**Question 1**

You've been contracted to build the core mobile experience for "GadgetGrove," an online retailer specializing in a curated selection of high-tech gadgets. The primary goal is to create a sleek browsing interface and a seamless "Add to Cart" and checkout simulation process.

The first step is to model the product data and display it in a compelling, grid-based catalog on the home screen.

1. In a new Flutter project named gadget\_grove, create a Product model class in lib/models/product.dart. It must contain: id (String), name (String), brand (String), price (double), imageUrl (String), and inStock (boolean). Create a corresponding mock data file in lib/data/ with a list of at least sixunique electronic gadgets.
2. On your HomePage, display the products from your mock data source using a GridView.builder. The grid must have two columns with a small amount of spacing between them. The UI should display a prominent "Top Picks" Text widget above the grid.
3. Create a reusable ProductTile widget for the grid. This widget must display the product's image within a Card. Below the image, show the product's name and brand. In the bottom-right corner of the card, overlay a small, circular Container that is green if inStock is true and grey if it's false, acting as a stock indicator. Use a Stack to achieve this overlay.

Users need to be able to view detailed information about a product and select options before adding it to a conceptual cart.

* 1. Implement go\_router to handle navigation from the ProductTile on the HomePage to a ProductDetailsPage. The route must be '/product/:id'.
  2. On the ProductDetailsPage, display the product's image in a large hero section at the top. Below, show the name, brand, and a longer, hard-coded description. Convert this page to a StatefulWidget.
  3. Add a section for user interaction on the ProductDetailsPage. It should contain a "Quantity" TextFormField (managed by a TextEditingController) that only accepts numbers and defaults to "1". Next to it, include an "Add to Cart" ElevatedButton. When pressed, it should print a confirmation message to the console, like: "Added 'Product Name' (Quantity: X) to cart."

To complete the user journey, you need to build a form that simulates the first step of a checkout process.

1. Create a CheckoutPage (/checkout) accessible from a button on the (not-yet-functional) cart page. This page must contain a Form with TextFormFields for "Full Name," "Shipping Address," and "Phone Number."
2. Implement robust validation. "Full Name" must contain at least two words. "Shipping Address" must be at least 15 characters long. "Phone Number" must not be empty and should only contain numbers and basic punctuation (e.g., +, -, ()).
3. Add a Checkbox to the form with the label "Use a different billing address." If this checkbox is **not** ticked, the form submission is complete. If it **is** ticked, a new TextFormField for "Billing Address" must appear below it. The "Place Order" button at the bottom should only be enabled once the entire form (including the conditional billing address field) is valid.

**Question 2**

You are upgrading a simple to-do list app, "TaskFlow," into a more powerful project management tool. Users need to organize tasks into projects, set deadlines, and attach files or notes.

The app's core structure must be refactored to handle a two-level hierarchy: Projects, which contain Tasks.

1. Create two model classes: Project (id, name, colorHex) and Task (id, title, description, dueDate, isCompleted). In your mock data file, create a List<Project> and a separate List<Task>, ensuring each Task has a projectId property that links it to a Project.
2. The app's HomePage should now display a list of Project cards. Each ProjectCard should be a ListTile wrapped in a Card, with a Container colored by the project's colorHex as the leading widget and the project name as the title.
3. Set up go\_router. When a ProjectCard is tapped, it should navigate to a ProjectDetailsPage using the route '/project/:projectId'.

The ProjectDetailsPage must display only the tasks associated with the selected project.

1. On the ProjectDetailsPage, receive the projectId from the router. Filter your global mockTasks list to get only the tasks whose projectId matches the one from the route.
2. Display the filtered tasks in a ListView.builder. Each task should be represented by a custom TaskTile widget. The TaskTile must show the task title and its dueDate (formatted as "Due: MMM dd, yyyy"). On the left, it should have a Checkbox to toggle its completion status (for this part, the state change can be local to the TaskTile widget and won't persist).
3. Add a FloatingActionButton to the ProjectDetailsPage. When tapped, it should navigate to an AddTaskPage, passing the current projectId in the route (e.g., /project/:projectId/add-task).

The form for adding a new task needs to be comprehensive, including the ability to attach a document.

1. Create the AddTaskPage. It must be a StatefulWidget containing a Form with TextFormFields for the "Title" and "Description." It must also include a "Deadline" TextFormField.
2. Make the "Deadline" field interactive. When the user taps this field, it should show a native date picker using showDatePicker(). Upon selection, the chosen date should be formatted and populated into the field's TextEditingController.
3. Add a button labeled "Attach File." Use the file\_picker package (a more versatile alternative to image\_picker) to allow the user to select any type of document (PDF, DOCX, etc.). Below the button, display the name of the selected file if one has been chosen. The "Save Task" button should validate the form and print all data, including the path of the attached file, to the console.

You are building the MVP (Minimum Viable Product) for "PhotoSphere," a photo-sharing social network. The core features for this build are the main feed, a user profile page, and the photo upload process.The central experience is an infinitely scrolling feed of photo posts.

1. Create two models: User (userId, username, avatarUrl) and Post (postId, userId, imageUrl, caption, likes). In your mock data files, create a list of users and a list of posts, linking them with userId.
2. Build a PostCard widget. This is a complex component. It must have a header Row showing the user's avatar (CircleAvatar) and username. Below that, display the large post image. Below the image, show another Row with action buttons (like, comment, share icons). Finally, at the bottom, display the post caption.
3. On the FeedPage (/), use a ListView.builder to display a list of all PostCards from your mock data. The username in the header of each PostCard must be tappable, navigating to that user's profile page via the route '/profile/:userId'.

This page should give a snapshot of a user, including a grid of all the photos they have posted.

1. Create the ProfilePage. It will receive a userId from the router. Use this ID to find the correct User object from your mock data.
2. The top of the ProfilePage must be a header section that displays the user's avatarUrl in a large CircleAvatar, their username, and some static metric counters (e.g., "150 Posts," "1.2k Followers").
3. Below the header, filter the global mockPosts list to get only the posts where the userId matches the current profile. Display these post images (and only the images) in a GridView.builder with three columns.

The most critical flow is allowing users to create a new post.

1. Create an UploadPage (/upload) accessible from a camera icon in the AppBar. This page must first present the user with a large image selection area. Use image\_picker to allow the user to pick an image from their gallery. The page should not proceed until an image is selected.
2. Once an image is selected and displayed, two TextFormFields should appear below it: one for the "Caption" and another for "Location (Optional)."
3. Create a "Share" button. When pressed, it must validate that the caption field is not empty (the location can be). If valid, it should print the caption, location, and the selected image's file path to the console, then navigate the user back to the FeedPage.

**Question 3**

You are the lead developer tasked with building the first version of "UrbanEats," a mobile app for discovering and reviewing local restaurants. Users will be able to browse restaurants, view details, and eventually, submit their own reviews. For this version, you will work with a local, mock data source. You will create your own mock data based on the following structure.

* **Restaurant Data Structure:**
  + id: String (e.g., "res-001")
  + name: String (e.g., "The Golden Spoon")
  + cuisine: String (e.g., "Italian")
  + address: String (e.g., "123 Main St, Metropolis")
  + rating: double (e.g., 4.7)
  + imageUrl: String (a URL to a placeholder image)
  + menu: List<MenuItem> (a list of menu item objects)
* **Menu Item Data Structure:**
  + id: String (e.g., "menu-001")
  + name: String (e.g., "Spaghetti Carbonara")
  + price: double (e.g., 18.99)

Your first major task is to establish a robust project foundation and build the main "Discovery" screen where users can browse a list of restaurants. The focus is on clean data modeling, efficient list rendering, and professional UI. The foundation of a good app is a well-organized project with clearly defined data structures.

1. Initialize a new Flutter project named urban\_eats. Create a clean and logical directory structure inside lib/ for your models (models/), data sources (data/), pages/screens (pages/), and reusable widgets (widgets/).
2. In lib/models/, create two model class files. restaurant.dart should contain the Restaurant class, and menu\_item.dart should contain the MenuItem class, both perfectly matching the data structures defined above.
3. In lib/data/mock\_restaurants.dart, create a final List<Restaurant> variable. Populate this list with at least **five** detailed Restaurant objects. Each restaurant must have a list of at least **three** MenuItem objects in its menu property. This will be your app's single source of truth for now.

The Discovery screen must display the restaurants from your mock data in a visually appealing and performant way.

1. Create a reusable RestaurantCard widget in lib/widgets/restaurant\_card.dart. This StatelessWidget must accept a single Restaurant object. The card should display the restaurant's imageUrl in a Container with a height of 150, using BoxFit.cover. Below the image, it should display the restaurant's name in a bold, larger font, and the cuisine type in a smaller, italicized font. At the bottom of the card, display the rating next to a filled star icon (Icons.star).
2. Create the main DiscoveryPage in lib/pages/discovery\_page.dart. This page must use a ListView.builder to render a vertical list of RestaurantCard widgets, using your mock data list as the source. Ensure you provide a unique ValueKey using the restaurant's id to each card for efficient updates.
3. Above the ListView.builder, add a static header section. This section should contain a prominent "Find Your Next Meal" Text widget and a TextField styled to look like a search bar (with a prefixIcon of Icons.search and a hint text of "Search restaurants..."). For this part, the TextField does not need to be functional.

The app needs a consistent visual theme and the basic plumbing for navigation to a future details screen.

1. In main.dart, define a ThemeData for the app. The theme should use a ColorScheme.fromSeed with a seedColor of Colors.teal. You must also download and integrate the "Roboto" font from Google Fonts, setting it as the default fontFamily. Configure the AppBarTheme to have a backgroundColor that matches the theme's primary color and white text.
2. Set up go\_router. Create lib/app\_router.dart and configure it with two routes: '/' which points to DiscoveryPage, and a route '/restaurant/:id' which will point to a RestaurantDetailsPage that you will create in the next section.
3. Make each RestaurantCard tappable. When a user taps on a card, it must navigate to the RestaurantDetailsPage, correctly passing the id of the tapped restaurant in the URL path.

With the discovery screen complete, you need to build the page that shows the details of a single restaurant. Additionally, you'll create the infrastructure for users to contribute by adding new restaurants to the platform via a form.

This page must provide a comprehensive view of a single restaurant, including its full menu.

1. Create the RestaurantDetailsPage in lib/pages/restaurant\_details\_page.dart. This widget will receive the restaurantId from the router. Your first job is to use this ID to find the correct Restaurant object from your mock data list. Gracefully handle the case where an invalid ID is passed by showing a "Restaurant not found" message.
2. The layout of this page must have a large image of the restaurant at the top (height 250). Below the image, display the restaurant's name, cuisine, and address. Use Theme.of(context).textTheme to apply appropriate styles (e.g., headlineSmall for the name).
3. Below the restaurant details, you must display its full menu. Use a ListView.builder to render the list of MenuItem objects associated with the current restaurant. For each menu item, use a ListTile widget to display its name as the title and its price (formatted as "$18.99") as the trailing widget. Place a Divider between each ListTile.

You need to build a complex form for users to submit new restaurants. This is a critical feature for growing the platform.

1. Create an AddRestaurantPage (/add-restaurant) and make it accessible via a FloatingActionButton on the DiscoveryPage. The page must be a StatefulWidget and contain a Form managed by a GlobalKey<FormState>.
2. The form must include TextFormFields for the Restaurant's "Name," "Cuisine," and "Address." Implement validators for each field: "Name" and "Cuisine" cannot be empty. "Address" must be at least 10 characters long.
3. Add an "Image Picker" UI to this form. It must be a Container with a dashed border that, when tapped, uses the image\_picker package to allow the user to select an image from their gallery. The selected image must then be displayed within this container.

The form needs to be fully functional, validating all inputs before "submitting" the data.

1. Create a "Submit Restaurant" ElevatedButton at the bottom of the form. The onPressed callback for this button must be a single function, e.g., \_submitForm().
2. In your \_submitForm() function, implement the complete validation logic. It must first check if an image has been selected. If not, it should show a SnackBar with the message "Please select a restaurant image." and stop. Then, it must trigger the form's validation using its GlobalKey.
3. If both the image is present and the form fields are valid, your function should print all the captured information to the console in a structured way: the text from all controllers and the file path of the selected image. Finally, it should show a success SnackBar saying "Restaurant submitted for review!" and navigate the user back to the DiscoveryPage using go\_router.