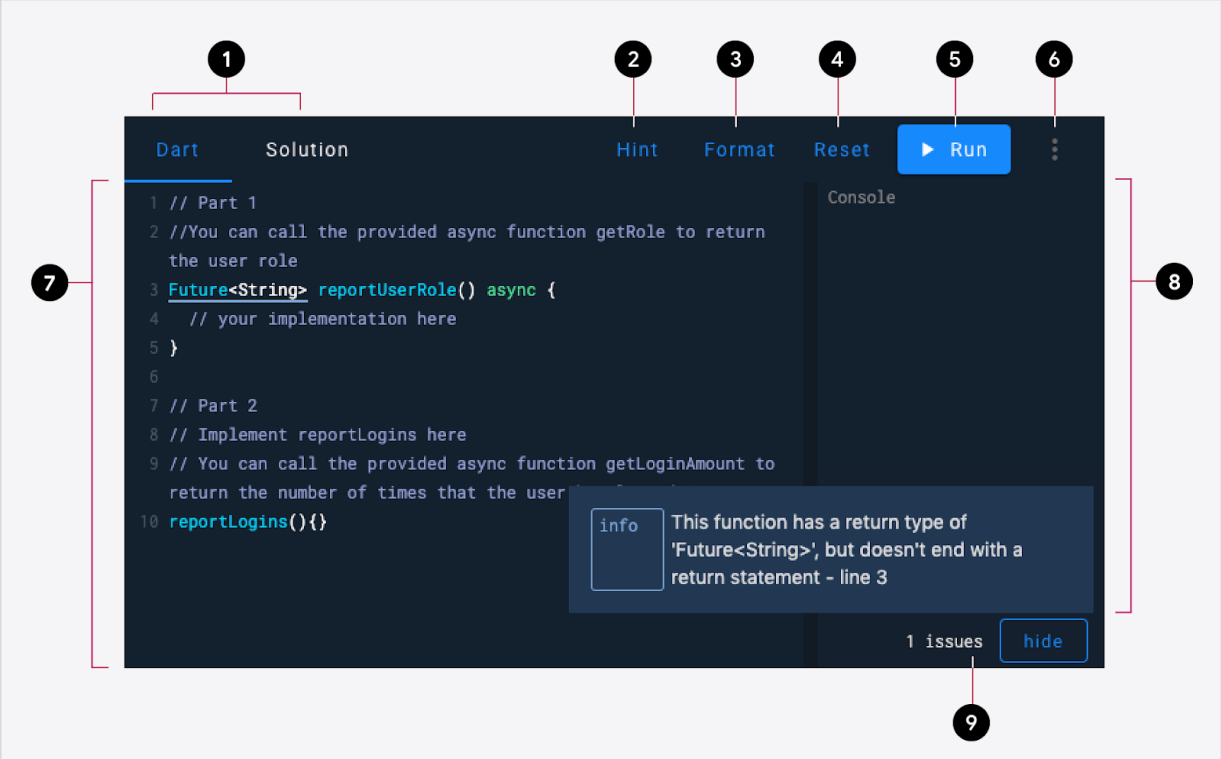
DartPad in Tutorials: Best Practices

This guide introduces DartPad, a tool for creating effective and engaging educational content for Dart and Flutter users. It provides you, the tutorial authors, with advice, tips, and examples for using DartPad.

What is DartPad?

DartPad is an online code editor for the Dart language. In addition to executing regular Dart programs, it can run Flutter programs and show graphic output. DartPad enables learners to engage with the content without having to set up a development environment. The following screenshot shows DartPad. You can also try it out.

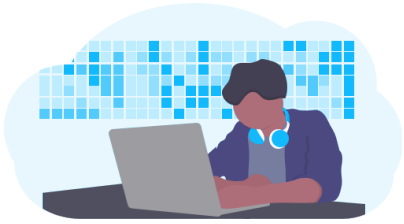


A screenshot of DartPad with annotations

1. Tabs: Dart, Solution, and Tests (hidden)
2. Hint: Offers help
3. Format: Runs the code formatter (dartfmt)
4. Reset: Erases your work and restores the editor to its original state
5. Run
6. Menu: The Tests tab toggle
7. Code pane
8. Output: Console, UI output
9. Analyzer: Instantly checks the code

DartPad is under development. If you find a bug or have suggestions, please create an issue. Learn more about DartPad at dart.dev, its GitHub repo, and the Embedding Guide.

Why use DartPad in a tutorial?



A traditional tutorial provides learners with step-by-step instructions and static code snippets in order to complete a software development task.

DartPad enables learners to test their knowledge by running example code and by completing exercises as they go through the steps in the tutorial.

Coding tutorials are sometimes referred to as codelabs.

What you'll learn from this guide

- Design principles for interactive tutorials
- Ways of using DartPad in tutorials
- Case study: DartPad in a Google tutorial

This guide doesn't provide general technical writing information. If you're new to technical writing, you can reference the following resources:

- Google Developer Documentation Style Guide: The primary writing style reference for Google-related developer documentation
- Cloud Shell Walkthrough Writing Guide: Suggestions about high-level tutorial structure
- Content guidelines: Content handbook for web.dev
- Other editorial resources

The guidance provided in this doc is based on Google's internal research and related academic research about instructional design. This document will evolve as we learn more about what works.

Design principles for interactive tutorials

DartPad enhances a traditional tutorial through its ability to support guided discovery learning, a recognized learning approach that calls for a balance between the student's freedom of exploration and the teacher's scaffolding of the learning process.

Past research (Mayer, Richard E., 2004) suggested that guided discovery learning was more effective than pure discovery learning in helping students learn and transfer their learning. Pedagogical principles of coding tutorials (Kim, Ada S., and Amy J. Ko., 2017) also emphasize the importance of utilization, actionability and feedback, transfer learning, and support.

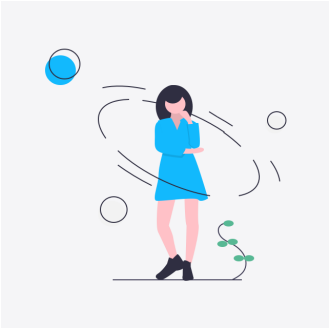
Based on the guided discovery learning approach and our application of it on coding tutorial development, we identified the following principles for using DartPad in interactive tutorials:

Enable hands-on exercises

Provide code examples and hands-on practice that engage learners in actively writing code.

Give just-in-time and just-enough help, and feedback

Offer precise, contextualized, and immediate feedback on learners' progress without taking away learning opportunities.



Facilitate reflection

Encourage meta-cognitive learning, the learners' ability to predict the outcomes of their learning and monitor their understanding.

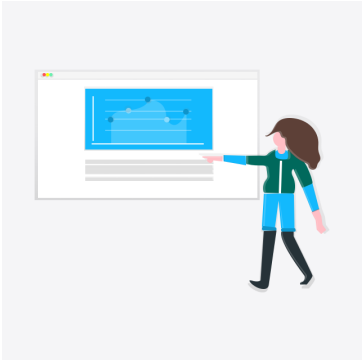
Support learning transfer

Help learners leverage accumulated knowledge gained from tutorials and apply it to new settings.

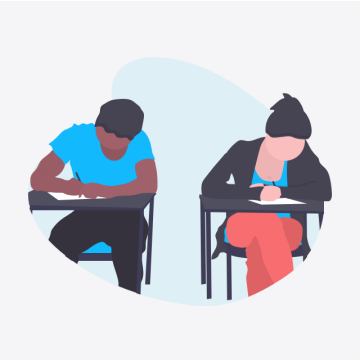
Ways of using DartPad in tutorials

The pedagogical principles previously described are realized by configuring DartPad to show demos, exercises and quizzes.

I. Demo

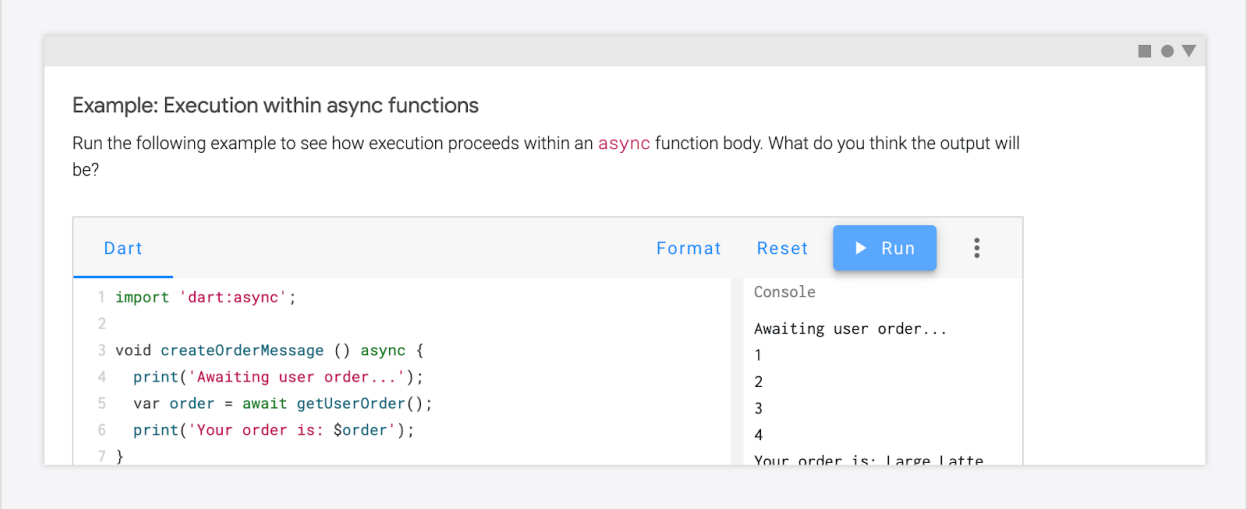


II. Exercise



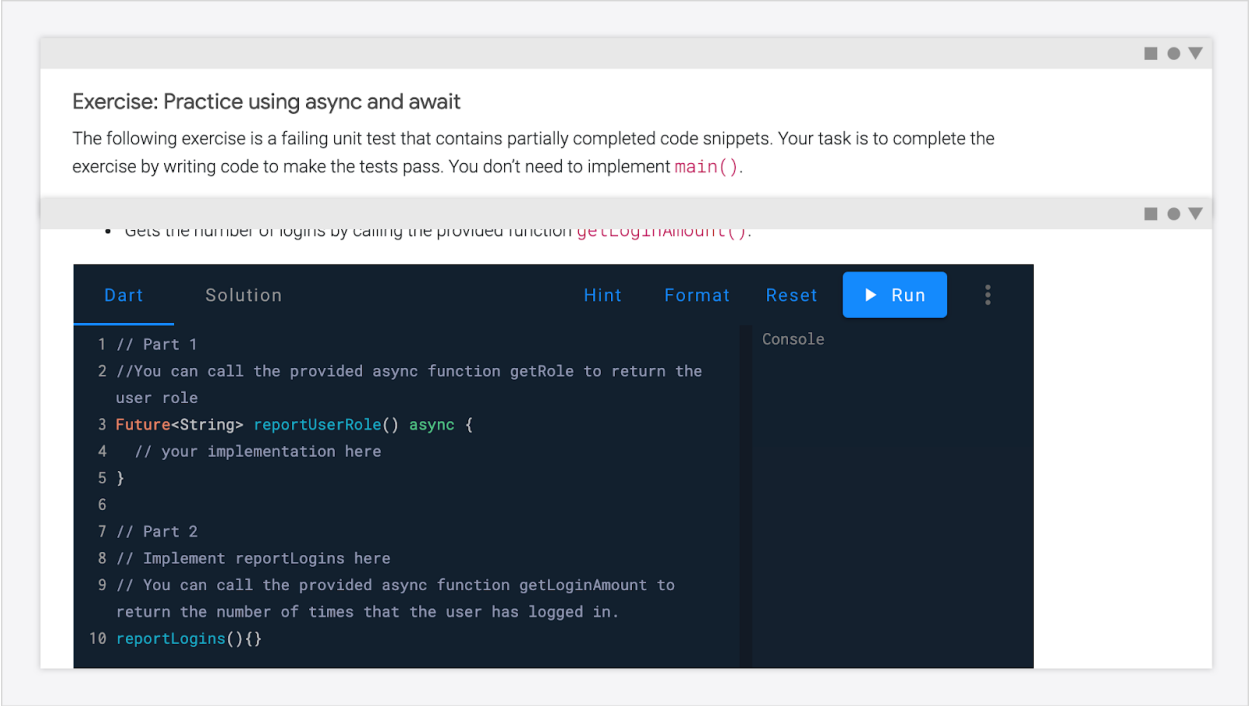
III. Quiz

The **demo** provides interactive code examples for concepts and feature use. This is similar to how science teachers bring experiments into the classroom to make concepts concrete and memorable.



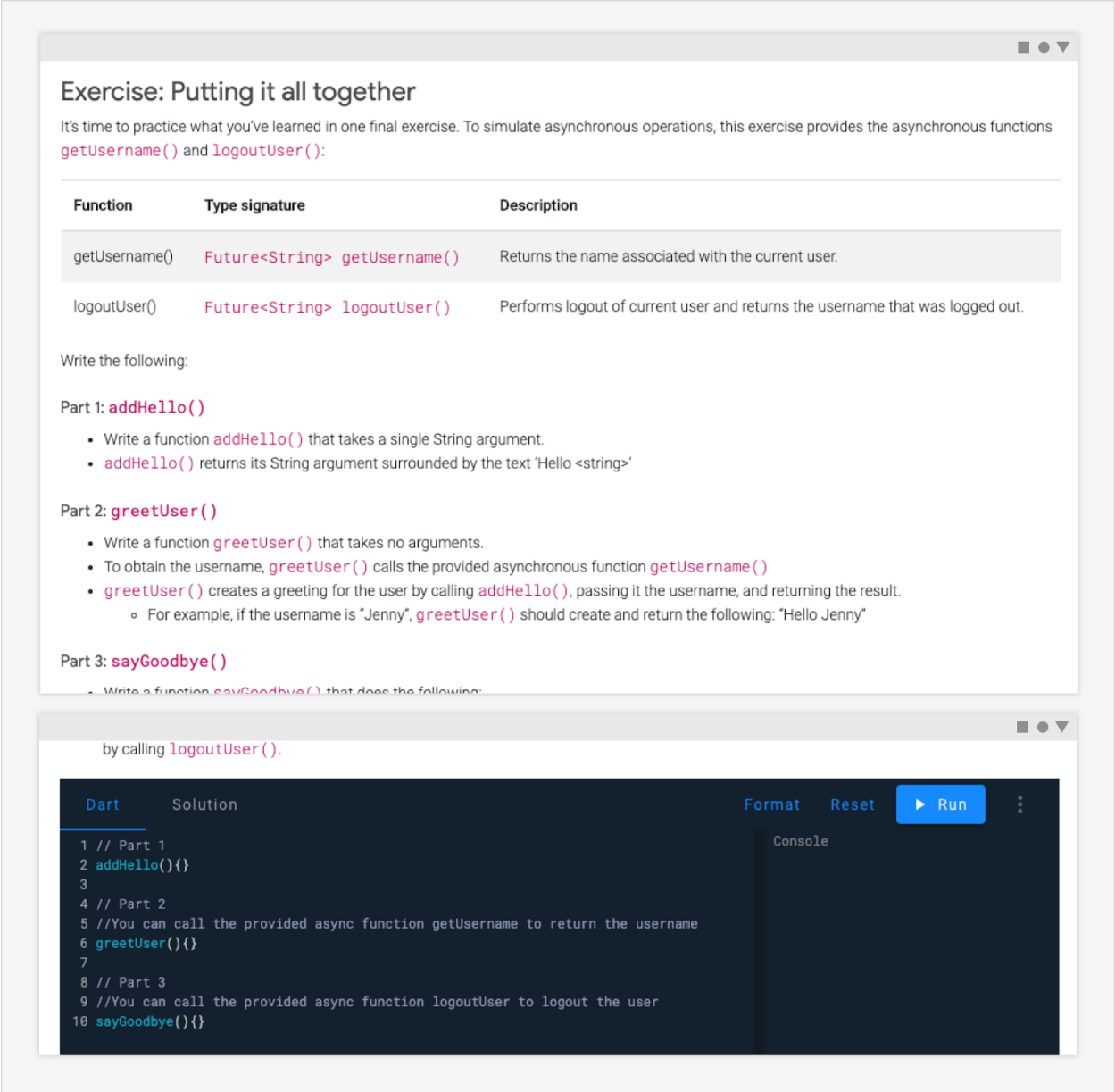
A code demo: abstract execution flow and instructions for changing code

The **exercise** provides the necessary scaffolds for the learner to carry out a specific task. The scaffolds can include a starting template, key information, check points, and other kinds of feedback and help.



A coding exercise: implementing two async functions

The **quiz** enables the learner to work on another similar problem, and automatically check if the solution is correct. This is useful to evaluate whether the transfer of learning happened.



A quiz on implementing 3 functions about async functions, async/await keywords and handling errors

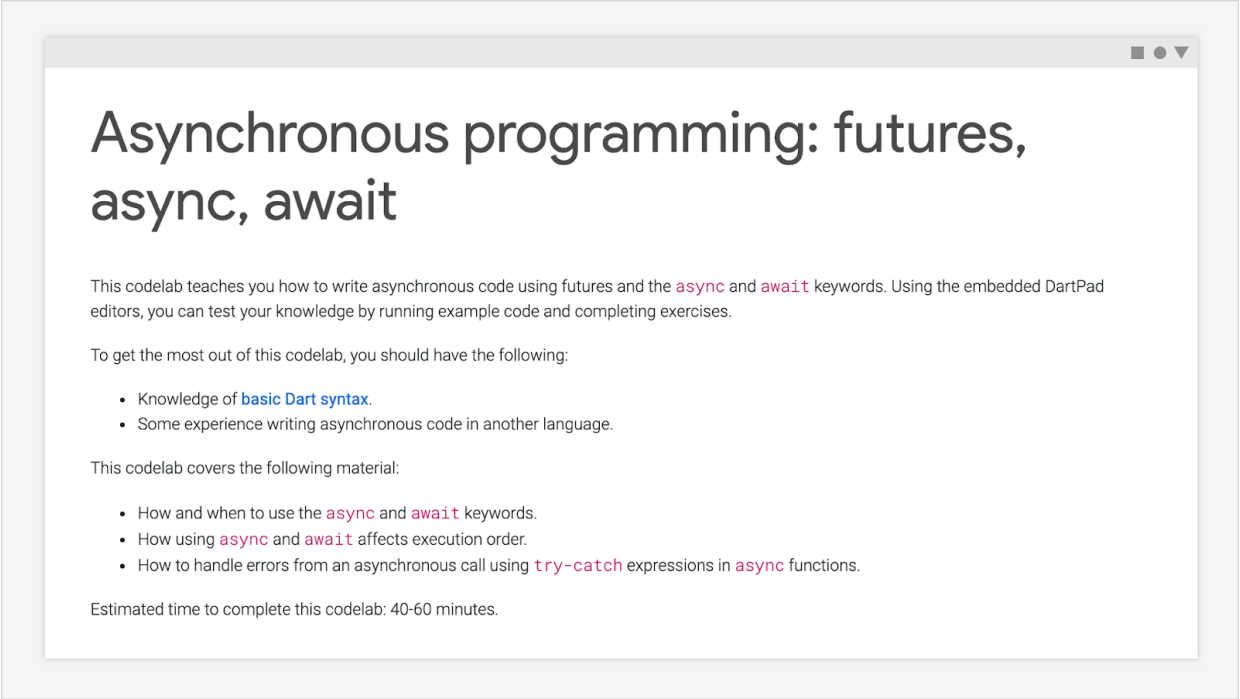
What's the difference between exercise and quiz? There are two main differences:

1. **Exercises provide more scaffolds.** When you write exercise instructions, in addition to describing the expected results (for example, add two boxes and ensure that the extra space between the two are divided evenly), you may step learners through how to get to that outcome (for example, enter two `Text` widgets inside the `Row` and use the `manAxisAlignment.SpaceBetween` property). Also, there might be more pre-filled code snippets in the starting state, and more hints in the exercises than in a quiz.
2. **A quiz is more challenging than an exercise.** A quiz tests the transfer of learning. The problem in a quiz is usually a bit harder than a problem in an exercise, and requires a

real understanding of how to complete a task, instead of just copying and pasting code snippets. Until now, we might not have explicitly claimed "this is a quiz" because we didn’t know how users would perceive the word "quiz," and we didn’t want to add too much pressure.

Case study: the Dart Futures codelab

So far, you learned about the guiding principles and ways of using DartPad for creating interactive tutorials. But, how can you apply them when you are developing a real-world tutorial? We'll explain that through a detailed case study. In the case study we used DartPad in the instructional design of a tutorial titled "Asynchronous programming: futures, async, await" or "the Dart Futures codelab" for short.



The "Asynchronous programming: futures, async, await" codelab

This tutorial teaches developers how to write asynchronous code in Dart using the `Future` class and the `async` and `await` keywords. The following chart shows the general structure of the codelab and where DartPad is used to support learning, including demos, exercises, and a quiz. In the following topics, we walk you through how each use case is used in the design of this tutorial.

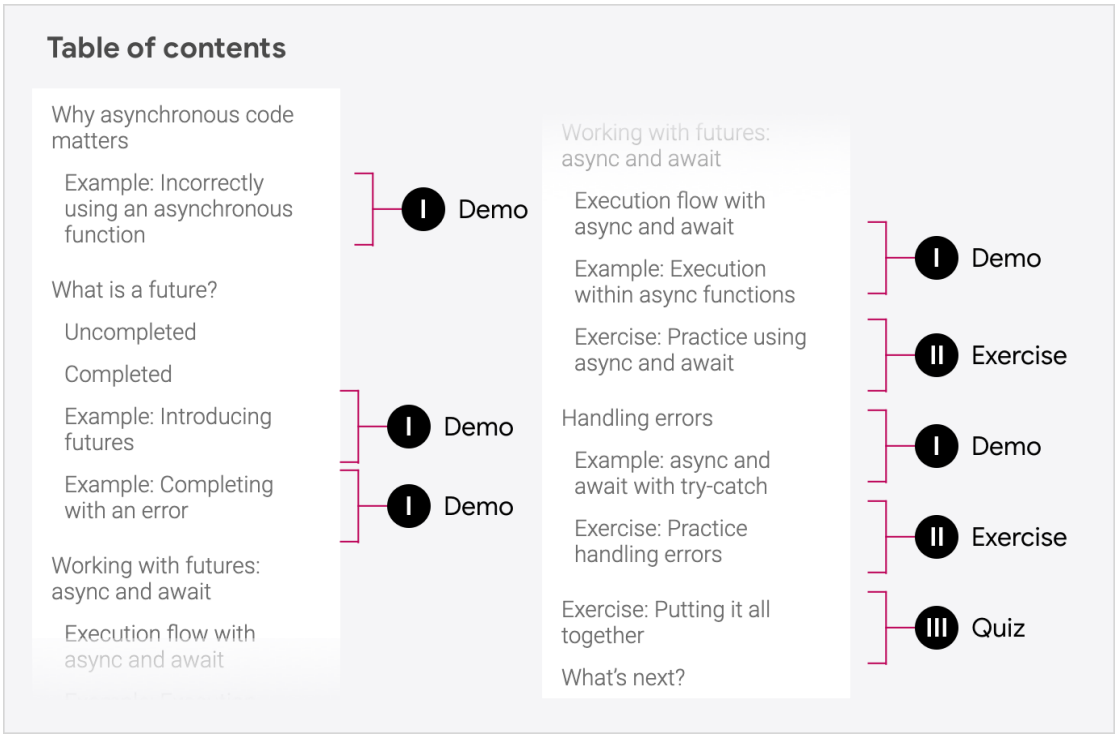


Table of contents: 5 demos, 2 exercises, and 1 final quiz

I. Demo

First, demos was used to accomplish the following goals:

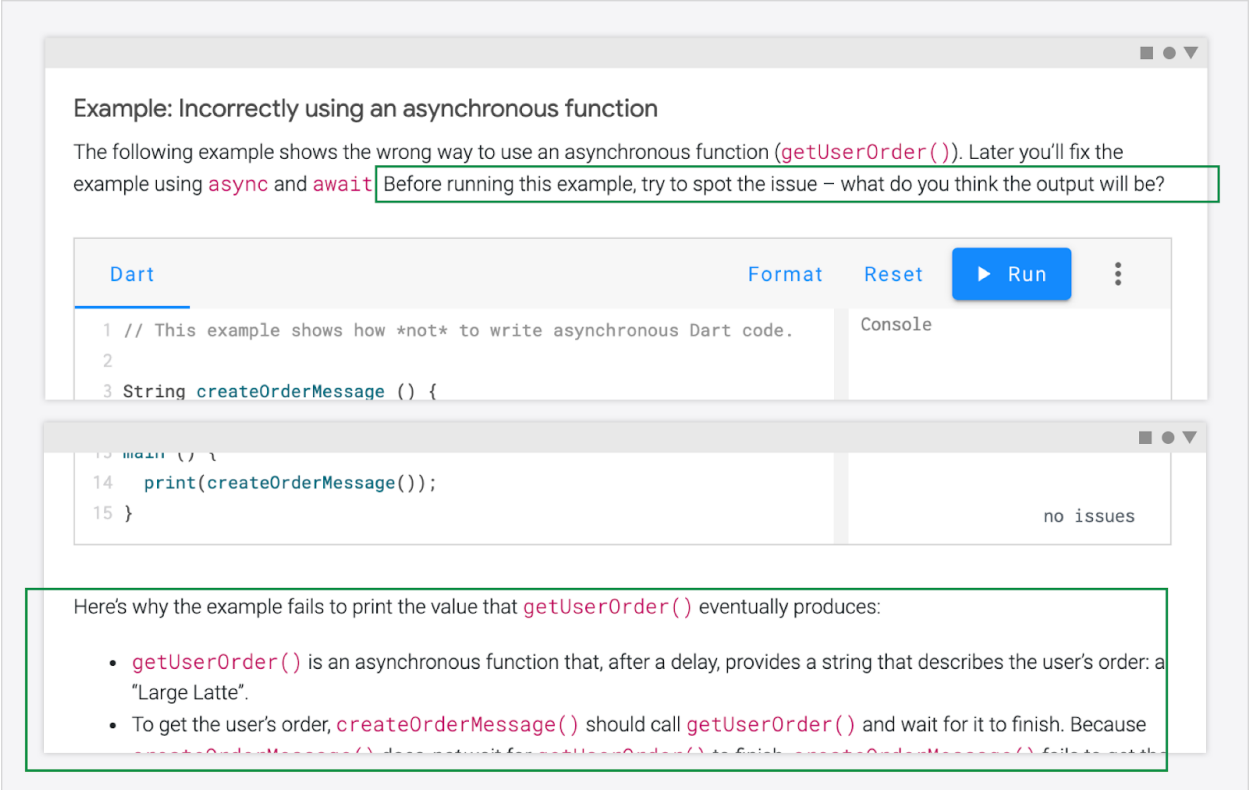
- Demonstrate how concepts work in action and show the use of features in concrete examples.
- Familiarize learners with the Dart syntax for `Future`, `async` and `await` keywords.
- Provide sample code used as a reference for hands-on coding exercises.

When you design a demo using DartPad, pay attention to the following things.

Facilitate reflection by explicitly telling learners what to do

Demos make important points about a concept, but to get those points across, we need to tell users what they should be looking for in the demo. Instead of unconsciously jumping into the output, explicit instructions encourage learners to think about what they thought would happen and what might not make sense to them.

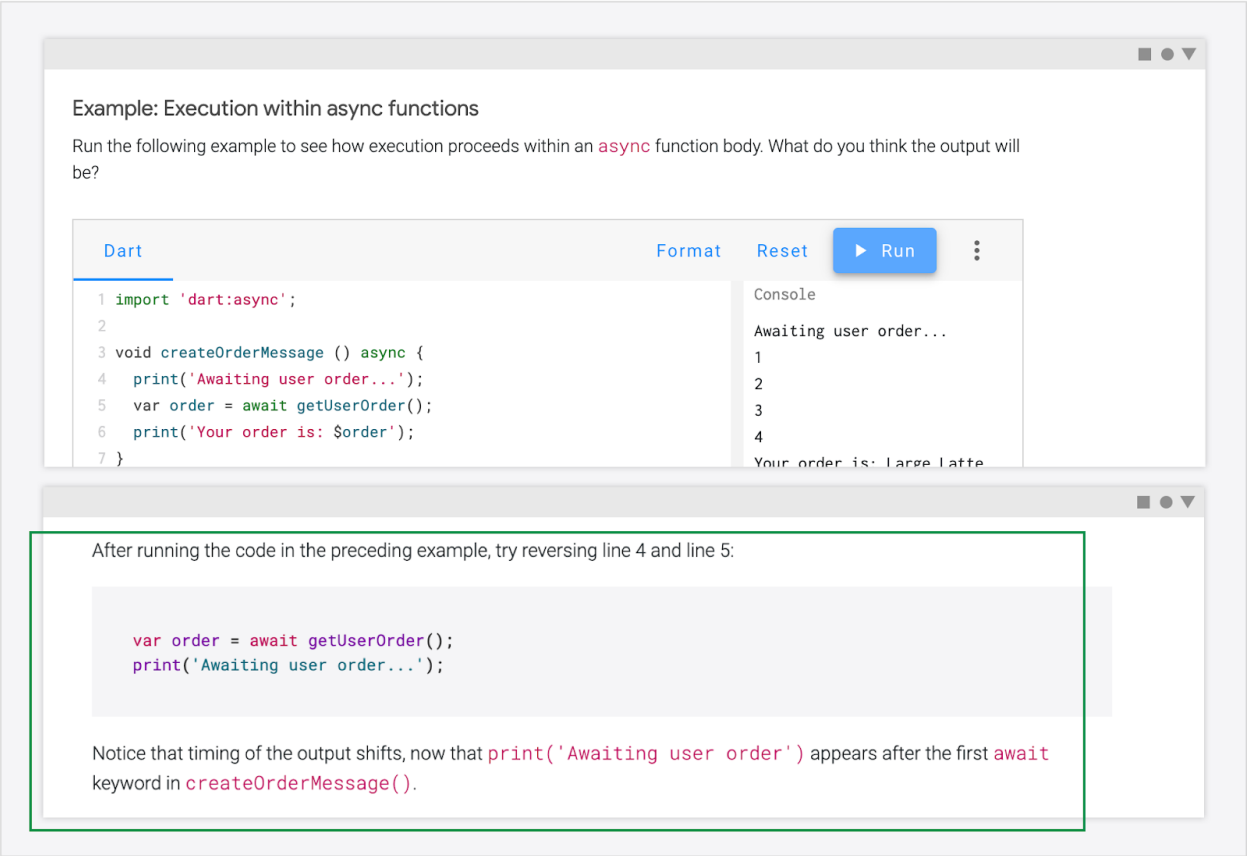
For example, the following demo presents learners with an example of incorrectly using an asynchronous function. Above the code snippet, there is a prompt: "*Before running this example, try to spot the issue – what do you think the output will be*?" Beneath the code example, there is a description of what the code example is about, how the code is executed, and an explanation of why the code fails to print the desired value.



A prompt to ask the user to consider the output of an example that incorrectly uses an asynchronous function, followed by an explanation.

Encourage learners to play with the code and observe the results

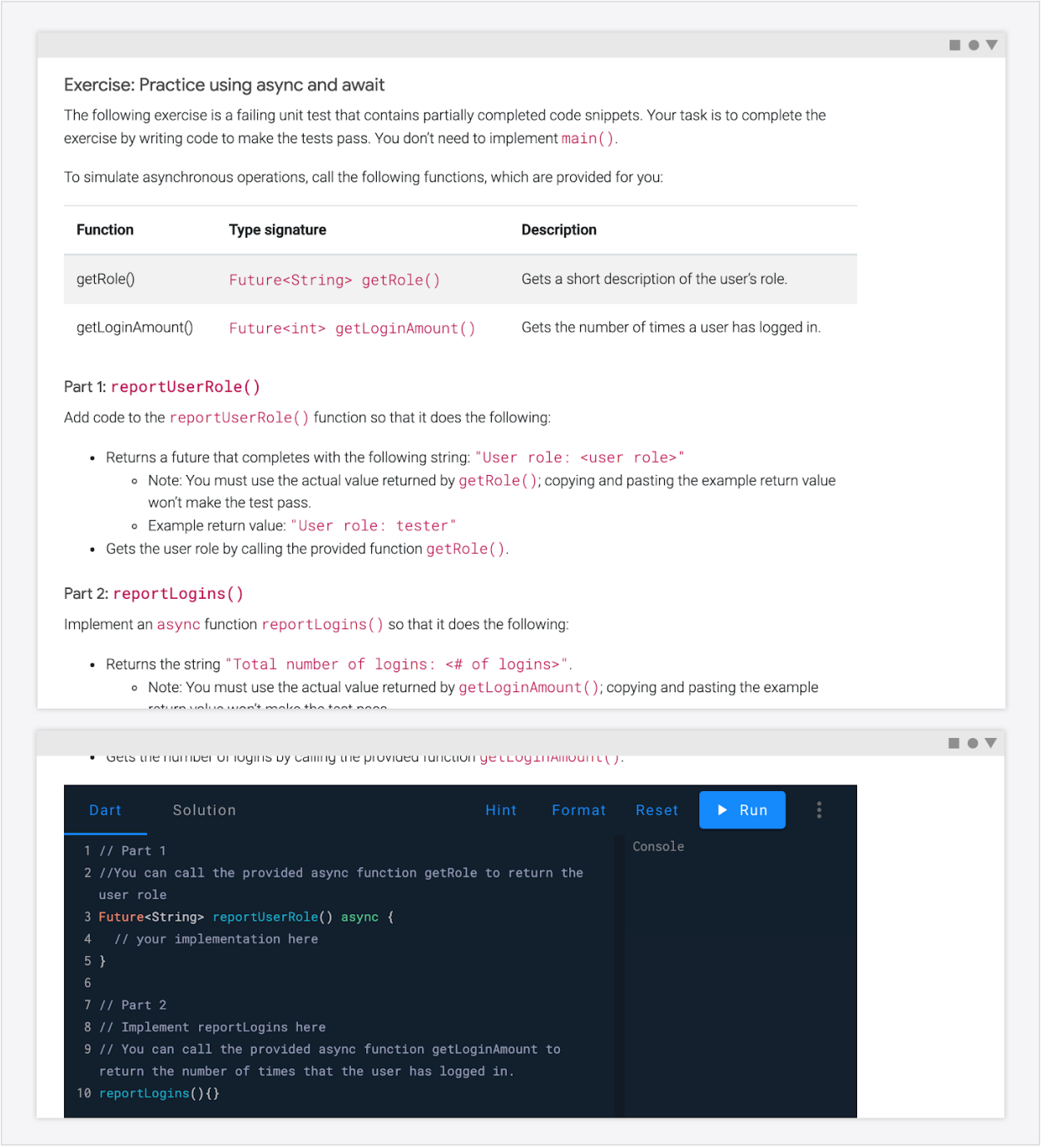
Interactive demos make hard-to-explain concepts concrete because they allow learners to actively explore and experiment with the code. The following example encourages learners to first think about how execution proceeds within an async function body. Next, learners are encouraged to reverse line 4 and line 5, and then observe the difference in the timing of the output. With this demo, learners can better understand the execution flow in asynchronous code.



A code example: abstract execution flow and instructions for changing code

II. Exercise

After the learner is exposed to the basic concepts and operations of asynchronous programming in Dart, the tutorial provides exercises to help them put this newly acquired knowledge into action. For example, the following exercise requires learners to implement two async functions, `reportUserRole()` and `reportLogins()`, using the `Future` class, the `async` keyword, and the `await` keyword. Learners have an opportunity to practice what they just learned from the demos.



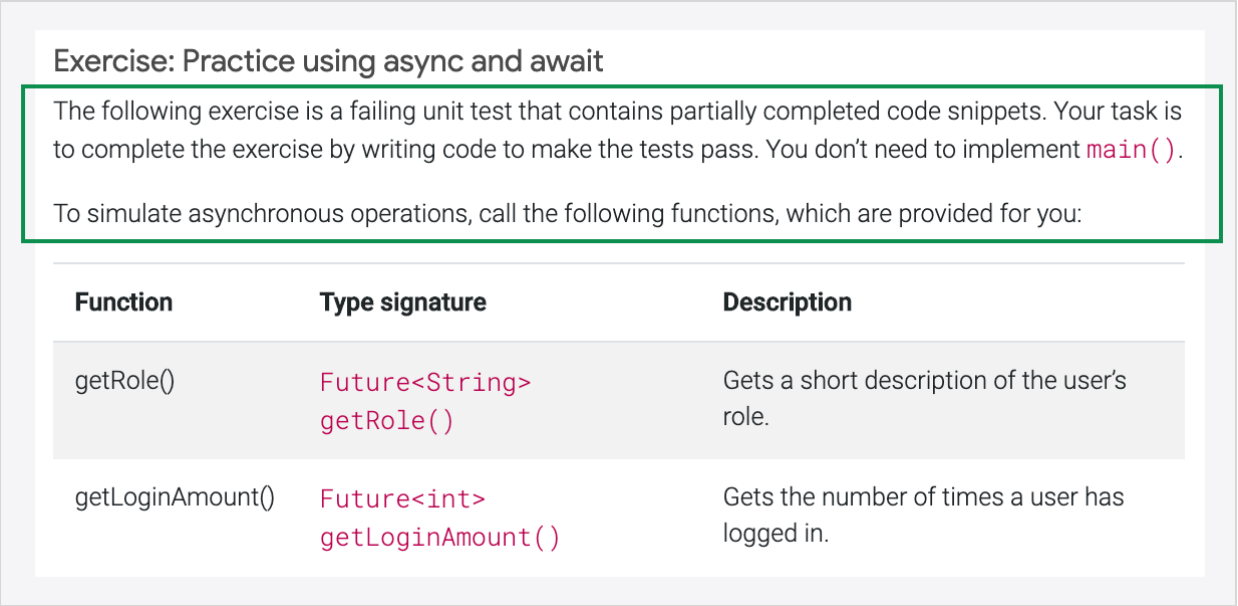
A coding exercise: implementing two async functions

When you design a coding exercise, there are a few things you need to pay attention to.

Describe the exercise workflow

Learners look for clear direction on what to expect next. When an exercise is first presented to the learner, provide a brief description of the exercise's workflow. In this case, it's important to point out that the goal is to modify the snippet, to make the unit test pass.

For example, the following exercise starts with an introduction: "*The following exercise is a failing unit test that contains partially completed code snippets. Your task is to complete the exercise by writing code to make the tests pass.*" In addition to explaining what the exercise is about, the author also clearly communicates that learners don't need to implement the hidden code that was provided, such as `main()` and two asynchronous functions, `getRole()` and `getLoginAmount()`.



Exercise: Practice using async and await

The following exercise is a failing unit test that contains partially completed code snippets. Your task is to complete the exercise by writing code to make the tests pass. You don't need to implement `main()`.

To simulate asynchronous operations, call the following functions, which are provided for you:

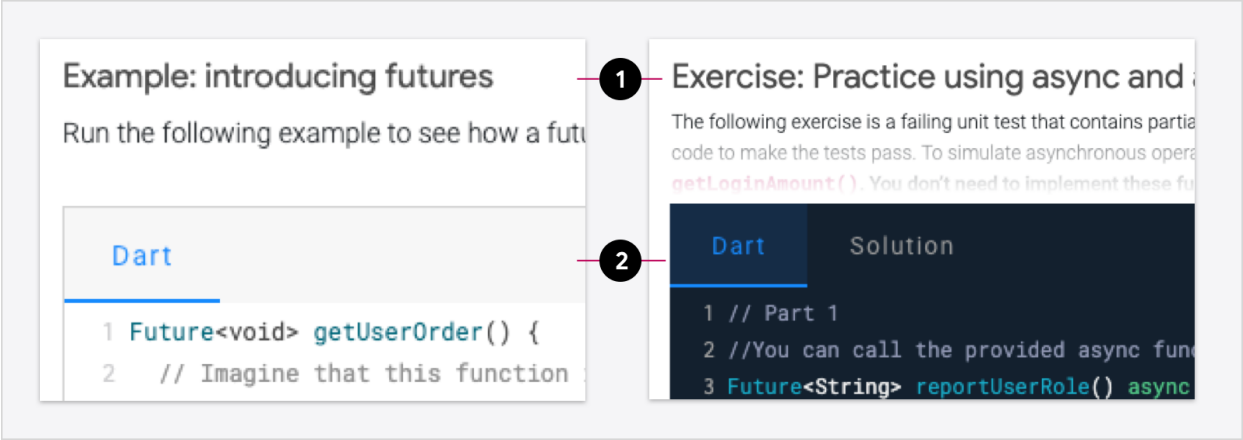
Function	Type signature	Description
getRole()	`Future<String> getRole()`	Gets a short description of the user's role.
getLoginAmount()	`Future<int> getLoginAmount()`	Gets the number of times a user has logged in.

Briefly explain what the exercise is about and what learners are supposed (or not supposed) to do.

Visually distinguish between demos and exercises

When DartPad is embedded in multiple places in a tutorial, a clear, visual distinction between demos and exercises help users quickly recognize the expected actions. When we used the same UIs for both demos and exercises, one of our study participants said, *"I wasn't sure whether I should just code something or I'm supposed to just run it to see it."*

In the published Futures codelab, all embedded DartPads are labeled with clear headings, such as "*Example: Introducing futures*" and "*Exercise: Practice using async and await.*" Also, we adopted the light DartPad theme for demos and the dark DartPad theme for exercises. The continuous improvement is tracked using this issue on GitHub.



Clearly label the titles, and use different themes for demos and exercises.

Steps toward greater confidence

Interactive tutorials provide hands-on practice so that learners can accumulate knowledge as they tackle more and more sophisticated problems. However, learners could get frustrated if the tutorial doesn't prepare them for bigger challenges. We learned four lessons from developing the Futures codelab.

First, including demos and exercises for each concept before the final quiz can provide a gradual progression that novices can follow. An earlier draft of this codelab had a demo for handling errors, but didn't have a corresponding exercise to practice the try-catch concept before the final quiz. One of our study participants who tried that version said, *"When you have to do something for the first time during the test, it doesn't feel good. Because I'm not confident that I'll get this part right."*

Second, exercises need to provide necessary scaffolds. When there are multiple tasks in an exercise, consider providing more support in the first task. In the following example, Part 1, `reportUserRole()`, has more pre-filled code snippets in the starting state, while in Part 2, `reportLogins()`, only the function name is offered.

Part 1 has more complete code snippets in the starting state than Part 2 does.

Third, learners appreciate that they can refer to code examples when working on exercises. As they get more familiar with the topic and syntax, they check the code examples less frequently. In the Futures codelab, all code examples and exercises are put on the same page. One of our study participants said, *"I liked how I could be typing here but then also refer back to the examples to see where I should put stuff."*

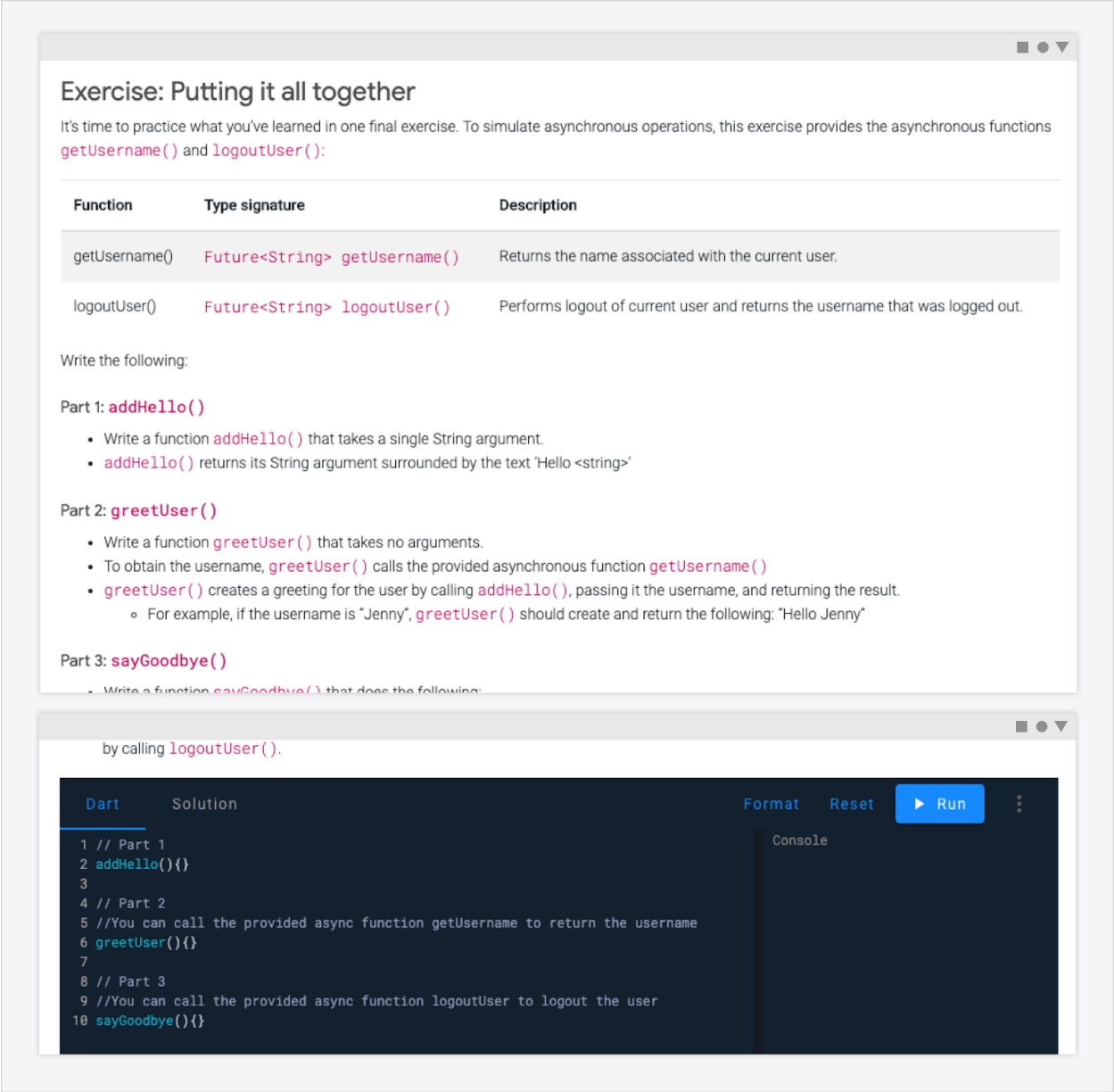
Last, demos and exercises (or quizzes) shouldn't be too similar, to avoid feeling redundant. Providing almost identical examples and exercises may confuse learners. Ultimately, they just copy and paste the code instead of practicing on their own. In our initial prototype, the demos and the first exercise used a similar context and function names, `getUserOrder()` and `reportChange()`. One of our study participants said, "*This exercise is slightly odd that they're looking for basically the exact same code as the example. I'm not sure what they're asking me to do."* We then improved this by changing the context of exercise to access control instead of ordering coffee.

III. Quiz

Before wrapping up, the tutorial provides a final quiz that covers all the concepts, to help learners apply everything that they learned in the tutorial to a new setting. For instance, the following quiz requires learners to implement three functions, including a non-async function, `addHello()`, and async functions, `greetUser()` and `sayGoodbye()`. Learners can practice when to use async functions, where to use the `Future` class, how to use the `async` and `await` keywords, and handling errors.

Even though this section is not explicitly labeled as a quiz, learners may still consider it as a final assessment. As one of our study participants said, *"Whenever you do like a tutorial and it's the*

final part, it's usually to me that means it's like a test of every single thing that you've written out and combining them.“



A coding quiz: implementing 3 functions about async functions, async/await keywords and handling errors

Minimal amount of help and scaffolds in a quiz

The problem in a quiz is usually a bit harder than the problem in an exercise and less guidance is provided. Compared to the previous “Exercise: Practice using async and await,” the partially completed code in the starting point for this quiz is minimized. Also, there is no **Hint** button in this quiz.

Final words

We solicited feedback about the tutorial using moderated UX studies and an embedded survey. Learners are satisfied overall with this interactive tutorial for the following reasons:

- Provides hands-on exercises without having to set up a development environment.
- Content is well-explained, straightforward, and easy to read.
- Gives just-in-time and just-enough help and feedback.

DartPad enables us to create effective tutorials that lower the learning curves, are easy to use and more engaging. If you write tutorials for Dart or Flutter, we encourage you to consider using DartPad to enhance your tutorial.

"Exercises are an excellent learning tool!!! Thank you!"

"I learn coding concepts the best when I have to think through the entire process versus just being handed the solution."

"I like the live coding parts are free form. As long as you get the return correct, being able to write how you write it is nice."