# V3 - Implementation Suggestions

RAFT

plementing full support for our JSON schema dynamic forms may at first look intimidating, but if you break the problem down in the right way, it's atively straightforward.  As a starting point, build 'up' rather 'down'.  So start with the form controls you're going to need, then build on top of those create the form.

## ontrols

a minimum you need the following controls:

- A text input
- A date input
- A number input
- A boolean input (e.g. a checkbox)
- A selection input (e.g. a select box)
- A file input that converts files to strings **- Required for verification & self service flows**

would also strongly recommend:

- A radio button input (for selections with just a few values)
- ???

ch control should be capable of broadcasting it's value as it changes.  Your higher level components will use this to update the form's model, and form actions like validation.

## e schema component

need a component to render a single schema.  A schema can contain several things, one of the five data types supported by JSON (& therefore ON schema):

- Strings
- Numbers
- Booleans
- Objects
- Arrays

d also both multiple schemas (allOf) and alternative schemas (oneOf)

- oneOf
- allOf

'll create a component for all of these types of schema.  The latter two are special cases, the former can use a simple switch statement on the e' property to determine which component to show.  Pseudocode:

```
if (schema.oneOf) {
  // Render oneOf schema component
  return
}

if (schema.allOf) {
  // Render allOf schema component
  return
}

switch (schema.type) {
  case "number"
    // Render number schema component
    break
  case "boolean"
    // Render boolean schema component
    break
  case "object"
    // Render object schema component
    break
  case "array"
    // Render array schema component
    break;
  default:
    // Render a string schema component
    // should only occur when type == "string"
}
```

ch of these components will need to know how to render the appropriate UX for that type, and each will be responsible for broadcasting changes t's internal value.

hin these components the behaviour will vary.  Let's take a look at the component for string types

### ring schema component

have a number of components that output string types:

- A text input
- A date input
- A file input that converts files to strings
- A selection input (e.g. a select box)

but selection are determined by the format property, so we'll make a special case for that.  To determine what to render we'll need something like following.

```
if (schema.values) {
  // Render selection component
  return
}

switch (schema.format) {
  case "date":
    // Render date input
  case "base64url":
    // Render file upload component
  default:
    // Render text input
}
```

each case we must listen to the change events being broadcast by the individual components, and propagate that new value out to the consumer.

#### lidation

e advantage of breaking down our components by data type is that we can now validate the broadcast values produced in a consistent way. ings can be validated with:

- minLength
- maxLength
- pattern
- required
- min (date)
- max (date)

don't care which component produced the value, we can just apply the relevant rules to the string that is broadcast.

**te: date iso8601 date strings can be compared with '<' and '>', so no extra logic is required.**

### umber schema component

mbers are even simpler, they can only be selection controls, or number inputs:

```
if (schema.values) {
  // Render selection component
  return
}

// Render number input
```

d the validation options are:

- min
- max
- required

### olean schema component

ere's only one control for boolean types so we simply render that.  The only applicable validation is 'required'

## oject schema component

e object type is where thing get more interesting, objects have properties, and each of those properties is another JSON schema.  Therefore, the ponsibility of the object type component is to iterate over the properties collection, and re-use the schema component for each property.

e component will need to maintain the state of the object model, keeping it updated as each child schema broadcasts new updated values.  It will pagate changes to it's model out to it's consumers.

```
foreach(schema.properties as nestedSchema) {
  // Render schema component using nestedSchema
}
```

## ray type schemas

DO simple types, complex types

## ultiple schemas

ltiple schemas are simple to implement, we simply need to iterate over the list of schemas, and reuse the schema component for each:

```
foreach(schema.allOf as nestedSchema) {
  // Render schema component using nestedSchema
}
```

## ternative schemas

ernative schemas are a little more complicated, we need to provide a choice between the possible schemas, we might do this with a tabbed out, a set of radio buttons, or a select box.  The chosen schema schema can be rendered using the schema component.

```
foreach(schema.oneOf as nestedSchema) {
  // Render a tab using the nestedSchema title
}
foreach(schema.oneOf as nestedSchema) {
  if (isSelected(nestedSchema)) {
    // Render nestedSchema using schema component
  }
}
```

## onclusion

u'll need a number of components for entering values:

- A text input
- A date input
- A number input
- A boolean input (e.g. a checkbox)
- A selection input (e.g. a select box)
- A file input that converts files to strings (for verification and self service domains)
- A radio button input (optional enhancement to select box)

d a second set of components for the different types of schema:

- String type component
- Number type component
- Boolean type component
- Object type component
- Array type component **- TODO Arrays of Basic types component ??**
- AnyOf component
- AllOf component

**DO More on props**

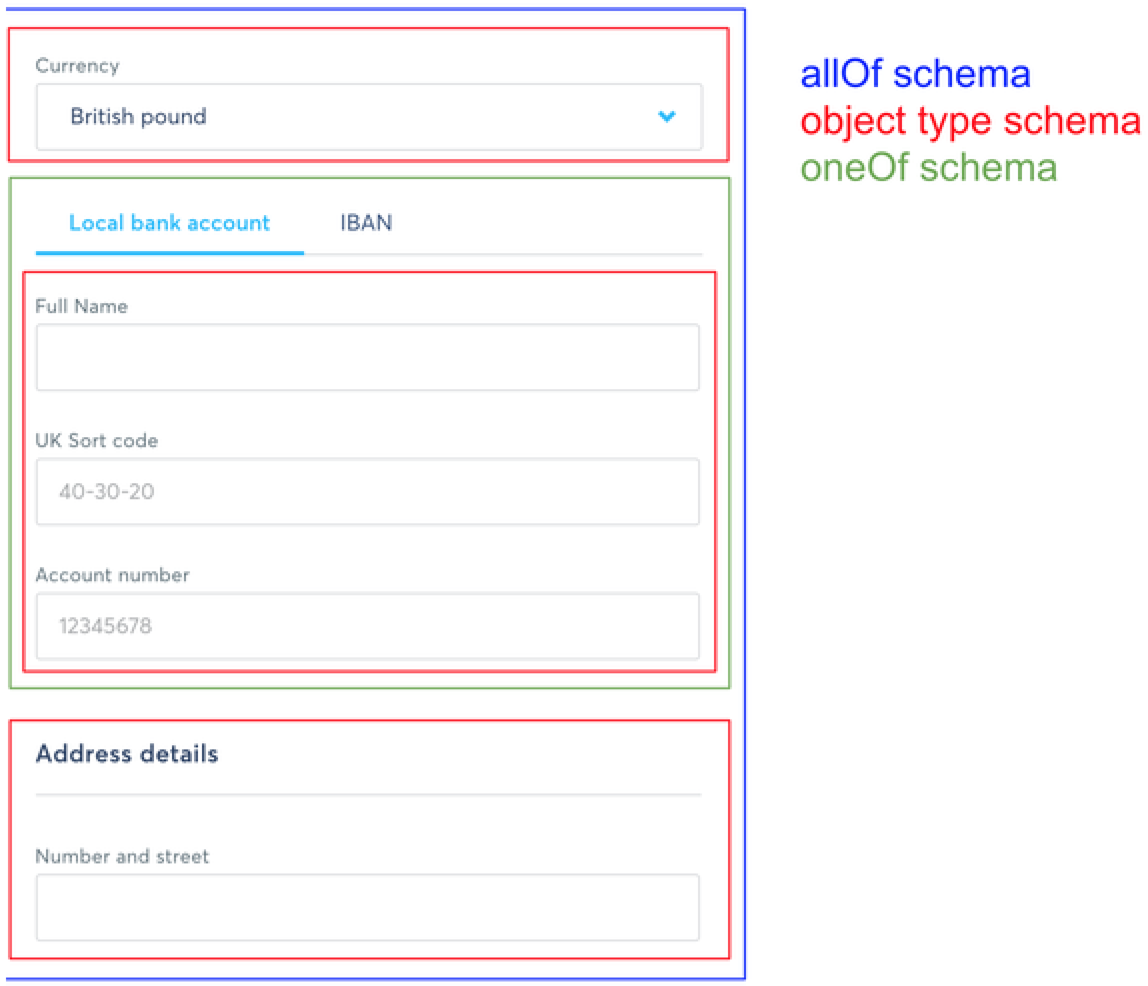
**ch of which accepts the schema, ...TODO ...  And broadcasts onChange**

## ıtting it all together

he following complex example, we use several object type schemas alongside both an allOf and a oneOf schema:

```
{
  allOf: [
    {
      type: "object",
      properties: {
        currency: { ... }
      }
    },
    {
      oneOf: [
        {
          type: "object",
          title: "Local bank account",
          properties: {
            fullName: { ... },
            sortCode: { ... },
            accountNumber: { ... },
          }
        },
        {
          type: "object",
          title: "IBAN",
          properties: { ... }
        }
      ]
    },
    {
      type: "object",
      title: "Address details",
      properties: {
        streetAddress: { ... }
      }
    }
  }
}
```

e form would render like this (assuming your implementation of oneOf uses tabs)


Currency

British pound

Local bank account | IBAN

Full Name

UK Sort code

40-30-20

Account number

12345678

Address details

Number and street

cause allOf and oneOf do not introduce nesting, the resulting model would look as follows:

```
{
  currency: "GBP",
  fullName: "John Smith",
  sortCode: "123456",
  accountNumber: "87654321",
  streetAddress: "22 Acacia Avenue"
}
```