7-Day Form Builder Development Plan

Project: Basic Form Builder using React

Duration: 7 Days (6-8 hours per day)

Tech Stack: React, Vite, Tailwind CSS, @dnd-kit, lucide-react

Project Scope

- Simple drag-and-drop interface for adding fields (text inputs, dropdowns, checkboxes, etc.)
- Storing form configurations as JSON
- Rendering forms based on configuration
- Collecting and storing responses

Day 1: Project Setup & Core Data Structure

Morning Session (3-4 hours)

Task 1.1: Initialize Project

- Create new React project using Vite with React template
- Install required dependencies:
  - @dnd-kit/core, @dnd-kit/sortable, @dnd-kit/utilities (for drag-drop)
  - lucide-react (for icons)
  - tailwindcss, postcss, autoprefixer (for styling)
- Configure Tailwind CSS in the project
- Remove default boilerplate code from App.jsx and index.css

Task 1.2: Create Folder Structure

- Create folders: `components/builder` , `components/fields` , `components/preview` , `components/responses`
- Create folders: `context` , `hooks` , `utils` , `constants`
- Each folder should be empty at this stage, just creating the structure

Task 1.3: Define Field Type Constants

- In `constants/fieldTypes.js` , define all supported field types (text, email, number, textarea, dropdown, checkbox, radio)
- Create a default configuration object for each field type with common properties
- This file exports constants that will be used throughout the app

Afternoon Session (3-4 hours)

Task 1.4: Create Data Schema Utilities

- In `utils/formSchema.js`, create a function that generates a new field object with unique ID, type, label, placeholder, required flag, order number
- For dropdown and radio types, include an options array with default values
- Create a function that generates a new form object with ID, title, description, empty fields array, and timestamp
- These utility functions will be called whenever creating new fields or forms

Task 1.5: Build Form Builder Context

- In `context/FormBuilderContext.jsx`, create a Context and Provider component
- Initialize state for: current form object, selected field ID
- Create function stubs (empty for now) for: addField, updateField, deleteField, reorderFields, updateFormMetadata
- Export a custom hook `useFormBuilder` that returns the context
- Wrap the context value with all state and functions

Task 1.6: Test Setup

- Update App.jsx to wrap content with FormBuilderProvider
- Add a simple console.log in the provider to verify it's working
- Verify Tailwind is working by adding a colored div

Deliverable: Clean project structure with data models and context ready to use

Day 2: Field Components & Form Preview

Morning Session (3-4 hours)

Task 2.1: Create Text-based Field Component

- In `components/fields/TextField.jsx` , create a component that accepts: field object, value, onChange handler, preview boolean
- Render a label with the field's label text, add asterisk if required
- Render an input element with type from field.type, placeholder, value, onChange
- Disable input if preview is false
- Style with Tailwind: proper spacing, borders, focus states

Task 2.2: Create TextArea Field Component

- Similar to TextField but use textarea element instead of input
- Make it resizable with appropriate rows (default 4)
- Same prop structure and styling approach

Task 2.3: Create Dropdown Field Component

- Accept same props as TextField
- Render select element with option elements mapped from field.options array
- Include an empty "Select..." option as the first option
- Handle value changes appropriately

Task 2.4: Create Checkbox Field Component

- Render a checkbox input with label
- Handle boolean value (checked/unchecked)

- Style differently - checkbox next to label, not stacked

Task 2.5: Create Radio Field Component

- Map through field.options array
- Render radio input for each option with same name attribute
- Each radio should have its value from the option
- Handle value changes when any radio is selected

Task 2.6: Create Field Renderer Component

- In `components/fields/FieldRenderer.jsx` , create a component that accepts field, value, onChange, preview
- Create an object mapping field types to their components
- Based on field.type, render the appropriate component
- Pass all props through to the selected component

Afternoon Session (3-4 hours)

Task 2.7: Create Form Preview Component

- In `components/preview/FormPreview.jsx` , create a component
- Use useFormBuilder hook to get current form
- Initialize local state to store form responses (object with fieldId: value pairs)
- Create handleFieldChange function that updates the response state
- Create handleSubmit function that prevents default and logs formData (will enhance later)

Task 2.8: Build Preview UI

- Render form title as h2 element
- Render form description if it exists
- Render a form element with onSubmit handler

- Sort fields by order property
- Map through sorted fields and render FieldRenderer for each, passing field, value from state, onChange handler, and preview=true
- Add Submit button at bottom (only show if fields exist)
- Style as a card with max-width, centered, with shadow

Task 2.9: Implement Context addField Method

- In FormBuilderContext, implement addField function
- Accept a field object parameter
- Update form state by adding field to fields array
- Set selectedFieldId to the new field's ID

Task 2.10: Test Preview Component

- In App.jsx, render FormPreview component
- Manually add a test field to initial form state to verify rendering
- Test each field type by manually adding them to initial state
- Verify form submission logs data correctly
- Test required field validation

Deliverable: All field types render correctly and form preview is fully functional with submission

Day 3: Form Builder Interface Layout

Morning Session (3-4 hours)

Task 3.1: Create Field Type Configuration

- Create array of field type objects with: type, display label, icon name
- This will drive the field palette UI

Task 3.2: Build Field Palette Component

- In `components/builder/FieldPalette.jsx` , create component that accepts onAddField callback prop
- Map through field types array
- Render a button for each field type with icon (from lucide-react) and label
- On click, call onAddField with the field type
- Style as vertical list with hover effects, icons aligned left
- Fixed width sidebar (256px), white background, border on right

Task 3.3: Create Field Editor Placeholder

- In `components/builder/FieldEditor.jsx` , create basic component
- Just render a div with "Properties Panel" heading
- Fixed width sidebar (320px), white background, border on left
- Will be implemented fully on Day 5

Task 3.4: Build Form Canvas Header

- In `components/builder/FormCanvas.jsx` , start building the component
- Use useFormBuilder hook to access form state
- Render editable input for form title (large, bold)
- Render editable textarea for form description (smaller, gray, optional)

- Create handlers to update form metadata (will implement in context later)
- Style as large centered card

Afternoon Session (3-4 hours)

Task 3.5: Build Form Canvas Field List

- Below the header, create a section for fields
- If fields array is empty, show centered message "No fields yet. Add fields from the left panel"
- If fields exist, sort by order and map through them
- For each field, render a container div with click handler to select field

Task 3.6: Style Field Items in Canvas

- Each field container should have:
    - Border that changes color when selected (blue when selected, gray otherwise)
    - Background that changes when selected (light blue tint)
    - Padding and rounded corners
    - Cursor pointer
    - onClick sets selectedFieldId in context
- Show grip icon on left (for future drag-drop)
- Show delete icon on right with click handler to delete field
- In the middle, render FieldRenderer with preview=false

Task 3.7: Implement deleteField in Context

- In FormBuilderContext, implement deleteField function
- Accept fieldId parameter
- Filter out the field from fields array
- If deleted field was selected, set selectedFieldId to null

Task 3.8: Create Main Builder Layout Component

- In `components/builder/FormBuilder.jsx` , create component
- Use useFormBuilder hook
- Create handleAddField function that creates new field using utility function and calls context addField
- Render three-column layout: FieldPalette, FormCanvas, FieldEditor
- Use flexbox for layout, FieldPalette and FieldEditor fixed width, FormCanvas flexible

Task 3.9: Implement updateFormMetadata in Context

- In FormBuilderContext, add updateFormMetadata function
- Accept updates object (title and/or description)
- Merge updates into form state

Task 3.10: Add Mode Switching to App

- In App.jsx, add state for current mode (builder or preview)
- Add navigation bar at top with buttons to switch between modes
- Conditionally render FormBuilder or FormPreview based on mode
- Style navbar with app title and mode buttons

Deliverable: Complete builder interface where you can add fields by clicking palette items, see them in canvas, select them, and delete them

Day 4: Drag-and-Drop Functionality

Morning Session (3-4 hours)

Task 4.1: Set Up DnD Context

- In FormBuilder.jsx, import DndContext from @dnd-kit/core
- Import sensors (PointerSensor, KeyboardSensor) and useSensors hook
- Set up sensors with proper activation constraints (minimum drag distance)
- Wrap the builder layout with DndContext
- Add onDragEnd handler (empty for now)

Task 4.2: Make Palette Items Draggable

- In FieldPalette.jsx, import useDraggable from @dnd-kit/core
- For each field type button, wrap with draggable functionality
- Set unique ID for each draggable (e.g., "palette-text", "palette-email")
- Pass field type as data
- Apply setNodeRef to button element
- Apply transform and transition styles from listeners and attributes
- Add visual feedback during drag (slight opacity change)

Task 4.3: Make Canvas a Drop Zone

- In FormCanvas.jsx, import useDroppable from @dnd-kit/core
- Create droppable zone for the entire field list area
- Set ID as "canvas-dropzone"
- Apply setNodeRef to the container
- Add visual feedback when dragging over (border color change)

Afternoon Session (3-4 hours)

Task 4.4: Implement onDragEnd Handler for Adding Fields

- In FormBuilder.jsx onDragEnd function:
  - Check if active.id starts with "palette-" and over.id is "canvas-dropzone"
  - Extract field type from active.data
  - Create new field using utility function with correct order (fields.length)
  - Call addField from context
- Test: drag from palette to canvas should add field

Task 4.5: Make Canvas Fields Sortable

- In FormCanvas.jsx, import SortableContext and arrayMove from @dnd-kit/sortable
- Wrap the field list with SortableContext
- Pass fields array mapped to IDs as items prop
- Use vertical list sorting strategy

Task 4.6: Create Sortable Field Item Component

- Create new component `SortableFieldItem.jsx` in components/builder
- Use useSortable hook with field.id
- Apply setNodeRef, transform, transition to container
- Render the same field item UI from FormCanvas but with sortable attributes
- Include grip icon that uses listeners to initiate drag

Task 4.7: Update Canvas to Use Sortable Items

- In FormCanvas.jsx, replace field mapping with SortableFieldItem components
- Pass all necessary props (field, isSelected, onSelect, onDelete)

Task 4.8: Implement reorderFields in Context

- In FormBuilderContext, implement reorderFields function
- Accept oldIndex and newIndex parameters
- Use arrayMove or manual array manipulation to reorder
- Update order property of each field to match new position
- Update form state with reordered fields

Task 4.9: Complete onDragEnd for Reordering

- In FormBuilder.jsx onDragEnd:
  - Add condition to check if both active and over are field IDs (not palette)
  - Get old and new indices from fields array
  - Call reorderFields with indices
- Test: dragging fields should reorder them

Task 4.10: Add Visual Feedback

- Add overlay component when dragging (shows what's being dragged)
- Add drop indicators between fields when dragging to reorder
- Style drag handles to look interactive (grab cursor)
- Test all drag-drop interactions for smooth UX

Deliverable: Fully functional drag-and-drop - can drag from palette to add fields, and drag within canvas to reorder fields

Day 5: Field Configuration & Editing

Morning Session (3-4 hours)

Task 5.1: Design Field Editor UI Structure

- In FieldEditor.jsx, use useFormBuilder to get selectedFieldId and form
- Find the selected field object from form.fields
- If no field selected, show message "Select a field to edit properties"
- If field selected, show field type badge at top with icon

Task 5.2: Create Common Property Editors

- Create input for Label: text input bound to field.label
- Create input for Placeholder: text input bound to field.placeholder
- Create toggle for Required: checkbox bound to field.required
- Each input should call updateField from context onChange with field ID and updated property
- Style as vertical form with labels and proper spacing

Task 5.3: Implement Conditional Property Editors

- Add conditional section that only shows for dropdown and radio types
- Create editable list of options
- Each option shows as input with delete button
- Add "Add Option" button below list
- Options should be stored as array in field.options

Task 5.4: Create Options Editor Logic

- Create local handlers for:
  - Adding new option (appends empty string to options array)

    - Updating option text (updates specific index in array)
    - Deleting option (filters out specific index)
    - Reordering options (future enhancement, can skip for basic version)
- Each handler calls updateField with updated options array

Afternoon Session (3-4 hours)

Task 5.5: Implement updateField in Context

- Complete updateField function in FormBuilderContext
- Accept fieldId and updates object
- Find field in fields array
- Merge updates into field object
- Update form state with modified fields array
- Ensure immutability (don't mutate original arrays/objects)

Task 5.6: Add Field Duplication Feature

- In SortableFieldItem, add duplicate button next to delete
- Create duplicateField function in context
- Function should:
    - Find the field to duplicate
    - Create new field object with same properties but new ID
    - Insert after the original field
    - Update order properties of subsequent fields
- Wire up button click to call duplicateField

Task 5.7: Add Field Type-Specific Options

- For number type, add min/max value inputs in editor
- For text/email type, add min/max length inputs
- For textarea, add rows input

- Update field schema to support these properties
- Update field components to respect these constraints

Task 5.8: Improve Editor UX

- Add section headers in editor ("Basic Properties", "Options", "Validation")
- Use accordions or collapsible sections for better organization
- Add help text/tooltips for properties
- Show field preview at bottom of editor panel
- Add keyboard shortcuts info

Task 5.9: Add Undo/Redo Capability (Optional)

- Create history state in context (array of previous form states)
- On each update, push current state to history
- Create undo function that restores previous state
- Add undo button in navbar (Ctrl+Z hint)
- Limit history to last 20 states

Task 5.10: Test All Editing Scenarios

- Test editing each property type
- Test editing fields of different types
- Test adding/removing/editing options for dropdown/radio
- Test duplication
- Test that changes reflect immediately in canvas
- Test switching between fields while editing

Deliverable: Fully functional field editor where all properties can be edited and changes reflect immediately

Day 6: Form Configuration Storage

Morning Session (3-4 hours)

Task 6.1: Create Storage Utility Functions

- In `utils/storage.js`, create functions:
  - saveForm(form): saves form object to localStorage with key "formBuilder_currentForm"
  - loadForm(): retrieves and parses form from localStorage, returns null if not found
  - saveFormList(forms): saves array of forms to localStorage with key "formBuilder_savedForms"
  - loadFormList(): retrieves saved forms list
  - exportFormJSON(form): returns formatted JSON string
  - importFormJSON(jsonString): parses and validates JSON, returns form object or throws error

Task 6.2: Implement Auto-Save

- In FormBuilderContext, use useEffect to watch form state
- Debounce saves by 1 second (use setTimeout/clearTimeout)
- On change, save form to localStorage after debounce delay
- Add "last saved" timestamp to context state
- Show "Saving..." or "Saved" indicator in navbar

Task 6.3: Implement Load on Mount

- In FormBuilderContext, use useEffect with empty dependency array
- On mount, attempt to load form from localStorage
- If found, set as initial form state
- If not found, use createForm() default

- Add loading state to prevent flash of wrong content

Task 6.4: Create Form Management UI

- In navbar, add buttons for:
    - New Form (clears current form)
    - Save As (saves to forms list with user-provided name)
    - Open (shows modal to select from saved forms)
    - Export JSON (downloads file)
    - Import JSON (opens file picker)

Afternoon Session (3-4 hours)

Task 6.5: Implement New Form

- Create handleNewForm function
- Show confirmation dialog if current form has unsaved changes
- Call createForm() utility and set as current form
- Clear selectedFieldId
- Clear localStorage current form

Task 6.6: Implement Save As

- Create modal component for SaveFormModal
- Include input for form name
- On save, add form to saved forms list in localStorage
- Each saved form should have: id, name, form object, savedAt timestamp
- Show success message
- Update forms list in context if maintaining one

Task 6.7: Implement Open Form

- Create modal component for OpenFormModal

- Load and display list of saved forms
- Show each form with: name, created date, field count
- Add delete button for each saved form
- On select, confirm if current form has unsaved changes
- Load selected form into context
- Close modal

Task 6.8: Implement Export JSON

- Create handleExport function
- Generate JSON string from current form using exportFormJSON utility
- Create blob from JSON string
- Create download link with filename "form-{timestamp}.json"
- Programmatically click link to download
- Clean up blob URL

Task 6.9: Implement Import JSON

- Create handleImport function
- Create hidden file input element with accept=".json"
- On file select, read file as text using FileReader
- Parse JSON using importFormJSON utility
- Validate structure (check required properties)
- If valid, show confirmation and load form
- If invalid, show error message with details

Task 6.10: Add Export/Import for Responses (Prep for Day 7)

- Create storage functions for responses:
  - saveResponse(formId, response): appends response to responses array in localStorage
  - getResponses(formId): retrieves all responses for a specific form
  - clearResponses(formId): deletes all responses for a form

> - exportResponses(formId, format): exports as JSON or CSV
> - Don't implement UI yet, just create the utilities

> **Deliverable:** Complete form persistence with save/load, export/import, and auto-save functionality

Day 7: Response Collection & Polish

Morning Session (3-4 hours)

Task 7.1: Update Form Preview to Save Responses

- In FormPreview.jsx handleSubmit function:
  - Create response object with: id, formId, submittedAt timestamp, data (the formData object)
  - Call saveResponse utility to store in localStorage
  - Show success message after submission
  - Reset form data state to empty
  - Add loading state during save

Task 7.2: Create Response Viewer Component

- In `components/responses/ResponseViewer.jsx` , create component
- Accept formId as prop
- Load responses for current form using getResponses utility
- Show count of total responses at top
- If no responses, show empty state message

Task 7.3: Build Response List UI

- Map through responses array
- For each response, show:
  - Submission timestamp (formatted nicely)
  - Preview of first 2-3 field values
  - Expand/collapse button
- Use accordion or expandable cards pattern

- Style with alternating backgrounds or cards

Task 7.4: Build Response Detail View

- When expanded, show all field values
- Display each field label and submitted value
- Handle different field types (show checkboxes as Yes/No, etc.)
- Format dates and times nicely
- Add "Delete" button for individual response

Afternoon Session (3-4 hours)

Task 7.5: Implement Response Export

- Add "Export Responses" button above response list
- Create dropdown or modal to choose format (JSON or CSV)
- For JSON: export as array of response objects
- For CSV: create headers from field labels, rows from response data
- Handle fields with multiple values (checkboxes, etc.) appropriately
- Trigger download with proper filename and extension

Task 7.6: Add Response Management

- Add "Clear All Responses" button with confirmation dialog
- Implement delete individual response
- Add filter/search functionality for responses (optional but nice)
- Show response statistics (total responses, completion rate if tracking partial submissions)

Task 7.7: Polish Form Builder UI

- Review all components for consistent spacing and sizing
- Ensure all buttons have hover states

- Add transitions for smooth interactions
- Check responsive behavior (though desktop-focused is fine)
- Ensure proper focus states for accessibility
- Add loading spinners where appropriate
- Fix any visual glitches in drag-drop

Task 7.8: Add Validation and Error Handling

- In FormPreview, validate required fields before submission
- Show error messages for validation failures
- Add try-catch blocks around localStorage operations
- Show user-friendly error messages
- Add error boundaries (optional but recommended)
- Validate field configurations (e.g., dropdown must have options)

Task 7.9: Add Help/Guidance Features

- Add tooltips to explain features
- Create a "?" help button that shows keyboard shortcuts and tips
- Add placeholder text and empty states throughout
- Consider adding a brief tutorial or onboarding flow
- Add confirmation dialogs for destructive actions

Task 7.10: Final Testing and Bug Fixes

- Test complete workflow: create form → add fields → edit properties → preview → submit → view responses → export
- Test edge cases: empty forms, forms with no fields, deleting all fields, invalid JSON import
- Test data persistence: refresh page at various stages
- Test localStorage limits (what happens with very large forms or many responses)
- Test all drag-drop scenarios

- Fix any bugs found
- Verify all features work as expected

Deliverable: Fully functional form builder with response collection, viewing, and export capabilities

Success Criteria

By end of Day 7, you should have:

- ✅ Working drag-and-drop interface for adding and reordering fields
- ✅ 7 field types all rendering and working correctly
- ✅ Property editor for customizing all field attributes
- ✅ Form preview mode with working form submission
- ✅ Persistent storage using localStorage with auto-save
- ✅ Export/import forms as JSON
- ✅ Response collection and storage
- ✅ Response viewer with export functionality
- ✅ Polished UI with good UX
- ✅ Error handling and validation

Daily Time Commitment

- **Each day:** 6-8 hours of focused development
- **Morning session:** 3-4 hours
- **Afternoon session:** 3-4 hours
- **Tasks:** Broken into 30-60 minute chunks

Important Notes

- Stick to the plan but be flexible if a task takes longer
- If you finish early any day, start next day's tasks

- Don't skip testing - verify each feature works before moving on
- Commit code at the end of each major task or session
- Take breaks between sessions to avoid burnout
- Document tricky decisions or complex logic as you go

Good Luck with Your Form Builder Project! 🚀

Remember: Progress over perfection. Build something that works first, then make it beautiful.