Front-End Engineering Challenge

Build an application that dynamically generates form inputs from a provided JSON configuration and autosaves them to a backing store using the browser's LocalStorage.

Implementation Details

This type of configuration would typically come from an API, but you may load it from a JSON file or hardcode them for this project. You can assume this configuration does not change during the lifecycle of your application. You are responsible for creating the input components as you see fit for the best user experience. Feel free to use any libraries that may help you with implementation as you would in a production codebase.

The input configuration is defined as:

```
[
  {
    entity: 'Loan', // 'Loan' | 'Borrower'
    display: 'Loan Amount',
    field: 'loanAmount',
    type: 'money', // 'string' | 'money' | 'date'
    conditions: { maxValue: 100000000, minValue: 1000 } // conditions schema valid if type == money
  },
  {
    entity: 'Borrower', // 'Loan' | 'Borrower'
    display: 'First Name',
    field: 'firstName',
    type: 'string', // 'string' | 'money' | 'date'
    conditions: { regex: '^[a-zA-Z]+$' } // conditions schema valid if type == string
  },
  ...
  ...
]
```

Entity is always either a Loan or a Borrower.
Display represents the human-friendly label for the field.
Field is a string that uniquely identifies an input on an Entity. (Entity, Field) will always be unique.
Type is always either string, money, or date.

- A **string** is any valid javascript string. Empty saved string inputs should store an empty string.
- A **money** is any valid javascript number. Empty saved money inputs should store null.
- A **date** is an object with the following schema. Empty saved date inputs should store null.

```
{
    month: 1, // 1-12
    day: 14, // 1-31
    year: 1988
 }
```

Conditions will vary depending on the type and will trigger validation errors.

- If the type is **money**, valid conditions are maxValue and minValue. (see sample input)
- If the type is **string**, a regex can be provided for validation. (see sample input)
- If the type is **date**, there are no validation conditions.

Each of these input fields will run validation and autosave **on blur**. Validation errors should block the saving of the field to the backing store. Typically, the save event will trigger an API call, but for this project you should persist the information to the browser LocalStorage.

The backing store should be formatted as follows:

```
{
  Loan: {
    loanAmount: 500000,
    loanType: 'Purchase',
    downPaymentAmount: 100000
  },
  Borrower: {
    firstName: 'Jane',
    lastName: 'Homeowner',
    birthDate: {
      month: 1, // 1-12
      day: 14, // 1-31
      year: 1988
    }
  }
}
```

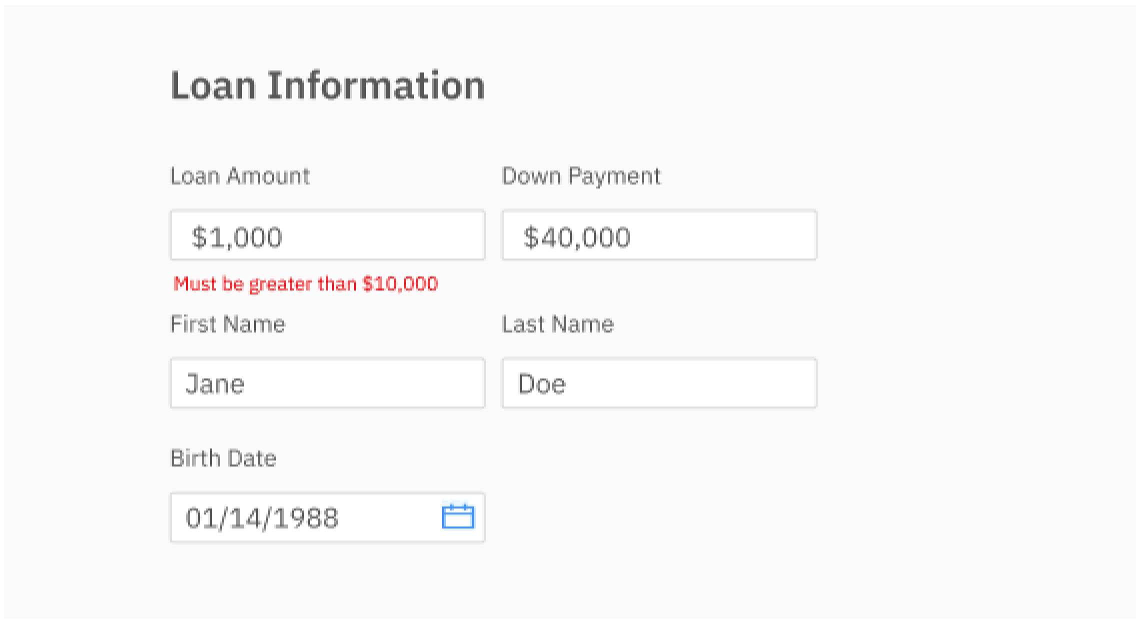
The top-level Loan and Borrower objects can optionally exist if they do not contain any saved fields. When an input field is saved, a value with the key of *field* should be added to the corresponding entity object.

Whenever an input field is successfully saved, please log the contents of the entire backing store to the console.

The application should reload any information from the backing store whenever the page is reloaded, even if the server is restarted.

User Interface

You may design the UI however you prefer with attention to usability. A sample user interface may look like this, but this is just an example.


Loan Information

Loan Amount	Down Payment
$1,000	$40,000
Must be greater than $10,000	
First Name	Last Name
Jane	Doe
Birth Date	
01/14/1988	

You will find that you will have to make some UX decisions around edge cases. For example, when an input validator fails, how does the user recover from that, and what is the impact on the backing store?

It is up to you to decide how you want to handle these scenarios, and you may be asked to justify your decisions.

Technology

Please use React with Typescript for this implementation. Create React App is a good starting point, though feel free to bootstrap however you wish. If you prefer to use another language or framework, prior approval must be received. You are free to use any UI component or code libraries that you prefer.

Deliverables

- Please include a README with your submission that explains how to run your code.
- Feel free to write as little or as many tests as you feel necessary to ensure your application behaves properly for all reasonable human behavior.
- Please submit code that is production quality -- it should follow best practices around syntax and style, without debug statements or TODOs.