**Enterprise-Level Food Delivery System: Automated Testing Roadmap**

**Overview & Objectives**

* **Application Under Test:**  
  A live food delivery mobile app (Flutter-based) that includes user authentication, browsing restaurants, viewing menus, adding items to a cart, order placement, payment, and order tracking.
* **Testing Goals:**
  + Validate the mobile app on both Android and iOS.
  + Build a modular, maintainable test framework using Appium with JavaScript (WebdriverIO) and Mocha.
  + Use the **Page Object Model (POM)** to organize tests by screen (Login, Home, Restaurant Detail, Cart, etc.).
  + Utilize external test data files (JSON/CSV) for both valid and invalid scenarios.
  + Integrate hooks to capture logs and screenshots on test failures.
  + Cover unit tests, integration tests, and load tests (with Artillery) and integrate reporting.
  + Integrate with a CI/CD system for continuous testing.
* **Timeline:**  
  **1 Month (4 Weeks)**
  + **Week 1:** Environment setup, basic Appium fundamentals, and project initialization.
  + **Week 2:** Implement modular tests using the POM and external test data.
  + **Week 3:** Build advanced test scenarios (end-to-end workflows, hooks, reporting) and CI/CD integration.
  + **Week 4:** Load testing, enterprise-level suite integration, documentation, and team review.

**Phase 1: Environment Setup & Fundamentals (Week 1)**

**1.1 Install Prerequisites**

* **Java Development Kit (JDK 11+):**  
  Download and install from the [Oracle website](https://www.oracle.com/java/technologies/javase-downloads.html) or [OpenJDK](https://openjdk.java.net/).  
  *Why?* Android tools and Appium depend on Java.
* **Node.js & NPM:**  
  Download and install the LTS version from [Node.js](https://nodejs.org/). Verify installation:
* node -v
* npm -v

*Why?* Node.js is the runtime for the Appium server and executing JavaScript tests.

* **Appium & Appium Doctor:**  
  Install globally:
* npm install -g appium
* npm install -g appium-doctor
* appium-doctor

*Why?* Appium is the automation engine; Appium Doctor verifies that all dependencies are correctly installed.

* **Platform-Specific Tools:**
  + **Android Studio & SDK:** Install from [Android Studio](https://developer.android.com/studio) and set environment variables (e.g., ANDROID\_HOME and update your PATH to include platform-tools and emulator).
  + **Xcode (for iOS):** Install via the Mac App Store and configure simulators.
* **Flutter & Appium Flutter Driver:**  
  Install Flutter from [flutter.dev](https://flutter.dev/docs/get-started/install) and then install the Flutter driver:
* npm install -g appium-flutter-driver

**1.2 Create a New Node.js Project**

1. **Initialize Project Directory:**
2. mkdir food-delivery-tests
3. cd food-delivery-tests
4. npm init -y

*Explanation:* Creates the project folder and initializes it with default settings.

1. **Install Dependencies:**
2. npm install appium mocha chai webdriverio
3. npm install mochawesome --save-dev

*Explanation:*

* + **Appium:** Core automation engine.
  + **Mocha:** Test framework to structure test suites and test cases.
  + **Chai:** Assertion library providing the expect syntax.
  + **WebdriverIO:** JavaScript client to interact with Appium.
  + **Mochawesome:** Reporting tool that generates interactive HTML reports.

1. **Project Structure Overview:**
2. food-delivery-tests/
3. ├── helpers/
4. │ ├── utils.js
5. │ └── pageObjects/
6. │ ├── loginPage.js
7. │ ├── homePage.js
8. │ ├── restaurantPage.js
9. │ └── cartPage.js
10. ├── tests/
11. │ ├── data/
12. │ │ ├── loginData.json
13. │ │ └── orderData.json
14. │ ├── unit/
15. │ │ └── inputValidationTest.js
16. │ └── integration/
17. │ ├── loginFlowTest.js
18. │ └── orderPlacementTest.js
19. ├── baseTest.js
20. ├── test-config.js
21. └── package.json

**Phase 2: Implementing a Modular Test Framework Using POM (Week 2)**

**2.1 Configuration & Base Test Setup**

1. **test-config.js:**  
   Create a file that holds device capabilities for Android and iOS.
2. // test-config.js
3. module.exports = {
4. androidCaps: {
5. platformName: "Android",
6. deviceName: "emulator-5554",
7. app: "/path/to/your/flutter-app.apk",
8. automationName: "Flutter",
9. noReset: true
10. },
11. iosCaps: {
12. platformName: "iOS",
13. deviceName: "iPhone Simulator",
14. platformVersion: "14.0",
15. app: "/path/to/your/FlutterApp.app",
16. automationName: "Flutter",
17. noReset: true
18. }
19. };

*Explanation:* This file centralizes configuration, making it easy to update capabilities without modifying test code.

1. **baseTest.js:**  
   Create a base class to initialize and tear down the Appium session.
2. // baseTest.js
3. const wdio = require("webdriverio");
4. const { androidCaps, iosCaps } = require("./test-config");
5. class BaseTest {
6. async init(platform = "android") {
7. const capabilities = platform === "ios" ? iosCaps : androidCaps;
8. this.driver = await wdio.remote({
9. path: "/wd/hub",
10. port: 4723,
11. capabilities
12. });
13. }
14. async tearDown() {
15. if (this.driver) {
16. await this.driver.deleteSession();
17. }
18. }
19. }
20. module.exports = BaseTest;

*Explanation:*

* + **wdio.remote(...)** creates a new Appium session using specified capabilities.
  + **tearDown()** ensures a clean session closure.

**2.2 Implementing Page Object Model (POM) Classes**

Create a dedicated folder /helpers/pageObjects/ and define separate classes for each key screen.

1. **Login Page (loginPage.js):**
2. // helpers/pageObjects/loginPage.js
3. class LoginPage {
4. constructor(driver) {
5. this.driver = driver;
6. }
7. // Define selectors using accessibility IDs
8. get usernameInput() { return this.driver.$("~username\_input"); }
9. get passwordInput() { return this.driver.$("~password\_input"); }
10. get loginButton() { return this.driver.$("~login\_button"); }
11. get errorMessage() { return this.driver.$("~error\_message"); }
12. // Methods to perform actions
13. async login(username, password) {
14. await (await this.usernameInput).setValue(username);
15. await (await this.passwordInput).setValue(password);
16. await (await this.loginButton).click();
17. }
18. }
19. module.exports = LoginPage;

*Explanation:*

* + **Selectors as getters:** Allow lazy evaluation of elements.
  + **Action method (login):** Encapsulates user interactions for logging in.

1. **Home Page (homePage.js):**
2. // helpers/pageObjects/homePage.js
3. class HomePage {
4. constructor(driver) {
5. this.driver = driver;
6. }
7. // Example element selectors for home screen
8. get welcomeBanner() { return this.driver.$("~welcome\_banner"); }
9. get searchBar() { return this.driver.$("~search\_bar"); }
10. // Method to verify the home screen is displayed
11. async isDisplayed() {
12. return await (await this.welcomeBanner).isDisplayed();
13. }
14. }
15. module.exports = HomePage;

*Explanation:*

* + Provides a method to verify that key elements are present on the home screen.

1. **Restaurant Detail Page (restaurantPage.js):**
2. // helpers/pageObjects/restaurantPage.js
3. class RestaurantPage {
4. constructor(driver) {
5. this.driver = driver;
6. }
7. // Selectors for restaurant details
8. get restaurantName() { return this.driver.$("~restaurant\_name"); }
9. get menuList() { return this.driver.$("~menu\_list"); }
10. async getRestaurantName() {
11. return await (await this.restaurantName).getText();
12. }
13. }
14. module.exports = RestaurantPage;

*Explanation:*

* + Encapsulates methods to interact with restaurant details, such as retrieving the restaurant name.

1. **Cart Page (cartPage.js):**
2. // helpers/pageObjects/cartPage.js
3. class CartPage {
4. constructor(driver) {
5. this.driver = driver;
6. }
7. // Selectors for cart elements
8. get cartItems() { return this.driver.$$("~cart\_item"); }
9. get checkoutButton() { return this.driver.$("~checkout\_button"); }
10. async getCartItemCount() {
11. return (await this.cartItems).length;
12. }
13. async checkout() {
14. await (await this.checkoutButton).click();
15. }
16. }
17. module.exports = CartPage;

*Explanation:*

* + Provides methods to validate the cart contents and trigger the checkout process.

**2.3 External Test Data**

Place your test data in JSON (or CSV) format under /tests/data/.

1. **Login Data (loginData.json):**
2. {
3. "valid": {
4. "username": "testUser",
5. "password": "password123"
6. },
7. "invalid": {
8. "username": "badUser",
9. "password": "wrongPassword"
10. }
11. }

*Explanation:*

* + Enables data-driven tests by separating valid and invalid login scenarios.

1. **Order Data (orderData.json):**  
   You might include details such as selected restaurant, items, and promo codes.
2. {
3. "order1": {
4. "restaurant": "Pizzeria",
5. "items": ["Margherita Pizza", "Garlic Bread"],
6. "promoCode": "SAVE10",
7. "invalidPromo": "BADCODE"
8. }
9. }

**Phase 3: Writing Tests with Hooks, Reporting, and CI/CD (Week 3)**

**3.1 Writing Integration Test Cases Using POM**

1. **Login Flow Test (tests/integration/loginFlowTest.js):**
2. // tests/integration/loginFlowTest.js
3. const { expect } = require("chai");
4. const BaseTest = require("../../baseTest");
5. const LoginPage = require("../../helpers/pageObjects/loginPage");
6. const HomePage = require("../../helpers/pageObjects/homePage");
7. const loginData = require("../data/loginData.json");
8. describe("Integration Test: Login Flow", function () {
9. let test, loginPage, homePage;
10. before(async function () {
11. test = new BaseTest();
12. await test.init("android");
13. loginPage = new LoginPage(test.driver);
14. homePage = new HomePage(test.driver);
15. });
16. it("should login successfully with valid credentials", async function () {
17. await loginPage.login(loginData.valid.username, loginData.valid.password);
18. const displayed = await homePage.isDisplayed();
19. expect(displayed).to.be.true;
20. });
21. it("should not login with invalid credentials", async function () {
22. await loginPage.login(loginData.invalid.username, loginData.invalid.password);
23. const errorText = await (await loginPage.errorMessage).getText();
24. expect(errorText).to.equal("Invalid credentials");
25. });
26. after(async function () {
27. await test.tearDown();
28. });
29. });

*Explanation:*

* + **POM Usage:** Utilizes page objects for login and home screens.
  + **Data-Driven:** Pulls test data from an external JSON file.
  + **Assertions:** Validates outcomes for both valid and invalid login attempts.

1. **Order Placement Test (tests/integration/orderPlacementTest.js):**
2. // tests/integration/orderPlacementTest.js
3. const { expect } = require("chai");
4. const BaseTest = require("../../baseTest");
5. const HomePage = require("../../helpers/pageObjects/homePage");
6. const RestaurantPage = require("../../helpers/pageObjects/restaurantPage");
7. const CartPage = require("../../helpers/pageObjects/cartPage");
8. const orderData = require("../data/orderData.json");
9. describe("Integration Test: Order Placement Flow", function () {
10. let test, homePage, restaurantPage, cartPage;
11. before(async function () {
12. test = new BaseTest();
13. await test.init("android");
14. homePage = new HomePage(test.driver);
15. restaurantPage = new RestaurantPage(test.driver);
16. cartPage = new CartPage(test.driver);
17. });
18. it("should select a restaurant and place an order", async function () {
19. // Assume that tapping on a restaurant navigates to the restaurant page.
20. const restaurantElement = await test.driver.$(`~restaurant\_${orderData.order1.restaurant}`);
21. await restaurantElement.click();
22. // Validate restaurant details.
23. const restaurantName = await restaurantPage.getRestaurantName();
24. expect(restaurantName).to.include(orderData.order1.restaurant);
25. // Add items to the cart (pseudo-code; replace with actual selectors/actions)
26. // For example, click on each item in the menu.
27. // await restaurantPage.addItem(orderData.order1.items[0]);
28. // await restaurantPage.addItem(orderData.order1.items[1]);
29. // Navigate to Cart and verify items
30. const cartButton = await test.driver.$("~cart\_button");
31. await cartButton.click();
32. const itemCount = await cartPage.getCartItemCount();
33. expect(itemCount).to.be.above(0);
34. // Optionally: Apply promo code and validate
35. // const promoInput = await test.driver.$("~promo\_input");
36. // await promoInput.setValue(orderData.order1.promoCode);
37. // const applyPromo = await test.driver.$("~apply\_promo");
38. // await applyPromo.click();
39. // Proceed to checkout
40. await cartPage.checkout();
41. // Validate that checkout succeeded by verifying an element on the checkout screen.
42. });
43. after(async function () {
44. await test.tearDown();
45. });
46. });

*Explanation:*

* + **Complex Workflow:** Simulates selecting a restaurant, adding items, and checking out.
  + **External Data:** Uses order details from orderData.json.

**3.2 Hooks for Logging & Screenshots**

Implement global Mocha hooks (in a separate file or within your test files) to capture logs and screenshots on failures.

1. **Global Hook Example (hooks.js):**
2. // hooks.js
3. const fs = require("fs");
4. const path = require("path");
5. module.exports = function () {
6. // After each test, check if it failed and capture a screenshot
7. afterEach(async function () {
8. if (this.currentTest.state === "failed" && this.test.ctx && this.test.ctx.driver) {
9. const screenshotDir = path.resolve(\_\_dirname, "screenshots");
10. if (!fs.existsSync(screenshotDir)) {
11. fs.mkdirSync(screenshotDir);
12. }
13. const screenshotPath = path.join(
14. screenshotDir,
15. `${this.currentTest.fullTitle().replace(/\s+/g, "\_")}.png`
16. );
17. await this.test.ctx.driver.saveScreenshot(screenshotPath);
18. console.log(`Screenshot saved: ${screenshotPath}`);
19. }
20. });
21. };

*Usage:*

* + Include this hook file in your Mocha configuration or import it at the beginning of your test suite.
  + **What It Does:** Automatically captures a screenshot for any failing test and logs the file path.

1. **Configure Mocha to Use Hooks:**  
   In your Mocha setup (for example, in a test helper file), include:
2. // testSetup.js
3. require("./hooks")();

Then, add this file to your Mocha command via the --file flag or import it in your test files.

**3.3 Reporting with Mochawesome**

1. **Configure Package.json Scripts:**
2. {
3. "scripts": {
4. "test": "mocha --reporter mochawesome --reporter-options reportDir=./mochawesome-report,reportFilename=index"
5. }
6. }
7. **Run Tests:**
8. npm test

*Explanation:*  
Mochawesome generates interactive HTML reports that you can use to analyze test outcomes, complete with screenshots from failures if available.

**3.4 CI/CD Integration**

Integrate with your CI/CD system (e.g., GitHub Actions, Jenkins, or GitLab CI) to automatically run tests on code pushes.

**Example GitHub Actions Workflow (.github/workflows/test.yml):**

name: Food Delivery Appium Tests

on:

push:

branches:

- main

jobs:

test:

runs-on: macos-latest # Use macos-latest for iOS tests or ubuntu-latest for Android-only

steps:

- name: Checkout Code

uses: actions/checkout@v3

- name: Set up Node.js

uses: actions/setup-node@v3

with:

node-version: '16'

- name: Install Dependencies

run: npm install

- name: Run Tests

run: npm test

*Explanation:*  
This workflow checks out your code, sets up Node.js, installs dependencies, and then runs your tests. Adjust the runner (macos-latest vs. ubuntu-latest) based on the platform you need.

**Phase 4: Load Testing and Enterprise Suite (Week 4)**

**4.1 Load Testing with Artillery**

1. **Install Artillery Globally:**
2. npm install -g artillery
3. **Create a Load Test Script (loadTest.yml):**
4. config:
5. target: "http://localhost:4723"
6. phases:
7. - duration: 60
8. arrivalRate: 5
9. scenarios:
10. - flow:
11. - post:
12. url: "/wd/hub/session"
13. json:
14. desiredCapabilities:
15. platformName: "Android"
16. app: "/path/to/your/flutter-app.apk"
17. automationName: "Flutter"
18. **Run Load Test:**
19. artillery run loadTest.yml

*Explanation:*  
This simulates multiple concurrent sessions (or actions) to stress test your application under load.

**4.2 Comprehensive Enterprise Test Suite**

* **Scope:**  
  Build an end-to-end test suite that covers:
  + User authentication (sign up, login, password recovery)
  + Browsing restaurants and viewing menus
  + Adding items to a cart and placing an order
  + Payment processing (with valid/invalid test cases)
  + Order tracking (real-time updates, notifications)
* **Best Practices:**
  + Reuse POM classes and external test data.
  + Use Mocha hooks for consistent logging and screenshot capture.
  + Write tests that are granular (unit) and then combine them into integration tests.
  + Document the test suite structure and instructions for adding new tests.

**Final Documentation & Maintenance Guidelines**

**Documentation:**

* **Project Wiki/Readme:**  
  Document the following:
  + **Environment Setup:** Installation steps for JDK, Node.js, Appium, Android Studio/Xcode, Flutter.
  + **Project Structure:** Describe the folder layout (helpers, tests, data, etc.).
  + **Writing New Tests:** Guidelines on how to use the Page Object Model, how to add test data files, and how to write new test cases.
  + **Running Tests:** Commands to run unit tests, integration tests, and load tests.
  + **CI/CD Instructions:** How the CI/CD pipeline works and how to interpret reports.
  + **Debugging Tips:** How to check Appium logs and locate screenshots for failed tests.

**Maintenance Guidelines:**

* **Modularity:**  
  Each page object and helper should be independent and reusable across tests.
* **Data Management:**  
  Keep test data external (JSON/CSV) so that changes in test cases do not require code modifications.
* **Version Control & CI/CD:**  
  Integrate with version control (Git) and ensure tests run automatically on each push.
* **Periodic Reviews:**  
  Schedule regular reviews to refactor tests, update selectors (if the UI changes), and improve coverage.
* **Team Training:**  
  Organize walkthroughs of the framework, discuss best practices, and update documentation as needed.

**Roadmap Timeline Summary**

| **Week** | **Focus Area** | **Key Deliverables** |
| --- | --- | --- |
| **1** | Environment Setup & Fundamentals | Install JDK, Node.js, Appium, Android Studio/Xcode, Flutter, create project structure, run basic "Hello World" tests on Android and iOS. |
| **2** | Modular Framework & Page Object Model Implementation | Create configuration files, baseTest.js, and POM classes for login, home, restaurant detail, and cart; add external test data files. |
| **3** | Integration, Hooks, Reporting, & CI/CD | Write integration tests for login and order placement using external data; implement hooks for screenshots on failure; integrate Mochawesome reporting; set up CI/CD. |
| **4** | Load Testing & Enterprise-Level Test Suite | Create load test scripts with Artillery; build a comprehensive test suite covering all major workflows; finalize documentation and team training. |

By following this detailed roadmap, your testing team will be well-equipped to build, execute, and maintain a robust automated test framework for your enterprise-level food delivery system. The use of modular POM classes, external data files, and hooks for logging and screenshots ensures that your tests remain maintainable, scalable, and easy to debug. This structure not only facilitates rapid development within a month but also provides a solid foundation for continuous integration and delivery.