Task overview

Overview

In this assessment, you are provided with:… will fill with comments as it goes

---> Implement basic navigation within the Flutter app based on the wireframe. - done
---> Create an architecture diagram and design documentation that explains how the backend and infrastructure would be structured to handle the user stories below. - below

---> I noticed some dissonance in between the ui_wireframe_design and design so decided to change a little and make more realistic (in my POW)

---> Use "user" login and blanc password to login to the app if you will sstart it.
---> also added I10n support

Architecture and design doc

rontend:

Flutter app for iOS and Android
Bottom navigation with Home, Sell, Map, and Wallet screens
Features: item listing, geo-tagging, transactions, and authentication
Uses Firebase Authentication and FCM (Cloud Messaging)

Backend:

.NET 8 Web API hosted on Azure (App Service or AKS) or others
Handles CRUD operations for items and transactions
Validates Firebase ID tokens for secure access
Stores data in Azure SQL/PostgreSQL
Optionally integrates with AMQP (RabbitMQ or EMQX) for messaging and future real-time features

Infrastructure:

Provisioned using Infrastructure as Code (IaC) tools
CI/CD pipelines automate testing and deployment
Prepared for horizontal and vertical scaling
Logging and monitoring handled via Azure Monitor for example

the diagram goes here

here is the link, description below
https://drive.google.com/file/d/1FXjS8-7rhKM1mvED5mmVy7kNJqhiTmRw/view?usp=drive_link

COMMENTS

Navigation Implementation Based on UI Wireframe

Home Screen – Shows list of items in a grid format
Buy Flow – Triggered via Buy button on both Home and Map screens, shows payment options dialog affects most of structures : balance lists of items etc
Sell Screen – Allows user to input item info + photo, and auto-fills location - affects the wallet
Map Screen – Shows pins for items + user, marker tap opens bottom sheet with item + Buy button

Wallet Screen – Shows balance, filterable item lists (for sale, sold, bought), transaction history
Conclusion: Basic navigation and screen flow fully follows the UI wireframe — Done.

asic App Logic & UI Interactions
- Buying items:
  - Can buy from both Home and Map
  - Buy dialog offers balance or card option (txt but what ever can be done with additional widget)
  - Balance deduction only on "pay with balance"
  - Adds to bought items list
  - Adds transaction
  - Item removed from available items
- Selling items:
  - Auth check enforced before opening Sell screen
  - Image picking from device
  - Auto location detection
  - New item appears on Home + Wallet screen
  - Location stored for Map
- Wallet:
  - Balance shown
  - View Details opens transaction list
  - Withdraw works with limit & user card selection
- Map screen:
  - Real user location (with permission)
  - Markers for items (with lat/lng)
  - Tap marker opens bottom sheet with image, des, price, buy
  - Selected marker color changes

Flutter Project Technical Practices
- Cubit + BLoC pattern used throughout (AuthCubit, MarketplaceCubit, WalletCubit, UserCubit)
- Clear separation of concerns: widgets vs cubits vs models
- UI is responsive (Grid, SafeArea, scrollable dialogs, keyboard handling)
- Async image loading supported (local file vs asset)
- Location handled safely and with permission prompts
- No critical crash/bug paths found (for the test app done in 10 hours)

User stories

would not repeat them all here but in UI covered 5 out of 5.

also a few things more like chat planned - BTW there is pattern description on the app chat page

TO BE CLEAR current logic lays fully on the front like balance check, transactions, filtering on load, etc etc it should also be done in the back end of course

TASKS

1. Flutter Navigation Implementation
- Navigation between Main Screens
  - Home (Buy) screen is default. As for the sell screen - i think its better descision to make it accessible via floating button as well as buy button is conlitely useless on the bot nav bar and it was moved to the item to buy.
  - Sell screen accessible via FAB.
  - Map and Wallet are navigable via buttons.

- Bottom Navigation Bar
  - Present and switches between major tabs.

Top Button Access (Map / Wallet)
Map - absolutely - added the search line as nice practice
wallet moved to the separate item in bot nav bar

Screen Placeholders or Full Views
Implemented views with logic — even beyond placeholders.

CI/CD & IaC Planning

Frontend (Flutter Mobile App)

1. Code Push / PR Git / or other Repos Trigger on main or feature branches.
2. Static Analysis flutter analyze Ensure code quality and linting.
3. Unit Tests flutter test Run Cubit and logic tests.
4. Build APK/AAB flutter build Generate Android/iOS artifacts.
5. Deploy App Distribution

we can use GitHub Actions or Azure DevOps Pipelines to automate steps. Secrets managed via GitHub Secrets or Azure Key Vault.

Backend (Web API + Serverless Functions)

1. Push / PR GitHub / Azure DevOps Trigger pipeline on push.
2. Build .NET SDK / Docker Build API or container image.
3. Test xUnit / NUnit Run unit & integration tests.
4. Deploy Azure CLI / ARM Deploy to App Service or Container App.

can be published and deployed via GitHub Actions or Azure Pipelines.

Infrastructure as Code

Tool of Choice: Terraform.
Terraform allows us to define cloud infrastructure in HCL and supports modular, environment-driven, and team-friendly workflows. We can also use Bicep or Pulumi, but Terraform has the widest community and multi-cloud support. just thoughts i have not touched the Bicep or Pulumi before.

Project Structure (terra)

```
infra/
  main.tf              # entry point, calls all modules
  variables.tf           # shared variables
  terraform.tfvars   # environment-specific config (e.g., dev/stage/prod)
  modules/
    app_service/        # App Service or Container App
    cosmos_db/
    storage/
    service_bus/
    monitoring/
```

Modules & Key Resources

App Service azurerm_app_service / azurerm_container_app Hosts .NET Web API in Docker container with autoscaling

Database azurerm_cosmosdb_account or azurerm_sql_database Stores listings, transactions, users and on

Blob Storage azurerm_storage_account + azurerm_storage_container Stores item images as well can be used the firebase easier to store moving our the load from the backend

Service Bus (optional) azurerm_servicebus_namespace + queue/topic Event-driven communication (purchases, notifications)

Monitoring azurerm_application_insights Logs, metrics, and tracing for backend services
D
NS / CDN (optional) azurerm_cdn_profile, azurerm_dns_zone If you want to expose API/images via custom domain or CDN

Firebase as External Dependency

Firebase Auth is not provisioned via Terraform, but our backend should validate ID tokens using public keys from Firebase.

Firebase Cloud Messaging is configured in the mobile app and can be extended via HTTPAPI or Admin SDK.

the other options not full replacement more like scaling

If Using AMQP (e.g., RabbitMQ, Azure Service Bus, EMQX).AMQP replaces Firebase Cloud Messaging (or complements it) for real-time, event-driven communication, especially chat, transactions, or notification fan-out.

How It Fits Into Architecture

The Flutter app publishes to Firebase or AMQP depending on scale & feature. Backend listens to AMQP queues, processes transactions (e.g., “user bought item”), and notifies users via FCM or SignalR.

With AMQP, we can add microservices like: Notification worker (push/email), Analytics processor and of couse chat broker.

SO Firebase for Auth and optionally FCM for basic push. AMQP for messaging, transactions, and backend communication where real-time throughput and scale matter.

Secrets Management

Azure Key Vault and azurerm_key_vault for: Firebase private keys (if needed for Admin SDK), DB connection strings, even for stripe if we would use that as likely google and apple gave the permission to use other providers

Terraform can reference secrets using data "azurerm_key_vault_secret".

Scaling Configs, App Service Autoscale example (very general)

```
resource "azurerm_monitor_autoscale_setting" "api_autoscale" {
  name                = "autoscale-app"
  resource_group_name = var.resource_group_name
  target_resource_id  = azurerm_app_service.api.id

  profile {
    name = "default"

    capacity {
      minimum = "1"
      maximum = "10"
      default = "1"
    }

    rule {
      metric_trigger {
        metric_name        = "CpuPercentage"
        metric_resource_id = azurerm_app_service.api.id
        operator           = "GreaterThan"
        statistic          = "Average"
```

```
      threshold         = 70
      time_aggregation  = "Average"
      time_grain        = "PT1M"
      time_window       = "PT5M"
    }

    scale_action {
      direction = "Increase"
      type      = "ChangeCount"
      value     = "1"
      cooldown  = "PT5M"
    }
  }
 }
}
```

Summary for CI/CD

Everything is provisioned declaratively via Terraform. Resources are grouped into logical modules for reusability and clarity. Secrets, config, and scaling are externalized and automatable. Environment promotion (dev → stage → prod) is handled with terraform.tfvars.

Summary and intro to another leveld

We use Terraform to define and provision our infrastructure — virtual machines, databases, storage, networking etc — as code. This is more about automation and reproducibility (same infrastructure in environments)

BUT for scalability we should use K8s to run and scale our app on top of it. So Pods automaticly, restarting on crash, scaling up and down depending on load (that users 100 and 10000 - this is definetely not constant) and updates without downtime.

For our case we can for example run some of backend modules (wallet, marketplace, messaging - its for sure) as separate Kubernetes deployments. That way, they can scale independently.

Example: Selling an Item
User presses “Sell Now” in the Flutter
Request goes to Marketplace API (running in Kubernetes)
API saves the item to the database (provisioned by Terraform)
Other request goes to Chat API (running in Kubernetes) if we need to inform some others who were chatting about that item or may be API that was watching for that

Kubernetes automatically scales this service if 1000 people are selling at the same time etc.

Example K8s Architecture you will find via the link
https://drive.google.com/file/d/1ITfkKTPmk5YEwGy9gLoBCtjrx0ihC70x/view?usp=drive_link

So what kubernetes for.

We have: a dlutter frontend using Firebase Auth + backend with buy/sell logic, wallet, transactions + potential real-time messaging via AMQP or SignalR + support 10,000+ concurrent users and grouring to 1 mln

Kuber will give us
Auto scaling pods based on CPU, memory, or custom metrics (e.g., incoming messages).
microservices (wallet, marketplace, chat others) are independently deployed and balanced updates without downtime

restarts
horizontal hcaling pods based on traffic/load

To run that we can use
Azure AKS or GCP GKE.

That was huge)

Ahh the link to the video also __

https://drive.google.com/file/d/1B2DcDoJCGk49SAC_b-iN0SbSSmAQHLos/view?usp=drivesdk