Practical Exam Instructions (CRM-like Features)

Practical 1: Basic CRM Features

1. **CRUD (Create, Read, Update, Delete)**
- Implement CRUD operations for a “Contacts” module (or similar) using **any PHP framework** (e.g., Laravel, CodeIgniter, Symfony, etc.).
- Ensure proper database schema design.
2. **Form Fields**
- **Standard fields**:
- Name
- Email
- Phone
- Gender (Radio button)
- Profile Image (file upload)
- Additional File (e.g., document upload)
- **Custom Fields**:
- Provide functionality for the user/administrator to **add and manage additional custom fields** (e.g., “Birthday,” “Company Name,” “Address,” or any user-defined field).
- Demonstrate how you store these custom fields in the database (e.g., a separate `contact_custom_fields` table, a JSON column, or another extensible approach).
- Ensure the UI can display and handle these custom fields dynamically.
3. **AJAX Integration for CRUD**
- Use AJAX for at least the **Insert, Update, and Delete** operations on contacts.
- Display success/error messages without a full page refresh.
4. **Filtering and Search**
- Implement filters for searching by **Name, Email, and Gender**.
- Perform the filtering via AJAX so that the results update without reloading the entire page.
- (Optional Bonus) Demonstrate filtering by **custom fields** if applicable.

Practical 2: Merging Contacts with Custom Fields

1. **Merge Feature**
- Provide a feature to **merge two contacts**.
- Include a button or link in the contact list to initiate the merge.
2. **Master Contact Selection**
- Upon initiating a merge, display a **popup** or modal that allows the user to select a **“master”** contact.
- This “master” contact will remain as the primary record after the merge.
3. **Confirmation and Final Merge**
- Show a **final confirmation** before committing the merge.
- If the user confirms:
- Retain all existing master contact data.
- **Add or merge** data from the secondary contact, including:
- Emails (store additional ones if they differ)
- Phone numbers (store additional ones if they differ)
- **Custom Fields**:
- If the secondary contact has values for custom fields the master lacks, add them.
- If both have the same custom field but different values, decide on a policy (e.g., keep the master’s value or append both values).

- **No data should be permanently lost**; secondary contact data should be preserved in some form under the master record.
4. **Data Integrity and UI**
- The merged (secondary) contact record should be **tracked** in some way (e.g., marked as merged or inactive), rather than simply deleted.
- Provide a clear UI indicating which fields/values were merged or overridden.
5. **Technical Considerations**
- Demonstrate how custom fields are merged. Examples:
- If using a separate table referencing `contact_id`, update or duplicate references for the merged contact.
- If using a JSON column, properly merge the JSON objects.
- Ensure your solution is **extensible** to accommodate future custom fields or additional data.

What We're Looking For

1. **Database Design & Extensibility**
- How well the schema handles dynamic custom fields.
- How merging is tracked (e.g., versioning, flags, etc.).
2. **Clean Code & Best Practices**
- Logical folder/file structure in your chosen PHP framework.
- Clear, reusable components (models, controllers, views).
- Properly handled AJAX calls with minimal page reload.
3. **UI/UX**
- Intuitive CRUD interfaces.
- Simple and informative merge workflow.
- Helpful validation/error messages.
4. **Merging Logic**
- Properly handle conflicts in custom field values.
- Guarantee no data is lost.
- Mark or store merged records accurately.
5. **Testing and Verification**
- Show that merged data appears correctly post-merge.
- Verify that filters/search still function correctly with merged contacts and custom fields.

Please record a video of whole functionality and show how database valued are changing after merge.

Good luck! Remember that clarity in your approach and code is just as important as delivering the final functionality.