

# Accurate Addressing Best Practices

By Gaston Hummel and Tom Thomas, February 7, 2019

## Contents

Abstract.....	1
Spectrum Flows for Accurate Addressing.....	2
Web Site Integration.....	3
Front End (Angular Application) .....	5
Building the Angular App and deploying it into Spectrum .....	24
Security Settings in Spectrum to Enable Accurate Addressing .....	26
Appendix A – Web Service Details.....	30
Shape of <i>AutoComplete/Type-Ahead</i> Web Service .....	30
Shape of <i>Address Resolution</i> Web Service .....	31
Appendix B – Download Locations .....	32
Sample Angular App using admin/admin credentials .....	32
Sample Angular App using no login credentials .....	33
Advanced Angular App using admin/admin credentials .....	33

## Abstract

Poor quality customer data imposes costs and risks on businesses. Gartner research, Harvard Business Review and Forbes Magazine agree that organizations estimate poor data quality to be responsible for an average of \$15 million per year in losses. These costs are not solely financial. Businesses experience loss of reputation, missed opportunities and higher-risk decision making as the result of low confidence in data.

Delivery of one in twenty online orders fails on the first attempt, costing organizations an average of \$15 per failed delivery.

18% of customer records in people databases such as CRM systems is duplicate. Corporate data is growing at around 40% annually and each duplicate record costs organizations \$20 to \$100 per year in terms of time spent on upkeep and storage, not to mention the impact this has on customer experience.

What are the root causes of these issues? Two root causes are not validating addresses (and other contact information) at the point of capture and not normalizing the formats of captured names and addresses. Solutions exist to both problems but effective solutions are not as simple as they might first appear.

A prime opportunity to capture and validate accurate address details is during the checkout process of an online order or the online signup process for a new service. Many organizations have taken a first step towards improving address capture at these customer touchpoints by implementing *AutoComplete/Type-Ahead* solutions where the customer can just start typing their first address and a list of address matches pops up for the customer to select. While such an *AutoComplete/Type-*

*Ahead* solution is convenient for the customer, it by no means solves all your addressing problems. Often the addresses presented by the *AutoComplete/Type-Ahead* datasets include building addresses both with and without suite or apartment numbers, meaning a customer selects their correct building address and overlooks that they selected a non-deliverable address by not including their apartment number.

Therefore when a customer selects an address presented in an *AutoComplete/Type-Ahead* solution it is important to validate that the selected address is actually a complete and verified as deliverable. It is possible to maintain an optimal customer experience in the event of a selected address being non-deliverable by either automatically fixing it or presenting the user with additional *validated* address options for selection. Automatic fixes can include automatically adding suite numbers to addresses based on entered company names or automatically fixing misfielded data such as name that have been included in the address field.

Beyond validation, normalizing captured names and addresses can ensure consistency in customer records across all systems and avoid creating duplicate records as the result of capturing slightly differently formatted customer names and addresses during different customer interactions across your different touchpoints.

There are two parts to this paper. The first looks at how the optimal Spectrum flows work to provide *AutoComplete/Type-Ahead* functionality and then validate selected addresses as well as seamlessly (auto) correct non-deliverable addresses. The second section looks at best practices for integrating the Accurate Addressing solution with your front-end and back-end systems.

## Spectrum Flows for Accurate Addressing

The Accurate Addressing solution relies on two Web services API workflows. When used together, they accomplish the desired result of “getting the right address at the right time.” The workflows integrate easily into a web interface as described in detail in this document.

Both the Interactive *AutoComplete/Type-Ahead* and Real-Time *Address Resolution* services are simple REST web services.

**AutoComplete/Type-Ahead service** – processes characters as they are typed into the address field of an online form. The engine searches its source data to dynamically predict and compile a list of address candidates. As the customer types their address, an address candidate list pops up for the user to select the correct address and minimize typing.

- **AutoComplete/Type-Ahead** expedites an address entry processes by reducing keystrokes. It resolves most common address entry problems by correcting mistypes and misspells. It eliminates parsing and misfielding of data by auto-populating fields on selection. It includes secondary addresses like apartment numbers when available to ensure selection accuracy. It is possible to filter the returned address candidates by postal code or geolocation for a faster, more intuitive experience and the maximum returned results is configurable.
- **AutoComplete/Type-Ahead** source data is optimized for returning fast results. Commonly, this data does not contain information about location accuracy or postal deliverability, making it critically important to complement the *AutoComplete/Type-Ahead* step with an *Address Resolution* service.

**Address Resolution service** - captures a selected address from an *AutoComplete/Type-Ahead* candidate list. It then standardizes and verifies the address against trusted third party source data (such as postal or geolocation data if desired) and returns a validated, postal deliverable or highly

accurate geo-locatable address. If the candidate address does not validate to postal delivery or to geolocation accuracy standards, a list of verified address suggestions is output and displayed for selection as available. Additionally, algorithms including both phonetic and non-phonetic heuristics determine the accuracy and relevance of address suggestions.

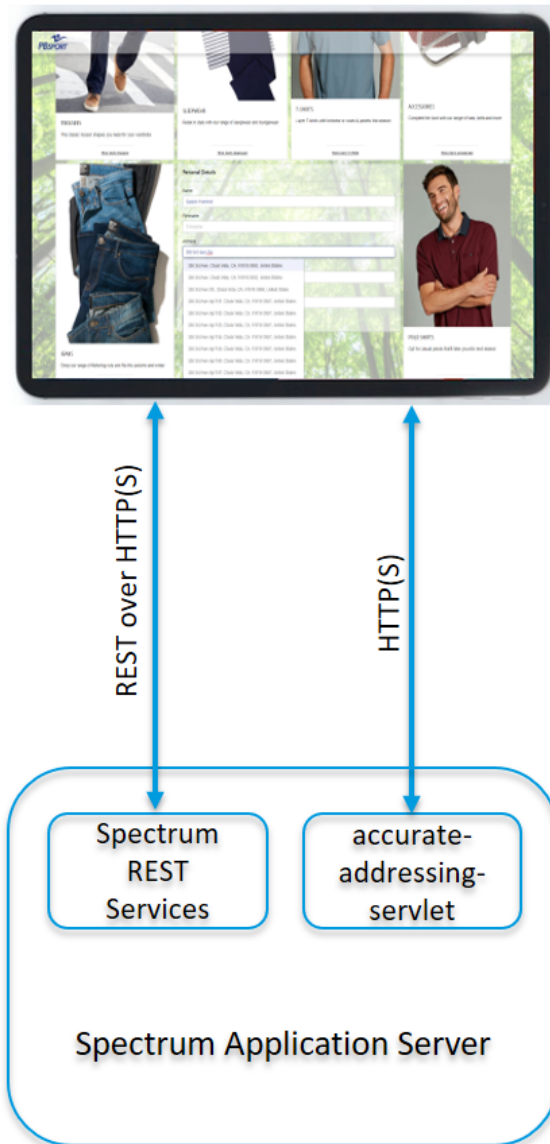
- **Address Resolution** benefits additionally by standardizing and validating military, highway contract, rural route and PO Box addresses. It permits additional data quality processing should an address be selected by override. It resolves multi-match addresses where a pre or post directional is missing or incorrectly entered, appends accurate secondary addresses based on business name, applies postal conversation changes when available to keep addresses current and it returns an abundance of additional information about how the address matched and about its accuracy.
- **The Address Resolution** service returns all the information necessary to make confident business decisions based on address accuracy.

## Web Site Integration

A video of a demo of a sample implementation of the Accurate Addressing solution is available at [https://youtu.be/70fDFt\\_3eio](https://youtu.be/70fDFt_3eio).

The Accurate Addressing solution calls two RESTful web services. One to return *AutoComplete/Type-Ahead* matches as the user enters their address, and an *Address Resolution service* to validate the deliverability of the selected address from the *AutoComplete/Type-Ahead*. The Spectrum flow behind the validation web service will also fix or propose alternated validated address options in case a selected address in the *AutoComplete/Type-Ahead* is non-deliverable.

Two components play a role in the Accurate Addressing solution.



Web browser capturing and validating customer address and other contact details using Accurate Addressing solution.

Angular application using PB Design System.

Angular app is served to the browser from servlet deployed in Spectrum

Once on browser, Angular app communicates with Spectrum REST services

Spectrum with UAM Loqate, Universal Address Module, DPV, LACSLink and SuiteLink DBs.

Accurate Addressing Spectrum Flow.

For the *AutoComplete/Type-Ahead*, you want your customer to see matched address results as they type with minimal delay, yet at the same time you want to minimize the load on your Spectrum server and minimize the impact of poor network bandwidth on the service. The solution is to use [Reactive JavaScript extensions](#), which includes Observables, enabling you to set things like:

- **debounceTime**: how long the input field should be unchanged before calling out to the server. If your customer is rapidly typing an address, it is best to wait for them to pause for say 300 milliseconds before sending a request for matches to the server. This limits server and network load.
- **distinctUntilChanged**: should your customer enter text but then delete/correct it, then there is no point sending it to the server unless it has changed since the last request to the server.
- **filter**: check that the length of the input is not zero to avoid sending empty requests to the server.

## Front End (Angular Application)

The frontend demo in the above video uses the PB Design System. The PB Design System is available for download from Github at <https://github.com/PBGUX/pb-design-system> and includes *unbranded* options so that you can configure it to match your brand. It is based on the [Angular](#) web framework, a complete modern web framework open sourced by Google with most necessary feature built-in. For example, the Reactive JavaScript extensions (RxJS) come packaged with Angular by default.

The *AutoComplete/Type-Ahead* part of the solution needs to pass the country of the address into the search query. The demo sets the country automatically to the country of the user's browser, but provides an option for the user to override the country if they are entering an address in a different country.

Pitney Bowes offers a [GeoLocation](#) service that can determine a user's location based on their IP address or Wifi Access Point MAC address. When the user connects to the web application it calls these services.

Components make up an Angular web application. Each section of the user interface or even element can be its own component and embedded into other components. Each component consists of four files by default:

1. HTML file containing the content.
2. TS, Typescript file containing the logic. Typescript is a type safe language from Microsoft that compiles down to JavaScript.
3. SCSS file(s) containing the styling.
4. SPEC.TS file containing unit tests.

This paper is not a thorough introduction to Angular but aims to share the basics that you will need to implement an Accurate Addressing solution inside your own portal.

Much of this paper will focus on a component I called *cardcontent*, which contains the address entry field. Let us start by looking at the *AutoComplete/Type-Ahead* part of *cardcontent.component.html*.

```
1 <div class="card-body">
2   <form class="form" [formGroup]="addressForm" autocomplete="off">
```

Styling is associated with the form using the class called *form*. The *formGroup* setting associates the form with settings in the Typescript. *autocomplete* is set to *off* to prevent the browser's saved values from appearing and interfering with the address suggestions made by the *AutoComplete/Type-Ahead* service.

```

3     <div class="form-group">
4         <label for="cardName">Name</label>
5         <input
6             type="text"
7             class="form-control"
8             [(ngModel)]="enteredName"
9             id="cardName"
10            placeholder="Enter your full name"
11            (input)="normalizeName($event)"
12            [ngModelOptions]="{ standalone: true }"
13        />
14    </div>
15    <div class="form-group">
16        <label for="firmName">Firmname</label>
17        <input
18            type="text"
19            class="form-control"
20            [(ngModel)]="firmName"
21            id="firmName"
22            placeholder="Firmname"
23            (blur)="saveFirmName($event)"
24            [ngModelOptions]="{ standalone: true }"
25        />
26    </div>

```

The above two *form-groups* display the input for Name and Firmname. The *ngModel* setting ties their values to the specified variable in the component's Typescript. The *(input)* attribute of the *cardName* input element specifies that the *normalizeName* function in the component's Typescript is called whenever text is typed into the field.

The *ngModelOptions'* *standalone: true* setting decouples the *Name* and *Firmname* controls from the *address* input with its sophisticated *AutoComplete/Type-Ahead* controls.

The *(blur)* attribute specifies that the *saveFirmName* function is called in the component's Typescript when the user moves their cursor outside the *Firmname* field.

```

27     <div class="form-group">
28         <label for="cardAddress"
29             >Address
30         <i
31             *ngIf="addressValid == 2"
32             class="pbi-icon-outline pbi-alert-circle"
33             container="body"
34             placement="right"
35             ngbPopover="{{ getAddressDescription }}"
36             triggers="mouseenter:mouseleave"
37         ></i>
38         <i
39             *ngIf="addressValid == 1"
40             class="pbi-icon-mini pbi-check-bold"
41             container="body"
42             placement="right"
43             ngbPopover="{{ getAddressDescription }}"
44             triggers="mouseenter:mouseleave"
45             style="color: greenyellow;"
46         ></i>
47         </label>
48         <div class="input-typeahead" role="alert" aria-live="assertive">
49             <input
50                 id="cardAddress"
51                 autocomplete="address-typeahead"
52                 placeholder="Start typing your address"
53                 autocorrect="off"
54                 type="text"
55                 class="form-control"
56                 [ngClass]="{
57                     'is-invalid': cardAddress.invalid && cardAddress.touched
58                 }"
59                 formControlName="cardAddress"
60                 [ngbTypeahead]="search"
61                 [resultFormatter]="resultFormatter"
62                 (selectItem)="selectItem($event)"
63                 required
64             />
65             <span *ngIf="searching" class="input-typeahead-loader">
66                 <i class="pbi-icon-mini pbi-loader-circle spin"></i>
67                 <p class="sr-only">Searching addresses...</p>
68             </span>
69             <p class="invalid-feedback">Enter an address.</p>
70         </div>
71     </div>
72     <app-addressoptions [hidden]="!showAddressOptions"></app-addressoptions>
73     <br />
74     <input
75         class="btn btn-outline-primary"
76         type="reset"
77         value="Reset"
78         (click)="resetForm()"
79     />
80 </form>
81 </div>

```

The *label* for this field contains the word “Address”, followed by icons with popover descriptions that conditionally show based on an address being selected and the validity of the selected address. We will later look at how the values *addressValid* and *getAddressDescription* are set in the Typescript.

The attributes of the *input* field are:

id	Associates the <i>input</i> with its label
autocomplete	Taken over from the PB Design System Sample
placeholder	The text that appears in the <i>input</i> field before the user enters any data.
autocorrect	Option to <i>autocorrect</i> the data the user enters as they type.
type	The type of <i>input</i> field.
name	Not required but included for completeness.
class	Used for styling this element.
ngClass	A way of applying the <i>is-invalid</i> class to the <i>input</i> element when the address is invalid and the address field has been touched.
formControlName	Links this field in the HTML to the <i>formControl</i> in the Typescript file.
ngbTypeahead	Specifies the function in the Typescript file to call when user enters new data.
resultFormatter	Specifies the function in the Typescript file called with each item of the array returned from the <i>AutoComplete/Type-Ahead</i> web service in order to return the string displayed in the <i>AutoComplete/Type-Ahead</i> matches for selection.
selectItem	Specifies the function in the Typescript file called when the user pick an address from the <i>AutoComplete/Type-Ahead</i> options. <i>\$event</i> is the selected item passed into the function.
required	Indicates not to submit the form when the address <i>input</i> field is blank.

The *span* shows a spinner to the right of the *input* field when the UI has sent an address string to the web server and is awaiting a response. *Searching* is toggled between false and true by functions in the component's Typescript file, which will be explained later.

The *p (paragraph)* tag is displayed if the user tries to submit the form with a blank *address* field.

The *app-addressoptions* is another Angular component. Here you can see how one Angular component gets included in another. As described in the Abstract, sometimes an address picked from the *AutoComplete/Type-Ahead* matches can be for a building, but not be a deliverable address. For example, it might be missing a suite or apartment number. These types of issues are resolved by calling *Address Resolution* service with the selected address. This flow returns an array of one or more corrected and validated addresses. If it is possible to fix the address without requiring additional user input then the returned array contains one verified address. This happens if, for example, a firm name is provided that can be used to automatically add the correct unit number and in such cases the address picked by the user is automatically corrected to the validated address with unit number.

However, when there are multiple possible unit numbers for a selected address then multiple validated address options get from the *Address Resolution* service. These displayed in the *app-addressoptions* component for the user to select. This component is normally hidden unless



functions in the Typescript file sets *showAddressOptions* to *true*, which will happen when more than one validated address is returned and the user needs to select the correct one.

The *input* with the reset button resets the form and calls the *resetForm* function in the Typescript to clear setting related to the form that are saved in other components like *appdataService*.

We will now look at how the functions called from the *input* field in the HTML file of the component work in the corresponding Typescript file of the component.

```
1 import { Component, Input, OnInit } from "@angular/core";
2
3 import { FormGroup, FormControl, Validators } from "@angular/forms";
4
```

The Typescript starts by importing the Angular directives common to most UI components.

It also imports the directives for controlling forms.

```
5 import { Observable, of } from "rxjs";
6 import "rxjs/add/operator/catch";
7 import "rxjs/add/operator/debounceTime";
8 import "rxjs/add/operator/distinctUntilChanged";
9 import "rxjs/add/operator/do";
10 import "rxjs/add/operator/map";
11 import "rxjs/add/operator/switchMap";
12 import "rxjs/add/operator/merge";
13 import "rxjs/add/operator/filter";
14
```

Next, we import the Reactive JavaScript extensions, mentioned earlier, to deal with when to send typed content to the server and how to wait for a response. RxJS is included with Angular.

```
15 import { AppdataService } from "../services/appdata.service";
16 import { NameService } from "../services/name.service";
17 import { AddressService } from "../services/address.service";
18
```

Here we are importing three service components that we wrote as part of this implementation. Service components in Angular are a common way of sharing data between multiple components. An example of how service components are useful here is when dealing with an *Address Resolution* service web-service-response. If the response array contains more than one validated address option, then two UI components need access to this response. The *cardcontent* component to show the first item in the response and provide data to the condition that decides if the *addressoptions* component is shown and the *addressoptions* component also needs the response to be able to display validated address alternatives. We will describe these service components in more detail later. Refer to Appendix A for details on the two web services used for Accurate Addressing.

```
19 @Component({
20   selector: "app-cardcontent",
21   templateUrl: "../cardcontent.component.html",
22   styleUrls: ["../cardcontent.component.scss"]
23 })
```

*Selector* defines how the *cardcontent* component is selected in its parent component. *templateUrl* defines the file containing the HTML content for this component and *styleUrls* specifies the files containing details how to style this component.

```

24 export class CardcontentComponent implements OnInit {
25   @Input()
26   showValidations: false;
27   searching = false;
28   searchFailed = false;
29   hideSearchingWhenUnsubscribed = new Observable(() => () =>
30     (this.searching = false)
31   );
32
33   addressForm: FormGroup;
34   cardAddress: FormControl;
35
36   showAddressOptions: boolean = false;
37   addresses: any[];
38   FormattedAddress: string = "";
39   selectedAddressObj: any = {};
40   enteredName: string = "";
41   firmName: string = "";
42   addressValid: number = 0;
43   getAddressDescription: string = "";
44   valAddrRes: any = {};
45   showWarning: boolean = false;
46   showSaved: boolean = false;
47

```

Each Angular component needs to export a class on initialization. Variables and their default values are set next. The search related fields and function deal with the showing and hiding of the spinner to the right of the *input* field when communicating with the server.

```

48   constructor(
49     private nameService: NameService,
50     private addressService: AddressService,
51     private appdataService: AppdataService
52   ) {
53     this.appdataService.newNaAddresses.subscribe(addresses => { ...
70   });
71
72     this.appdataService.newNaAddressSelected.subscribe(address => { ...
85   });
86   }
87
88   ngOnInit() {
89     this.createFormControls();
90     this.createForm();
91   }
92

```

The constructor initializes the services for use in the *cardcontent* component and upon initialization, subscribes to events emitted by these services. We will discuss these subscriptions in more detail later.

Upon initialization of this component, the *createFormControls* and *createForm* functions are called to create these. See below.

```

93     resetForm() {
94         this.showWarning = false;
95         this.showSaved = false;
96         this.valAddrRes = {};
97         this.addressValid = 0;
98         this.getAddressDescription = "";
99         this.firmName = "";
100        this.showAddressOptions = false;
101        this.appdataService.firmName = "";
102        this.appdataService.name = "";
103    }
104

```

The *resetForm* function is called to reset the variables to their original values when the form is reset.

```

105    normalizeName(event) {
106        if (event.target.value.length > 4) {
107            this.nameService.parseName(event.target.value);
108            this.nameService.normalizeName(event.target.value);
109        }
110    }
111

```

The *normalizeName* function is called when the text in the *Name input* field changes. If the length of the entered text is longer than 4 characters then two functions in the *nameService* component are called. The *parseName* service calls the Spectrum name parsing web service, requesting the name parts but not requesting normalization of the first name. The *normalizeName* service calls the same Spectrum name parsing web service, but requests the name parts including normalization of the first name.

```

105    normalizeName(event) {
106        if (event.target.value.length > 4) {
107            this.nameService.parseName(event.target.value);
108            this.nameService.normalizeName(event.target.value);
109        }
110    }
111

```

The *saveFirmName* function shares the entered Firmname with the *appdataService*, which in turn shares it with the *addressService* for inclusion in its *validateAddress* function and respective web service call to the *Address Resolution service*.

```

116     search = (text$: Observable<string>) => {
117         return text$
118             .debounceTime(300)
119             .distinctUntilChanged()
120             .filter(term => term.length > 0)
121             .do(() => (this.searching = true))
122             .switchMap(term =>
123                 this.addressService
124                     .getAddress("USA", term)
125                     .map((response: any) => {
126                         return response.addresses;
127                     })
128                     .do(addresses => {
129                         this.searchFailed = false;
130                         return of(addresses);
131                     })
132                     .catch(() => {
133                         this.searchFailed = true;
134                         return of([]);
135                     })
136             )
137             .do(() => (this.searching = false))
138             .merge(this.hideSearchingWhenUnsubscribed);
139     };
140

```

The *search* function is where the *AutoComplete/Type-Ahead-magic* starts to happen. An *observable* observes the text in the *address input* field. The *debounceTime* waits for a pause of 300 milliseconds in receiving text (the user typing) before proceeding. *distinctUntilChanged* holds off doing anything unless the text has changed since the last request was sent to the server. So in a scenario where a user types two characters and then deletes them since the last server request, then no new request is sent to the server. Likewise, the *filter* function prevents server requests when the *input* field is blank. Upon passing the previous filters, the Reactive JavaScript extensions prepare to send a request to the server and the *do* function sets the *searching* variable to *true*, causing the spinner to the right of the input field to show. The *switchMap* function unsubscribes from the previous *observable* and subscribes to the new *observable* for the web service request it is about to send. This is another best practice given to us by the Reactive JavaScript extensions. It prevents the UI from responding to lots of previous (possibly slow) server requests and only listens to the most recent one.

The *getAddress* function in the *addressService* is called with the country and the search term from the *input* field. The *map* function receives the *AutoComplete/Type-Ahead* web service response and returns just the *addresses* array from the response to the next *do* function. This *do* function switches any *searchFailed* messages off. The *return of (addresses)* returns an *observable* of the *addresses* for other functions to observe. The *catch* function is called if the web service request to *getAddress* fails and when called will switch the *searchFailed* messages on and return an empty array to indicate no matches are available for the user to pick. The *do* function on line 146 hides the spinner in the right of the address *input* field. The *merge* function calls a function that hides the search spinner whenever the UI code unsubscribes from an *observable*.

```

141 resultFormatter = result => {
142     let FormattedAddress = "";
143     result.AddressLine1
144     ? (FormattedAddress += result.AddressLine1 + ", ")
145     : (FormattedAddress += "");
146     result.AddressLine2
147     ? (FormattedAddress += result.AddressLine2 + ", ")
148     : (FormattedAddress += "");
149     result.AddressLine3
150     ? (FormattedAddress += result.AddressLine3 + ", ")
151     : (FormattedAddress += "");
152     result.AddressLine4
153     ? (FormattedAddress += result.AddressLine4 + ", ")
154     : (FormattedAddress += "");
155     result.City
156     ? (FormattedAddress += result.City + ", ")
157     : (FormattedAddress += "");
158     result.StateProvince
159     ? (FormattedAddress += result.StateProvince + ", ")
160     : (FormattedAddress += "");
161     result.PostalCode
162     ? (FormattedAddress += result.PostalCode + ", ")
163     : (FormattedAddress += "");
164     result.Country
165     ? (FormattedAddress += result.Country)
166     : (FormattedAddress += "");
167
168     return `${FormattedAddress}`;
169 };
170

```

The *resultFormatter* function constructs a *FormattedAddress* from the address parts of each address object in the returned array and returns this in the *AutoComplete/Type-Ahead* selection list in the UI.

```

171 selectItem = $event => {
172     $event.preventDefault();
173     // $event.item is the selected address object
174     this.selectedAddressObj = $event.item;
175     this.appdataService.setSelectedAddress(this.selectedAddressObj);
176     this.appdataService.addressSelected.emit(this.selectedAddressObj);
177     this.addressService.validateAddress(this.selectedAddressObj);
178 };
179

```

The *selectItem* function specifies what to do when a user selects one of the addresses displayed in the *AutoComplete/Type-Ahead* matches. The selected address object in the matches-array is accessed using *\$event.item*. This is then made available in the *appdataService* for other components to and it is also emitted as an event should other components need to take action when an address is selected. Finally, the *validateAddress* function is called with the *selectedAddressObject*.

```

180 = ..createFormControls() {
181 = ..  this.cardAddress = new FormControl("", [Validators.required]);
182 = ..}
183
184 = ..createForm() {
185 = ..  this.addressForm = new FormGroup({
186 = ..    cardAddress: this.cardAddress
187 = ..  });
188 = ..}
189
190 = ..ngOnChanges() {
191 = ..  if (this.showValidations) {
192 = ..    this.addressForm.controls["cardAddress"].markAsTouched();
193 = ..  } else if (this.showValidations === false) {
194 = ..    this.addressForm.controls["cardAddress"].markAsUntouched();
195 = ..  }
196 = ..}
197 }

```

The remaining lines of the *cardcontent* component's Typescript file create the form and form-controls and are boilerplate code taken from the PB Design System.

Now we will examine the *getAddress* and *validateAddress* functions in the *addressService* component.

```

1  import { Injectable } from "@angular/core";
2  import { HttpClient } from "@angular/common/http";
3  import { AppdataService } from "../appdata.service";
4  import { spectrumUrl } from "../config";
5

```

*Injectable* and *HttpClient* are standard node modules for dealing with web service requests. *AppdataService* is a service for sharing data between the different components and *spectrumUrl* is the base URL for accessing Spectrum. In the deployed version of the Angular app this will simply be "/" but during development of the Angular app it will not yet be deployed in Spectrum.

```

6  @Injectable()
7  export class AddressService {
8  ..  addresses: any;
9  ..  autoCompleteLoqate: any = {};
10 ..  queryString: string;
11 ..  validatedAddress: any = {};
12 ..  validatedAddresses: any = [];
13 ..  res: any;
14 ..  validatedFormattedAddress: string;
15

```

*@Injectable* is an Angular directive that makes this service *injectable* and therefore usable by other componets like the *cardcontent* component described earlier. After declaring the name of the class to export, we define the variable used by the class and their initial values.

```

16  constructor(
17  ..  private httpClient: HttpClient,
18  ..  private appdataService: AppdataService
19  .. ) {}
20

```

The *constructor* makes the *HttpClient* and *appdataService* available for use throughout the rest of the class.

```

21   getAddress(country: string, addressInput: string) {
22     return this.httpClient.get(
23       `${spectrumUrl}rest/AutoCompleteLoqate/results.json?Data.AddressLine1=${encodeURIComponent(
24         addressInput
25       )}&Option.HomeCountry=${country}`
26     );
27   }
28 }

```

The `getAddress` function is called with the *country* and the address from the *AutoComplete/Type-Ahead* address-input field. This returns an *observable* of the response of the web service call to Spectrum. The *observable* is further handled by the *search* function in the *cardcontent* component. Later we will look at how the NodeJS web server deals with this request.

```

29 = validateAddress(selectedAddress) {
30   this.queryString = `${spectrumUrl}rest/NameAddress_StandardizeValidateRecommend/results.json?`;
31   this.appdataService.getName()
32   ? (this.queryString +=
33     "Data.Name=" +
34     encodeURIComponent(this.appdataService.getName()) +
35     "&")
36   : (this.queryString += "");
37   this.appdataService.firmName
38   ? (this.queryString +=
39     "Data.FirmName=" +
40     encodeURIComponent(this.appdataService.firmName) +
41     "&")
42   : (this.queryString += "");
43   selectedAddress.AddressLine1
44   ? (this.queryString +=
45     "Data.AddressLine1=" +
46     encodeURIComponent(selectedAddress.AddressLine1) +
47     "&")
48   : (this.queryString += "");
49   selectedAddress.AddressLine2
50   ? (this.queryString +=
51     "Data.AddressLine2=" +
52     encodeURIComponent(selectedAddress.AddressLine2) +
53     "&")
54   : (this.queryString += "");
55   selectedAddress.City
56   ? (this.queryString +=
57     "Data.City=" + encodeURIComponent(selectedAddress.City) + "&")
58   : (this.queryString += "");
59   selectedAddress.StateProvince
60   ? (this.queryString +=
61     "Data.StateProvince=" +
62     encodeURIComponent(selectedAddress.StateProvince) +
63     "&")
64   : (this.queryString += "");
65   selectedAddress.PostalCode
66   ? (this.queryString +=
67     "Data.PostalCode=" +
68     encodeURIComponent(selectedAddress.PostalCode) +
69     "&")
70   : (this.queryString += "");
71   selectedAddress.Country
72   ? (this.queryString +=
73     "Data.Country=" + encodeURIComponent(selectedAddress.Country))
74   : (this.queryString += "");
75 }

```

The first part of the *validateAddress* function builds the *Address Resolution* service Query string required to call this Spectrum flow. It does this by getting the *Name* and *Firmname* values captured in the form and shared via the *appdataService*. It then adds the address components in the selected address item from the selected *AutoComplete/Type-Ahead* response to the query string. The syntax checks if each field has a value and if it has, adds it to the query string and if it does not then the query string remains unchanged.



```

76     ...return this.httpClient
77     ...    .get(`${this.queryString}`)
78     ...    .subscribe((validatedAddress: any) => {
79     ...        this.validatedAddresses = validatedAddress.Output;
80     ...        let validatedAddressesComplete = [];
81
82     ...        this.validatedAddresses.map(validatedAddress => {
83     ...            let validatedFormattedAddress = "";
84     ...            validatedAddress.AddressLine1
85     ...            ? (validatedFormattedAddress +=
86     ...                validatedAddress.AddressLine1 + ", ")
87     ...            : (validatedFormattedAddress += "");
88     ...            validatedAddress.City
89     ...            ? (validatedFormattedAddress += validatedAddress.City + ", ")
90     ...            : (validatedFormattedAddress += "");
91     ...            validatedAddress.StateProvince
92     ...            ? (validatedFormattedAddress +=
93     ...                validatedAddress.StateProvince + ", ")
94     ...            : (validatedFormattedAddress += "");
95     ...            validatedAddress.PostalCode
96     ...            ? (validatedFormattedAddress += validatedAddress.PostalCode + ", ")
97     ...            : (validatedFormattedAddress += "");
98     ...            validatedAddress.Country
99     ...            ? (validatedFormattedAddress += validatedAddress.Country)
100     ...            : (validatedFormattedAddress += "");
101
102     ...            validatedAddress.FormattedAddress = validatedFormattedAddress;
103     ...            validatedAddressesComplete.push(validatedAddress);
104     ...        });
105
106     ...        this.appdataService.newNaAddresses.emit(validatedAddressesComplete);
107
108     ...        this.appdataService.setNaAddress(validatedAddressesComplete[0]);
109     ...    });
110 }
111 }

```

The second part of the *validateAddress* function returns an *observer* for the response of the web service call to the *Address Resolution Spectrum* flow. The input parameters to this web service call *Name*, *Firmname* and *Address Components*. We immediately *subscribe* to this *observer* and when we get the web service response (see Appendix A for the response format). Lines 82 to 104 enrich each address object in the response array with a formatted address.

Line 106 *emits* the enhanced response for other components to use. Finally, we store the first returned address from the address validation in the *appdataService* for other components to use. In the case of the *Address Resolution* flow having validated the address, or auto-corrected it then the returned array contains a single element. If the address cannot be validated as deliverable then the first address in the response array is the non-deliverable address including codes and a message why the address is not deliverable. In this, last scenario the subsequent addresses in the response-array are validated address, for example with apartment numbers added and intended for presentment to the user for selection.

Now, let us look at how the *cardcontent* component subscribes and processes the returned addresses.



```

48     constructor(
49         private nameService: NameService,
50         private addressService: AddressService,
51         private appdataService: AppdataService
52     ) {
53         this.appdataService.newNaAddresses.subscribe(addresses => {
54             const address = addresses[0];
55             if (addresses.length > 1) {
56                 this.showAddressOptions = true;
57                 this.addressForm.patchValue({
58                     cardAddress: address.FormattedAddress
59                 });
60             } else {
61                 this.showAddressOptions = false;
62                 if (address.FirmName) {
63                     this.firmName = address.FirmName;
64                 }
65                 if (address.Name) {
66                     this.enteredName = address.Name;
67                 }
68                 this.appdataService.newNaAddressSelected.emit(address);
69             }
70         });
71
72         this.appdataService.newNaAddressSelected.subscribe(address => {
73             this.showAddressOptions = false;
74             this.addressForm.patchValue({
75                 cardAddress: address.FormattedAddress
76             });
77             this.FormattedAddress = address.FormattedAddress;
78             this.appdataService.setNaAddress(address);
79             if (address.DPV == "Y") {
80                 this.addressValid = 1;
81             } else {
82                 this.addressValid = 2;
83             }
84             this.getAddressDescription = address.Message;
85         });
86     }
87

```

In line 53 of *cardcontent* component's Typescript we are subscribing to the *newNaAddresses* event emitted by the *validateAddress* function of the *AddressService*. We set the variable *address* to the first address in the array returned from *validateAddress*. If the array contains more than one address then we want to show the *showAddressOptions* component in the UI for the user to choose the correct address. We also *patch* the formatted address of this address object into the web form's address *input* field. If the response only contains one address, we hide the *showAddressOptions* component from the UI. Next, we see if the *validateAddress* response contains a Firmname and Name and if present, populate these into these respective fields in the UI.

Line 68 emits the address so that the *newNaAddressSelected* subscription below receives the address and can update the UI accordingly. This *emit* / *subscribe* programming pattern is used so that the *newNaAddressSelected* subscription can deal with both single returned addresses (from the *Address Resolution Spectrum* web service) as well as when a user selects one of the address options in the *addressoptions* component.

Line 72 listens for when a new address of multiple validated address options is selected by the user or for when the *validateAddress* function returns a single address. Line 73 hides the *addressoptions* component because the final validated address is now ready for display.

The *FormattedAddress* field's value is patched into the address *input* field in the HTML. Next it is stored in the component's state and in the *appdataService* for use by other components. Finally, we set the checkmark or error symbols in the address *input* field's *label* and set the *getAddressDescription*, which displays on hovering over the symbol to the *Message* field value of the address object of the *validateAddress* response-array.

Let us now look at the *addressoptions* component. The *addressoptions* component shows validated address options for the user to select when the originally selected address in the *AutoComplete/Type-Ahead* is incomplete. It is responsible for the highlighted content below.



The screenshot shows a web form with an "Address" label and a text input field containing "350 Third Ave, Chula Vista, 91910-3900, United States Of America". Below the input field, a dropdown menu is open, displaying a warning message and three address suggestions. The dropdown menu is highlighted with a red border. The warning message states: "It might not be possible to deliver packages & mail to the address you selected. Did you mean to enter one of the following addresses? Select your correct address below to ensure that packages & mail get delivered." The three suggestions are: "350 Third Ave Apt 101, Chula Vista, 91910-3957, United States of America", "350 Third Ave Apt 104, Chula Vista, 91910-3957, United States of America", and "350 Third Ave Ofc, Chula Vista, 91910-3966, United States of America". Each suggestion is preceded by a green checkmark and a radio button.

Address

350 Third Ave, Chula Vista, 91910-3900, United States Of America

⚠ It might not be possible to deliver packages & mail to the address you selected. Did you mean to enter one of the following addresses? Select your correct address below to ensure that packages & mail get delivered.

- ☐ ✓ 350 Third Ave Apt 101, Chula Vista, 91910-3957, United States of America
- ☐ ✓ 350 Third Ave Apt 104, Chula Vista, 91910-3957, United States of America
- ☐ ✓ 350 Third Ave Ofc, Chula Vista, 91910-3966, United States of America

Below is the HTML for the *addressoptions* component.

```

1  <p>
2  <i
3  <class="pbi-icon-outline pbi-alert-circle"
4  <container="body"
5  <placement="right"
6  <ngbPopover="{{ address?.Message }}"
7  <triggers="mouseenter:mouseleave"
8  ></i>
9  >&nbsp;It might not be possible to deliver packages & mail to the address
10 >you selected. Did you mean to enter one of the following addresses? Select
11 >your correct address below to ensure that packages & mail get delivered.
12 </p>
13 <div
14 <class="custom-control custom-checkbox"
15 <*ngFor="let address of addressOptions"
16 >
17 <input
18 <type="checkbox"
19 <class="custom-control-input"
20 <id="{{ address?.id }}"
21 <(change)="changeAddress(address?.index)"
22 </>
23 <label class="custom-control-label" for="{{ address?.id }}"
24 <><i
25 <ngIf="address?.DPV != 'Y'"
26 <class="pbi-icon-outline pbi-alert-circle"
27 <container="body"
28 <placement="right"
29 <ngbPopover="{{ address?.Message }}"
30 <triggers="mouseenter:mouseleave"
31 ></i>
32 <i
33 <ngIf="address?.DPV == 'Y'"
34 <class="pbi-icon-mini pbi-check-bold"
35 <container="body"
36 <placement="right"
37 <ngbPopover="{{ address?.Message }}"
38 <triggers="mouseenter:mouseleave"
39 <style="color: greenyellow;"
40 ></i>
41 >&nbsp;{{ address?.FormattedAddress }}</label>
42 >
43 </div>

```

First, we display a paragraph explaining that the address selected from the *AutoComplete/Type-Ahead* options is not a deliverable address. Preceding the paragraph is a warning icon and a popup message that appears when the user hovers over this icon. Next, we iterate over the *addressOptions* returned from the web service call to the *Address Resolution Spectrum* flow and output a checkbox item for each address, except the first address, which we write into the address *input* field. Each checkbox is followed by a symbol indicating whether the address has been validated as deliverable. This, along with a descriptive message comes from the response to the *Address Resolution Spectrum* web service call. See Appendix A for details about the structure of the response.

Below is the associated Typescript containing the logic for the *addressoptions* component.

```

1  import { Component, OnInit } from "@angular/core";
2  import { AppdataService } from "../services/appdata.service";
3
4  @Component({
5    selector: "app-addressoptions",
6    templateUrl: "../addressoptions.component.html",
7    styleUrls: ["../addressoptions.component.scss"]
8  })
9  export class AddressoptionsComponent implements OnInit {
10    addressOptions: any = [];
11
12    constructor(private appdataService: AppdataService) {
13      this.appdataService.newNaAddresses.subscribe(addresses => {
14        if (addresses.length > 1) {
15          this.addressOptions = [];
16          let i = -1;
17          addresses.map(address => {
18            address.index = i;
19            address.id = `addressOption${i}`;
20            this.addressOptions.push(address);
21            i++;
22          });
23          this.addressOptions.shift();
24        }
25      });
26    }
27
28    changeAddress(i) {
29      const address = this.addressOptions[i];
30      if (address.FirmName) {
31        this.appdataService.enteredFirmName.emit(address.FirmName);
32      }
33      if (address.Name) {
34        this.appdataService.enteredName.emit(address.Name);
35      }
36      this.appdataService.newNaAddressSelected.emit(address);
37    }
38    ngOnInit() {}
39  }

```

We start by importing the standard Angular directives for any visual component along with *AppdataService* that we use to share data across the different components.

The *selector* specifies how to select this component its parent components, the *templateUrl* points to the HTML for the component and *styleUrls* to the styling.

*addressOptions* is initialized as an empty array on line 10 to prevent errors when no address options have been returned from the *Address Resolution* web service yet.

The *constructor* initializes the *appdataService* and makes it available for use throughout the component. On initialization, this component also subscribes to the *newNaAddresses* event emitter, which emits the response to the Accurate Address Spectrum web service call. Unlike, the *cardcontent* component which is interested in the length of this response to know whether to show the *addressoptions* component or not, the *addressoptions* component is interested in the length in order to decide if it needs to prepare any content for display! If there is more than one *addressOption* returned, we add an *index* and *id* to each address object in the array so that we can

easily associate *labels* with the *checkbox inputs* in the HTML and the HTML can easily communicate with the Typescript what address is selected. Line 23 removes the first item from the array of returned addresses because the *cardcontent* component deals with displaying this and the *addressoptions* component only needs to concern itself with the subsequent validated addresses and present these for selection.

On selecting a *checkbox* in the HTML, the *changeAddress* function in the Typescript is called with the index of the selected address. The *changeAddress* function then proceeds with emitting the Firmname and Name from the response so that the *cardcontent* component can subscribe to these updates and update them in the HTML of the *cardcontent* component. Finally, the *changeAddress* function emits the newly selected address object. It is the *cardcontent* component, discussed earlier which subscribes to this event and handles it further including hiding this *addressoptions* component and setting the value in its address *input* to the *FormattedAddress* value of the selected address.

We will now look at the *AppdataService* component that we use to share data and updates between the different components of the Angular application.

```
1  import { EventEmitter } from "@angular/core";
2
3  export class AppdataService {
4    selectedAddress: any = {};
5    validatedAddress: any = {};
6    name: string = "";
7    nameObj: any = {};
8    address: string = "";
9    naAddress: any = {};
10   firmName: string = "";
11 }
```

The *EventEmitter* is imported so that the *AppdataService* can manage event emitters that are used via the *emit* and *subscribe* methods in the different components to notify each other of changes to data. Next, we define the variables used in this component and initialize their values.

```

12     setSelectedAddress(address) {
13         this.selectedAddress = address;
14     }
15
16     getSelectedAddress() {
17         return this.selectedAddress;
18     }
19
20     setValidatedAddress(key, value) {
21         this.validatedAddress[key] = value;
22     }
23
24     getValidatedAddress(key) {
25         return this.validatedAddress[key];
26     }
27
28     setName(name) {
29         this.name = name;
30     }
31
32     getName() {
33         return this.name;
34     }
35
36     setNameObj(type, nameObj) {
37         // type can be parsed or normalized
38         this.nameObj[type] = nameObj;
39     }
40
41     getNameObj() {
42         return this.nameObj;
43     }
44
45     setAddress(address) {
46         this.address = address;
47     }
48
49     getAddress() {
50         return this.address;
51     }
52
53     setNaAddress(naAddress) {
54         this.naAddress = naAddress;
55     }
56
57     getNaAddress() {
58         return this.naAddress;
59     }
60

```

The above functions allow the different Angular components to set and get the variable values and thereby share them.

```

61 addressSelected = new EventEmitter<{}>();
62 enteredAddressChanged = new EventEmitter<string>();
63 newFormattedValidatedAddr = new EventEmitter<string>();
64 selectedValidatedAddress = new EventEmitter<{}>();
65 newNaAddresses = new EventEmitter<{}>();
66 newNaAddressSelected = new EventEmitter<{}>();
67 enteredName = new EventEmitter<string>();
68 enteredFirmName = new EventEmitter<string>();
69 parsedName = new EventEmitter<{}>();
70 normalizedName = new EventEmitter<{}>();
71 }

```

The *EventEmitters* allow the different Angular components to *emit* and *subscribe* to events on the named channels. Some of the *EventEmitters* emit *string* values and others *JavaScript Objects* as *JSON*. It is also possible to share *boolean* values using this technique.

The Angular application uses the *app.module.ts* file to define all the components that make up the app. This file imports and initializes the components and services used by the application. The one used for the demo can be previewed below.

```

1 import { BrowserModule } from "@angular/platform-browser";
2 import { NgModule } from "@angular/core";
3 import { FormsModule, ReactiveFormsModule } from "@angular/forms";
4 import { CommonModule } from "@angular/common";
5 import { HttpClientModule } from "@angular/common/http";
6
7 import { NgbModule } from "@ng-bootstrap/ng-bootstrap";
8
9 import { AppRoutingModule } from "../app-routing.module";
10 import { AppComponent } from "../app.component";
11 import { NameService } from "../services/name.service";
12 import { AddressService } from "../services/address.service";
13 import { AppdataService } from "../services/appdata.service";
14 import { FormControlValueDirective } from "../directives/form.directive";
15 import { AddressoptionsComponent } from "../addressoptions/addressoptions.component";
16 import { CardcontentComponent } from "../cardcontent/cardcontent.component";
17
18 @NgModule({
19   declarations: [
20     AppComponent,
21     FormControlValueDirective,
22     AddressoptionsComponent,
23     CardcontentComponent
24   ],
25   imports: [
26     BrowserModule,
27     CommonModule,
28     FormsModule,
29     ReactiveFormsModule,
30     NgbModule.forRoot(),
31     AppRoutingModule,
32     HttpClientModule
33   ],
34   providers: [NameService, AddressService, AppdataService],
35   bootstrap: [AppComponent],
36   entryComponents: [],
37   exports: [AppComponent]
38 })
39 export class AppModule {}

```

Line 37 in *app.module.ts* exports the *AppComponent*, the entry point into the Angular app.

The *AppComponent*'s Typescript is below.



```

1 import { Component } from "@angular/core";
2
3 @Component({
4   selector: "app-root",
5   templateUrl: "../app.component.html",
6   styleUrls: ["../app.component.scss"]
7 })
8 export class AppComponent {
9   title = "app";
10 }

```

The *selector* is how the *index.html* file connects to the Angular app.

```

1 <!doctype html>
2 <html lang="en">
3
4 <head>
5   <meta charset="utf-8">
6   <title>Retail CI</title>
7   <base href="/">
8
9   <meta name="viewport" content="width=device-width, initial-scale=1">
10  <link rel="icon" type="image/png" href="assets/favicon/favicon_16.png" sizes="16x16">
11  <link rel="icon" type="image/png" href="assets/favicon/favicon_32.png" sizes="32x32">
12  <link rel="icon" type="image/png" href="assets/favicon/favicon_57.png" sizes="57x57">
13  <link rel="icon" type="image/png" href="assets/favicon/favicon_72.png" sizes="72x72">
14  <link rel="icon" type="image/png" href="assets/favicon/favicon_96.png" sizes="96x96">
15  <link rel="icon" type="image/png" href="assets/favicon/favicon_120.png" sizes="120x120">
16  <link rel="icon" type="image/png" href="assets/favicon/favicon_128.png" sizes="128x128">
17  <link rel="icon" type="image/png" href="assets/favicon/favicon_144.png" sizes="144x144">
18  <link rel="icon" type="image/png" href="assets/favicon/favicon_152.png" sizes="152x152">
19  <link rel="icon" type="image/png" href="assets/favicon/favicon_195.png" sizes="195x195">
20  <link rel="icon" type="image/png" href="assets/favicon/favicon_228.png" sizes="228x228">
21  <link rel="apple-touch-icon" sizes="57x57" href="assets/favicon/favicon_57.png" />
22  <link rel="apple-touch-icon" sizes="114x114" href="assets/favicon/favicon_114.png" />
23  <link rel="apple-touch-icon" sizes="72x72" href="assets/favicon/favicon_72.png" />
24  <link rel="apple-touch-icon" sizes="144x144" href="assets/favicon/favicon_144.png" />
25  <link rel="mask-icon" href="assets/favicon/pb_icon_16.svg" color="#314183">
26 </head>
27
28 <body>
29   <app-root class="site-root"></app-root>
30 </body>
31
32 </html>

```

Finally, we will look at how the *AppComponent*'s HTML links the subsequent *cardcontent* component for the rest of the demo:

```

1 <app-cardcontent></app-cardcontent>

```

## Building the Angular App and deploying it into Spectrum

If you adapt this sample Angular app for your own use, you will need to build a deployable version when you have finished your updates.

```

1 // export const spectrumUrl = "http://54.246.236.11:8181/";
2 export const spectrumUrl = "/";

```

In *config.ts*, comment out line 1 and uncomment line 2. Once deployed in Spectrum, your Angular app can access Spectrum web services at a relative path to its deployed location.



```

5 export const environment = {
6   production: true
7 };

```

Set *production: true* in the *environment.ts* file.

From a command prompt in the root of your Angular app run the following command:

*ng build*

When the build completes, the production Angular app will be in the */dist/ci5* folder relative to the root of your Angular app.

Open */dist/ci5/index.html* and add a "." In line 6. This tells the Angular app to access CSS, JS, Images and other assets at a path relative to the *index.html* file.

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8" />
5     <title>Accurate Addressing</title>
6     <base href="." />
7

```

Next, we need to create a Web Archive (WAR) for deployment into Spectrum's Java application server. Follow the steps below:

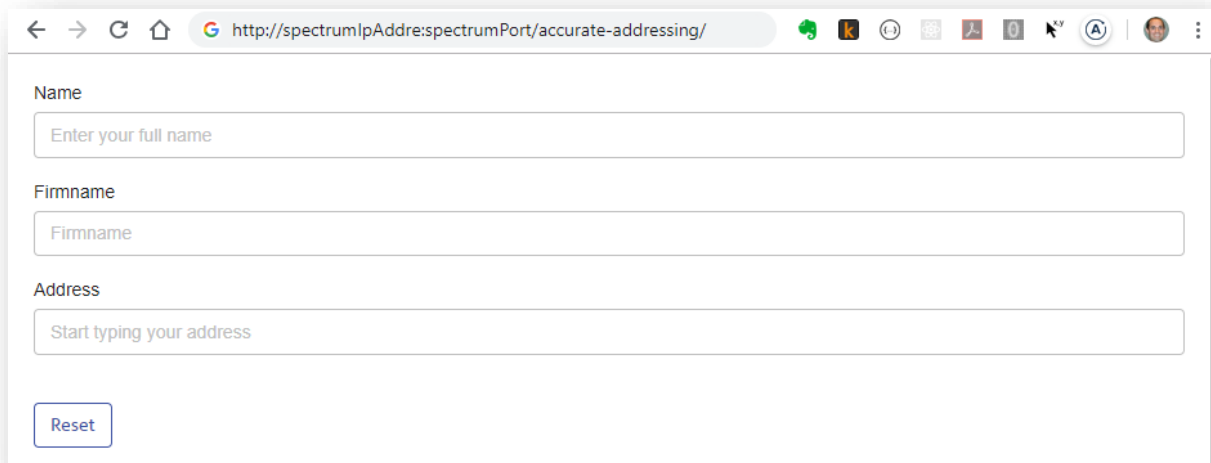
1. Create a folder called *accurate-addressing*.
2. Copy the contents of your */dist/ci5* folder including the updated *index.html* into this folder.
3. Add a subfolder called *WEB-INF*.
4. In the *WEB-INF* folder, create a new file called *web.xml* and paste the following content into the file (you can download a copy of *web.xml* from [here](#)):

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app version="2.4"
3   xmlns="http://java.sun.com/xml/ns/j2ee"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" >
6
7   <display-name>Accurate Addressing User Interface</display-name>
8   <context-param>
9     <param-name>unsecuredPatterns</param-name>
10    <param-value>/**</param-value>
11  </context-param>
12
13  <servlet-mapping>
14    <servlet-name>default</servlet-name>
15    <url-pattern>/*</url-pattern>
16  </servlet-mapping>
17
18 </web-app>

```

5. Zip up the contents of the *accurate-addressing* folder (do not Zip the folder itself but only its contents)
6. Rename the Zip file *accurate-addressing.war*.
7. Copy and paste *accurate-addressing.war* into the following Spectrum folder:  
*D:\Program Files\Pitney Bowes\Spectrum\server\app\deploy*
8. You will be able to access the Accurate Addressing sample application at  
<http://<spectrumIpAddress>:<spectrumPort>/accurate-addressing/> and the user interface will look like this:



Note that the *Name* and *AutoComplete/Type-Ahead* address entry might not work yet due to some security issues. These security issues can be resolved in several ways, three ways described in the next section. You can also combine these techniques.

### Security Settings in Spectrum to Enable Accurate Addressing

#### Option 1 – Disable Web Service Authentication in Spectrum

You can disable Basic Authentication for web services by following [these](#) instructions in the [Spectrum Administration Guide](#).

You will also need to disable authentication for web services by following [these](#) instructions in the [Spectrum Administration Guide](#).

Your Angular app will not yet be deployed inside the Spectrum web application server during development. Therefore, you will need to configure CORS (Cross Origin Resource Sharing) in Spectrum during development. Detailed instructions are in the [Spectrum Administration Guide](#). By default, an Angular development environment serves web pages on <http://localhost:4200>, so you will need to add *localhost:4200* to the *spectrum.jetty.cors.allowedOrigins* setting in *spectrum-advanced.properties* as indicated below. You will also need to set *spectrum.jetty.cors.enabled=true* in this same file. Note that once you have deployed your Angular app to Spectrum then you can remove the CORS settings. It is only required while you are developing your Angular app outside of Spectrum.

```
spectrum.jetty.cors.enabled=true
spectrum.jetty.cors.allowedOrigins=http://localhost:8080,http://localhost:443,http://localhost:4200
spectrum.jetty.cors.allowedMethods=POST,GET,OPTIONS,PUT,DELETE,HEAD
spectrum.jetty.cors.allowedHeaders=X-PINGOTHER, Origin, X-Requested-With, Content-Type, Accept
spectrum.jetty.cors.preflightMaxAge=1800
spectrum.jetty.cors.allowCredentials=true
```

#### Option 2 – Include Basic Authentication in the Web Service Calls to Spectrum from the Angular App

Include Basic Authentication for the web service call your Angular app makes to Spectrum by following these steps below.

#### Important Note

During development of the Angular app, you must disable authentication as outlined in [Option 1](#)

[above](#) and not include the `{ headers }` setting in `name.service.ts` and `address.service.ts`. Even with *CORS* enabled in Spectrum, web browsers will throw a CORS error and block access to resources on a different domain, if the Angular web app attempts to include an authentication header in its request to a different domain. You **will** need to apply these authentication and `{ headers }` settings to your app prior to building it and deploying it as a WAR to Spectrum as outlined [above](#) if the Spectrum instance to which you deploy the WAR has *Basic Authentication* enabled for its web services.

Update the `user` and `password` in the `config.ts` file with the user credentials you want to use for accessing the Accurate Addressing web services. It is advisable to create a new Spectrum user that only has access to these two web services. That is because the user credential are in the client side Angular app and therefore not secure. This is also the reason many organization use [Option 3](#) below so that no user credentials get share with the end-user's web browser.

```
1 // export const spectrumUrl = "http://54.246.236.11:8181/";
2 export const spectrumUrl = "/";
3
4 const user = "admin";
5 const password = "admin";
6 const authString = `${user}:${password}`;
7
8 export const Authorization = `Basic ${btoa(authString)}`;
```

Next, you need to make some minor changes to your `name.service.ts` and `address.service.ts` files to make them include the authentication credential in the web service calls they perform. The required updated are highlighted below.

name.service.ts

```
1 import { Injectable } from "@angular/core";
2 import { HttpClient, HttpHeaders } from "@angular/common/http";
3 import { AppdataService } from "../appdata.service";
4 import { spectrumUrl } from "../config";
5 import { Authorization } from "../config";
6 const headers = new HttpHeaders().append("Authorization", Authorization);
7
8 @Injectable()
9 export class NameService {
10   --parsedName: any = {};
11   --normalizedName: any = {};
12   --res: any;
13   --Output: any;
14
15   --constructor(
16   --  private httpClient: HttpClient,
17   --  private appdataService: AppdataService
18   --) {}
19
20   parseName(name: string) {
21     -- this.appdataService.enteredName.emit(name);
22     -- return this.httpClient
23     -- .get(
24     --   `${spectrumUrl}rest/NameParsingandCorrection/results.json?Data.Name=${encodeURIComponent(
25     --     name
26     --   )}&Data.CorrectionFlag=false`,
27     --   { headers }
28     -- )
29     -- .subscribe((parsedName: any) => {
30     --   this.parsedName = parsedName;
31     --   this.appdataService.parsedName.emit(this.parsedName);
32     --   this.appdataService.setNameObj("parsed", this.parsedName.Output[0]);
33     -- });
34   }
35
36   normalizeName(name: string) {
37     -- return this.httpClient
38     -- .get(
39     --   `${spectrumUrl}rest/NameParsingandCorrection/results.json?Data.Name=${encodeURIComponent(
40     --     name
41     --   )}&Data.CorrectionFlag=true`,
42     --   { headers }
43     -- )
44     -- .subscribe(normalizedName => {
45     --   this.normalizedName = normalizedName;
46     --   this.appdataService.normalizedName.emit(this.normalizedName);
47     --   this.appdataService.setName(this.normalizedName.Output[0].Name);
48     --   this.appdataService.setNameObj(
49     --     "normalized",
50     --     this.normalizedName.Output[0]
51     --   );
52     -- });
53   }
54 }
```

address.service.ts

```
1 import { Injectable } from "@angular/core";
2 import { HttpClient, HttpHeaders } from "@angular/common/http";
3 import { AppdataService } from "../appdata.service";
4 import { spectrumUrl } from "../config";
5 import { Authorization } from "../config";
6 const headers = new HttpHeaders().append("Authorization", Authorization);
7
8 @Injectable()
9 export class AddressService {
10   addresses: any;
11   autoCompleteLoqate: any = {};
12   queryString: string;
13   validatedAddress: any = {};
14   validatedAddresses: any = [];
15   res: any;
16   validatedFormattedAddress: string;
17
18   constructor(
19     private httpClient: HttpClient,
20     private appdataService: AppdataService
21   ) {}
22
23   getAddress(country: string, addressInput: string) {
24     return this.httpClient.get(
25       `${spectrumUrl}rest/AutoCompleteLoqate/results.json?Data.AddressLine1=${encodeURIComponent(
26         addressInput
27       )}&Option.HomeCountry=${country}`,
28       { headers }
29     );
30   }
31 }
```

and

```
79 return this.httpClient
80   .get(`${this.queryString}`, { headers })
81   .subscribe((validatedAddress: any) => {
82     this.validatedAddresses = validatedAddress.Output;
83     let validatedAddressesComplete = [];
84   });
```

After the authentication updates you will need to build it for production again and redeploy it to Spectrum by following the steps described [above](#).

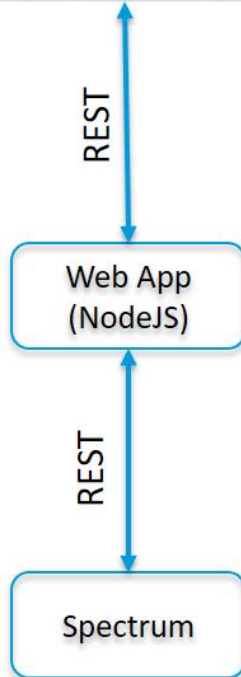
### Option 3 – Host the Angular App on a Separate Web Server to Spectrum

In this scenario, the Angular app is not deployed to Spectrum but on a separate web server. That web server serves the Angular app to the end user's browser and relays web service requests to Spectrum, adding authentication credentials in the process. A sample NodeJS / ExpressJS web server implementation along with an appropriately adjust Angular app is available on request.

### Example of deploying Accurate Addressing UI on separate web server



Web browser capturing and validating customer address and other contact details using Accurate Addressing solution. Angular application using PB Design System.



Web application hides Spectrum login credentials from the web and sets cross origin headers if required. Uses ExpressJS a NodeJS web server. Serves the Angular web application to the browser and relays the REST web service requests to Spectrum.

Spectrum with UAM Loqate, Universal Address Module, DPV, LACSLink and SuiteLink DBs. Accurate Addressing Spectrum Flow.

## Appendix A – Web Service Details

### Shape of *AutoComplete/Type-Ahead* Web Service

Called as the user is inputting their address.

#### Request

/rest/AutoCompleteLoqate/results.json?Data.AddressLine1=350%203rd%20Ave%20chu&Option.HomeCountry=USA

## Response

```
{
  "addresses": [
    {
      "ProcessedBy": "LOQATE",
      "HouseNumber": "350",
      "AddressLine1": "350 3rd Ave",
      "AddressLine2": "",
      "AddressLine3": "",
      "AddressLine4": "",
      "FirmName": "",
      "City": "Chula Vista",
      "StateProvince": "CA",
      "PostalCode": "91910-3900",
      "PostalCode.AddOn": "3900",
      "Country": "United States",
      "ApartmentLabel": "",
      "user_fields": [],
      "FormattedAddress": "350 3rd Ave, Chula Vista, CA, 91910-3900, United States"
    },
    {
      "ProcessedBy": "LOQATE",
      "HouseNumber": "350",
      "AddressLine1": "350 3rd Ave",
      "AddressLine2": "",
      "AddressLine3": "",
      "AddressLine4": "",
      "FirmName": "",
      "City": "Chula Vista",
      "StateProvince": "CA",

```

## Shape of *Address Resolution* Web Service

Called when the user selects on address from the list returned from the *AutoComplete/Type-Ahead* service.

## Request

/rest/NameAddress\_StandardizeValidateRecommend/results.json?Data.AddressLine1=350%203rd%20Ave&Data.City=Chula%20Vista&Data.StateProvince=CA&Data.PostalCode=91910

## Response

```
{
  "Output": [
    {
      "Name": "",
      "AddressLine1": "350 Third Ave",
      "City": "Chula Vista",
      "StateProvince": "CA",
      "PostalCode": "91910-3900",
      "Country": "United States Of America",
      "Latitude": "32.639440",
      "Longitude": "-117.081051",
      "LocationCode": "AP02",
      "MatchCode": "S80",
      "CMRA": "U",
      "DPV": "D",
      "DPVFootnote": "AAN1",
      "DPVNoStat": "Y",
      "DPVVacant": "N",
      "USLACS.ReturnCode": "",
      "RDI": "R",
      "SuiteLinkReturnCode": "",
      "Message": "Missing Apt/Suite Number",
      "Status.Code": "UnableToDPVConfirm",
      "Status.Description": "Missing Apt/Suite Number",
      "user_fields": []
    },
    {
      "FirmName": "",
      "Name": "",
      "AddressLine1": "350 Third Ave Apt 101",
      "City": "Chula Vista",
      "StateProvince": "CA",
      "PostalCode": "91910-3957",
      "Country": "United States of America",
      "Latitude": "32.639440",
      "Longitude": "-117.081051",
      "LocationCode": "AP02",
      "MatchCode": "A80",
      "CMRA": "N"
    }
  ]
}
```

## Appendix B – Download Locations

### Sample Angular App using admin/admin credentials

A sample web archive (WAR) file that works on a Spectrum instance with the default user credentials *admin/admin*, can be downloaded [here](#). Copy and paste *accurate-addressing.war* into the following Spectrum folder:

*D:\Program Files\Pitney Bowes\Spectrum\server\app\deploy*

You will be able to access the Accurate Addressing sample application at <http://localhost:8080/accurate-addressing/>

The source code for this Angular app is available [here](#).



### Sample Angular App using no login credentials

A sample web archive (WAR) file that works on a Spectrum instance with authentication disabled is available [here](#). Copy and paste *accurate-addressing.war* into the following Spectrum folder:

*D:\Program Files\Pitney Bowes\Spectrum\server\app\deploy*

Also, follow the steps described in the [above](#) section to disable web service authentication on your Spectrum instance. Sample [spectrum-advanced.properties](#) and [spectrum-container.properties](#) with web service authentication disabled and CORS enabled for the default Angular development host are also available for download.

You will be able to access the Accurate Addressing sample application at

<http://localhost:8080/accurate-addressing/>

### Advanced Angular App using admin/admin credentials

A more advanced Angular app that supports Accurate Addressing for international addresses and includes a *Details* button to view the actual web service requests and responses being exchanged with the Spectrum server can be downloaded from [here](#).

Copy and paste *accurate-addressing-plus.war* into the following Spectrum folder:

*D:\Program Files\Pitney Bowes\Spectrum\server\app\deploy*

You will be able to access the Accurate Addressing sample application at

<http://localhost:8080/accurate-addressing-plus/>