Name: Minh Nguyen, Nina Nguyen, Pongpichit Poonpiriyasup, Shaista Usman
CPSC 5200

Detailed Design Document

Introduction

Every country has a particular, and peculiar, way to structure a postal address. This paper proposes a software design to handle the validation of address formats from different countries. It will include three parts, a web-based user interface, an API callable via HTTP, and a database to store addresses.

Design overview:

A. We are adopting **client-server style** for our software solution. On client particularly, we are using **event-based style** to wait and react to user's input or selection.

B. Web Service Design Patterns

- **Web Service API Styles:**
  - Message API: due to the requirement of the project, the client doesn't need to execute any procedure. The client only needs data to use for validation. Therefore, message API is appropriate for the project
- **Client-Service Interaction Styles:**
  - Request / Response: client needs data from server per each request to either populate custom address fields or validate addresses. So request/response seems to fit the implementation and it's also simple
- **Request and Response Management:**
  - Data Transfer Object: This allows for getter/setter methods to simplify the request and response structures. This also accommodates requests with multiple parameters (e.g. sending to validate multiple fields of an address at once)
- **Web Service Implementation Styles:**
  - Datasource Adapter: client needs some source of access to the the database to get address components. So an adapter is needed

Algorithm:

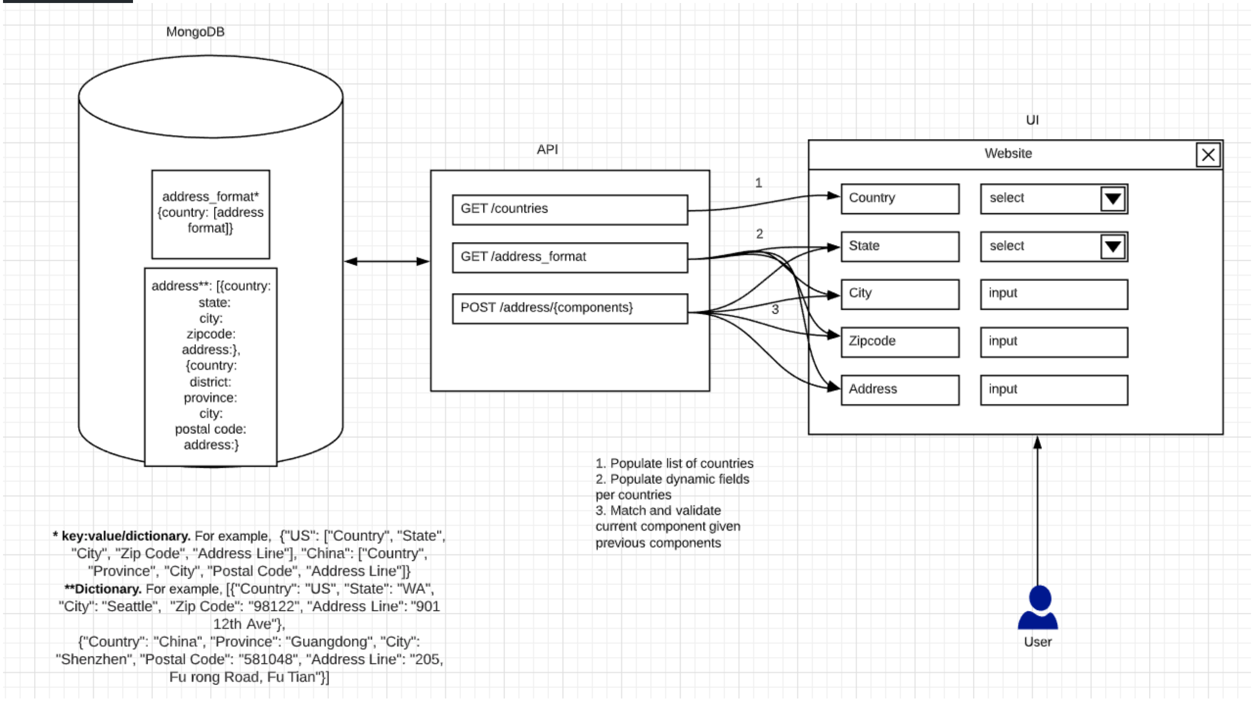
- Our scope of design is to accommodate exact-match searches so we will use Hashmap Search algorithm since our database is going to be a key-value database. This will provide fast lookup with O(1) complexity.

Technologies:

- HTML/Bootstrap for the UI design
- Node JS is used for the backend

Given these styles, we are considering two design choices and patterns:

Option 1


MongoDB
address_format* {country: [address format]}
address**: [{country: state: city: zipcode: address:}, {country: district: province: city: postal code: address:}
API
GET /countries
GET /address_format
POST /address/{components}
UI
Website
Country
select
State
select
City
input
Zipcode
input
Address
input
1
2
3
User

1. Populate list of countries
2. Populate dynamic fields per countries
3. Match and validate current component given previous components

*** key:value/dictionary.** For example, {"US": ["Country", "State", "City", "Zip Code", "Address Line"], "China": ["Country", "Province", "City", "Postal Code", "Address Line"]}
****Dictionary.** For example, [{"Country": "US", "State": "WA", "City": "Seattle", "Zip Code": "98122", "Address Line": "901 12th Ave"},
{"Country": "China", "Province": "Guangdong", "City": "Shenzhen", "Postal Code": "581048", "Address Line": "205, Fu rong Road, Fu Tian"}]

API Design:

GET /countries

- Description: get a list of countries to pre-populate in country field. Client will have this list ready on load
- Parameters: None
- Response: 200 successful with JSON data; 404: not found

GET /address_format

- Description: get a list of fields representing address parts per each country to send to client to populate dynamic address field
- Parameters: country
- Response: 200 successful with JSON data; 404: not found

POST /address/{components}

- Description: get a list of values/patterns to help validate current based on the parameters users select in the previous fields.
- Parameters: name of “lookup” field and list of values in previous fields (e.g. to populate city, send over “city” and list of selected values {“country”: “US”, “state”: “WA”})
- Response: 200 successful with JSON data; 404: not found

POST/address/{components}

- Description: validate the address text as last layer in hierarchy based on the parameters users have selected and types
- Parameters: list of values all fields

    - Response: 200 successful; 404: not found

We organize data fields in the hierarchy of Country => State/Province/District => City/Village => Postal code => Address line.

Country, State/Province/District, City will be populated as a constrained list based on the user’s input of the previous fields. Postal code and address line fields will be free-form text with underlying validators.

As a user chooses Country, a request call is made to an API on server with the country as a parameter. The server will query Mongo Db on Country as key and return a list of valid inputs in State/Province/District field to send back to client to populate into State/Province/District. Same mechanism will be applied to populate the drop down list of valid cities in “City” field (given prior input of Country and State/Province/District).

For postal code, an API call will be made to server to query the regex pattern allowed (given prior input of Country, State/Province/District, and City). Using this regex pattern, client will impose constraint on input allowed in freeform text

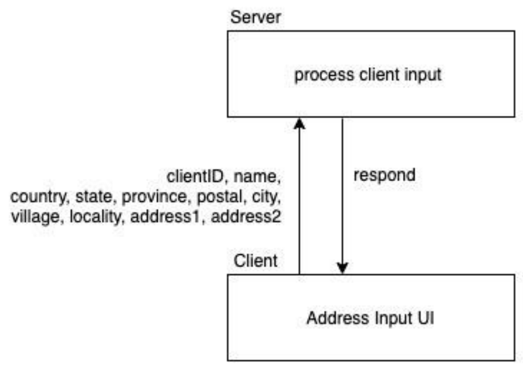
For the address line field, an API call will be made to server to query to the exact match (given prior input of Country, State/Province/District, City, and Postal Code PLUS current input of address text). Server will return “valid” or “not valid” to the client as a response.

Trade-off:

- Multiple API calls as user input in each field vs. just one API call to validate all address components as one. May consider caching to address the number of calls
- Need to track state of the current address component given states of others in the upper chain
- The benefits however are immediate response on validity if one of the address components is incorrect

<u>Option 2</u>

Style: Client-Server


Server
process client input
clientID, name,
country, state, province, postal, city,
village, locality, address1, address2
respond
Client
Address Input UI

Record Address Procedure

1. Client receive user input via UI
2. Client send data to the server upon submission
3. Server validate the input data
4. If the input data is valid, then the server stores the data into the database
5. If the server stores the input data, then responds success. Otherwise, responds fail.

Search Address Procedure

1. Client receive user input via UI
2. Client send the query data to the server upon search submission
3. Server searches the database according to the query data.
4. Server responds with all the matched data according to the query

Pattern:

- Abstract Factory: The client provides the interface, whereas the server create the object (address)
- Facade: The server serve as a sub-system to the client
- Observer: One server can serve multiple clients.

Trade-off:

- Only one API call is made so latency won't be affected

API design:

```
POST /address
    - description: validate the input data format, and record the data onto the
                   database if the format is valid
    - parameters (body): { country: String, state: String, province: String,
                           postal: String, city: String, village: String,
                           locality: String, address1: String, address2: String }
    - response: 200; for a successful record.
                404; otherwise.

POST /search/address
    - description: search the database according to the input query
    - parameters (body): { country: String, state: String, province: String,
                           postal: String, city: String, village: String,
                           locality: String, address1: String, address2: String }
    - response: 200; a list of matched addresses. An empty list otherwise.
```