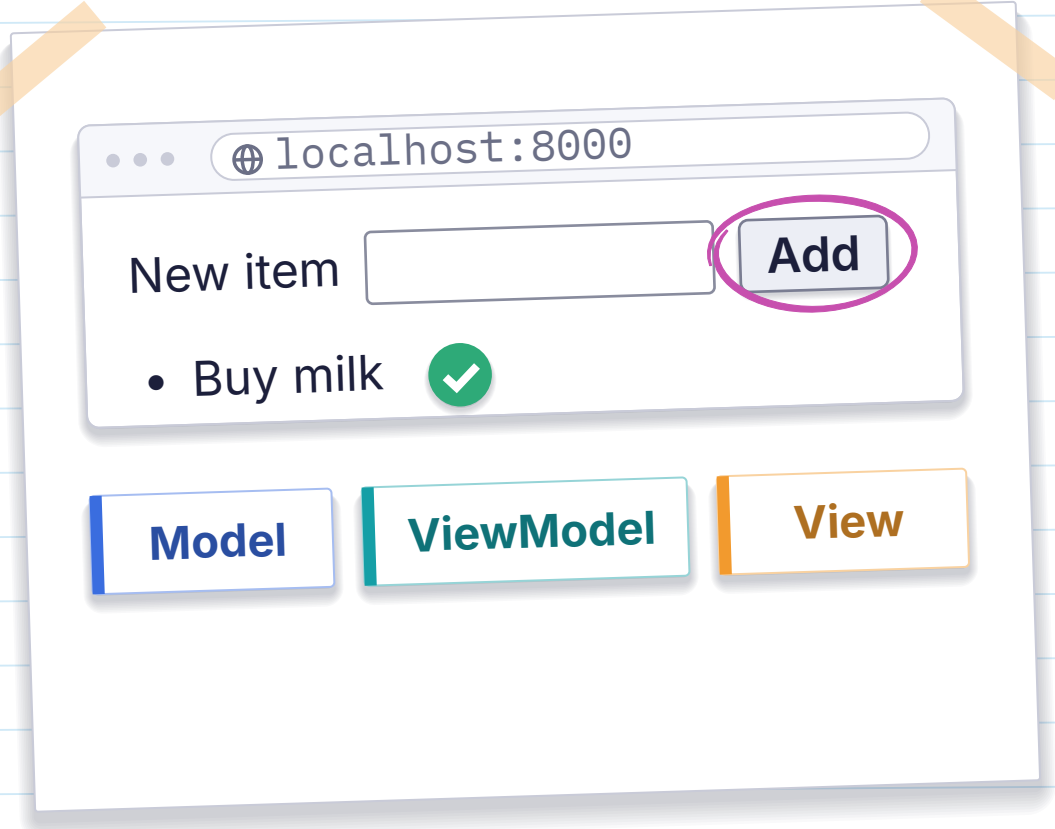
Prof. Aviv Zohar

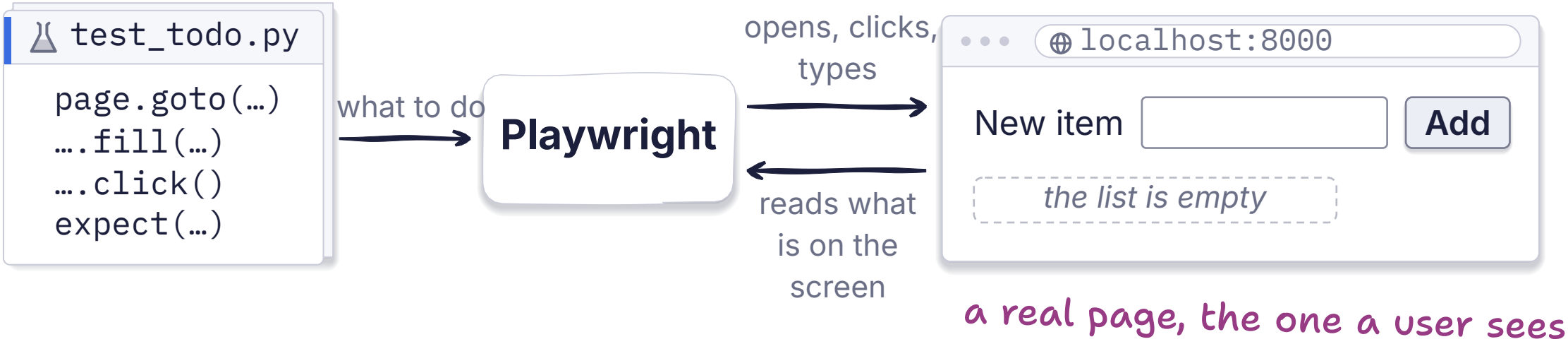
Semester A, 2026–7

Testing GUIs and their logic

Playwright tests, and MVVM for testable logic



A test can drive a real browser



The test finds controls by the names on the screen



index.html

```
<label for="new">New item</label>
<input id="new">                    a
<button>Add</button>                b
<ul></ul>                           c
```

ⓐ a box, labelled New item

ⓑ an Add button

ⓒ a list, empty for now

The test finds controls by the names on the screen



index.html

```
<label for="new">New item</label>
<input id="new">
<button>Add</button>
<ul></ul>
```

The test asks for the box with the label New item.

test_todo.py

```
page.get_by_label("New item")
```

The test finds controls by the names on the screen



index.html

```
<label for="new">New item</label>
<input id="new">
<button>Add</button>
<ul></ul>
```

The button named Add:
the names a person reads.

test_todo.py

```
page.get_by_label("New item")
page.get_by_role("button", name="Add")
```

Meaningful labels make these tests easier to write.

One browser test: open, type, click, check

test_todo.py

```python
from playwright.sync_api import expect
def test_add_item(page):
    page.goto("http://localhost:8000")
    page.get_by_label("New item").fill("Buy milk")
    page.get_by_role("button", name="Add").click()
    expect(page.get_by_role("listitem")).to_have_text(["Buy milk"])
```

page: a browser page that Playwright supplies



The app runs on this computer.
Its list starts empty.

One browser test: open, type, click, check

test_todo.py

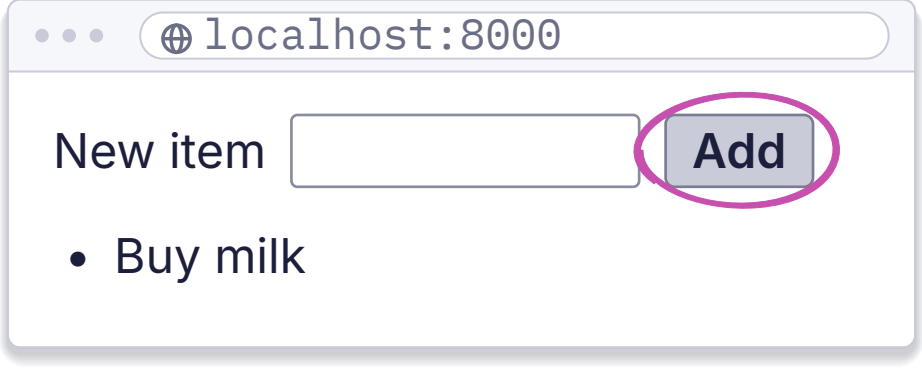
```python
from playwright.sync_api import expect
def test_add_item(page):
    page.goto("http://localhost:8000")
    page.get_by_label("New item").fill("Buy milk")
    page.get_by_role("button", name="Add").click()
    expect(page.get_by_role("listitem")).to_have_text(["Buy milk"])
```



One browser test: open, type, click, check

test_todo.py

```python
from playwright.sync_api import expect
def test_add_item(page):
    page.goto("http://localhost:8000")
    page.get_by_label("New item").fill("Buy milk")
    page.get_by_role("button", name="Add").click()
    expect(page.get_by_role("listitem")).to_have_text(["Buy milk"])
```

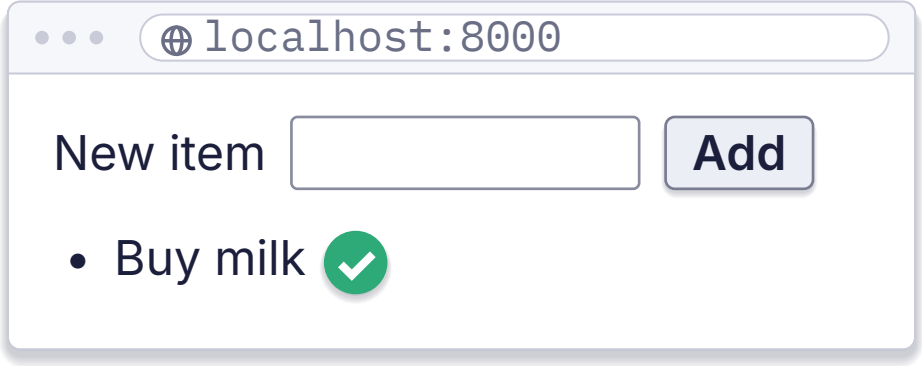


Buy milk now appears in the list.

One browser test: open, type, click, check

test_todo.py

```python
from playwright.sync_api import expect
def test_add_item(page):
    page.goto("http://localhost:8000")
    page.get_by_label("New item").fill("Buy milk")
    page.get_by_role("button", name="Add").click()
    expect(page.get_by_role("listitem")).to_have_text(["Buy milk"])
```



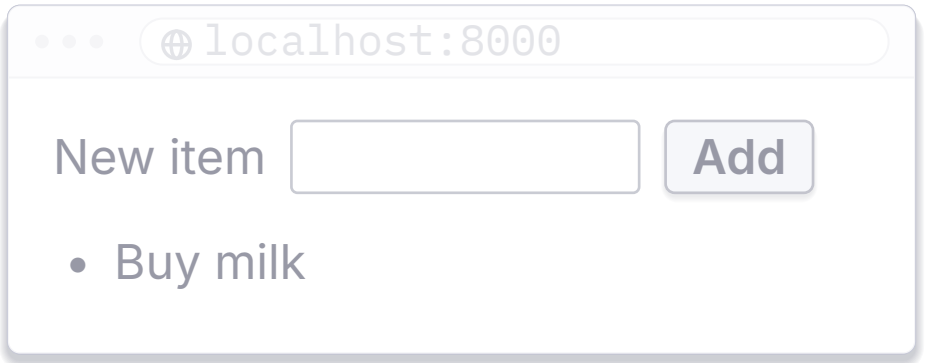
It waits for this, up to a timeout.

This check makes it a test. If the item never appears, the test fails, and the agent can read that failure.

One browser test: open, type, click, check

test_todo.py

```python
from playwright.sync_api import expect
def test_add_item(page):
    page.goto("http://localhost:8000")
    page.get_by_label("New item").fill("Buy milk")
    page.get_by_role("button", name="Add").click()
    expect(page.get_by_role("listitem")).to_have_text(["Buy milk"])
```



you typed by you

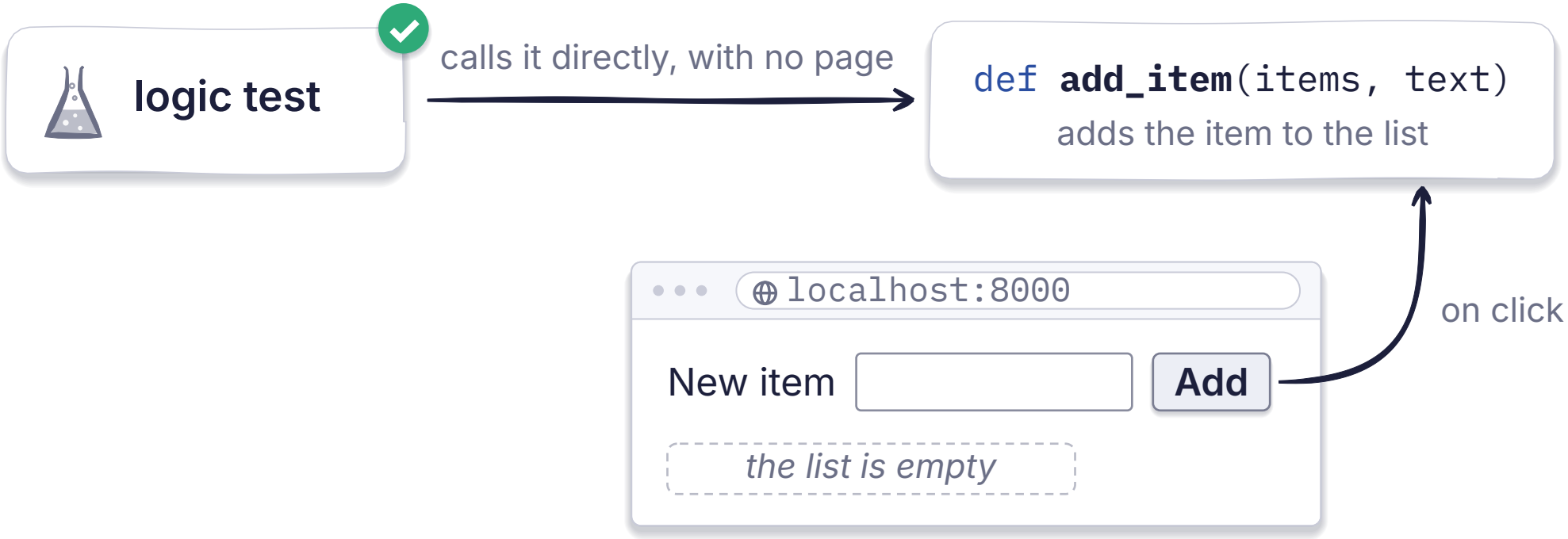
Write and run Playwright tests for adding an item.

ⓐ Type Buy milk into New item and click Add.

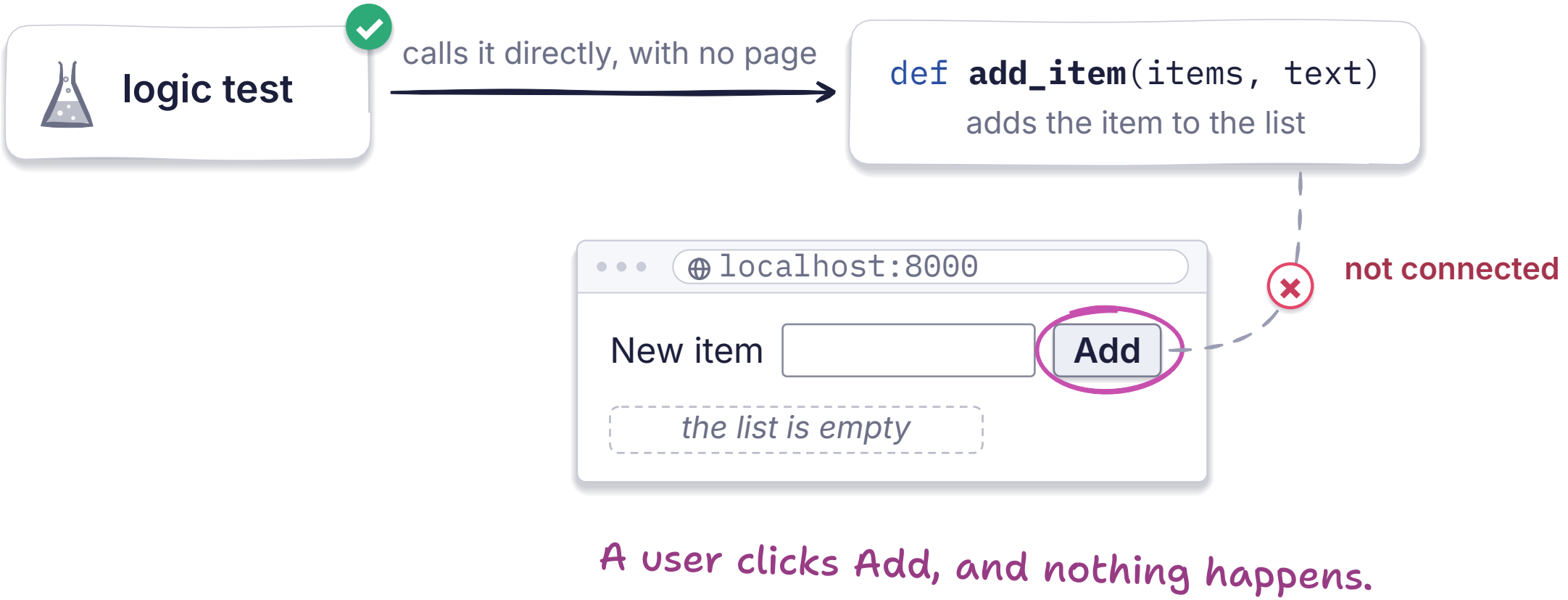
ⓑ Check that the list shows exactly Buy milk.

ⓐ the action ⓑ the result you expect

A logic test can pass while the button is broken

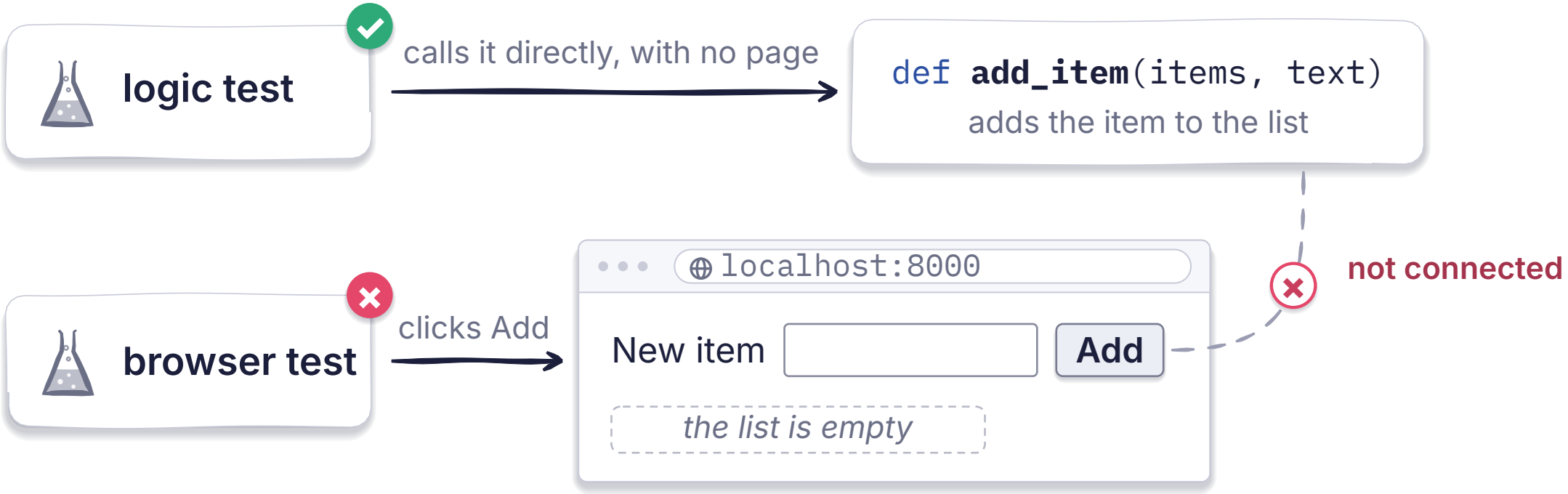


A logic test can pass while the button is broken



A user clicks Add, and nothing happens.

A logic test can pass while the button is broken

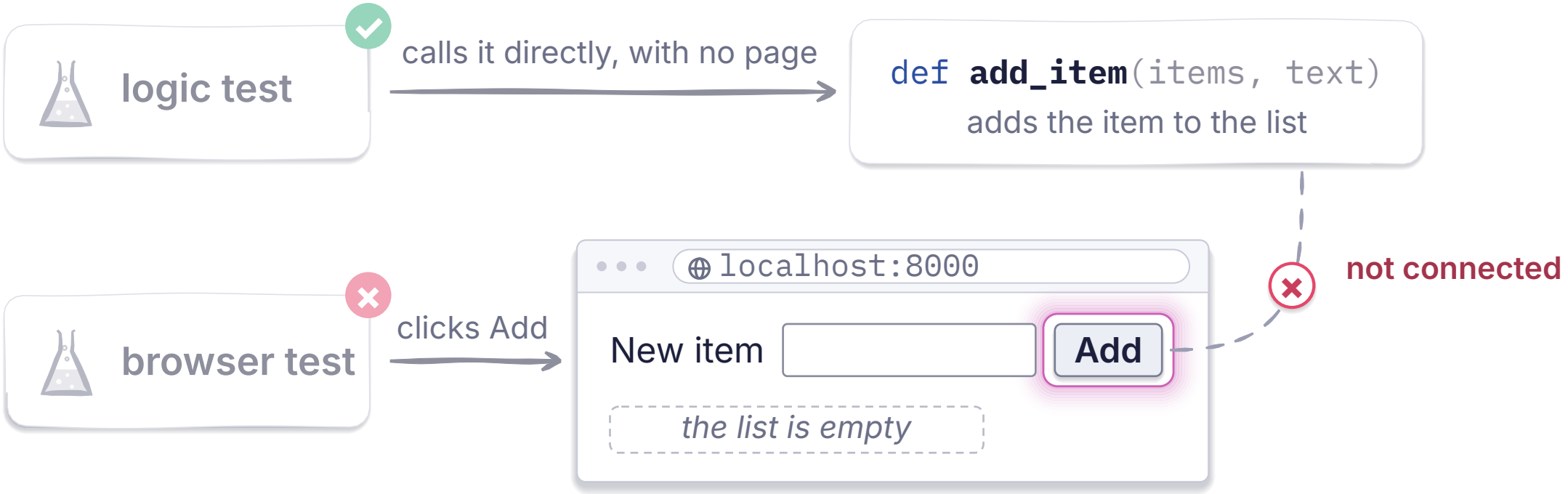


shell

```
FAILED test_add_item
Locator expected to have text '['Buy milk']'
Actual value: []
```

the actual output, with the button disconnected

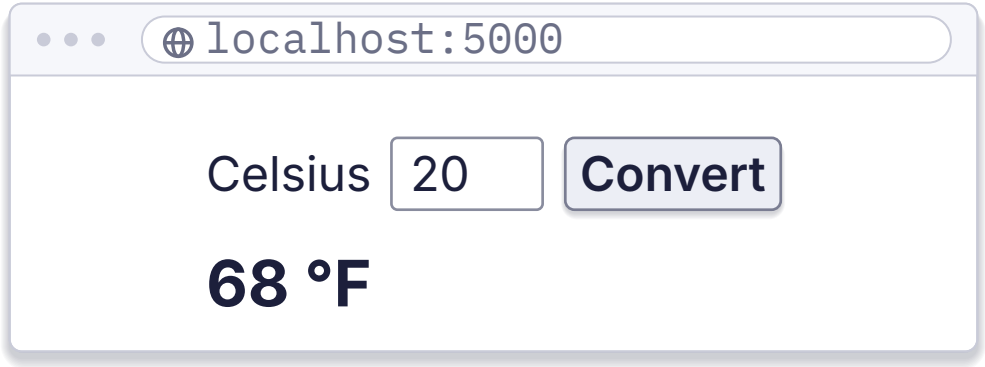
A logic test can pass while the button is broken




```
shell
FAILED test_add_item
Locator expected to have text '['Buy milk']'
Actual value: []
```

A browser test checks the interaction as well as the logic.

A smaller example: a temperature converter



Celsius in, Fahrenheit out: 20 °C is 68 °F.

Even this small converter has three jobs

1

The calculation
a temperature as a number

20 → 68

a number in, a number out

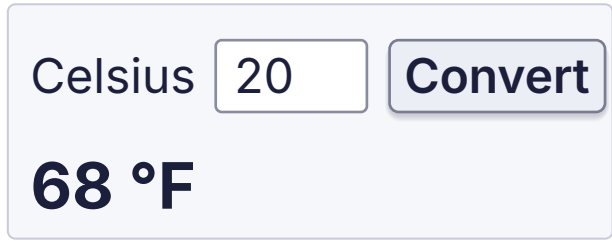
2

The screen's values
plain values in the program

text	`"20"`
convert	`convert()`
result	`"68 °F"`
error	`""`

3

The controls
input box, button, label



Even this small converter has three jobs

1

The calculation
a temperature as a number

20 → 68

a number in, a number out

2

The screen's values
plain values in the program

```
text     "20"
convert  convert()
result   "68 °F"
error    ""
```

3

The controls
input box, button, label

Celsius 20 Convert

68 °F

tested without creating any controls

exist only when the screen is drawn

Model, View, ViewModel

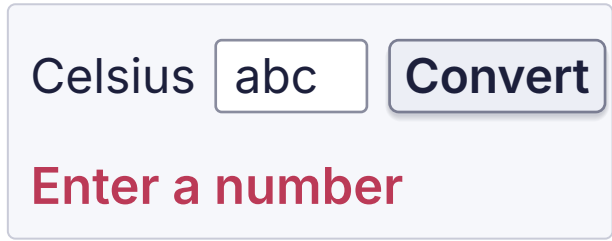
Model
data and rules

```
def to_fahrenheit(c):
    return c * 9 / 5 + 32
```

ViewModel
what the view needs

text	`"abc"`
convert	`convert()`
result	`""`
error	`"Enter a number"`

View
what the user sees and uses



Model, View, ViewModel

Model
data and rules

```
def to_fahrenheit(c):
    return c * 9 / 5 + 32
```

ViewModel
what the view needs

```
text     "abc"
convert  convert()
result   ""
error    "Enter a number"
```

View
what the user sees and uses

Celsius [abc] [Convert]

Enter a number

model in the usual software sense, not an AI model

Model, View, ViewModel

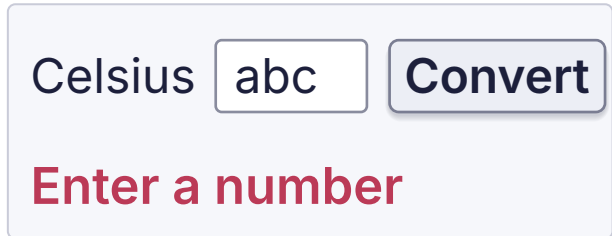
Model
data and rules

```
def to_fahrenheit(c):
    return c * 9 / 5 + 32
```

ViewModel
what the view needs

```
text      "abc"
convert   convert()
result    ""
error     "Enter a number"
```

View
what the user sees and uses



widgets, drawn on the screen

Model, View, ViewModel

Model
data and rules

```
def to_fahrenheit(c):
    return c * 9 / 5 + 32
```

ViewModel
what the view needs

text	`"abc"`
convert	`convert()`
result	`""`
error	`"Enter a number"`

View
what the user sees and uses

Celsius [abc] [Convert]

Enter a number

the screen's state as values;
it never touches a control

Model, View, ViewModel

Model
data and rules

```
def to_fahrenheit(c):
    return c * 9 / 5 + 32
```

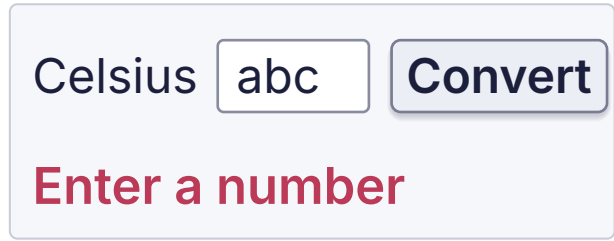
a temperature as a number belongs to the calculation

ViewModel
what the view needs

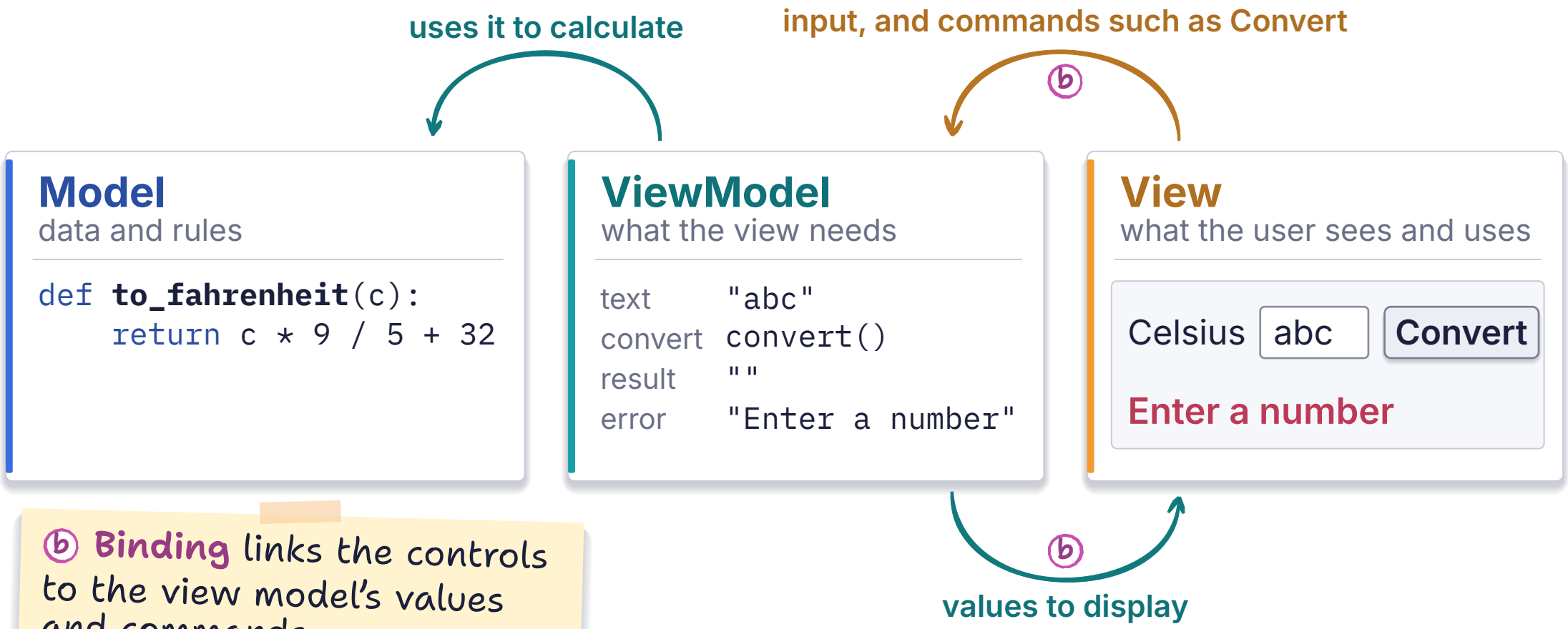
text	"abc"
convert	convert()
result	""
error	"Enter a number"

typed text, its error, the formatted result: the screen's

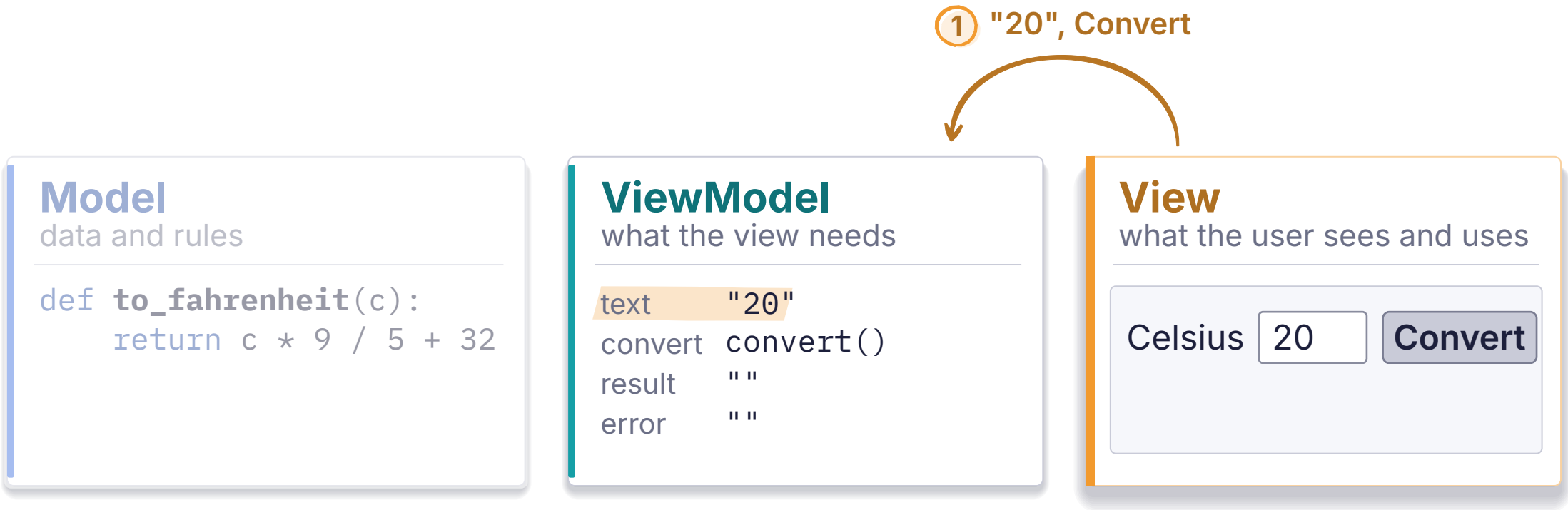
View
what the user sees and uses



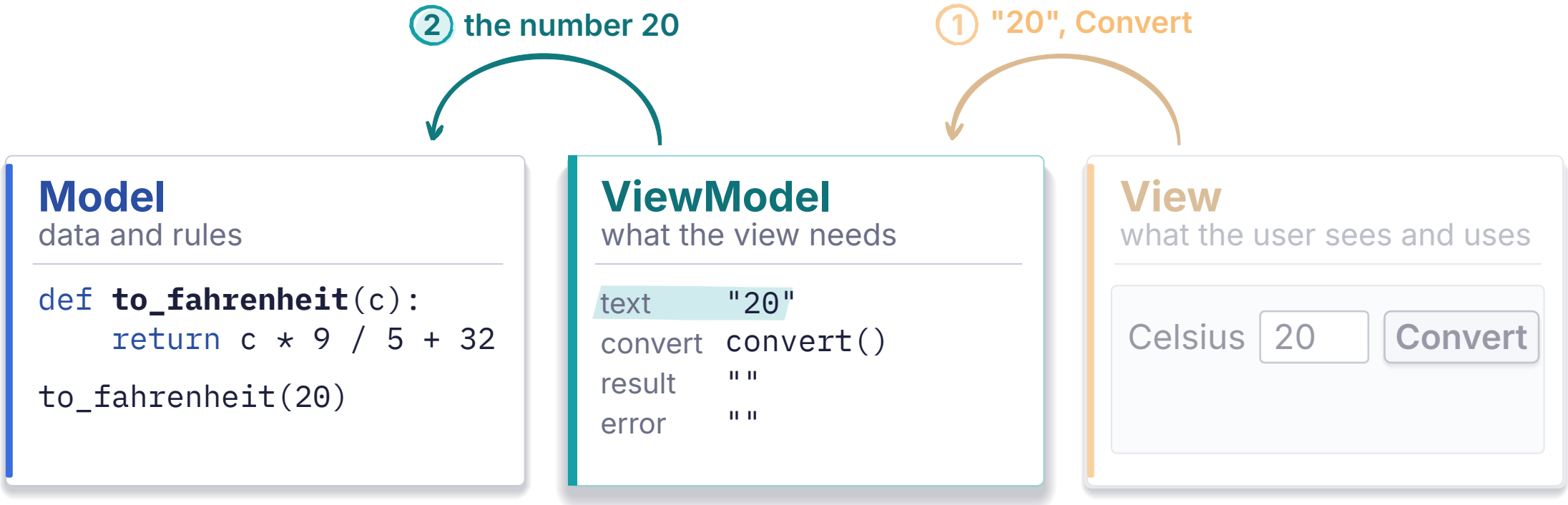
Model, View, ViewModel



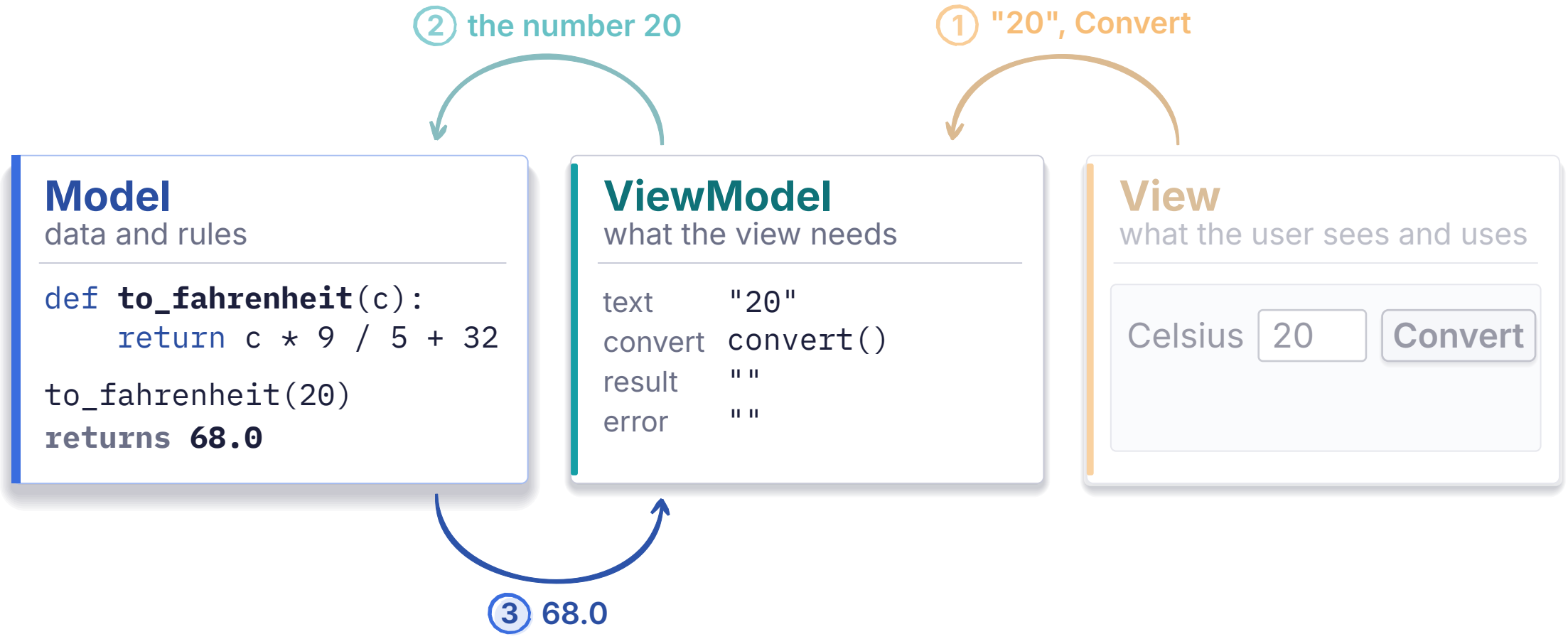
One interaction through the three parts



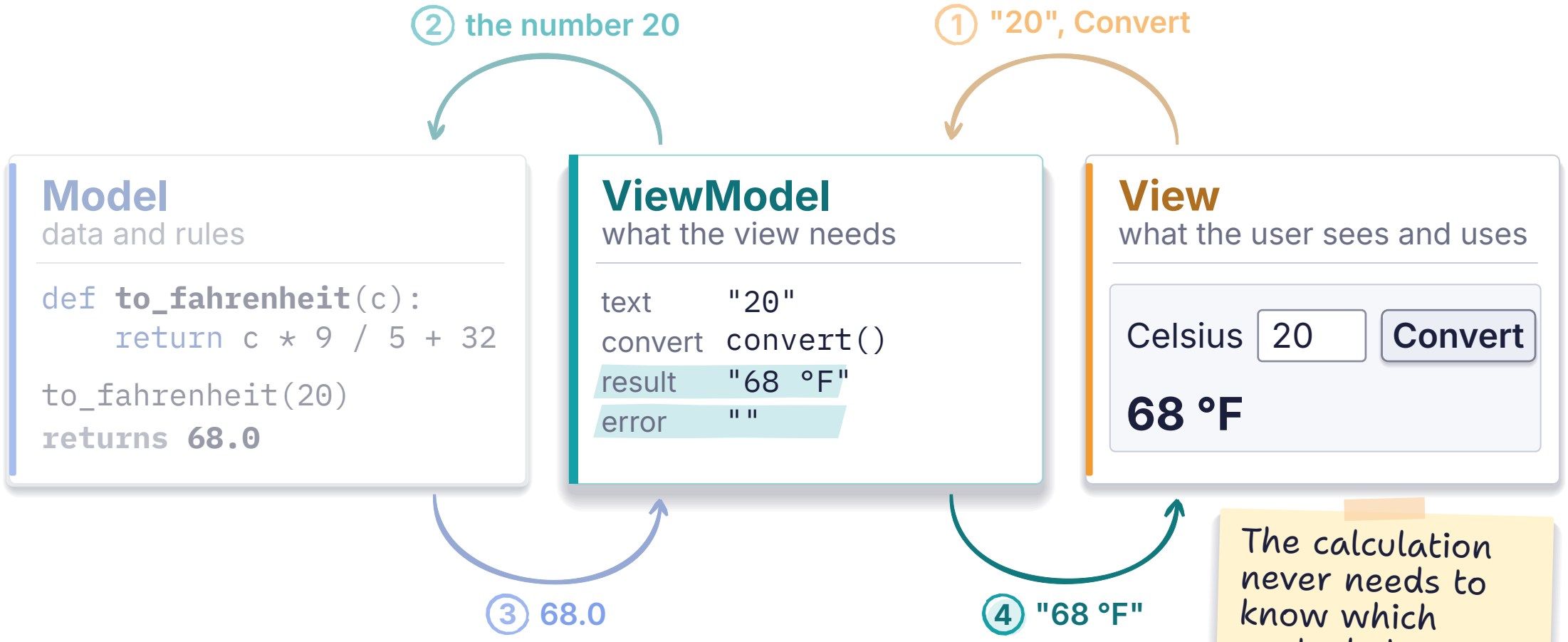
One interaction through the three parts



One interaction through the three parts



One interaction through the three parts



A test can do what the view did, with no page open

Model
data and rules

```
def to_fahrenheit(c):
    return c * 9 / 5 + 32
```

ViewModel
what the view needs

text	" 20 "
convert	convert()
result	"68 °F"
error	""

View

no page open

test_converter.py

```
vm.text = " 20 "
vm.convert()
assert vm.error == ""
assert vm.result == "68 °F"
```

Spaces ignored, no error, the result with its unit.

The browser test stays: it checks that the real controls call Convert and show the result.

A test can do what the view did, with no page open

Model
data and rules

```
def to_fahrenheit(c):
    return c * 9 / 5 + 32
```

ViewModel
what the view needs

text	"abc"
convert	convert()
result	""
error	"Enter a number"

View

no page open

test_converter.py

```
vm.text = " 20 "
vm.convert()
assert vm.error == ""
assert vm.result == "68 °F"
```

test_converter.py

```
vm.text = "abc"
vm.convert()
assert vm.error == "Enter a number"
assert vm.result == ""
```

Bad input: the error is set, the old result cleared.

Which test catches which failure

	checked directly, no GUI	left for a GUI test
Model	`to_fahrenheit(0) == 32.0`	the user can reach the calculation through the page

Which test catches which failure

	checked directly, no GUI	left for a GUI test
Model	`to_fahrenheit(0) == 32.0`	the user can reach the calculation through the page
ViewModel	`"abc"` sets an error and clears the result	the error is actually shown on the screen

Which test catches which failure

	checked directly, no GUI	left for a GUI test
Model	`to_fahrenheit(0) == 32.0`	the user can reach the calculation through the page
ViewModel	`"abc"` sets an error and clears the result	the error is actually shown on the screen
View and its connections	*no direct check*	clicking Convert runs the command and shows the result

Which test catches which failure

	checked directly, no GUI	left for a GUI test
Model	`to_fahrenheit(0) == 32.0`	the user can reach the calculation through the page
ViewModel	`"abc"` sets an error and clears the result	the error is actually shown on the screen
View and its connections	*no direct check*	clicking Convert runs the command and shows the result
Cost: time and computing, even when the agent writes the test.	a calculation check **under a millisecond**	a GUI test **a few seconds** app, screen and controls ready

illustrative timings

Which test catches which failure

	checked directly, no GUI	left for a GUI test
Model	`to_fahrenheit(0) == 32.0`	the user can reach the calculation through the page
ViewModel	`"abc"` sets an error and clears the result	the error is actually shown on the screen
View and its connections	*no direct check*	clicking Convert runs the command and shows the result
	a calculation check **under a millisecond**	a GUI test **a few seconds** app, screen and controls ready

Many calculation cases directly. GUI tests for the interactions that need them.

illustrative timings

What to ask the agent

you first request

Propose a separation between the app's rules, the screen's values and actions, and the visible controls. Use MVVM where it helps. Explain the responsibilities before you change any code.

Does the separation work?

- Can the rules run without a screen?
- Can the screen display a state it is given?

Ask for a small separation, not a rewrite that only gives files new names.

What to ask the agent

you first request

Propose a separation between the app's rules, the screen's values and actions, and the visible controls. Use MVVM where it helps. Explain the responsibilities before you change any code.

you second request

Add and run tests for the rules and the screen behaviour, and Playwright tests for the main interactions and what they show. Report the commands you ran, the results, and any tests you could not run.

Does the separation work?

- ? Can the rules run without a screen?
- ? Can the screen display a state it is given?

GUI testing tools

web Playwright, in the browser

desktop, mobile their own testing libraries

The same idea: perform the user's actions, check what appears.

What to ask the agent

you first request

Propose a separation between the app's rules, the screen's values and actions, and the visible controls. Use MVVM where it helps. Explain the responsibilities before you change any code.

you second request

Add and run tests for the rules and the screen behaviour, and Playwright tests for the main interactions and what they show. Report the commands you ran, the results, and any tests you could not run.

Name the interactions and the results you expect, then review the tests and their results.

Does the separation work?

- ? Can the rules run without a screen?
- ? Can the screen display a state it is given?

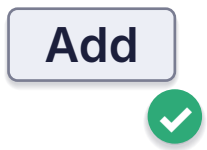
test_gui.py

```
def test_convert(page):
    page.goto(…)
    …
```

A test file the agent wrote is not evidence that it passed.

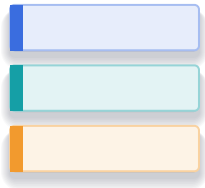
Could not run it?
Ask what is missing.

Ask for both, and check what ran



GUI tests

perform the user's actions and check the result, through the real controls



Separated logic

the rules and the screen's state, tested directly in many more cases

Ask for both, and check what the agent actually ran.