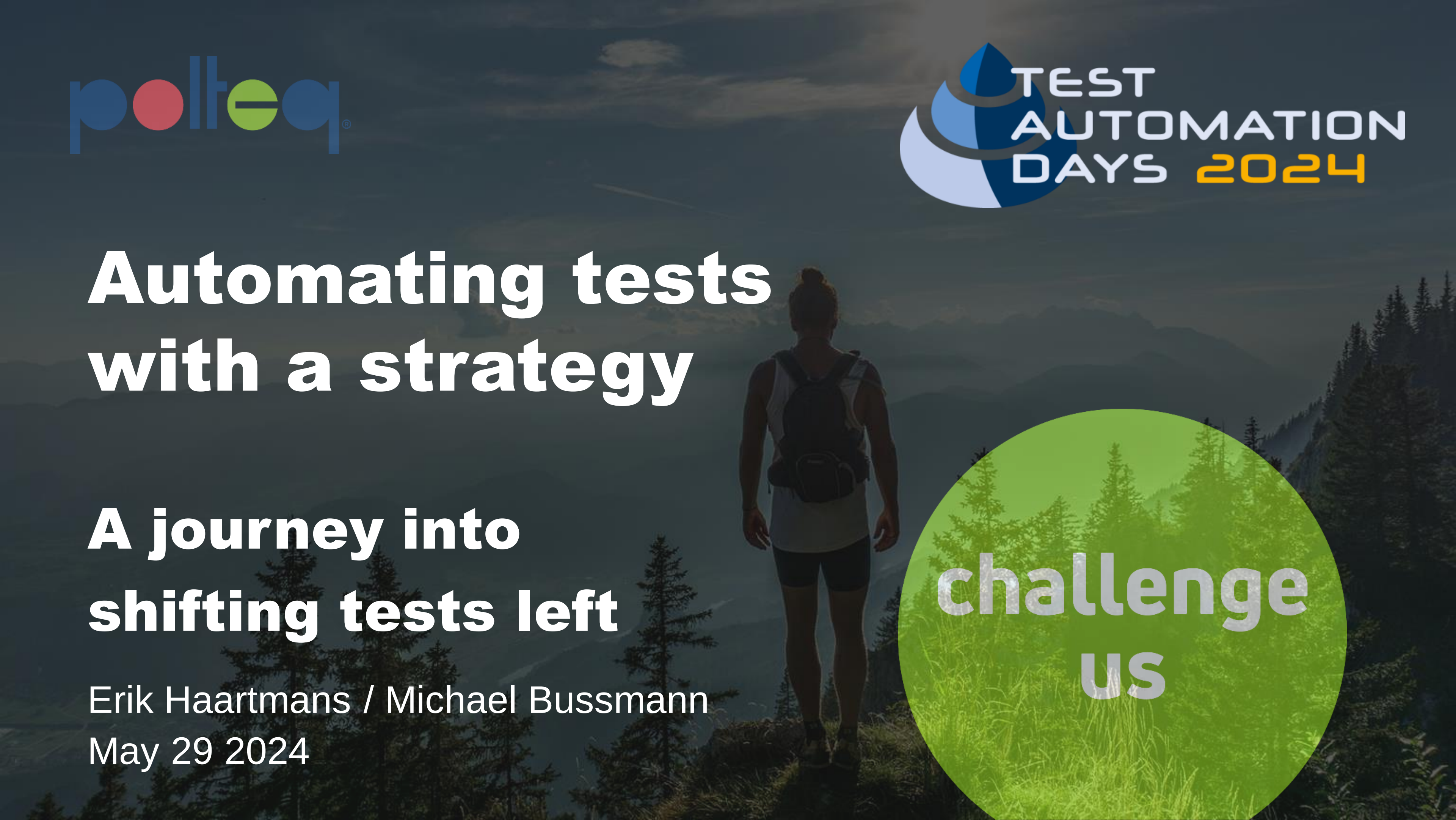
test automation

supports

the test approach

defined in the test strategy

back to the topic



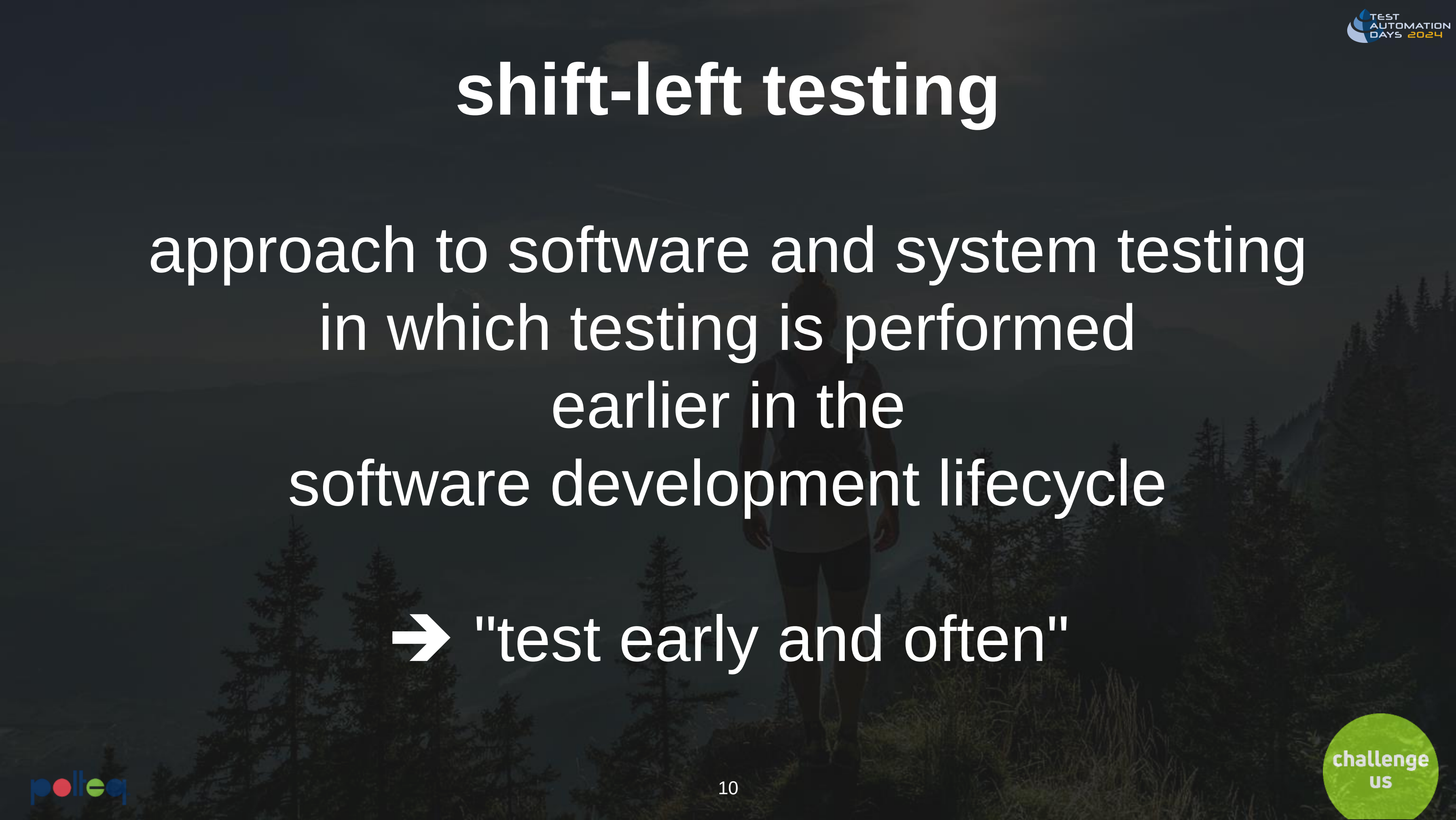
Automating tests with a strategy

A journey into shifting tests left

Erik Haartmans / Michael Bussmann
May 29 2024

What does "shifting tests left" mean to you?

challenge us



shift-left testing

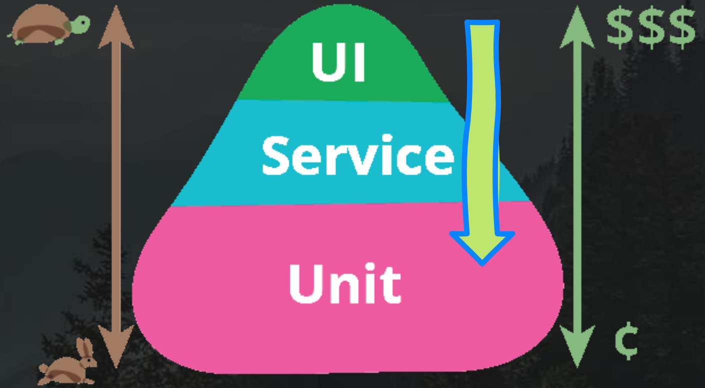
approach to software and system testing
in which testing is performed
earlier in the
software development lifecycle

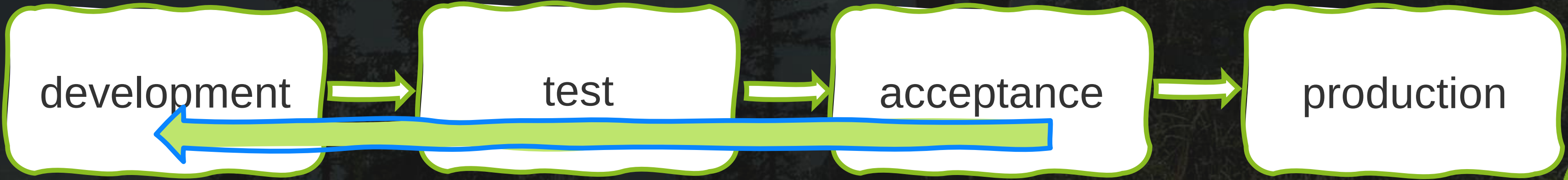
➔ "test early and often"

shift-left testing

approach to software and system testing
in which testing is performed
earlier in the
software development lifecycle

➔ "test early and often"





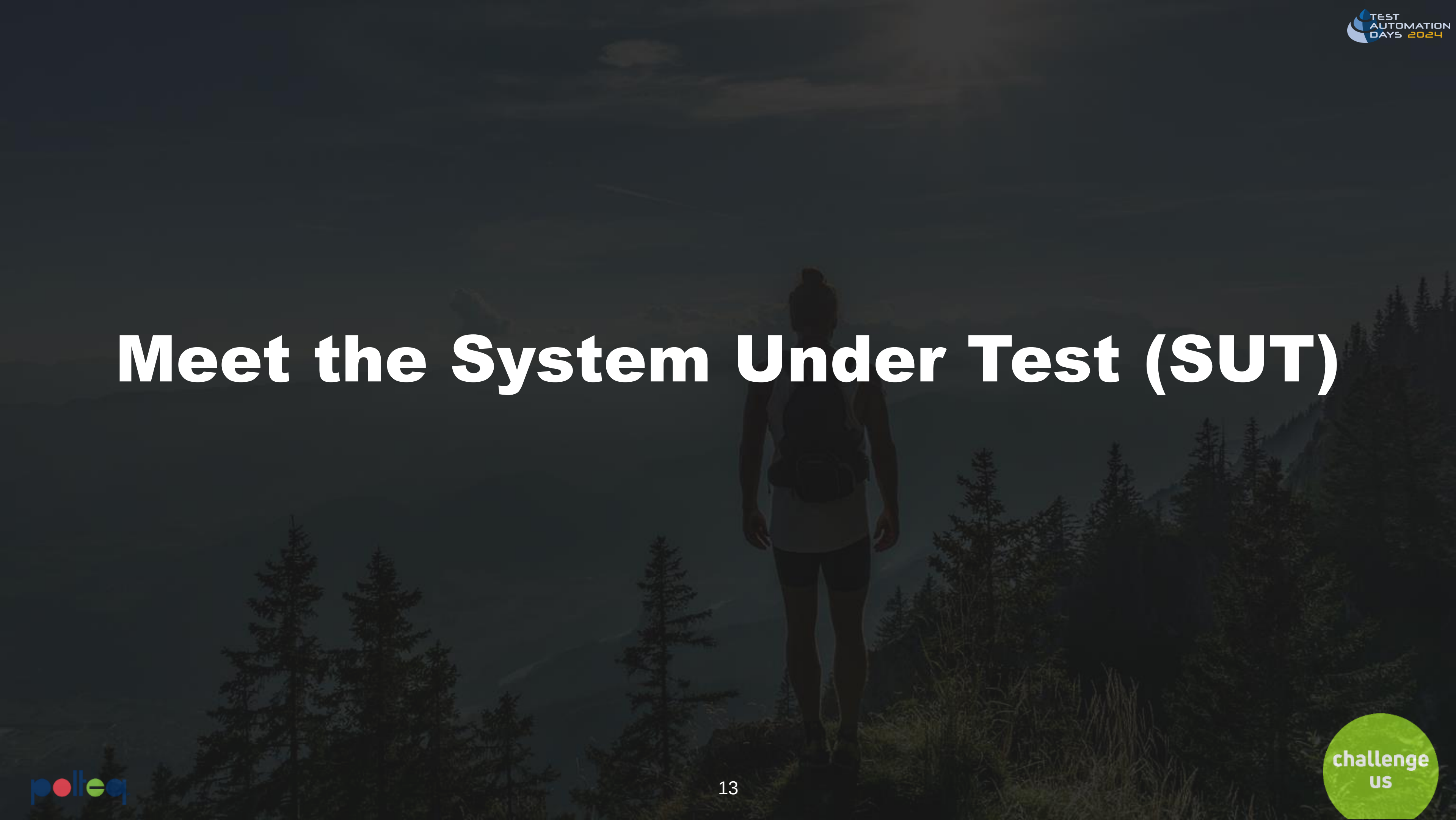
challenge us

Goal of this masterclass

- Not about creating a lot of tests

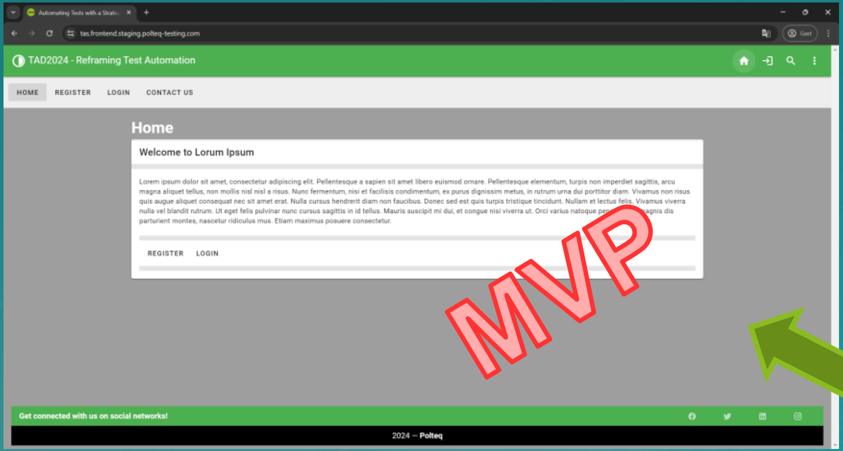
- It's about: Learn using approaches how to
  - test on various "levels"
  - use different test automation techniques & tools
    - Java/junit/Playwright/REST-assured/Docker/mocking
    - TypeScript/Playwright/Docker/mocking

challenge us



Meet the System Under Test (SUT)

Meet the System Under Test (SUT)



Automating Tests with a Strate

tas.frontend.staging.polteq-testing.com

TAD2024 - Reframing Test Automation

HOME REGISTER LOGIN CONTACT US

Home

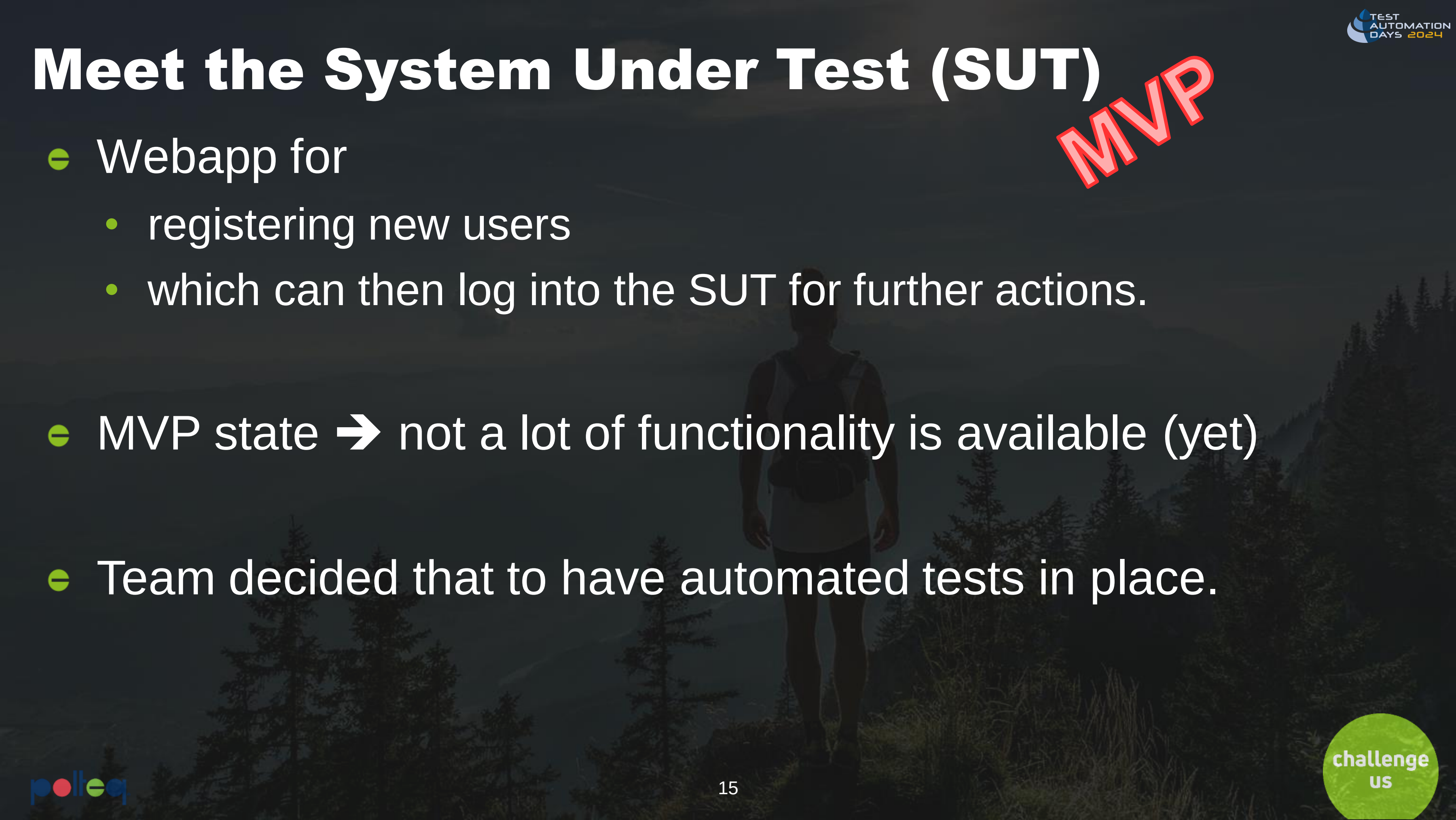
Welcome to Lorum Ipsum

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque a sapien sit amet libero euismod ornare. Pellentesque elementum, turpis non imperdiet sagittis, arcu magna aliquet tellus, non mollis nisl nisl a risus. Nunc fermentum, nisi et facilisis condimentum, ex purus dignissim metus, in rutrum urna dui porttitor diam. Vivamus non risus quis augue aliquet consequat nec sit amet erat. Nulla cursus hendrerit diam non faucibus. Donec sed est quis turpis tristique tincidunt. Nullam et lectus felis. Vivamus viverra nulla vel blandit rutrum. Ut eget felis pulvinar nunc cursus sagittis in id tellus. Mauris suscipit mi dui, et congue nisi viverra ut. Orci varius natoque pen[illegible] magnis dis parturient montes, nascetur ridiculus mus. Etiam maximus posuere consectetur.

REGISTER LOGIN

Get connected with us on social networks!

2024 — Polteq



Meet the System Under Test (SUT)

MVP

- Webapp for
  - registering new users
  - which can then log into the SUT for further actions.
- MVP state → not a lot of functionality is available (yet)
- Team decided that to have automated tests in place.

challenge us

Meet the System Under Test (SUT)

- The SUT is deployed in the staging environment:
  - https://tas.frontend.staging.polteq-testing.com
- Please explore the SUT

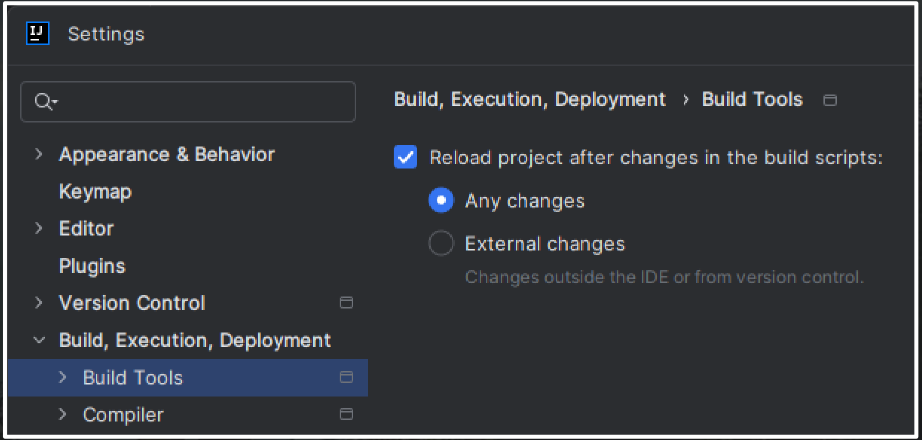
E2E testing repository

- End-to-end (E2E) testing repository already available.
  - This repo can be found here: https://github.com/erik-haartmans/tas-e2e-testing-repo-java

- Clone or download this repo to your laptop
- Open this repo with IntelliJ IDEA

Check the E2E testing repo

- Set the maven settings via menu `File / Settings` to `Any changes` to reload the project after changes are made to the pom.xml file.



Settings

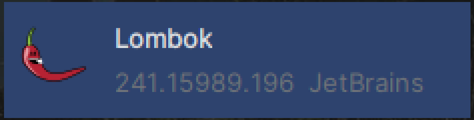
Build, Execution, Deployment › Build Tools

[✓] Reload project after changes in the build scripts:

[✓ Any changes] External changes

Changes outside the IDE or from version control.

- Also install the Lombok plugin
  - File / settings / plugin

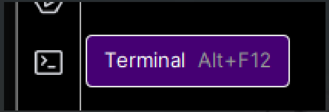


Lombok

241.15989.196 JetBrains

Check the E2E testing repo

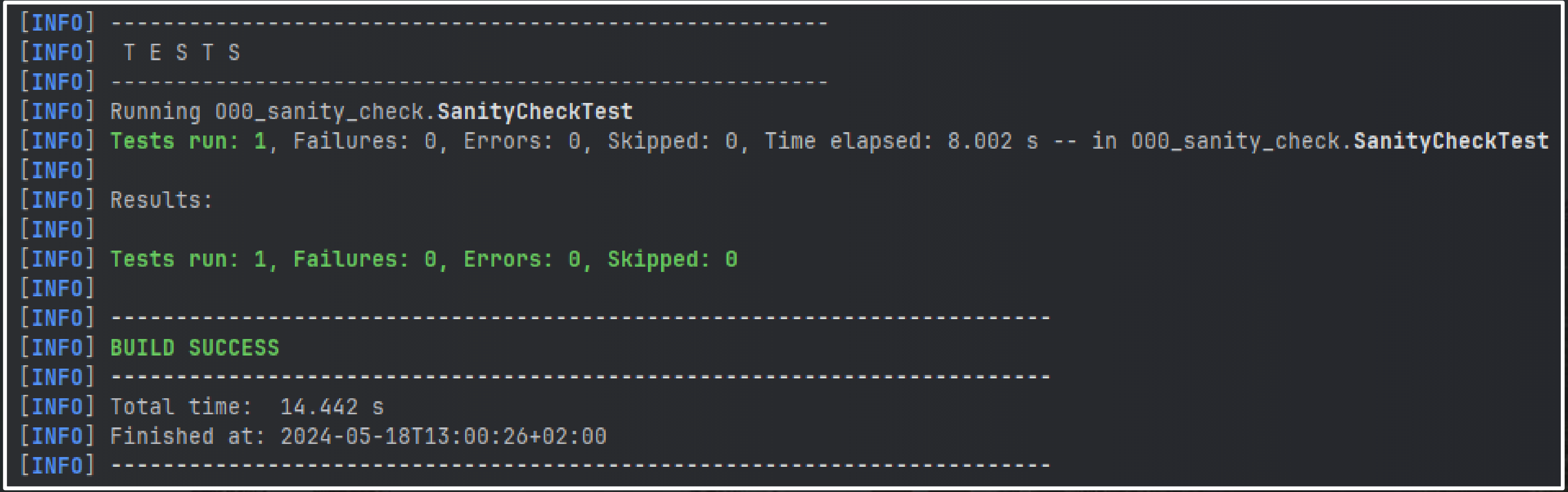
- Open the terminal (View / Tools Windows / Terminal) in IntelliJ and compile the project using the following command:
  - **`./mvnw clean test -Psanity-tests`**
    - mac users might have to give more permissions to mvnw ➔ **`chmod +x mvnw`**



- This will check that all settings are correct
  - A simple Playwright test runs on your machine

Check the E2E testing repo

- The output of the test should look like this:



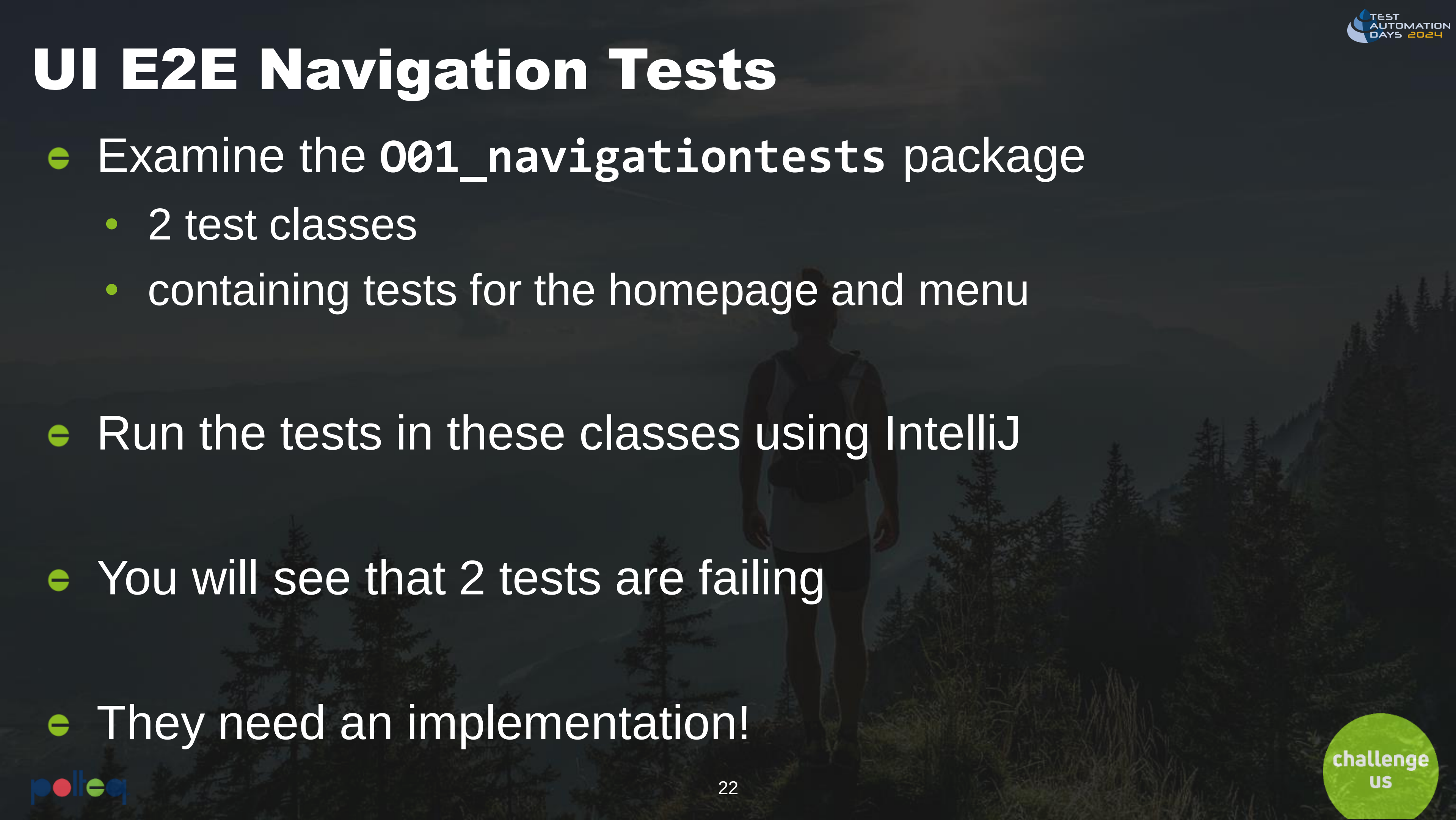
```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running 000_sanity_check.SanityCheckTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 8.002 s -- in 000_sanity_check.SanityCheckTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  14.442 s
[INFO] Finished at: 2024-05-18T13:00:26+02:00
[INFO] ------------------------------------------------------------------------
```

UI E2E Tests of the SUT

- Used tooling
  - Java / Playwright
  - Because most developers develop in Java
    - Easy to get support from the devs

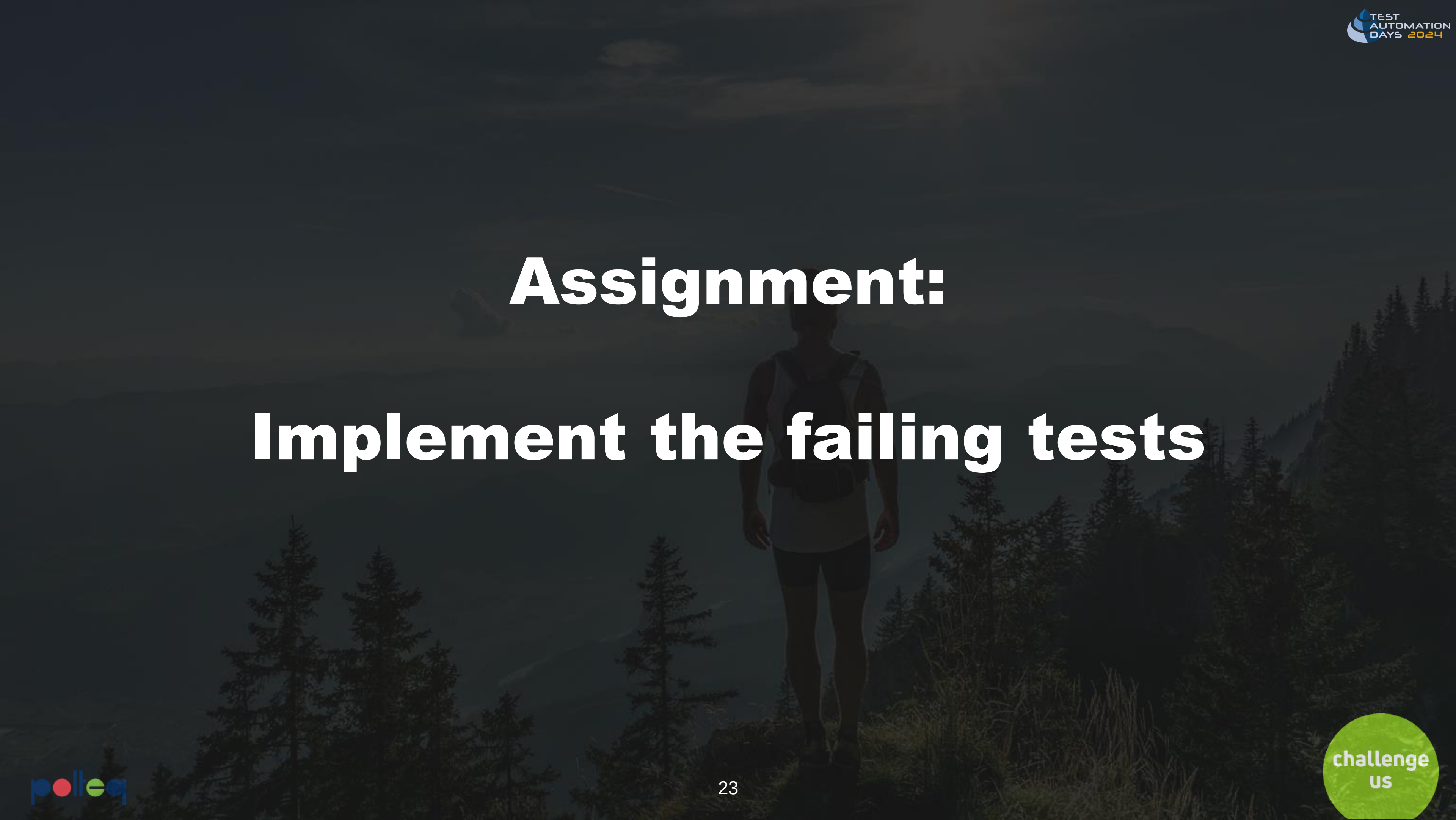
- The UI tests are located in the **`001_uitests`** package
  - Located in **`/src/test/java`**
  - Modular Playwright setup
    with tests using the Page Object Model

challenge us



UI E2E Navigation Tests

- Examine the **`001_navigationtests`** package
  - 2 test classes
  - containing tests for the homepage and menu
- Run the tests in these classes using IntelliJ
- You will see that 2 tests are failing
- They need an implementation!



Assignment:

Implement the failing tests

UI E2E Registration Tests

- Examine the `002_registrationtests` package
  - 3 test classes containing various tests
  - RegisterFormValidationTests
  - InvalidRegistrationTests
  - ValidRegistrationTests

UI E2E Registration Tests

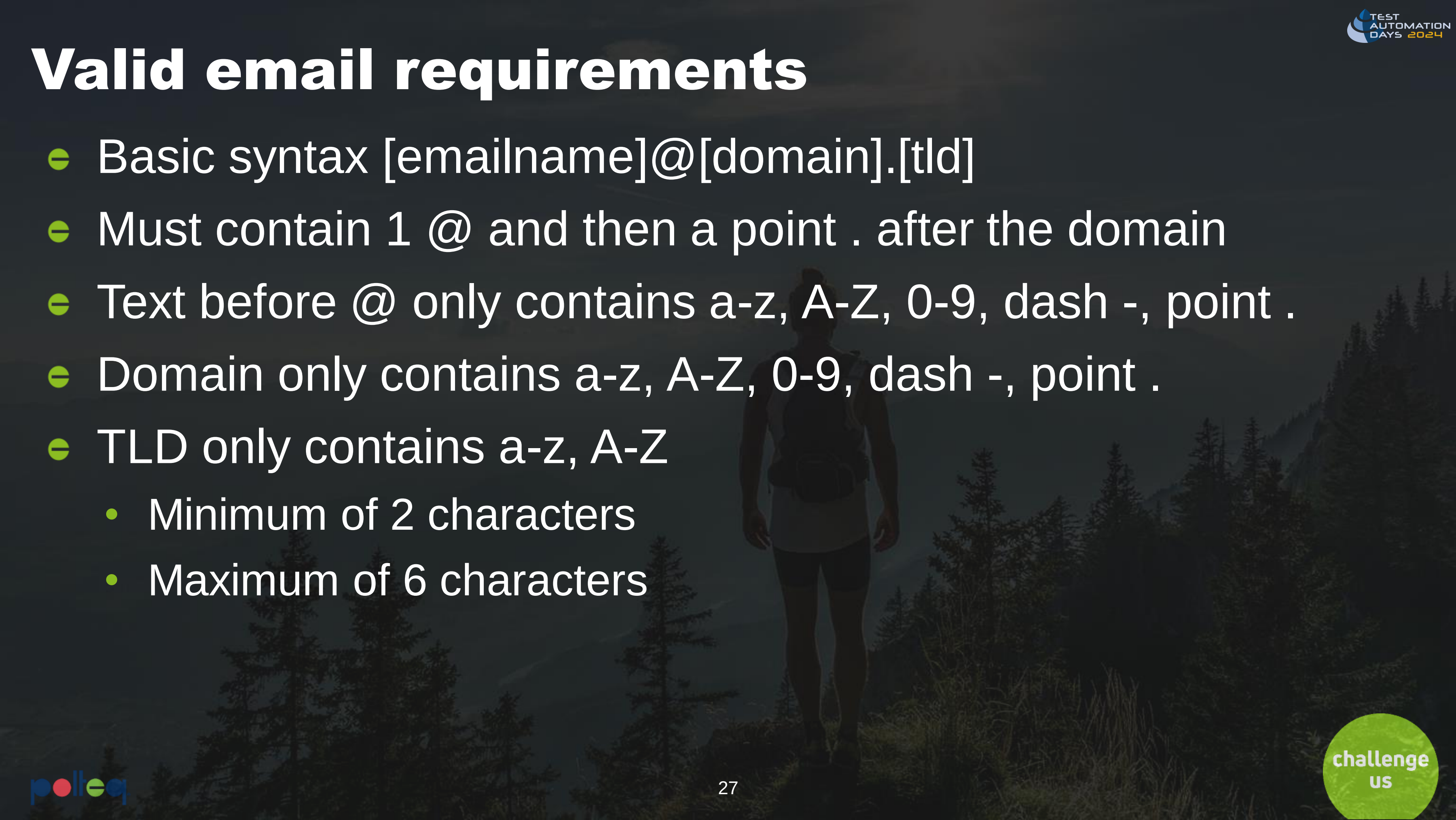
- Test `itShouldBePossibleToRegisterWithValidEmail`
  - in `ValidRegistrationTests` class
  - parameterized tests testing the requirements of a valid email

- Test `anErrorShouldBeShownWhenAnInvalidEmailIsUsed`
  - in `InvalidRegistrationTests` class
  - parameterized test testing the requirements of an invalid email

- Requirements on next pages

Register form requirements

- Fields with * are required
  - Error '<field> is required' will be shown
- Email must be according to the format
  - Error 'Email is not valid' will be shown
- You can only click register in the form when:
  - Terms are accepted
  - Both passwords are the same
- A user cannot register with the same email more than once
  - Error 'User <email> already registered' will be shown

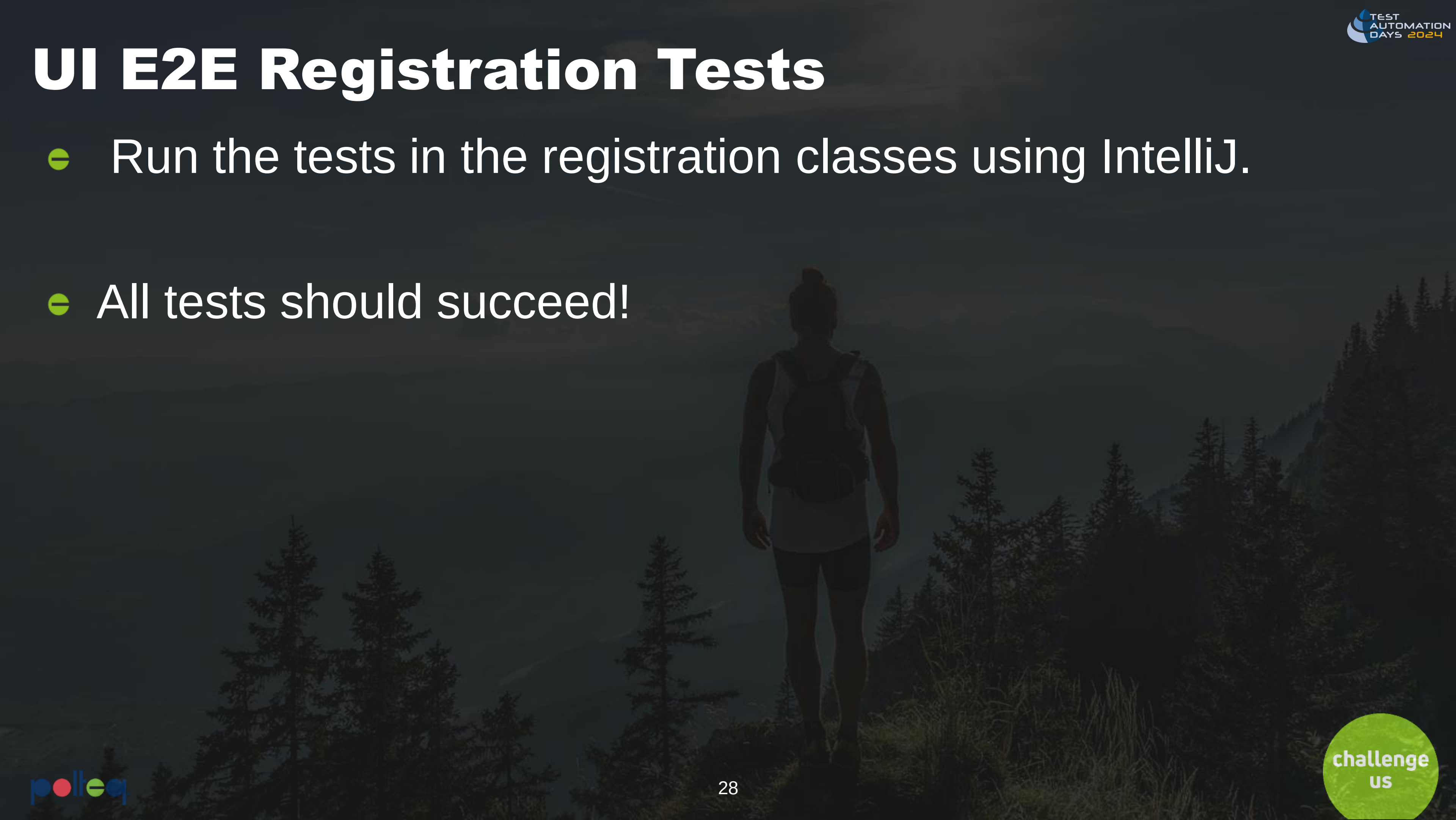


Valid email requirements

- Basic syntax [emailname]@[domain].[tld]
- Must contain 1 @ and then a point . after the domain
- Text before @ only contains a-z, A-Z, 0-9, dash -, point .
- Domain only contains a-z, A-Z, 0-9, dash -, point .
- TLD only contains a-z, A-Z
  - Minimum of 2 characters
  - Maximum of 6 characters

UI E2E Registration Tests

- Run the tests in the registration classes using IntelliJ.
- All tests should succeed!

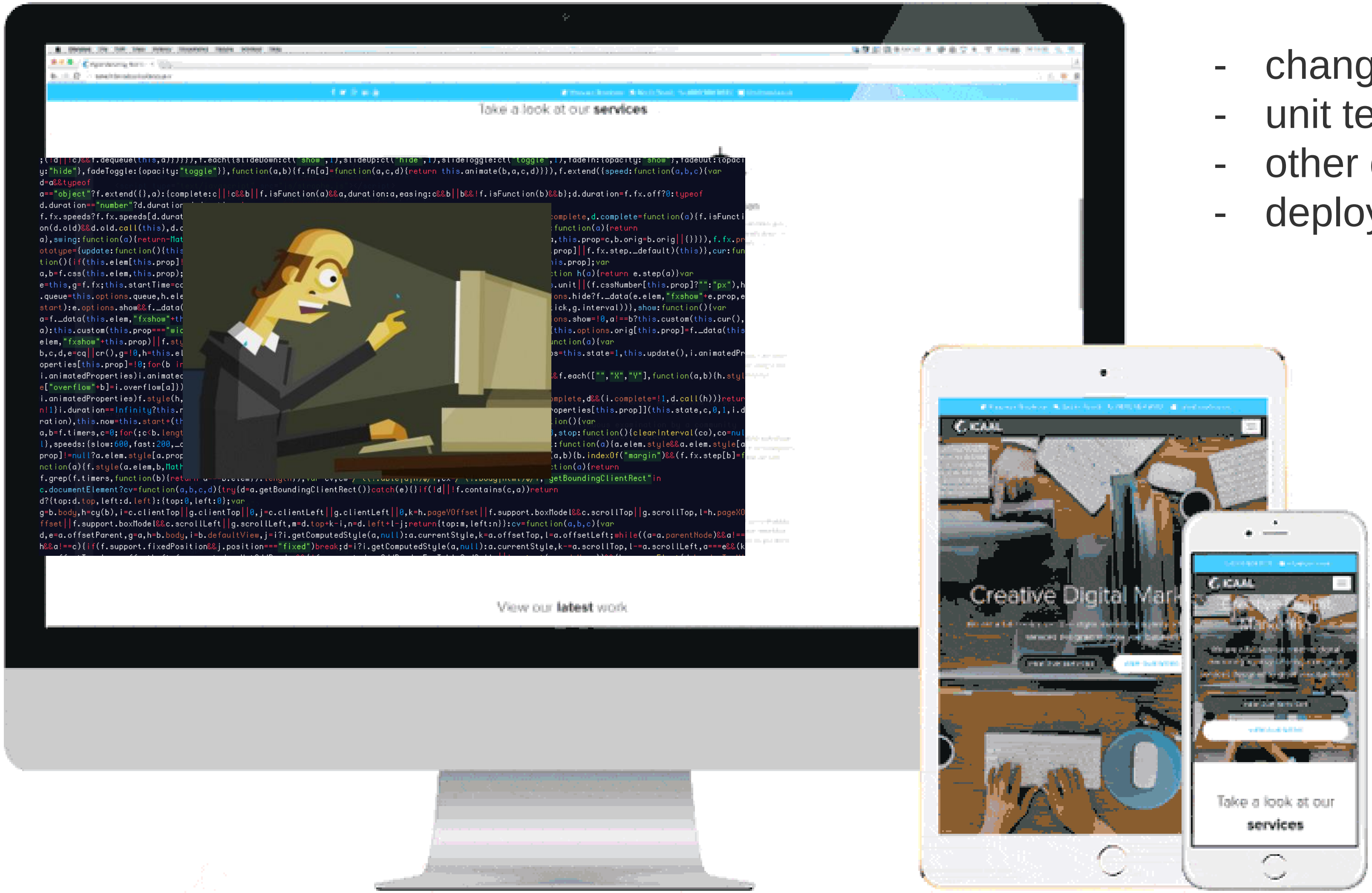


Bug ticket - New

Headline	Error text not correct for missing email
Reporter	An E2E tester
Description	Text is showing 'Email is mandatory' This should be 'Email is required'
Bug assigned to	A developer who can fix the bug!

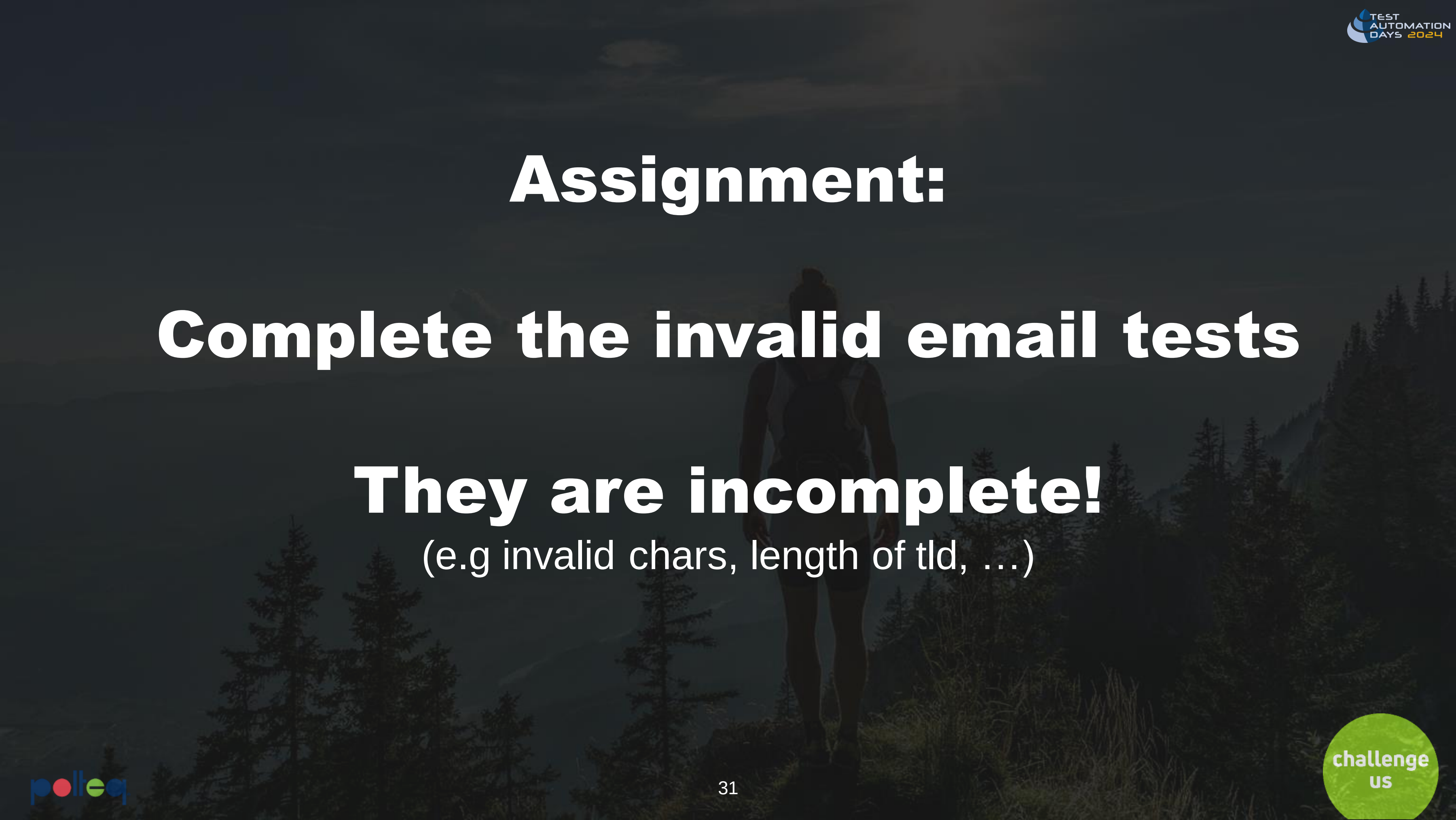
challenge us

TEST AUTOMATION DAYS 2024



- change code
- unit test
- other dev tests
- deploy!

challenge us



Assignment:

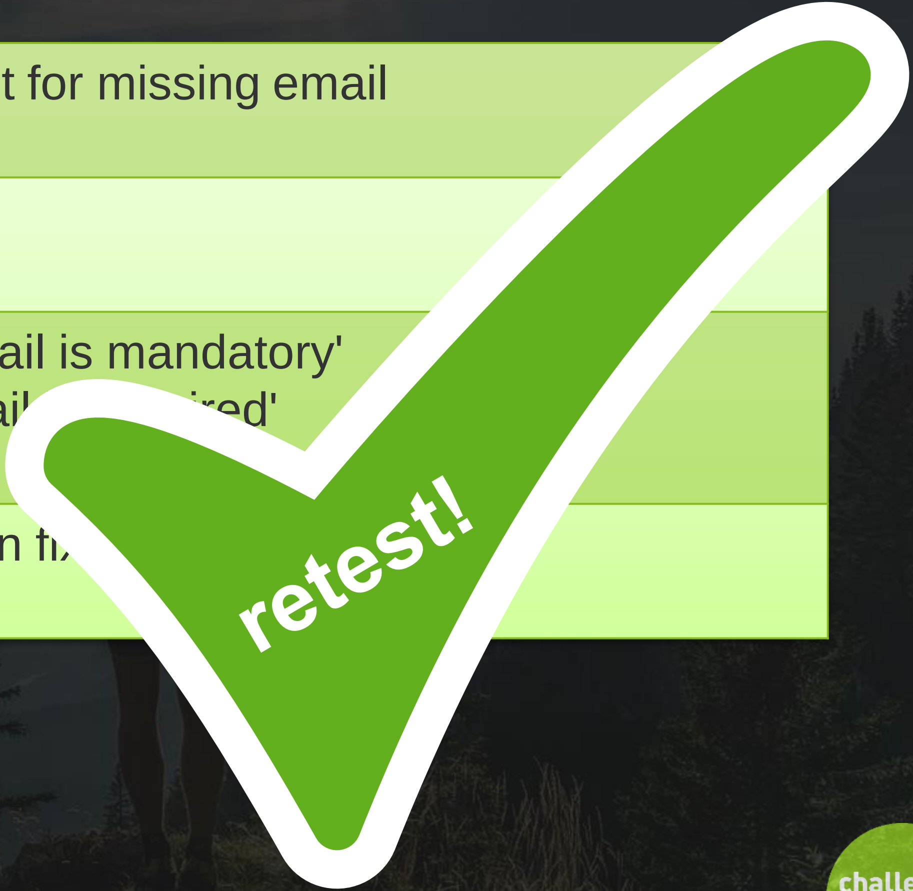
Complete the invalid email tests

They are incomplete!

(e.g invalid chars, length of tld, …)

Bug ticket - New

Headline	Error text not correct for missing email
Reporter	An E2E tester
Description	Text is showing 'Email is mandatory' This should be 'Email [illegible]red'
Bug assigned to	A developer who can fi



challenge us

Can we test more efficiently?

Can we test more efficiently?

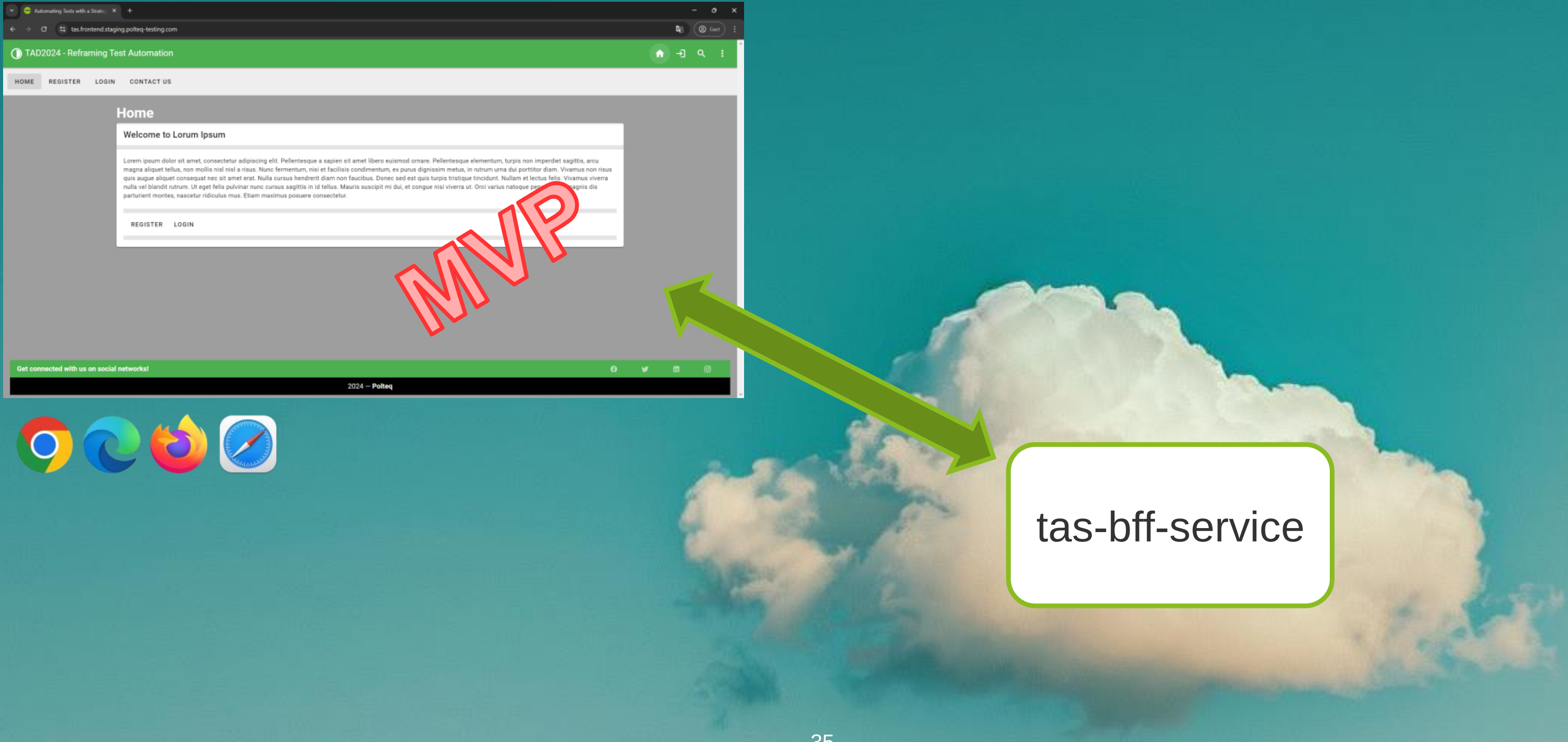
- Testing through the UI takes too long
- Maybe we can speed up the testing

- Let's have a closer look at how our system is built
  - Inspect the system landscape

- The frontend of the SUT communicates with a service in the cloud ➔ calles **`tas-bff-service`** (backend for frontend)

challenge us

The SUT in more detail



Automating Tests with a Strate

tas.frontend.staging.polteq-testing.com

TAD2024 - Reframing Test Automation

HOME REGISTER LOGIN CONTACT US

Home

Welcome to Lorum Ipsum

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque a sapien sit amet libero euismod ornare. Pellentesque elementum, turpis non imperdiet sagittis, arcu magna aliquet tellus, non mollis nisl nisl a risus. Nunc fermentum, nisi et facilisis condimentum, ex purus dignissim metus, in rutrum urna dui porttitor diam. Vivamus non risus quis augue aliquet consequat nec sit amet erat. Nulla cursus hendrerit diam non faucibus. Donec sed est quis turpis tristique tincidunt. Nullam et lectus felis. Vivamus viverra nulla vel blandit rutrum. Ut eget felis pulvinar nunc cursus sagittis in id tellus. Mauris suscipit mi dui, et congue nisi viverra ut. Orci varius natoque pe [illegible] agnis dis parturient montes, nascetur ridiculus mus. Etiam maximus posuere consectetur.

REGISTER LOGIN

Get connected with us on social networks!

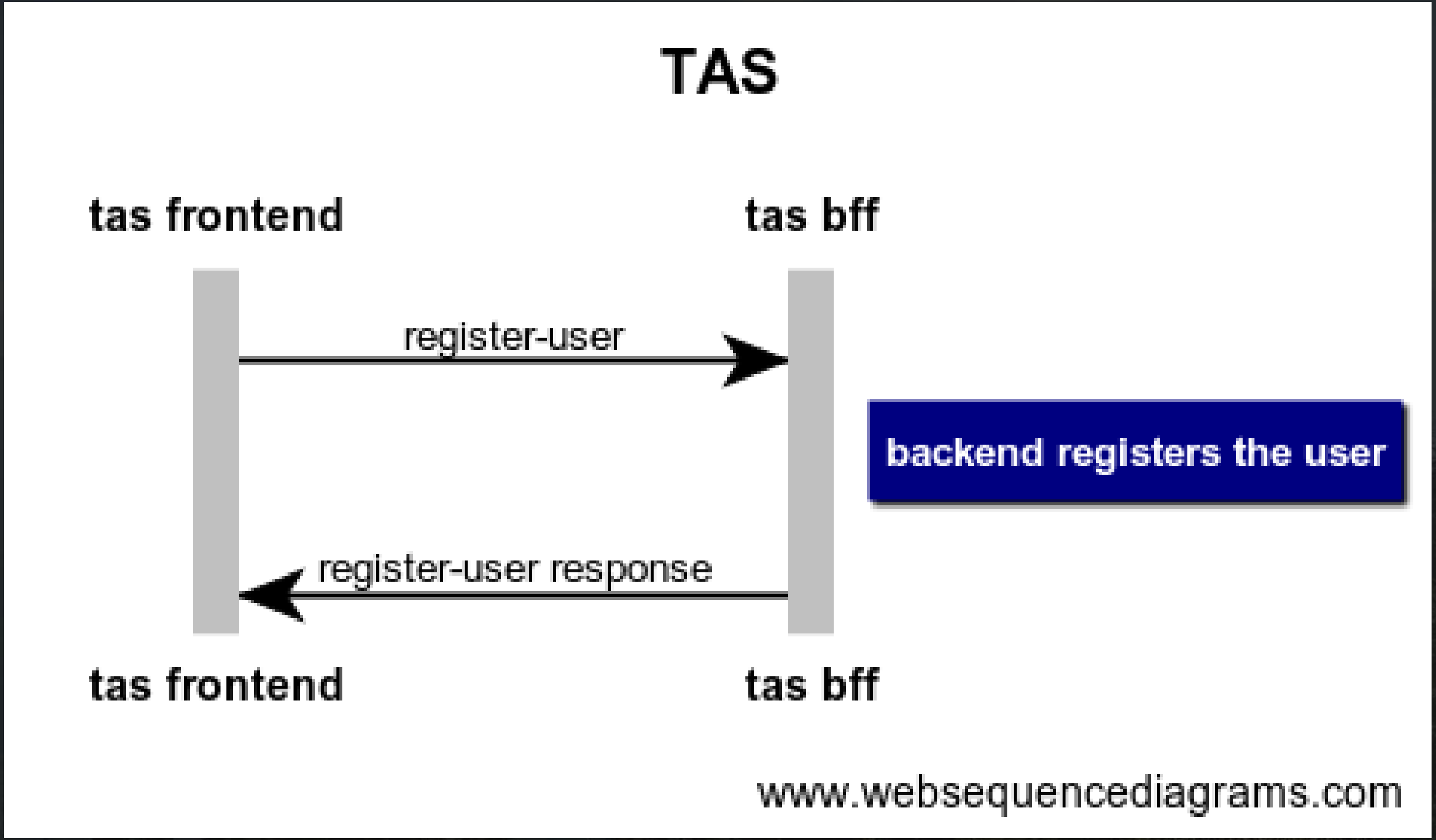
2024 — Polteq

Can we test more efficiently?

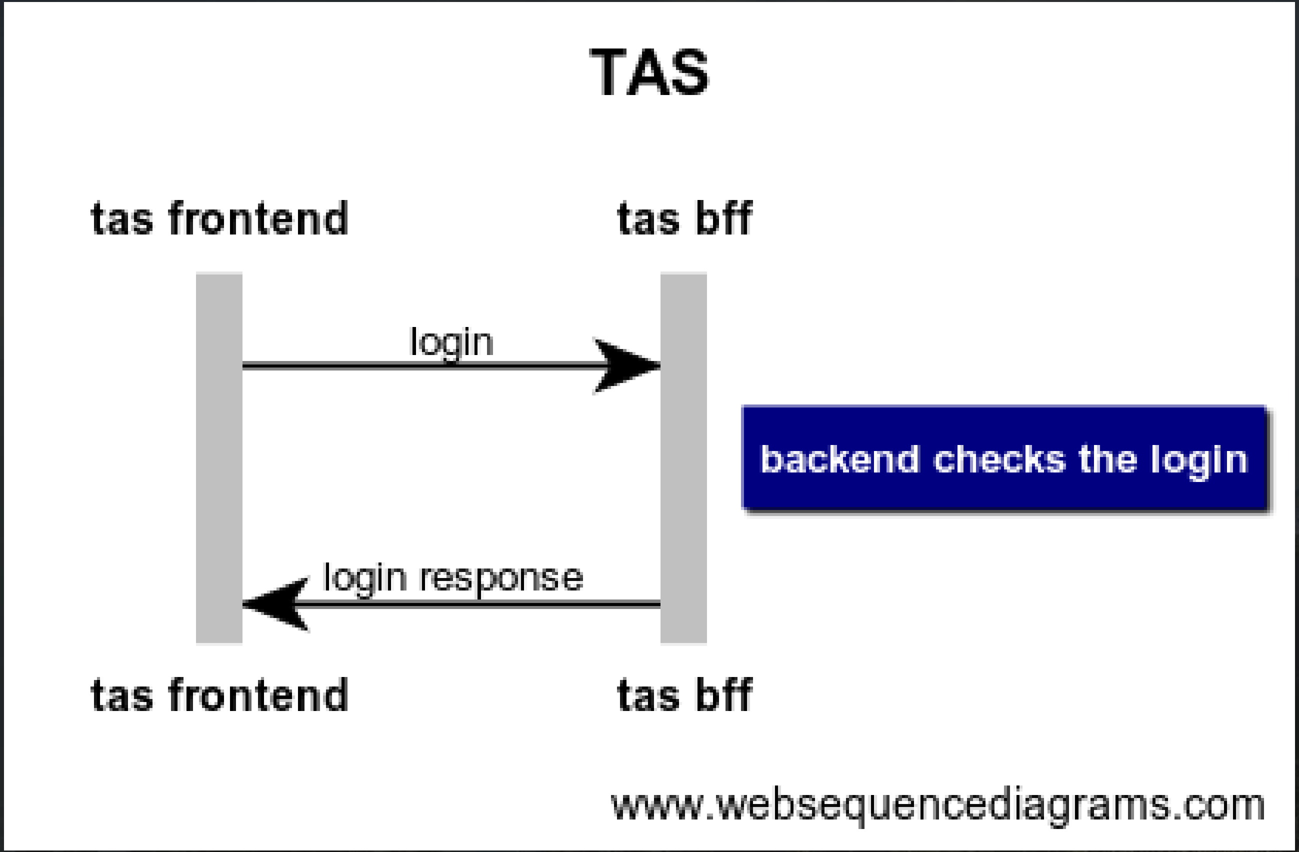
- **`tas-bff-service`**
  - exposes 2 ***api*** endpoints ➔ used by the **`tas-frontend`**
- The base url for the **`tas-bff-service`** is
  - https://tas.bff.staging.polteq-testing.com
- **`/register-user`** endpoint ➔ register a new user
- **`/login`** endpoint ➔ login an existing user

challenge us

/register-user flow



/login flow

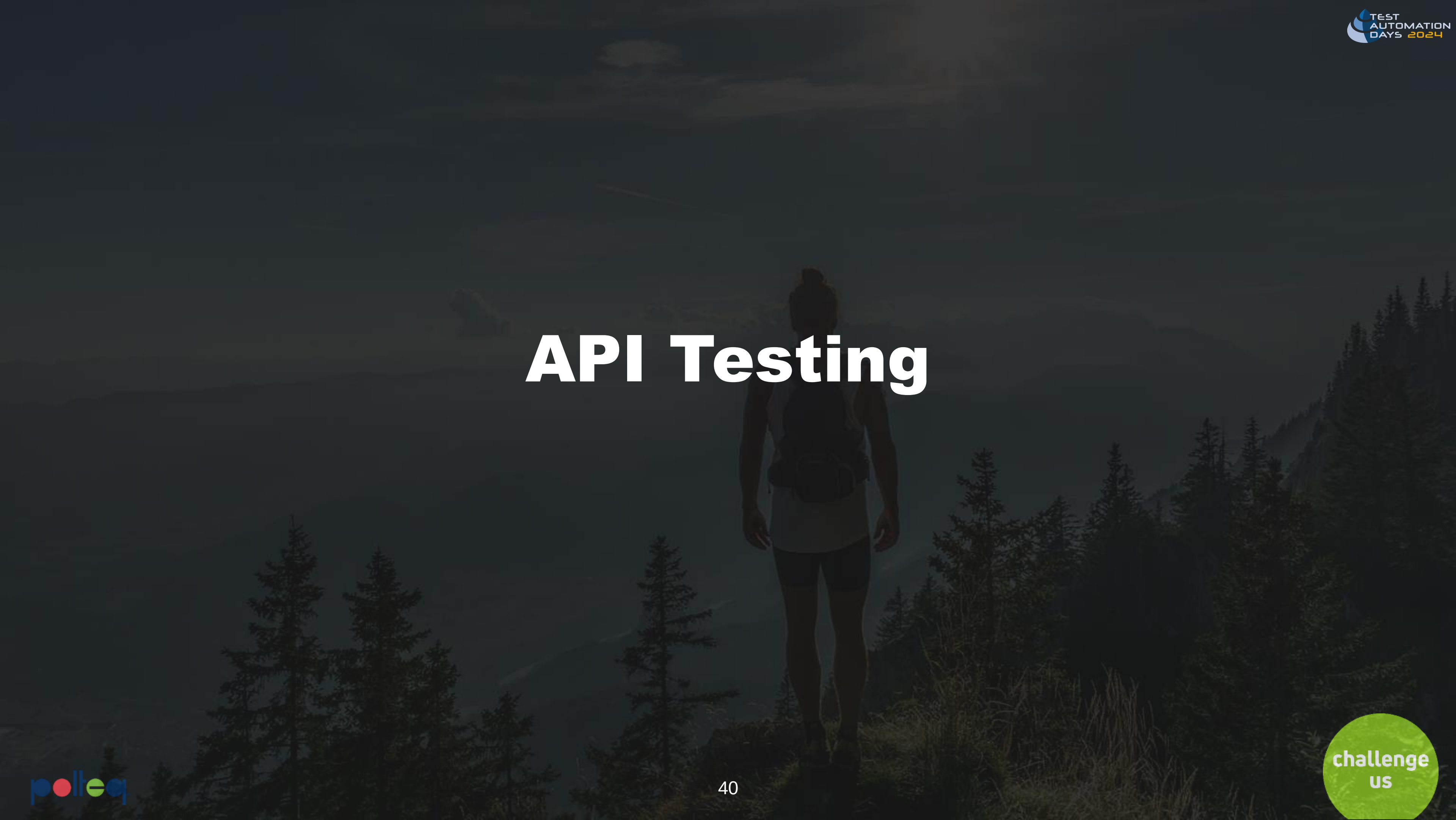

TAS
tas frontend
tas bff
login
backend checks the login
login response
tas frontend
tas bff
www.websequencediagrams.com

Description endpoints

```
- path: /register-user
  method: POST
  request:
    body:
      application/json:
        schema: |
          {
            "name": "Max Verstappen",
            "email": "m.verstappen@redbullracing.com",
            "password": "password123",
            "phoneNumber": "555-12345"
          }
  responses:
    - status: 200
      body:
        application/json:
          schema: |
          {
            "name": "Max Verstappen",
            "email": "m.verstappen@redbullracing.com",
            "password": "password123",
            "phoneNumber": "555-12345"
          }
    - status: 400
      body:
        application/json:
          schema: |
          {
            "statusCode": 400,
            "message": "Name is required, Email is required"
          }
```

```
- path: /login
  method: POST
  request:
    body:
      application/json:
        schema: |
          {
            "email": "m.verstappen@redbullracing.com",
            "password": "password123",
          }
  responses:
    - status: 200
      body:
        application/json:
          schema: |
          {
            "name": "Max Verstappen",
            "email": "m.verstappen@redbullracing.com",
            "phoneNumber": "555-12345"
          }
    - status: 403
      body:
        application/json:
          schema: |
          {
            "statusCode": 403,
            "message": "Could not login with these credentials"
          }
```

challenge us



API Testing

API Tests

- Package **`002_apitests`** contains tests which directly use the defined endpoints

- For now, we only focus on **`/register-user`**

- The tests are REST-assured tests
  - REST-assured is a library for API testing

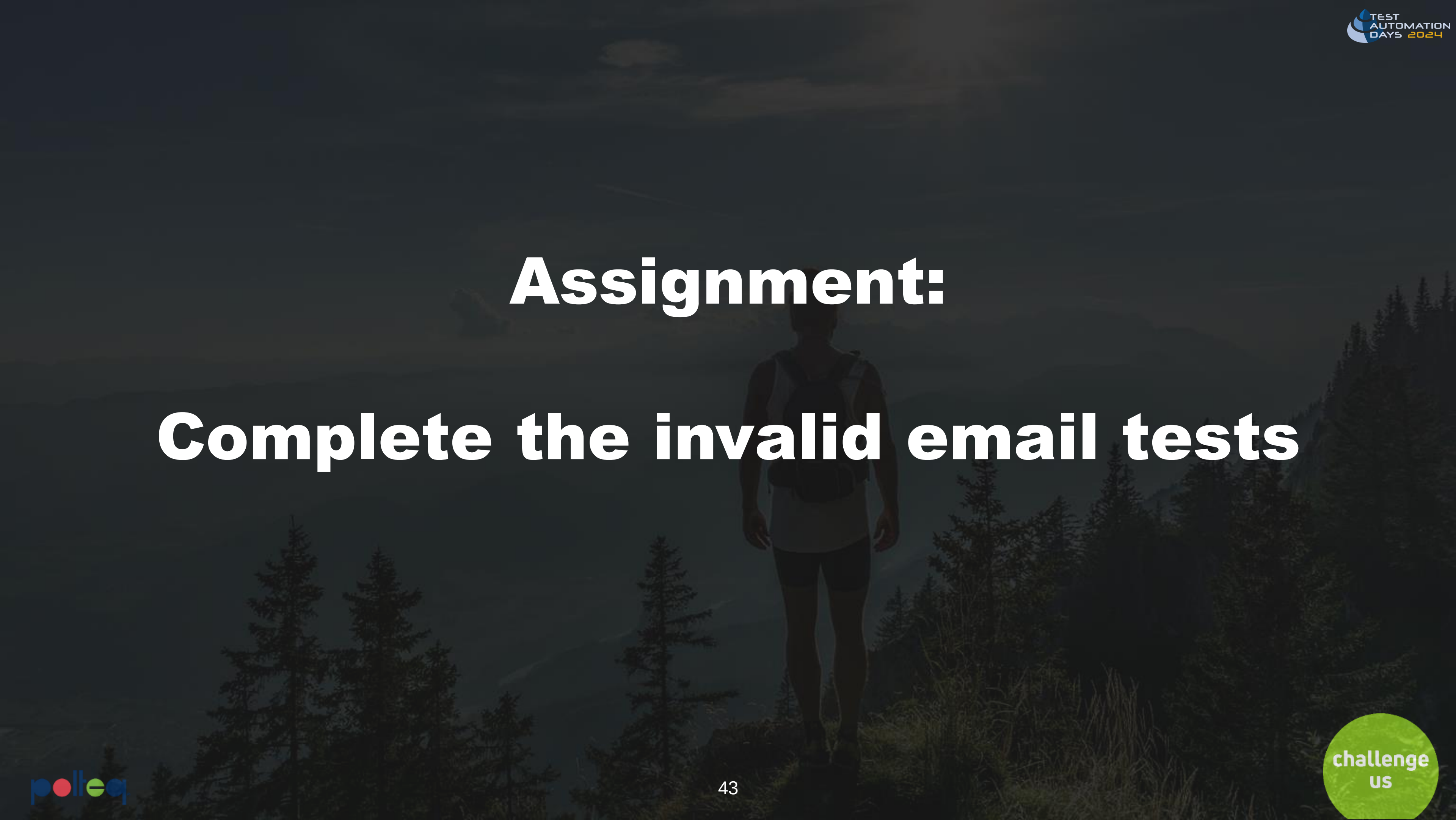
API Tests

- Examine the **`001_registertests`** package

- Run the tests ➔ they should all pass
- The `itShouldBePossibleToRegisterWithValidEmail` test validates the valid email formats.
- The `anErrorShouldBeReturnedWhenAnInvalidEmailIsPassed` test validates the invalid email formats.

- Check if they are complete

challenge us



Assignment:

Complete the invalid email tests

What tests are now obsolete?

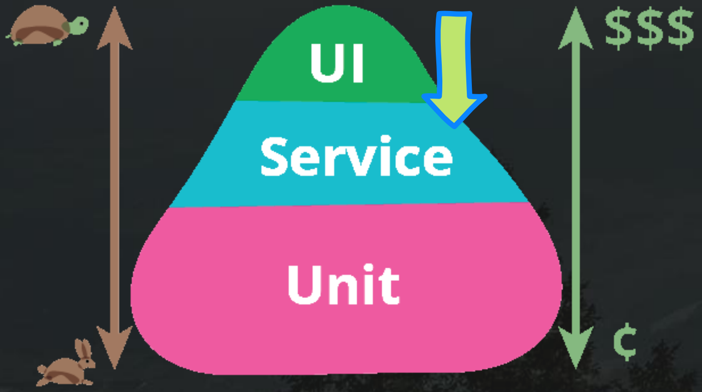
Why?

Delete them!

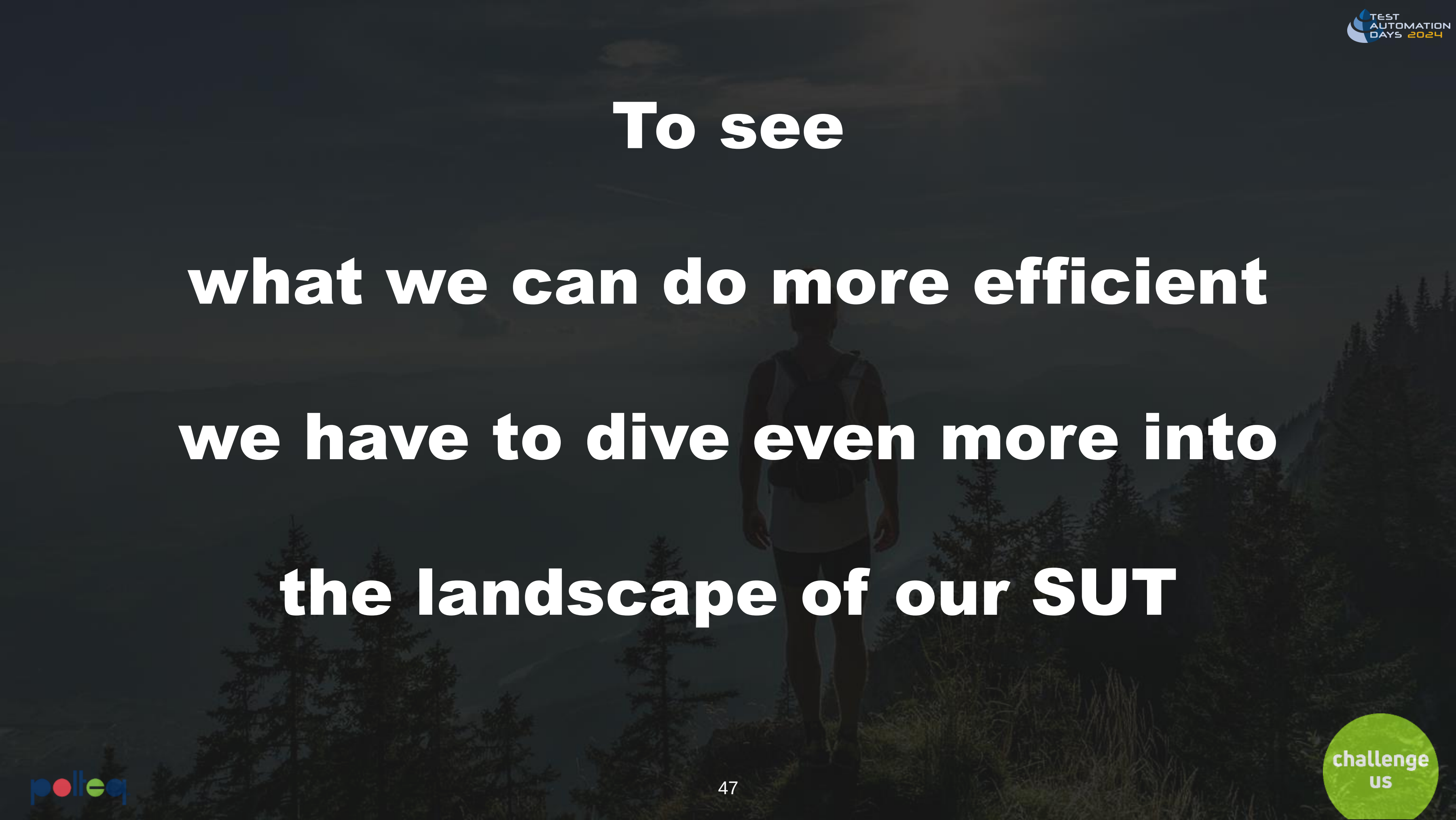
So, we moved our tests

from the UI

to the API ...



Is this *E2E* approach the best solution for testing the app?



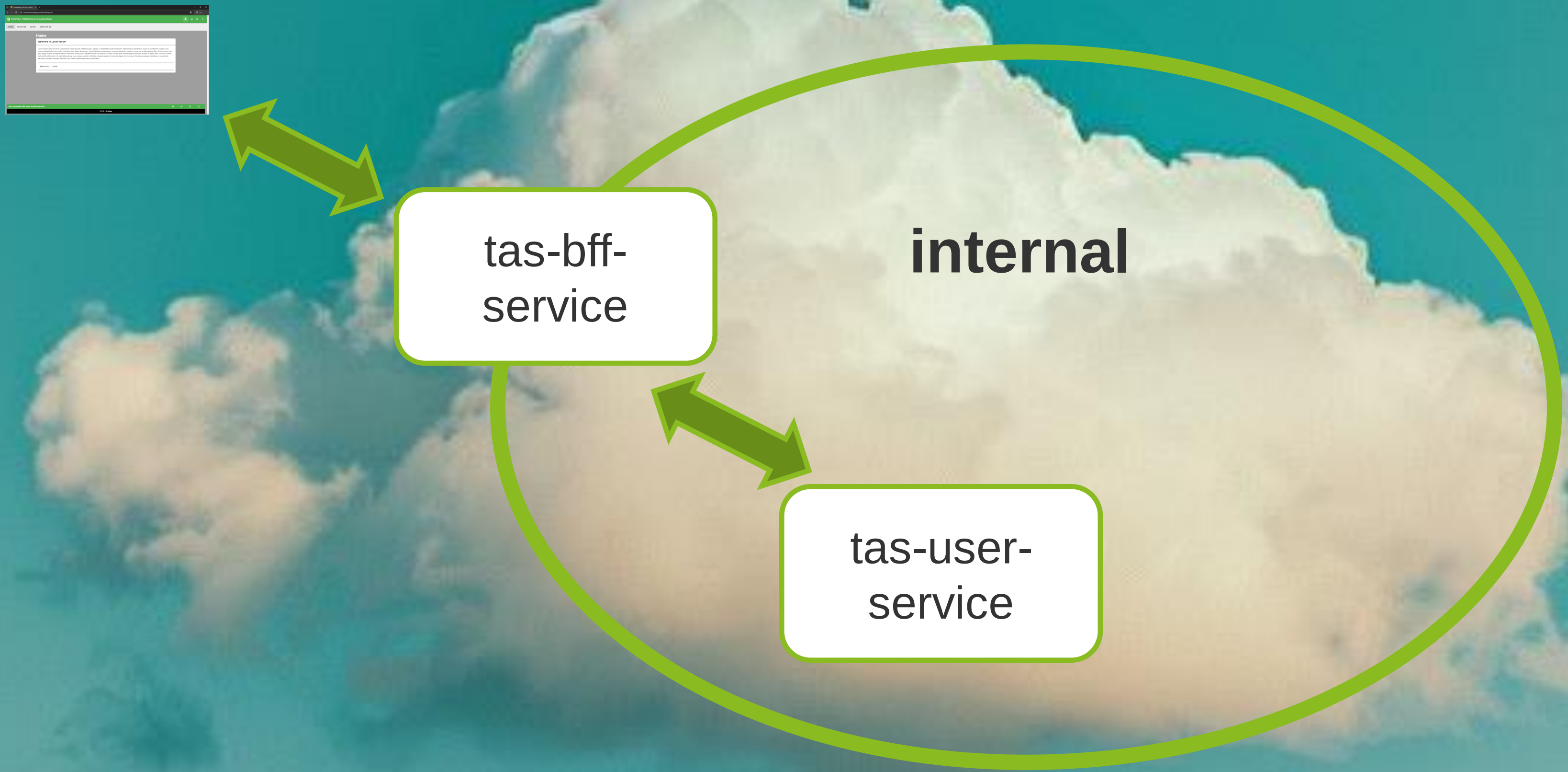
To see

what we can do more efficient

we have to dive even more into

the landscape of our SUT

The SUT in even more detail



SUT in even more detail

- `tas-bff-service`
  - uses `tas-user-service`
    - to register a new user
  - is a proxy / gateway for request to internal services
    - it has no logic!
  - is reachable from the outside world
    - `tas-user-service` is not
      - so we cannot directly test this service with our e2e test repo

challenge us

SUT in even more detail

- `tas-user-service`
  - Actually registers new users
  - Performs all the validations when registering a new user

- How nice would it be if we could test this service!

How can we test the tas-user-service?

challenge us

Component Testing

Component Testing

- Component Testing is testing a part of the system in isolation

- In our case we will use the **`tas-user-service`** as a component

- To be able to test the **`tas-user-service`** we need to know more about this service

testing

the `tas-user-service`

as a component

tas-user-service

- Get to know more about the **`tas-user-service`**
  - read the specifications
  - talk to the developer

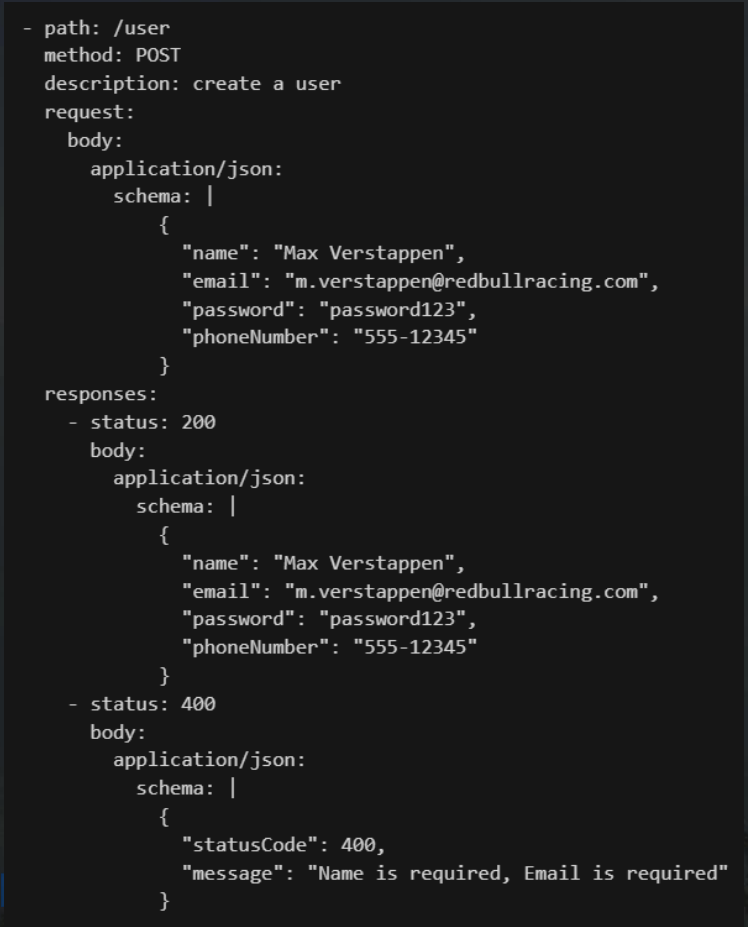
    ➔ for component testing input from devs is crucial

- **`tas-user-service`** has endpoint **`/user`** used by the **`tas-bff-service`**

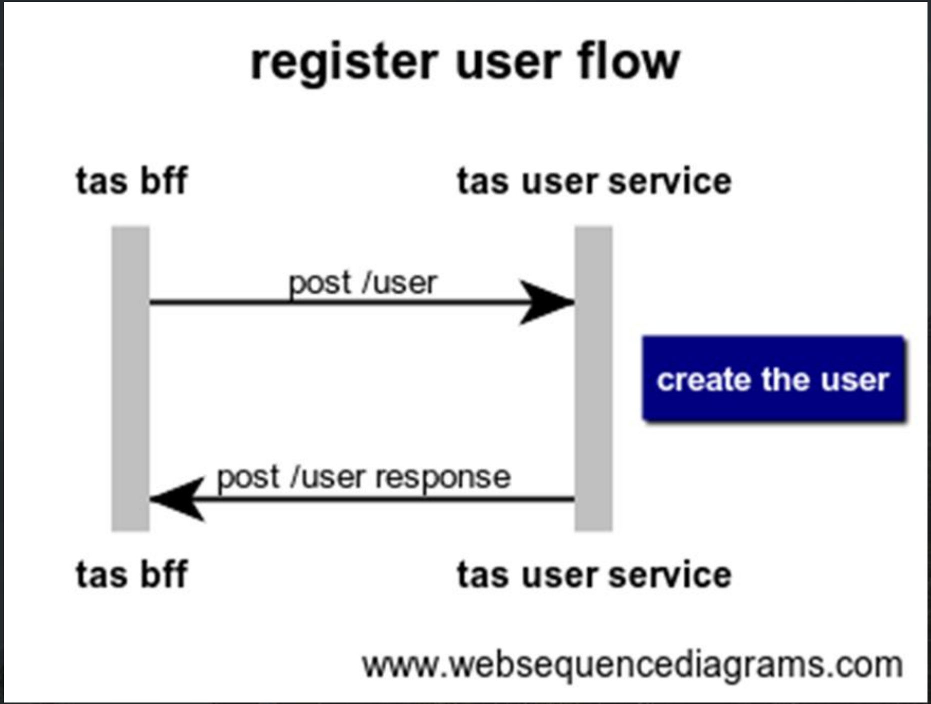
- This is also 'just' an api

challenge us

tas-user-service endpoint: /user



```
- path: /user
  method: POST
  description: create a user
  request:
    body:
      application/json:
        schema: |
            {
              "name": "Max Verstappen",
              "email": "m.verstappen@redbullracing.com",
              "password": "password123",
              "phoneNumber": "555-12345"
            }
  responses:
    - status: 200
      body:
        application/json:
          schema: |
            {
              "name": "Max Verstappen",
              "email": "m.verstappen@redbullracing.com",
              "password": "password123",
              "phoneNumber": "555-12345"
            }
    - status: 400
      body:
        application/json:
          schema: |
            {
              "statusCode": 400,
              "message": "Name is required, Email is required"
            }
```

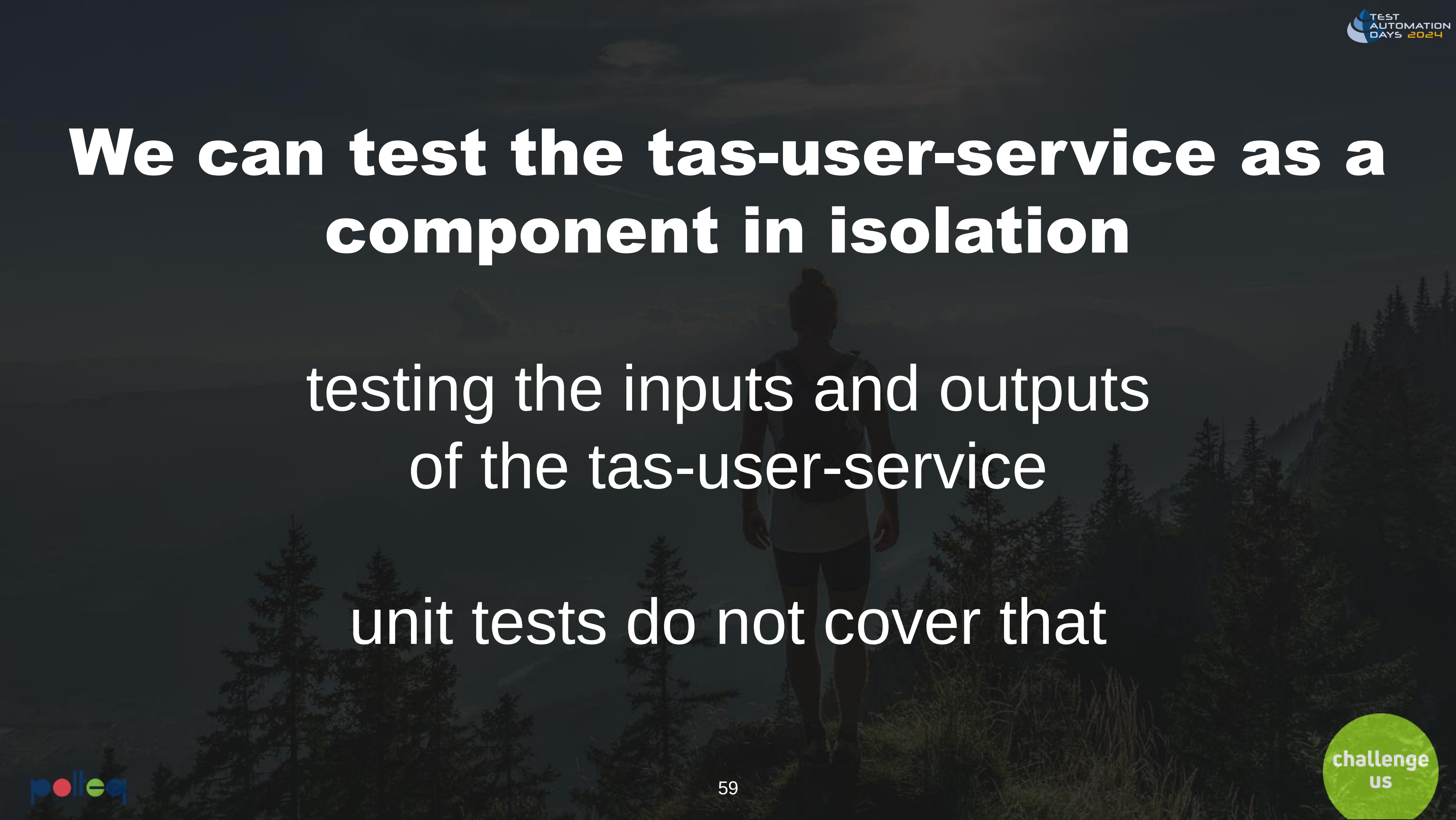


tas-user-service

- Create user → post /user

- Unit tests in place for:
  - name is required
  - email is required
  - password is required
  - email must be valid according to the email validation rules
  - an email cannot be registered more than once

The requirements are being unit tested. What else can we test here?

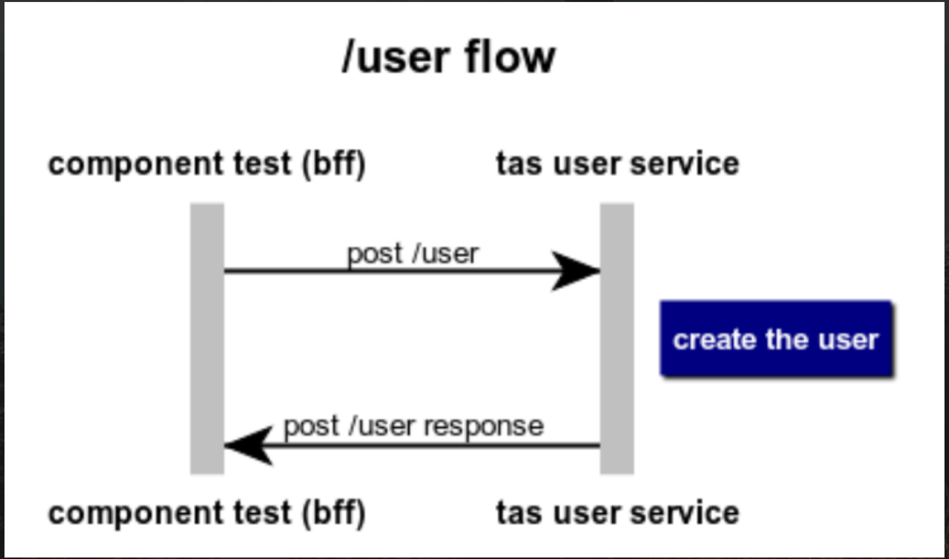


We can test the tas-user-service as a component in isolation

testing the inputs and outputs
of the tas-user-service

unit tests do not cover that

`tas-user-service` component will be tested just like the `tas-bff-service` uses it



Component Test of `tas-user-service`

- Technique used for component test:
  - Containerization with Docker
  - API testing with Java / REST-assured

- The **`tas-user-service`** repo can be found here:
  - https://github.com/erik-haartmans/tas-user-service

- Clone or download this repo to your own laptop
  - This repo contains a java spring boot api application

challenge us

tas-user-service repo

- Open the repo in IntelliJ and set the maven reload settings

- Open a terminal in IntelliJ and enter the command:
  - `./mvnw clean verify`

- You should see a build success and this verifies that the repo settings are correct.

tas-user-service using docker

- Next ➔ run the `tas-user-service` locally using docker
- Repo has files for docker
  - to create docker ***image*** from the source code
- Docker image is used to create a ***container*** which will be the actual ***tas-user-service*** running on your machine!

tas-user-service using docker

```
FROM maven:3-eclipse-temurin-21 as build
WORKDIR /build
COPY . .
RUN mvn clean package

FROM eclipse-temurin:21-jre-alpine as run

EXPOSE 8080

WORKDIR /app
COPY --from=build /build/target/tas-user-service-0.0.1.jar /app

ENTRYPOINT ["java","-jar","/app/tas-user-service-0.0.1.jar"]
```

Dockerfile = how to create image

```
services:

  tas-user-service:
    build:
        context: .
        dockerfile: Dockerfile
    container_name: tas-user-service
    ports:
      - 8081:8080
```

docker-compose.yml = how to start

challenge us

tas-user-service using docker

- In the terminal you can enter the following command to (re-)create image and start the container:
  - `docker compose up --build`

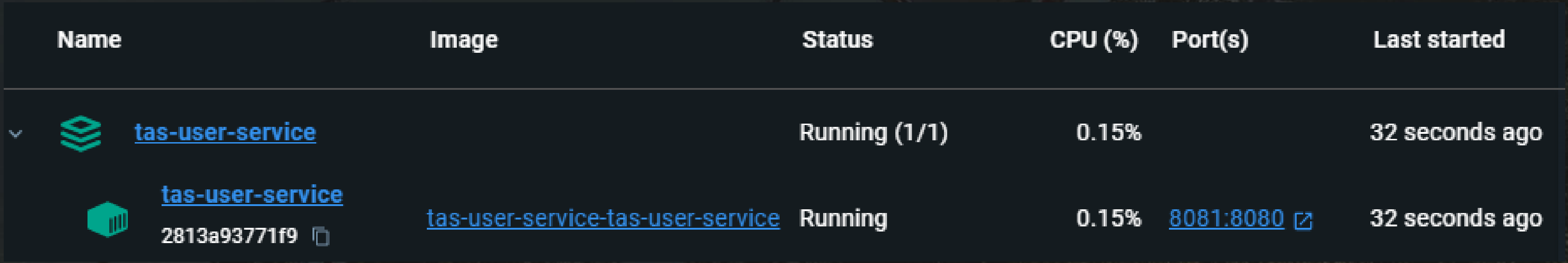
- This will trigger docker image downloads which are needed to create the **`tas-user-service`** image

- After running this command (will take a while) you should see the `Started TasUserServiceApplication` message in the log. The container now has started in the non detached mode. This results in logging in our terminal.

challenge us

Result:

`tas-user-service`
is now running on your own machine in isolation!



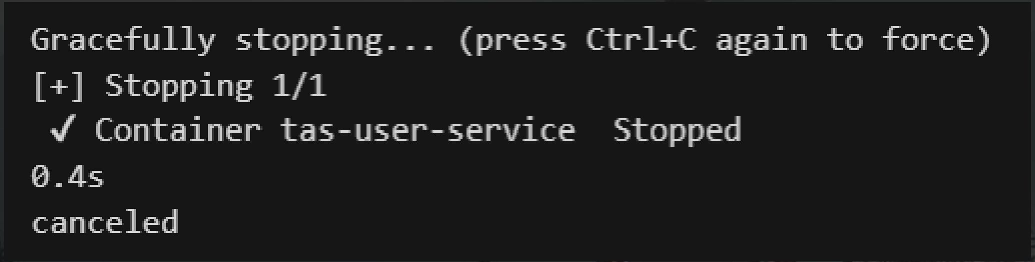
Name	Image	Status	CPU (%)	Port(s)	Last started
tas-user-service		Running (1/1)	0.15%		32 seconds ago
tas-user-service 2813a93771f9	tas-user-service-tas-user-service	Running	0.15%	8081:8080	32 seconds ago

Why is this useful?

Stopping the containers

- How to stop the running container
  ➔press CTRL-C in terminal where you started it

- You should see something like this:



```
Gracefully stopping... (press Ctrl+C again to force)
[+] Stopping 1/1
 ✓ Container tas-user-service  Stopped
0.4s
canceled
```

- Typing the following command in the same terminal will also remove the container:
  - `docker compose down`

Stopping the containers - recap

- The steps for building an image from the sources and starting and stopping the container, are:
  - `docker compose up --build`
  - CTRL-C
  - `docker compose down`

Start / Stopping containers alternative

- You could also use the following command to start the container:
  - `docker compose up -d --build`

- Result is that it returns to the command line.
- No logging will be visible.
- **`docker compose down`** will stop and delete the container

challenge us

tas-user-service component test

- `tas-user-service` is running locally on your machine
  - Now you can develop and execute tests for the service as a component

- Advantage of component tests
  - It tests the inputs, outputs and internals of the service
- It is used just like another service (like the bff) would use it.

➔ "Gray box testing"

tas-user-service component test

- Another big advantage of component tests
  - They can run in the ***ci/cd pipeline***

- They could run right after the unit tests which means that we have fast feedback
  - after each push / PR

- Tests can fail even before any deployment has been done!

challenge us

tas-user-service component test

- Check the package **`componenttest`** in the test folder
- **`001_registerusertests`** package contains component tests
  - In this case all component test classes end with `CTCase`
- Run the component tests (valid and invalid)
  - Check logging of running container in terminal

challenge us

Do we need more tests?

What tests can be deleted?

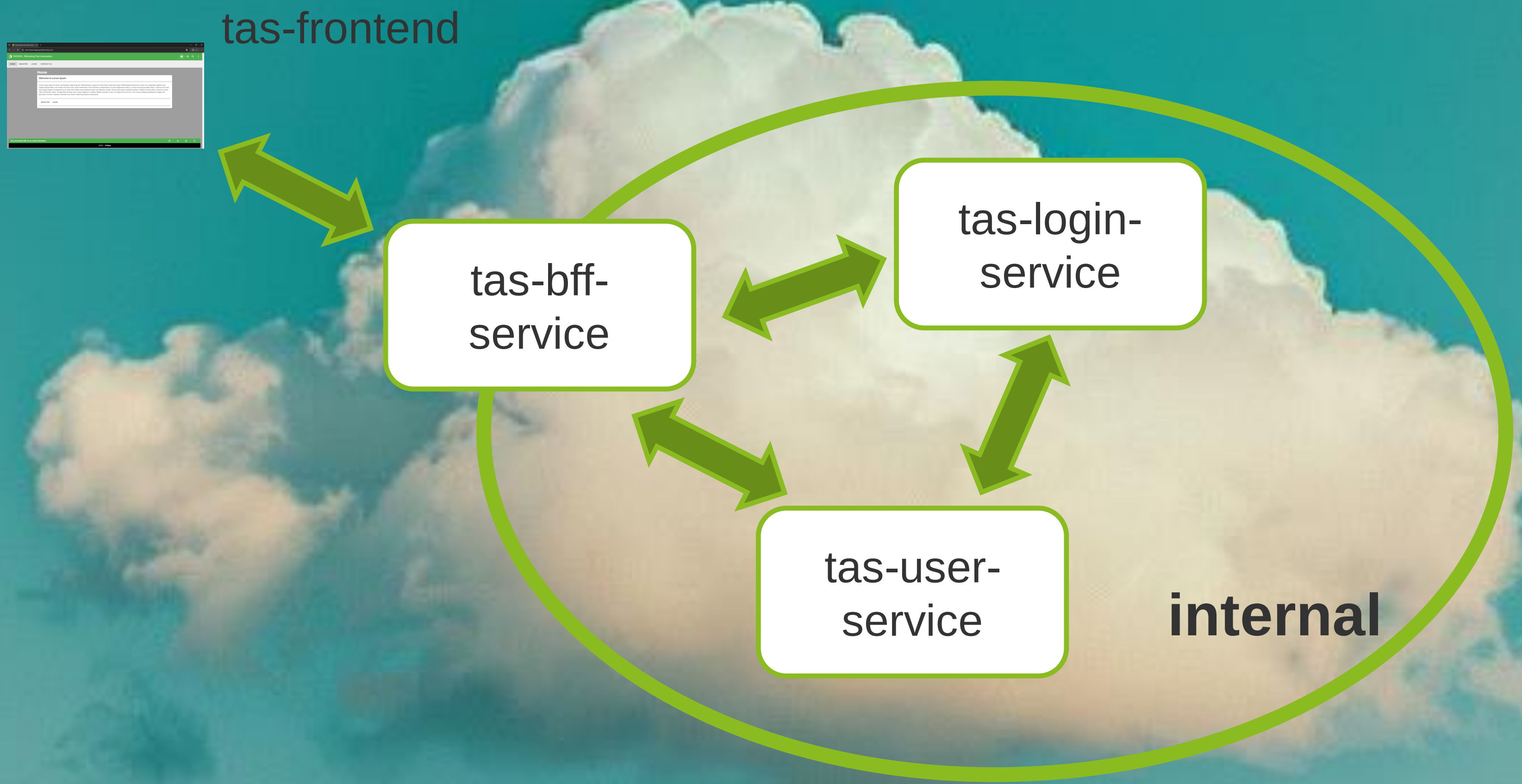
challenge us

testing

the frontend

as a component

The SUT in complete detail



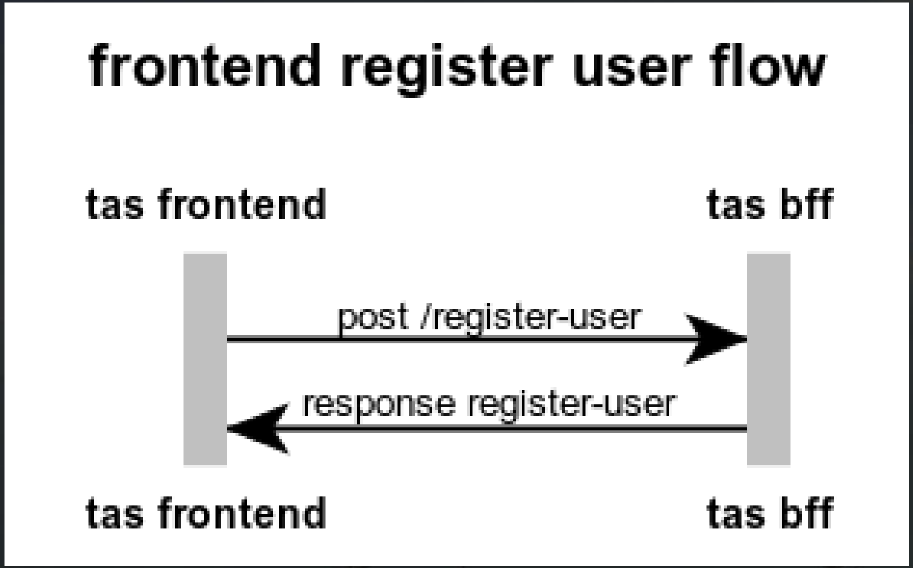
tas-frontend

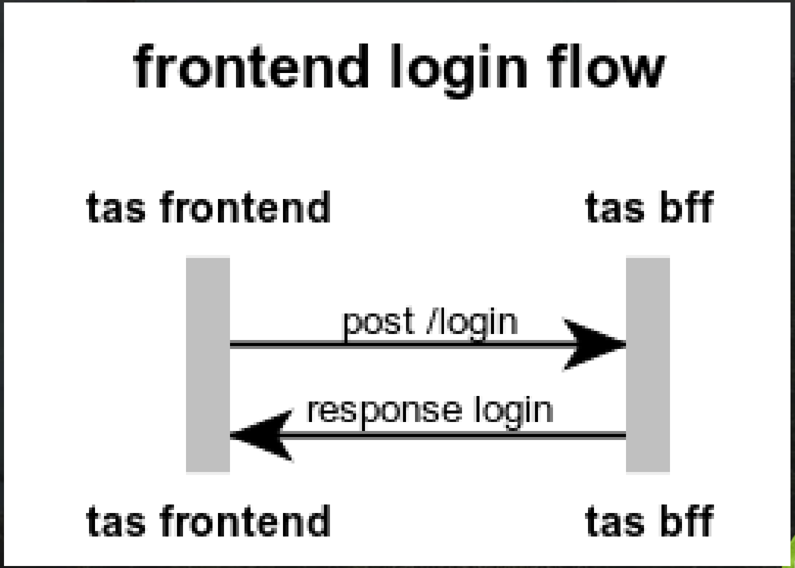
- The frontend has several functionalities
  - navigating internally
  - api calls to the **`tas-bff-service`**

- **`tas-frontend`** calls the **`tas-bff-service`** for:
  - post `/register-user`
  - post `/login`

challenge us

tas-frontend flows





tas-bff-service: /register-user & /login

```
- path: /register-user
  method: POST
  request:
    body:
      application/json:
        schema: |
          {
            "name": "Max Verstappen",
            "email": "ver@redbullracing.com",
            "password": "password123",
            "phoneNumber": "555-12345"
          }
  responses:
    - status: 200
      body:
        application/json:
          schema: |
          {
            "name": "Max Verstappen",
            "email": "ver@redbullracing.com",
            "password": "password123",
            "phoneNumber": "555-12345"
          }
    - status: 400
      body:
        application/json:
          schema: |
          {
            "statusCode": 400,
            "message": "Name is required, Email is required"
          }
```

```
- path: /login
  method: POST
  request:
    body:
      application/json:
        schema: |
          {
            "email": "ver@redbullracing.com",
            "password": "password123"
          }
  responses:
    - status: 200
      body:
        application/json:
          schema: |
          {
            "name": "Max Verstappen",
            "email": "ver@redbullracing.com",
            "phoneNumber": "555-12345"
          }
    - status: 403
      body:
        application/json:
          schema: |
          {
            "statusCode": 403,
            "message": "This can be any message"
          }
```

challenge us

Component Testing: tas-frontend

- How to create component tests for `tas-frontend`
  - add tests to this repo!

- This repo is a TypeScript repo!

Should we stick to java testrepos or also learn TypeScript in this case?

tas-frontend

- github: https://github.com/erik-haartmans/tas-frontend

- Clone or download this repo to your own laptop
  - This repo contains the frontend webapp (TS)

- Open the repo in VSCode

- Open a terminal and enter the command:
  - `npm install`
  - `npx playwright install`

tas-frontend

- Starting the **`tas-frontend`** as a component
  - `docker compose up --build`
  - **`Dockerfile`** and **`docker-compose.yml`** contain instructions to create the frontend image and start it as a container

- After starting the frontend it's already available via:
  - `http://localhost:8080`

- Click through the app and see what happens

challenge us

tas-frontend

- Contains Playwright component tests
  - uses the same type of UI testing library

- Open a new terminal and type:
  - `npx playwright test --ui`
- This starts the Playwright Test Runner
  - It shows the available tests in the project
  - The are located in the `` `playwright-e2e-ct-tests` `` folder

- Run the tests!

tas-frontend

- Examine the tests present

- We need to mock/stub the calls to the **`tas-bff-service`** apis
  → **`tas-bff-service`** is not available

- Playwright has its own mocking/stubbing functionality
  - See next pages

challenge us

tas-frontend

- Playwright mocking

```
const body = `
{
  "name": "name",
  "email": "a@a.com",
  "password": "1234",
  "phoneNumber": "555-4711",
}
`

// define the mocked response for the register endpoint
await page.route('*/**/register-user', async route => {
  await route.fulfill({
    status: 200,
    body
  });
});
```

tas-frontend

- Playwright mocking

```
// open the register page
await menu.openRegisterPage();

// enter register data
await registerPage.registerNewAccount(
  'name',
  'a@a.com',
  '1234',
  '1234',
  '555-4711',
  true
);

// mock returns a success message which the frontend can handle
// expect success page
await expect(registerSuccessPage.getPageContainer).toBeVisible();
```

challenge us

Exercise

Implement the invalid registration test

(check the valid tests for how to mock)

After being able to test the frontend in isolation …

Which tests can be deleted?

After expanding this also for the login functionality …

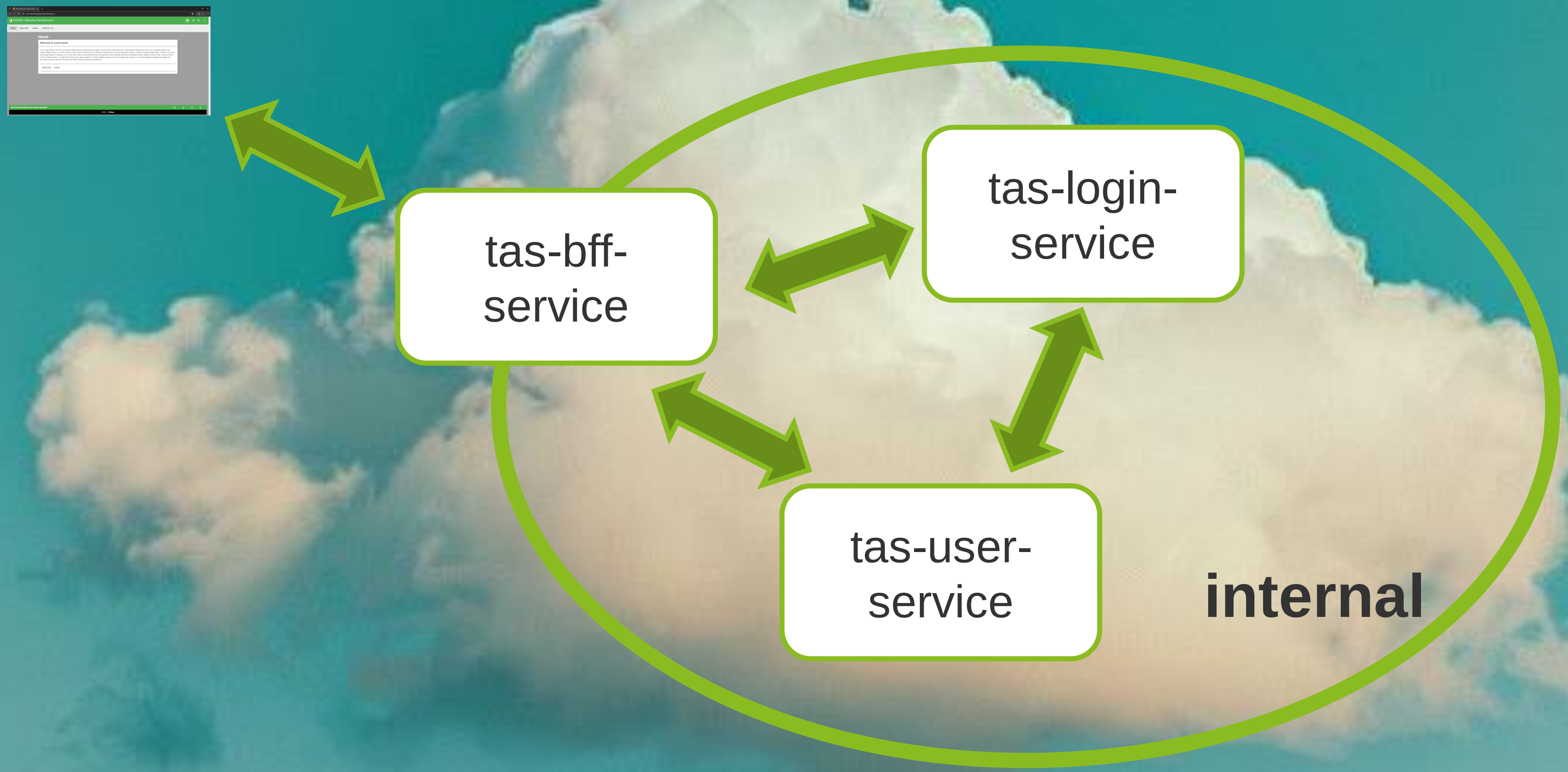
Which tests can be deleted?

Component Test

tas-login-service

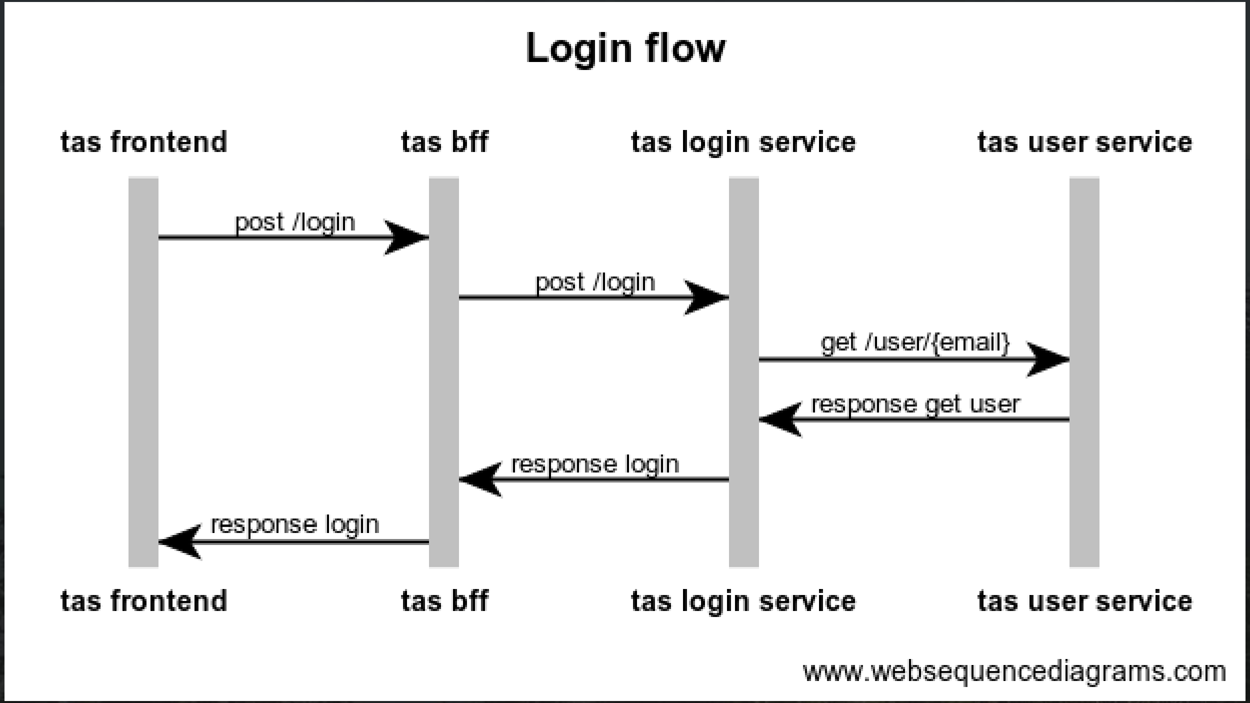
using WireMock

The SUT in complete detail



Login flow

- We will focus on the tas-login-service



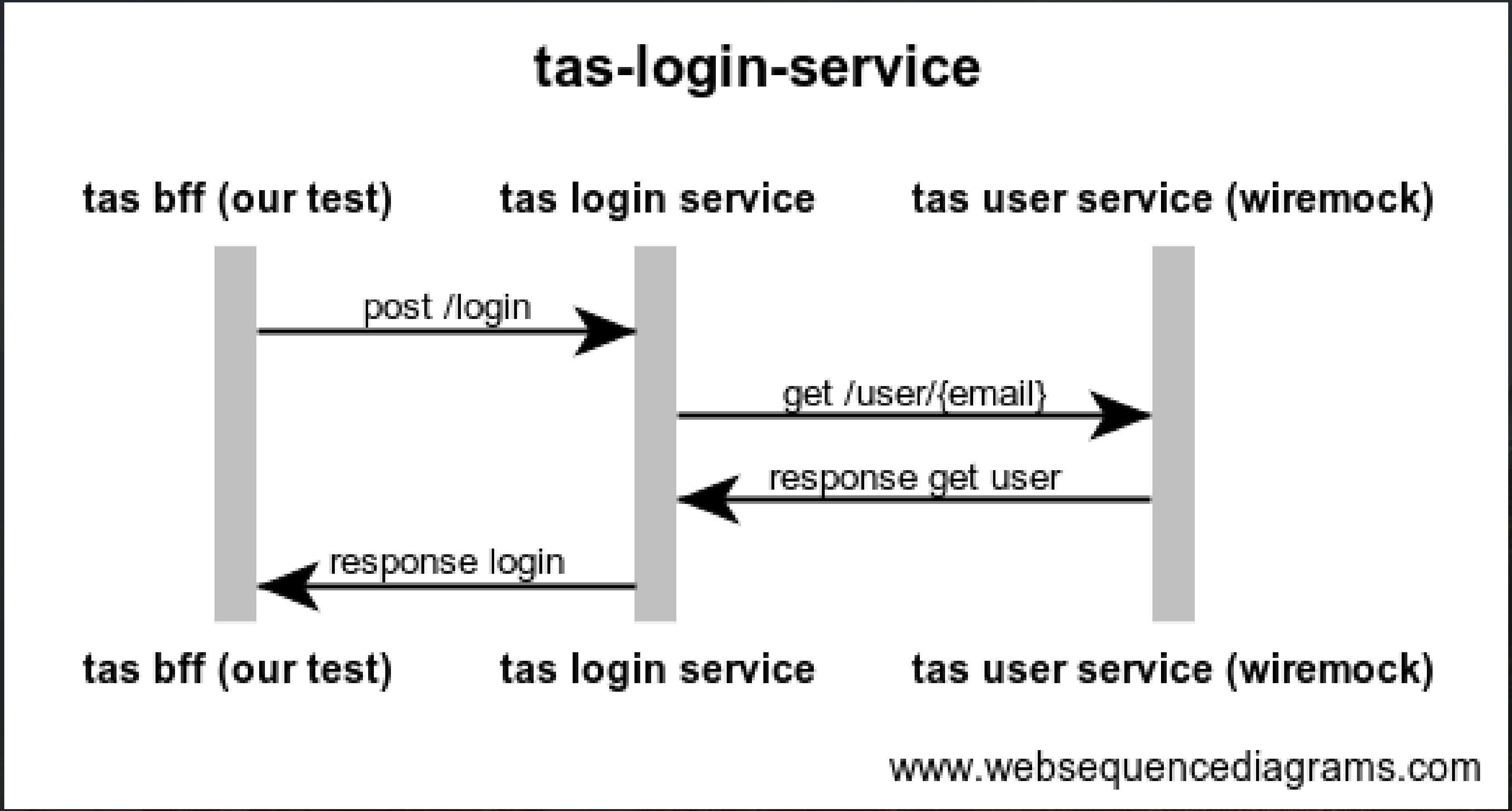
Component Test tas-login-service

- The tas-login-service uses the the tas-user-service to get user data

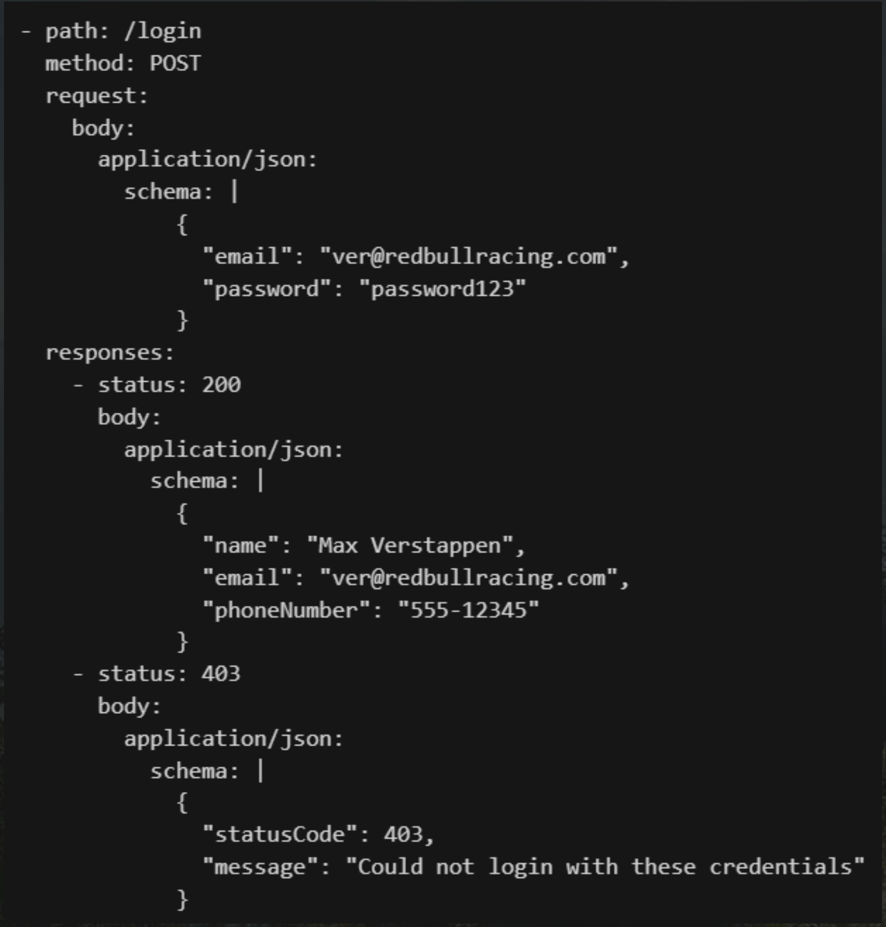
- Testing in isolation results in not having the tas-user-service

- The component test uses a mock tool (WireMock) to mock reponses from the tas-user-service

Component Test tas-login-service



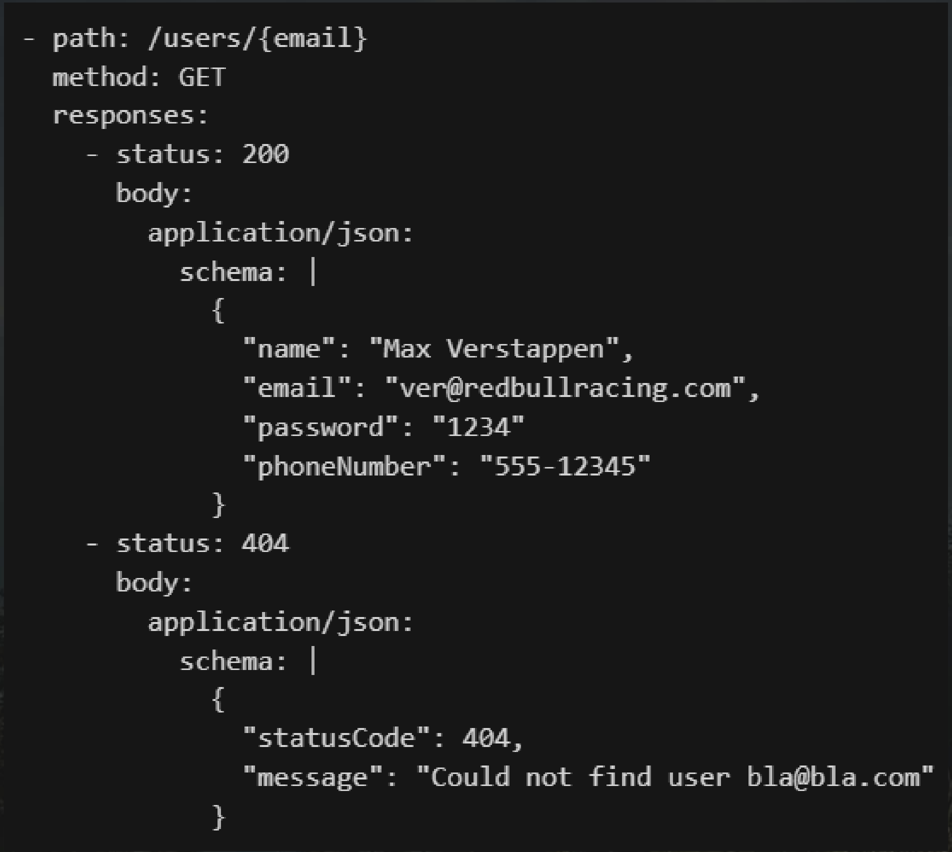
api spec tas-login-service /login



```
- path: /login
  method: POST
  request:
    body:
      application/json:
        schema: |
          {
            "email": "ver@redbullracing.com",
            "password": "password123"
          }
  responses:
    - status: 200
      body:
        application/json:
          schema: |
          {
            "name": "Max Verstappen",
            "email": "ver@redbullracing.com",
            "phoneNumber": "555-12345"
          }
    - status: 403
      body:
        application/json:
          schema: |
          {
            "statusCode": 403,
            "message": "Could not login with these credentials"
          }
```

challenge us

api spec tas-user-service /users/{email}



```
- path: /users/{email}
  method: GET
  responses:
    - status: 200
      body:
        application/json:
          schema: |
            {
              "name": "Max Verstappen",
              "email": "ver@redbullracing.com",
              "password": "1234"
              "phoneNumber": "555-12345"
            }
    - status: 404
      body:
        application/json:
          schema: |
            {
              "statusCode": 404,
              "message": "Could not find user bla@bla.com"
            }
```

docker-compose with mock

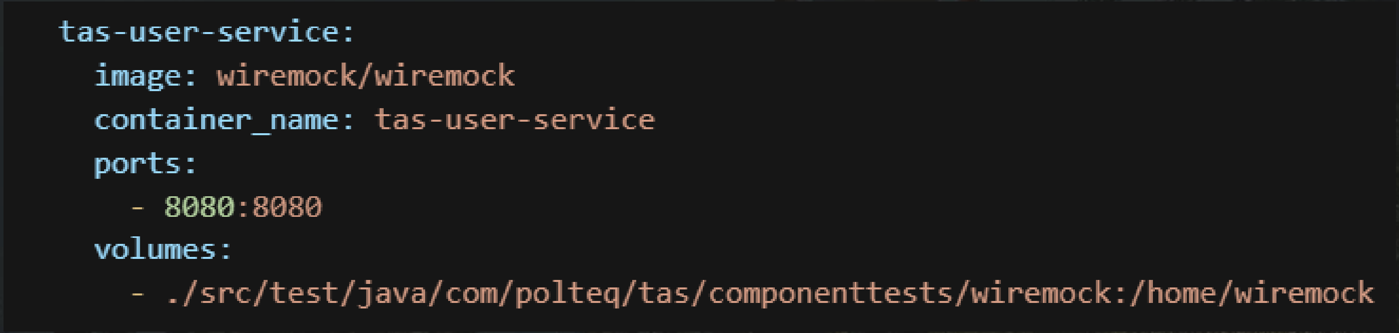
```yaml
services:

  tas-login-service:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: tas-login-service
    ports:
      - 8083:8080
    depends_on:
      - tas-user-service

  tas-user-service:
    image: wiremock/wiremock
    container_name: tas-user-service
    ports:
      - 8080:8080
    volumes:
      - ./src/test/java/com/polteq/tas/componenttests/wiremock:/home/wiremock
```

docker-compose with mock

- The login service communicates with the tas-user-service (which is actually wiremock)
- The volumes represent mock files defined in our test project which are passed through to WireMock
- These files contain predefined responses from the user service



```
tas-user-service:
  image: wiremock/wiremock
  container_name: tas-user-service
  ports:
    - 8080:8080
  volumes:
    - ./src/test/java/com/polteq/tas/componenttests/wiremock:/home/wiremock
```

Mocked response: user found

```
{
  "request": {
    "method": "GET",
    "url": "/users/a@a.nl"
  },
  "response": {
    "status": 200,
    "headers": {
      "Content-Type": "application/json"
    },
    "jsonBody": {
      "name": "fake user a",
      "email": "a@a.nl",
      "password": "password",
      "phoneNumber": "0612345678"
    }
  }
}
```

challenge us

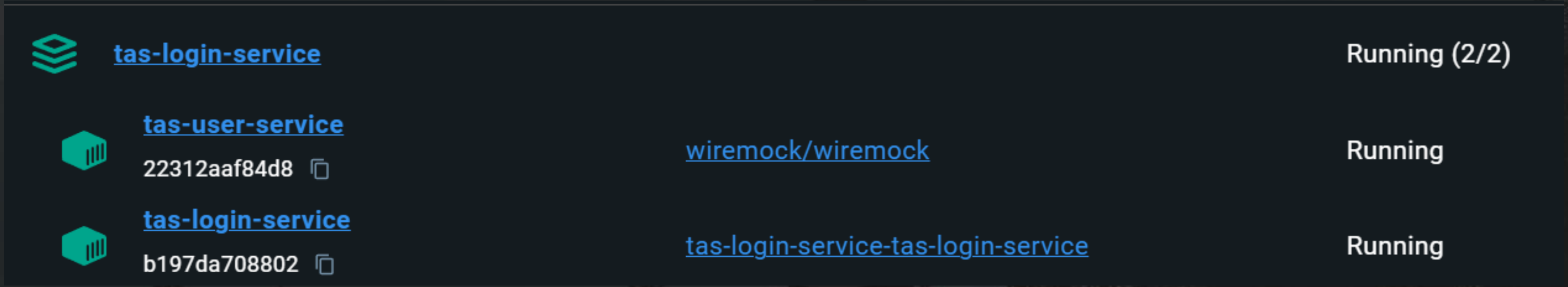
Mocked response: user not found

```json
{
  "request": {
    "method": "GET",
    "url": "/users/not@successful.login"
  },
  "response": {
    "status": 404,
    "headers": {
      "Content-Type": "application/json"
    },
    "jsonBody": {
      "statusCode": 404,
      "message": "Could not find user not@successful.login"
    }
  }
}
```

Running tas-login-service in isolation

- With mocking in place!

- `docker compose up --build`



tas-login-service Running (2/2)

tas-user-service 22312aaf84d8	wiremock/wiremock	Running
tas-login-service b197da708802	tas-login-service-tas-login-service	Running

- `CTRL-C`
- `docker compose down`

tas-login-service component tests

- Check the component test class LoginCTCase
- Run the test:
  - itShouldBePossibleToLoginWithValidData
- Complete the test:
  - itShouldNotBePossibleToLoginWithAnUnknownUser

challenge us

What tests can be deleted?

recap

The shift left journey

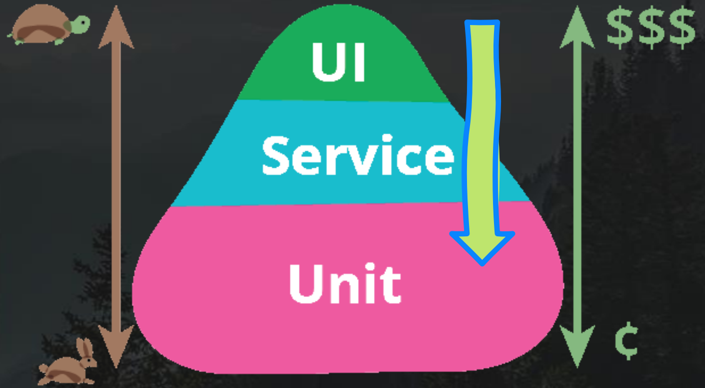
- We went from testing fully UI end-to-end
- To API testing (still E2E on api level)
- To testing a service in isolation with no related services
- To testing a frontend webapp in isolation using Playwright and its mocking capability
- To testing in isolation with related service using WireMock
- And reduced our E2E suite!

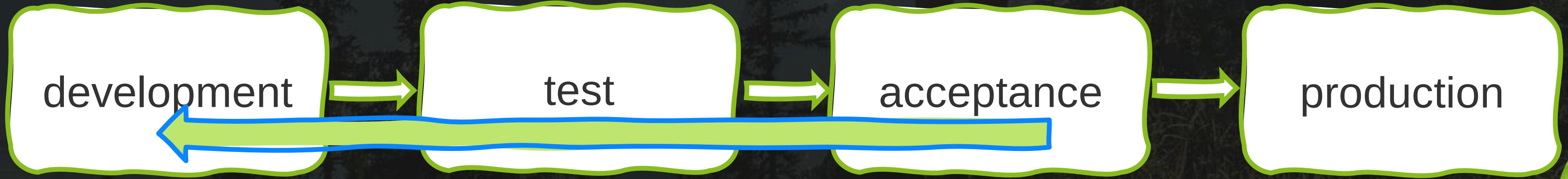
challenge us

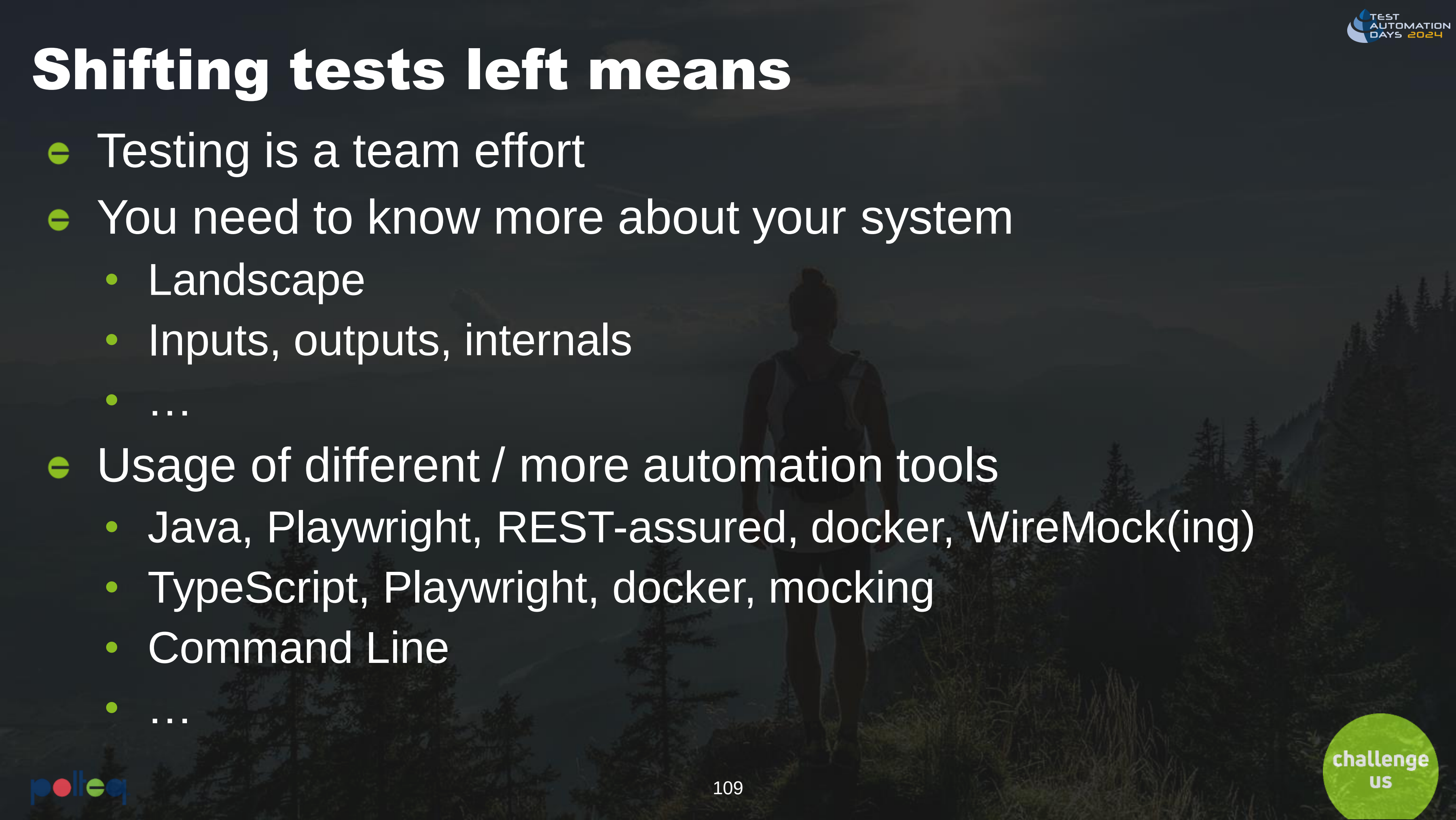
shift-left testing

approach to software and system testing
in which testing is performed
earlier in the
software development lifecycle

→ "test early and often"







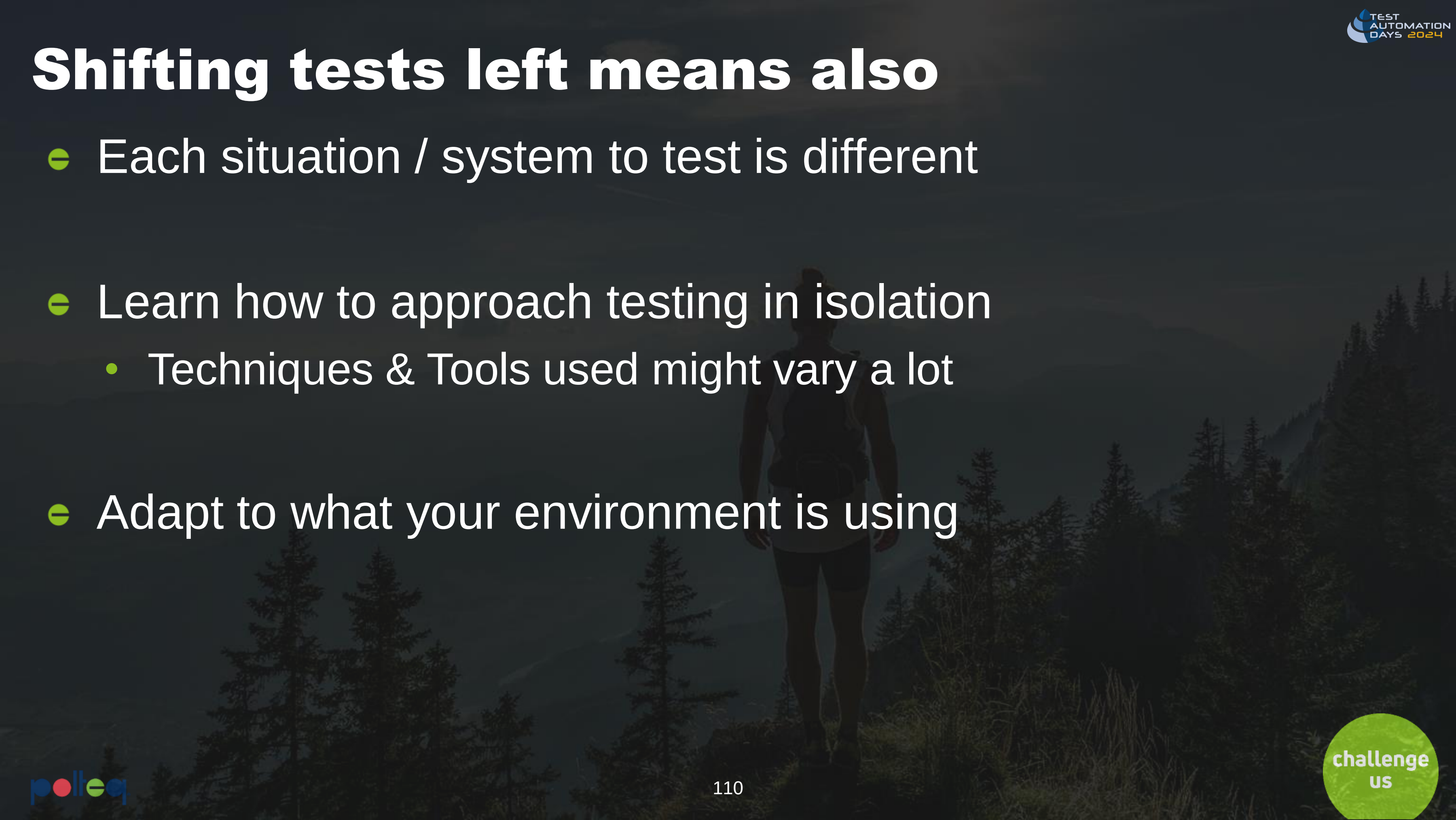
Shifting tests left means

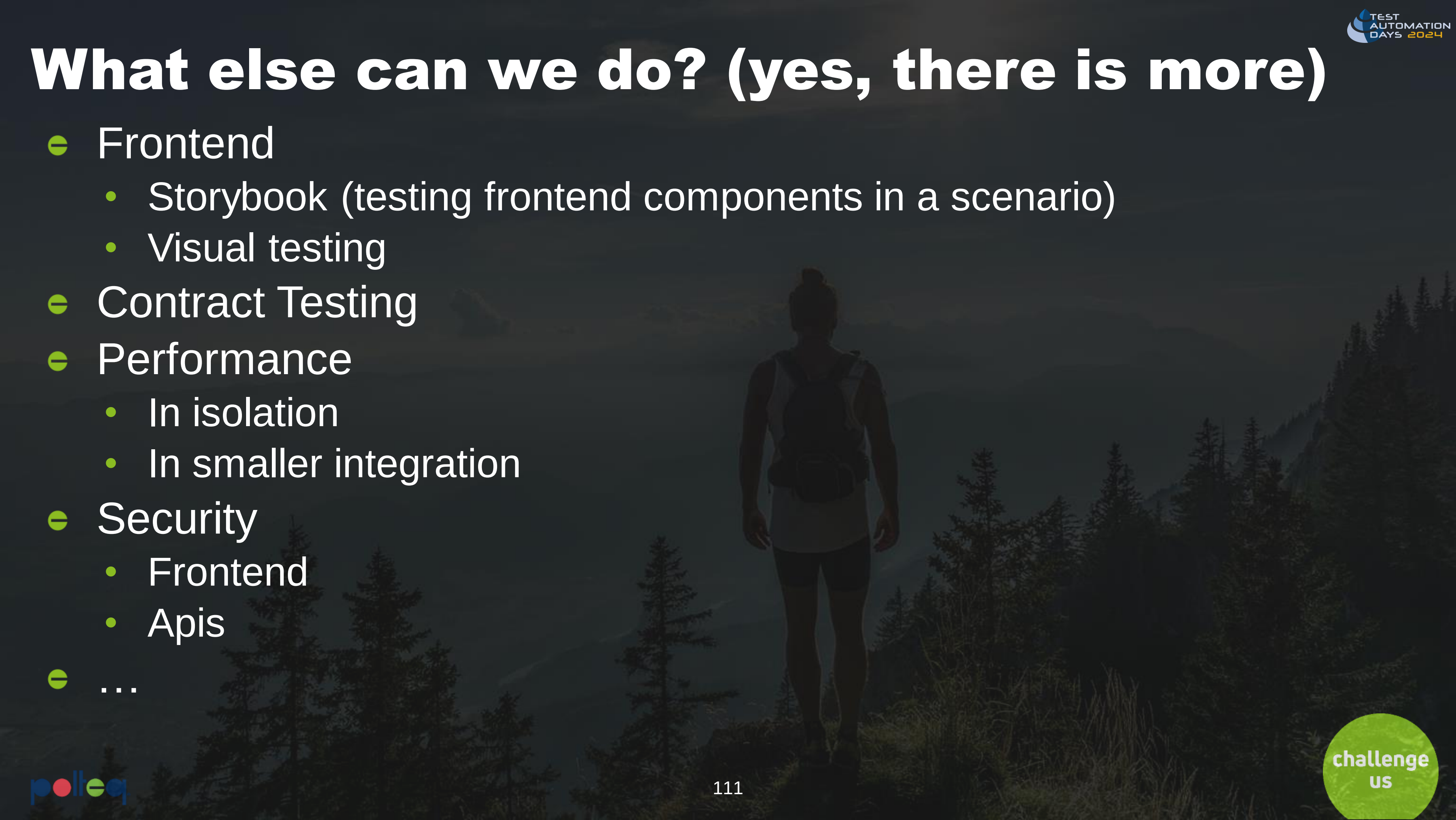
- Testing is a team effort
- You need to know more about your system
  - Landscape
  - Inputs, outputs, internals
  - …
- Usage of different / more automation tools
  - Java, Playwright, REST-assured, docker, WireMock(ing)
  - TypeScript, Playwright, docker, mocking
  - Command Line
  - …

challenge us

Shifting tests left means also

- Each situation / system to test is different
- Learn how to approach testing in isolation
  - Techniques & Tools used might vary a lot
- Adapt to what your environment is using





What else can we do? (yes, there is more)

- Frontend
  - Storybook (testing frontend components in a scenario)
  - Visual testing
- Contract Testing
- Performance
  - In isolation
  - In smaller integration
- Security
  - Frontend
  - Apis
- …

That's it folks!

Any questions?

THANK YOU!
TEST AUTOMATION DAYS 2024
polteq
challenge us

polteq®

Expertise

More

Fun

Sincerity

Sharing Knowlegde

Local

Personal

Focus

Groningen
Amsterdam
Deventer
Amersfoort
Rotterdam
Eindhoven
Leuven
Maastricht

challenge us