iTalanta High Potential Apprenticeship Program

Candidate Assessment – 29/11/2025

Thank you for your interest in the iTalanta high potential apprenticeship program role at Elewa and congratulations on making it to the assignment.

This assessment is designed to evaluate how you approach **real-world engineering challenges** you'll encounter in our work: balancing architecture, user experience, technical trade-offs, and business outcomes. The scenarios are based on a project we invested in in the past, but which was determined not commercially viable at the time and has since been open sourced.

The repository we'll study in this assignment can be found here: **https://github.com/italanta/kujali**

As part of this assignment, you will need to submit two parts:

1. You will be given some questions to submit in a form before the end of the assignment.
2. You will fork the repository and submit two pull requests **in your fork**[1] that correspond with the two coding assignments. In the form, please already submit the link to your fork. Ensure the fork is public or it won't be reviewed and you will be disqualified. Any commit made after the exercise time will be disregarded when we review. When creating the code, the app should not be runnable so don't lose time on trying to compile it beyond an `npm install`. We are reviewing your assignment for structural thinking, cleanliness and coding style more than compilation.

Estimated Time Commitment : 3 hours today. When is flexible but only **code commited** on your fork/PR **on or before 7:59pm 29/11/2026** will be reviewed.

Submission Format: Microsoft Form + GitHub PRs (repo linked in form)

[1] This means, please update your Pull Requests to remain inside of your own fork. For sake of clarity, if your branch is called **janedoe:kujali/refactor-signal-state**, your PR should **compare to janedoe/kujali:main** and **NOT** to italanta/kujali:main.

<u>Application Context</u>

Kujali is a cashflow management tool for start-ups and scale-ups. A core feature allows users to define budgets and then manage revenue realization through CRM, sales forecasting, invoicing, etc.

Kujali is built as a monorepo which is deployed to Angular, Firebase and Google Cloud. For data storage, Kujali uses Firestore with readers that stream data up to Postgres and BigQuery for BI & analysis.

In the root of the monorepo, you will find two important folders:

- **Apps**: In line with the use of a monorepo (https://monorepo.tools/), these are lightweight containers which roll up different features into a deployable application. The role of the apps is to configure cross-application behaviour and not to define features.

- **Libs**: This is the "body" of the app. Under features all the logic and features are contained.
  Libs has a few important subfolders of it's own:

Elements	- Reusable components (e.g. buttons, ...)
Features	- Actual logical frontend features
State	- Link between front and back
Model	- Cross-application models
Util	- Reusable libraries can be used across different client repos

Candidate Task

As a developer under the iTalanta HPAP program, your mission is to create a new feature applying the repo's architecture.

Part 1: Feature Implementation (Architectural Pattern Application - Backend)

Objective: This task evaluates your ability to understand and apply existing architectural patterns (CQRS), design clean interfaces, and structure code for a new feature within an Nx monorepo.

Background: The Kujali application uses a CQRS (Command Query Responsibility Segregation) pattern, supported by the @iote/cqrs and @ngfire/functions library (in utils). Your task is to implement the "Command" side of a new feature i.e. **"Add a Note to a Budget"**.

Your Tasks:

1. **Scaffold a New Library**

Create a new feature library to house this logic. From the root of the repository, run the following Nx command:

```
nx generate @nrwl/node:library budget-notes --directory=model/budgetting/notes
```

-- or (depending on NX version) --

```
nx generate @nx/node:library budget-notes --directory=model/budgetting/notes
```

This will create the necessary files and configuration under libs/features/budgetting/budget-notes/

2. **Define the Command and Handler:**

Inside the new library (libs/models/budgetting/budget-notes/src/lib/), create a domain folder.

Within this folder, create the following files and classes:

- add-note.command.ts

  Define a class AddNoteToBudgetCommand that encapsulates the data needed to add a note to a budget.

- add-note.handler.ts

Define a generic ICommandHandler<TCommand> interface with a single method: execute(command: TCommand): Promise<void>;.

- Define a class AddNoteToBudgetHandler that extends FunctionHandler<AddNoteToBudgetCommand, AddNoteToBudgetResult>.
- Implement the execute method in the handler. This method should perform basic validation on the command's data (e.g., check for empty content) and then call the repository's addNote method. It will need a data access service, for which you can use the getRepository in the handler's toolkit (which comes as part of execute payload).

Submission for Part 1:

- Create a new branch in your fork named feat/add-budget-note-command.
- Commit your new library and files to this branch. Use as many commits as you need. Remember a good commit is a single atomic functional change.
- Open a Pull Request in your forked repository titled: **"Feature: Add Note to Budget Command"**.

Don't forget!:

In the PR description:

- Briefly explain how you understand the backend development process, your design choices for the command, handler, and its dependencies.
- Also include how you would deploy the handler you built based on the current repo's architecture.
- What do you know about serverless deployment? How would this scale on a serverless architecture?

Part 2: Feature refactoring (Architecture refactoring - Frontend)

Objective: This task evaluates your ability to quickly adopt new technologies by focussing on the refactor of an existing component architecture from a traditional RxJS-based approach to use the modern, fine-grained reactivity of Angular Signals.

Background: In the Kujali application, there is a common pattern where a parent "container" component fetches data and passes it down to smaller "presentational" child components. We want to modernize by converting its state management to use Signals.

Find the following components that are to be refactored:

(libs/features/budgetting/budgets/src/lib):

- SelectBudgetPageComponent (Parent): Fetches and prepares the budget data.
- BudgetTableComponent (Child): A simple component that renders the data passed on to it.

Currently, this feature uses RxJS Observables and the async pipe to manage and propagate state changes. Your task is to refactor this entire component tree to use signal(), computed(), and signal-based inputs[2].

If you can also refactor both component to "zoneless" components, that wins you a big bonus.

Your Task:

1. SelectBudgetPageComponent (Parent)**:**
   - Remove the constructor dependencies in favour of inject()
   - Load the RxJs observable streams (overview$, sharedBudgets$, allBudgets$) into angular signals and use properly throughout the component
   - Refactor the subscribe functionality into a declarative, signal-based approach with proper use of effect() and computed()
   -
   - Update the template to pass signals to the child component

2. BudgetTableComponent (Child):
   - Update it to receive signal based inputs instead of observables
   - Remove the subscription logic and replace with signal-reactive patterns

[2] The key thing we are looking to see in this assignment is how fast you can read yourself into a new technology(ies) and how you approach refactoring to a different paradighm, from an imperative reactive style to a more declarative style of programming. At elewa, we are technology agnostic and may be coding one day in Flutter, the next in Angular, and the day after in Python. What we are looking for here therefore also applies to developers who have past experience in React/Vue/Svelte/... How fast can you express your fundamental knowledge in a new syntax/developer paradighm?

    - Bonus: Make HTML template zoneless and remove Angular CommonModule

Submission:

- Create a new branch in your fork named feat/refactor-signals-state.
- Commit both refactored component files.
- Open a Pull Request in your forked repository titled: **"Refactor: Modernize budget viewer state with Signals"**.

Don’t forget!:

In the PR description,

- Briefly explain how you went about this assignment and the key benefits of using Signals over the previous RxJS approach in this UI scenario.
- Explain how you understand the difference in coding styles between the two.
- Explain how signals and observables works and if there’s a fundamental difference between the two.
- Mention an example of another coding paradighm or style you’ve used in the past. What is better/worse about that one?

--

We look forward to seeing how you approach these challenges and wish you all the best with this challenge!

Team Elewa