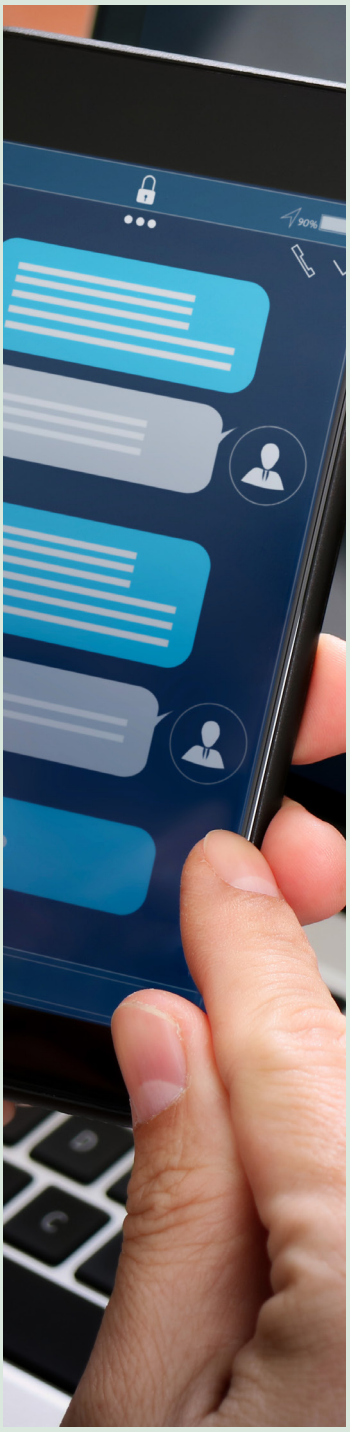
WHITE PAPER



LEVERAGING MICROFRONTENDS FOR BUILDING FLEXIBLE AND INNOVATIVE USER INTERFACES

Infosys®
Navigate your next

Table of Contents

Flexible user interfaces

User interfaces of modern applications typically tend to be one large monolith that caters to the business requirements of that specific application. However, such large monoliths rarely lend themselves to easy extension.

Modern applications need to be flexible and easily extensible. In this regard there is a benefit for the User Interface of modern applications to be designed to be as modular and independent as possible. Having application User Interface composed of several pieces, each handling a specific functionality, allows for ease of extensibility as well as maintenance in that a specific piece can be either enhanced or completely switched out and replaced with another without having significant impact on the remaining application functionality.

Functional example of flexible user interfaces

Let us illustrate such flexible user interface design with the help of a functional use-case of a modern contact center. Modern contact centers are meant to deliver value and ensure excellent customer-experiences, to the customers of today's enterprises. Customers nowadays would expect a contact center agent to know beforehand about the history of their past interactions, the nature of issues they have faced, their preferences, etc. The contact center agent often must use multiple enterprise applications to fetch all the information about the customer to ensure she is able to deliver the experience that the customer may have come to expect.

Today's contact centers employ a wide variety of tools to assist the agents deliver the value demanded by the customer. Any tool, which may be introduced into a contact center, must be capable of delivering immediate value while minimizing disruption to existing processes that the agents may be accustomed to.

Tools must be able to deliver assistance contextually, while still not getting in the way. Any tool that is constructed with a one-size-fits-all paradigm is bound to fail within contact centers that already have a suite of applications being employed. The reason for the failure has little do with the quality of the tool itself, but with the quantum of change in existing processes that the tool may entail.

In contrast, a tool that can plug into existing systems in a seamless way and provide incremental value to the agents will see better uptake and better success among agents who typically have little time to climb a steep learning curve that a completely new system or tool would demand.

Hence it becomes imperative for solution providers, who are aiming to deliver value-systems to contact center agents, to devise solutions which employ an extremely flexible user interface capable of plugging into any systems that the contact centers already employ.

Examples of Contact Center tools

To illustrate how such flexibility and ability to plug-in user interface elements may be beneficial, let's consider a few examples:

Toolbars

Agent tools that are built to be plugged into any existing application must be organized to enable easy access while still being as non-intrusive and non-disruptive to the agent. Simple toolbars that stay away from the main field of view and only come into play when accessed by the agent are useful mechanisms to embed multiple tools into the agent application.

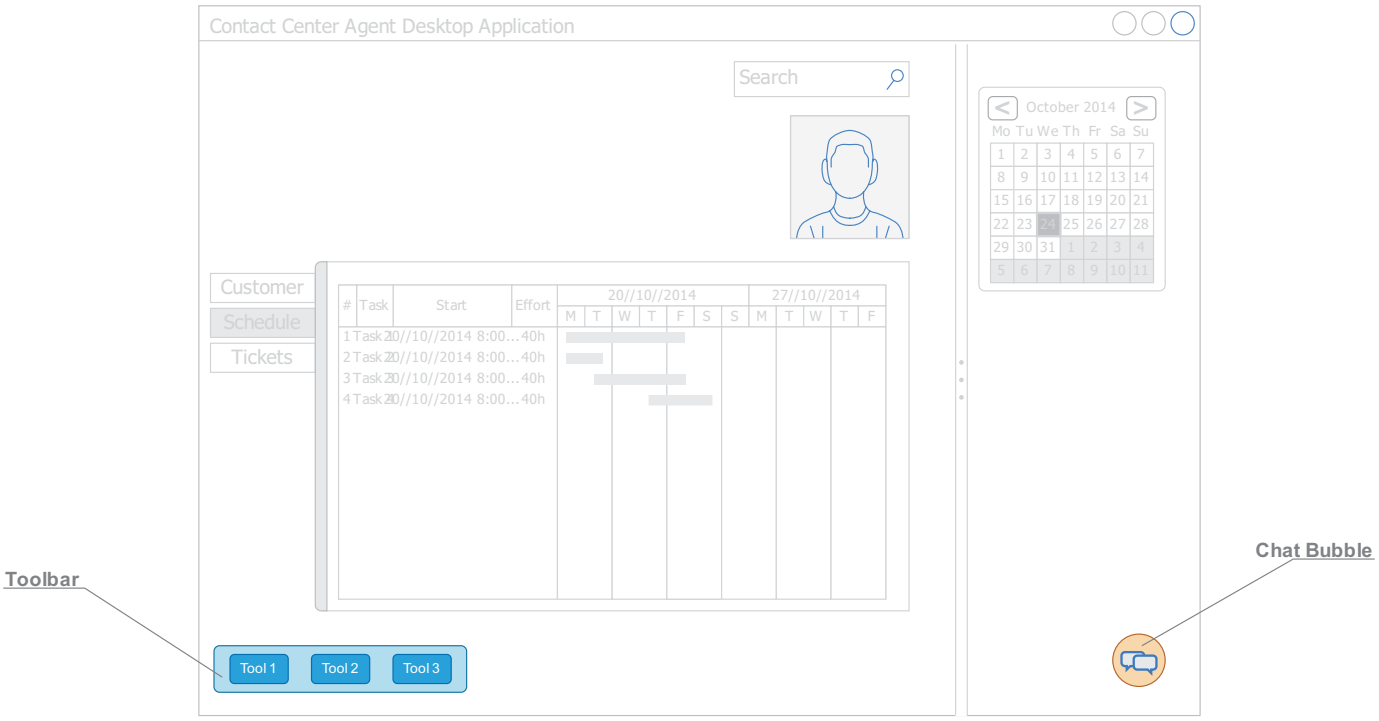


Figure 1: Illustration of user interface plug-in embedded into existing Agent's Desktop application

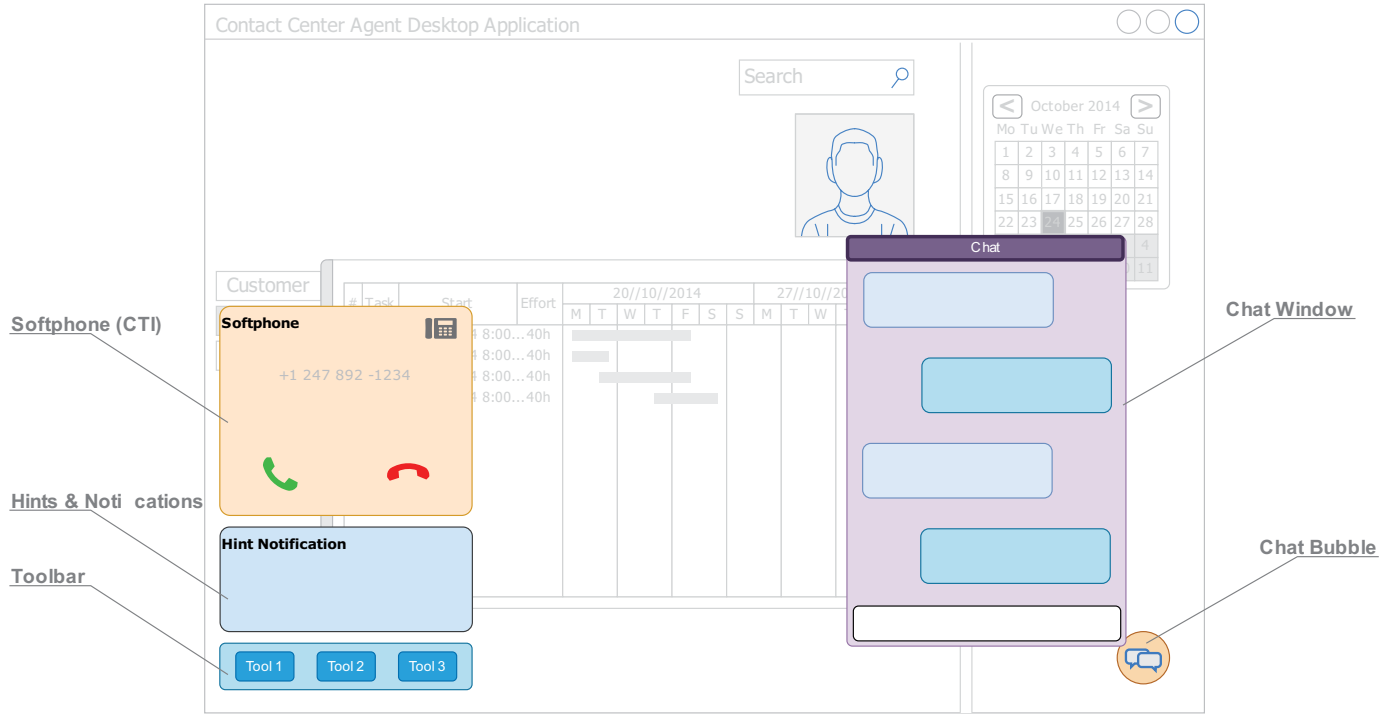


Figure 2: Illustration of user interface tools being used

Chat

Agents typically have the need to speak with others within the organization, or with an AI-based virtual assistant or a bot, and chat-based systems enable this. Plugins should allow for chat tools to be embedded within the agent application, such that the chat systems are made available on demand while also staying away from view when not needed.

Softphone

Agents also need to handle voice calls with customers. Customer Telephony tools (CTI) need to be embedded into the agent application. These could either be one more tool on the Toolbar or can be embedded with dedicated space within the screen real estate.

Prompts & Hints

Agents can also be provided with hints or contextual notifications as they work through resolving customer issues. Such tools would employ AI-driven contextual insights derived based off enterprise data, which is then personalized for the specific agent. Such notifications can be delivered to the agent's user interface via a toast window or hint overlay which appears contextually, stays active briefly, and then fades away after its purpose has been served.

Search Tools

Agents can also be provided with unified search capabilities to quickly find useful and required information as they work with customers on resolving issues. Such search tools must be equipped to provide quick and relevant results fast, in overlays that can be dismissed or hidden by the agent when she is done with the information.

Considerations while building user interfaces for contact center solutions

Broadly there are three key aspects that must be considered while building user interfaces for contact center solutions.

Integration with existing agent tools

Contact center agents typically make use of a wide variety of tools during their interactions with customers. Agents will go through multiple steps and perform numerous activities as they work through the processes mandated for dealing with the customer's query. It is important for any new user interface to be able to seamlessly integrate with the existing tool suite without distracting from the execution flow and experience that the agent may already be accustomed to.

Flexible tools can also help alleviate some cumbersome steps that the existing tools may entail, including eliminating the necessity for the agents to navigate complex multi-layer menus, jumping across multiple applications, etc., by providing a more seamless experience using timely pop-ups, overlays, and toasts, that present contextual information and insights.

Integration with CRM systems

Contact center agents will typically make use of CRM systems to manage client accounts. CRM systems may either form the core of the agent's day-to-day work, or an integral part of it.

User interfaces for contact center solutions should be built in a modular fashion to allow quick and easy integration with established CRM systems in the market.

Integration with CTI tools

Contact center agents make use of CTI technology (Computer-Telephony Interface) to for interactions with customers. There are several well-established vendors that provide CTI tools: Amazon Connect, Avaya, Cisco, Genesys, Twilio, to name a few. Contact center solutions must be able to easily integrate with CTI tools that the customers are making use of.

Building flexible user interfaces

After having laid out the rationale for building user interfaces to be flexible and "pluggable", in the subsequent sections of this document we will explore the tools and techniques that can help us build these flexible user interfaces for contact center solutions.

Technical assumptions

Before we discuss the mechanisms employed for building flexible user interfaces, we must preface it with the basic technical assumptions that helps define the boundary of the scope we are considering for this paper.

HTML5 & CSS3 Compatibility

We assume that the contact center tools are primarily web-based; while it cannot be denied that many contact centers still make use of desktop applications, it is a safe assumption to make that most contact centers these days are progressively moving towards modern technical stacks and will make use of web-based applications for their day-to-day operations. This enables the leveraging of modern HTML5 and CSS3 based technologies to build advanced solutions for contact centers.

Ability to modify HTML

We assume that the systems being employed within the contact centers lend themselves to a few minor modifications. This allows us to easily integrate any new solution user interfaces with existing systems without the need for extensive rewrites.

This allows us to **plug-in** other contact center tools into the existing applications and allow the users to receive one unified user experience.

Many systems have been found to allow creation of custom additions using some form of custom widgets that can be created and embedded within existing user interfaces. Such capabilities can easily be leveraged to plug-in additional solutions and tools that may be built for contact centers.

Ability to customize Content Security Policy headers

Web technologies today enforce security through Content Security Policies (implemented through CSP headers). These policies enforce how web applications interact with one another and with servers. When we create extensible user interfaces, often there can be scenarios that a pluggable user interface component needs to be 'hosted' within the user interface of another system.

In such scenarios, it becomes essential for the CSP headers of the host system to be tweaked to allow such embedded user interface components to function correctly. Fortunately, most established web applications today allow customization of CSP headers through the admin interfaces; case in point is Salesforce.com interface which allows for addition/modification of CSP headers through their configuration screens.

Options for building flexible user interfaces

While building our contact center solution, we leveraged the powerful UI framework: Angular. Angular provides the tools necessary to build modular user interface components which allows packaging core pieces of functionality into distinct pieces.

While we made use of the Angular framework, the options and the approach discussed within this paper needn't be restricted to Angular alone. It can easily be implemented in another framework, say React, or even using vanilla Javascript and the HTML5 APIs.

The two main options we employed were to build:

1. Micro frontends: these were complete applications that cater to a specific business requirement.
2. Web components: these are small reusable pieces of functionality, such as a user interface plugin, a reusable control, etc.

Micro frontends

Micro frontends are a relatively new concept in software engineering which aims to bring the benefits of the micro services paradigm to frontend engineering and web development (Mezzalira, 2019). Micro frontends essentially comprise of an empty shell which is purely a technical component and serves the purpose of bringing together multiple micro-applications together and stitching them into a cohesive web application.

Each individual micro application typically handles a specific business function and can be developed, maintained, and deployed independently by distinct teams.

Web components

The HTML5 specification introduces a new concept of custom HTML elements that can have self-contained functionality. These packets of custom functionality are termed Web components. Supported by all modern browsers, web components are an ideal way to build modular user interface components with specific functionality built into them, and then deploy the desired functionality across existing systems.

Five key technical considerations while building flexible user interfaces

While micro frontends and web components provide an ideal mechanism for building modular, self-contained user interfaces, the following aspects can be key in the success of flexible user interfaces.

1. Run-time loading of configuration

Typically, in a web development project the configuration files are stored within the codebase. However, with such an approach, any change in the configuration values will require a recompilation of the entire codebase, and a redeployment of the application code for the new configuration values to take effect.

While there is no real harm in having to rebuild a web development project, the need to do so every time a config value needs to be changed can quickly become cumbersome.

```
import { Component } from '@angular/core';
import { environment } from '../environments/environment';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  private readonly config;
  constructor() {
    this.config = environment;
  }
}
```

The need to recompile the application every time a configuration value needs to be modified becomes particularly tedious if the application being recompiled is meant to be a flexible plugin that should be embedded within other applications.

Hence, it is advisable to separate out the configuration files from the codebase and load them at run-time.

```
import { HttpClient } from '@angular/common/http';
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  private config;
  constructor(http: HttpClient) {
    http.get('path/to/run/time/configuration/file')
      .subscribe(config => (this.config = config));
  }
}
```

Such a mechanism allows for the configuration files to be maintained separate from the code and allows for quick configuration changes within the client environment during the life of the web application.

2. Dynamic rendering & positioning

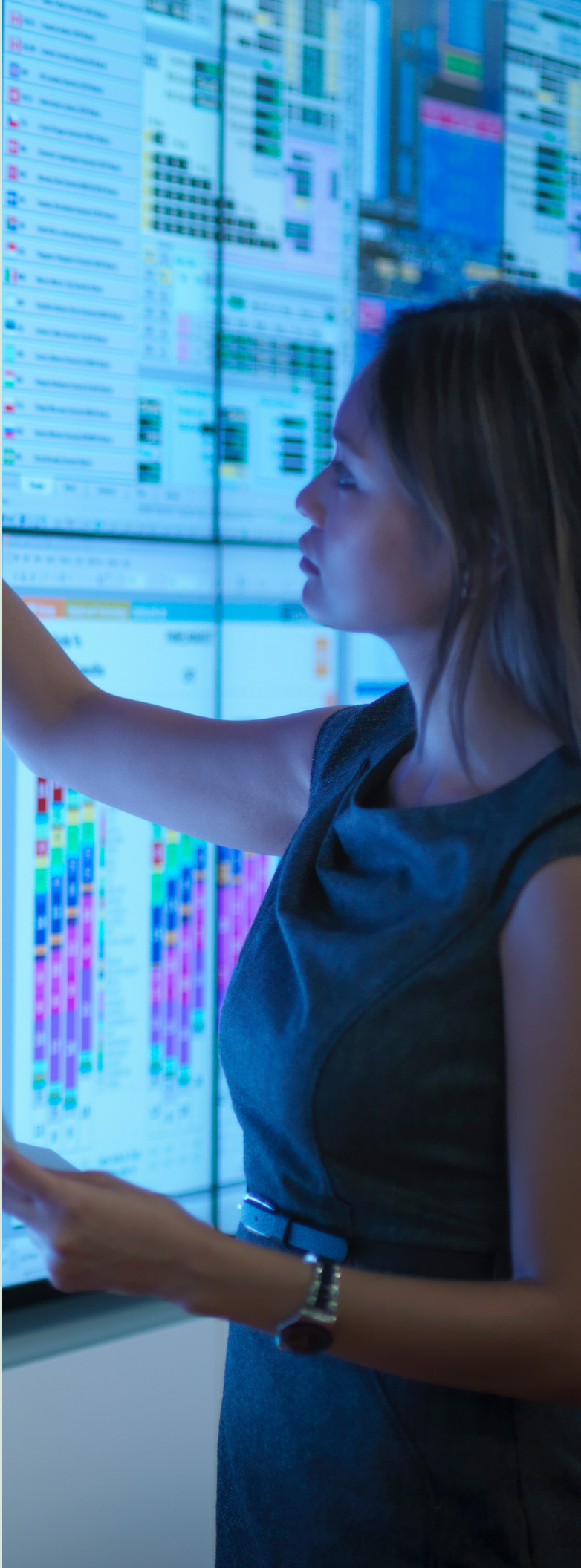
Another important aspect of building flexible user interfaces is to have a dynamic rendering capability. Dynamic rendering allows applications to be flexible in terms of the type of widgets or layouts that are rendered on screen depending on either configuration or the business scenario.

Ability to render different widgets or layouts dynamically expands the ability of the application to adapt and respond differently to different use cases.

Most frontend frameworks provide mechanisms to dynamically instantiate user interface components and inject them into the page at run time. This capability must be systemically leveraged to provide the capability to build the page and user experience based on business needs, user preference, localization needs, etc.

The following is an example of dynamic rendering implemented using the Angular framework:

```
@Component({
  selector: 'crtx-component-renderer',
  template: '<ng-container #renderSlot></ng-container>',
  styleUrls: ['./component-renderer.component.scss']
})
export class ComponentRendererComponent
  implements AfterViewInit, OnChanges {

  @Input() model!: ComponentModel;
  @Output() output = new EventEmitter();

  @ViewChild('renderSlot', { read: ViewContainerRef })
  renderSlot!: ViewContainerRef;

  private cmpRef!: ComponentRef<{}> | null;

  constructor(
    private readonly resolver: ComponentFactoryResolver,
    private readonly injector: Injector
  ) {}

  ngAfterViewInit(): void {
    Promise.resolve().then(() => this.renderView());
  }

  ngOnChanges(changes: SimpleChanges): void {
    if (changes.model && this.cmpRef) {
      this.cmpRef?.instance.model = changes.model.currentValue;
      this.renderView();
    }
  }

  private renderView(): void {
    const cmpRef = this.cmpRef = this.resolver
      .resolveComponentFactory(this.model.component)
      .create(this.injector);

    cmpRef.instance.model = this.model.data;
    cmpRef.instance.output
      .subscribe(msgValue => this.output.emit(msgValue));

    this.renderSlot.insert(cmpRef.hostView);
    cmpRef.changeDetectorRef.detectChanges();
  }
}
```



3. Communication between user interface components

Building flexible user interfaces using either micro frontends or web components essentially involves building separate pieces and stitching them together at run time to build the overall functionality. It is often necessary for these separate pieces to be able to communicate with one another for the overall functionality to work.

The following sections discuss a few options that can be employed to get around this limitation.

Communication via the Route definitions

When user interfaces are built using Angular, communication between the individual pieces can be established via the Angular Router. The routing definition already has provision to pass in a data property along with the individual routes. This data property is used to communicate key information into the individual micro applications.

An example route definition would look something as below:

```
RouterModule.forRoot([
 { path: '', component: HomeComponent, pathMatch: 'full' },
 {
   path: 'one',
   loadChildren: () => loadRemoteModule({...})
                       .then(m => m.AppModule),
   data: { hostname: 'http://micro-host1.com' }
 },
 {
   path: 'two',
   loadChildren: () => loadRemoteModule({...})
                       .then(m => m.AppModule),
   data: { hostname: 'http://micro-host2.com' }
 }
])
```

Obviously, this approach only works within Angular applications. If the micro frontends are built using a different set of frameworks this wouldn't be applicable.

Communication using CustomEvents

Another option for implementing inter-application communication is to use CustomEvents that can be triggered on the window object. The shell and the various micro applications can be built to listen to these events and responding to them as necessary.

When any of the applications needs to send out a message, it would broadcast the message using a CustomEvent object and using the window.dispatchEvent API.

In the first application:

```
window.addEventListener('app2-msg', msg => handleMsg(msg));

// Assuming variable message contains
// the value to be communicated to
// application-2
const event = new CustomEvent('app1-msg', message);

window.dispatchEvent(event);
```

by code in the second application to listen to that event:

```
window.addEventListener('app1-msg', evt => {
 const value = handleEvent(evt);

 // Creating a custom event with the
 // response from application-2
 const resp = new CustomEvent('app2-msg', value);

 window.dispatchEvent(resp);
});
```

4. Module federation: sharing of common libraries & modules

Another consideration is the use of Module federation to share common libraries and modules across the individual pieces of the user interface that may be deployed.

Module federation is a new technique that has become supported as part of Webpack version 5 and Angular version 13. This technique ensures that common libraries and modules --- for example: **@angular/core, @angular/common, @angular/router, rxjs** to name only a few --- are downloaded only once from a server and then shared across all applications that are loaded subsequently.

Module federation across applications thus helps in reducing the overall bundle sizes of the individual user interface pieces that would otherwise have to be loaded into the browser.

Module federation can be setup by tweaking the Webpack configuration files appropriately to define "remotes" and "shared modules". Webpack is capable of then packing these pieces of code in such a way as to ensure that shared modules are downloaded only once and then shared across the various bundles that may be loaded within the browser.

```
…
// Plugin definition within the Webpack Configuration file to define
// ModuleFederationPlugin & its configuration
plugins: [
 new ModuleFederationPlugin({
  library: { type: "module" },

  // Remote modules that will be bundled and loaded separately
  remotes: {
    // "mfe1": "mfe1@http://localhost:3000/remoteEntry.js",
  },

  // Shared libraries that will be downloaded only once but
  // shared across all the bundles that are loaded
  shared: share({
   "@angular/core": {
    singleton: true,
    strictVersion: true,
    requiredVersion: 'auto'
   },
   "@angular/common": { singleton: true, …},
   "@angular/router": { singleton: true, …},
   "@angular/common/http": { singleton: true, …}
  })
 }),
 sharedMappings.getPlugin(),
],
```

While working within an Angular ecosystem, there are some very powerful tools that enable us to quickly enable and configure Module federation for Angular applications. The **@angular-architects/module-federation** library can be easily incorporated into any Angular application to do the heavy lifting while enabling module federation (Streyer, 2020).

5. Common technical infrastructure

All user interface components typically share some technical plumbing code. Be it for authentication, configuration management, server communication, etc., the tasks to be done are typically the same across individual pieces irrespective of the business functionality handled. Such common services may be packaged into a separate module/library which can then be leveraged across all components.

Using such a common technical infrastructure, while not essential, does help in maintaining a uniform developer experience across all user interface components.

Building micro frontends in angular

In the following sections we will discuss the approach to build micro frontends using the Angular framework. We will rely heavily on the approach and tools laid out by Manfred Streyer (Streyer, 2020).

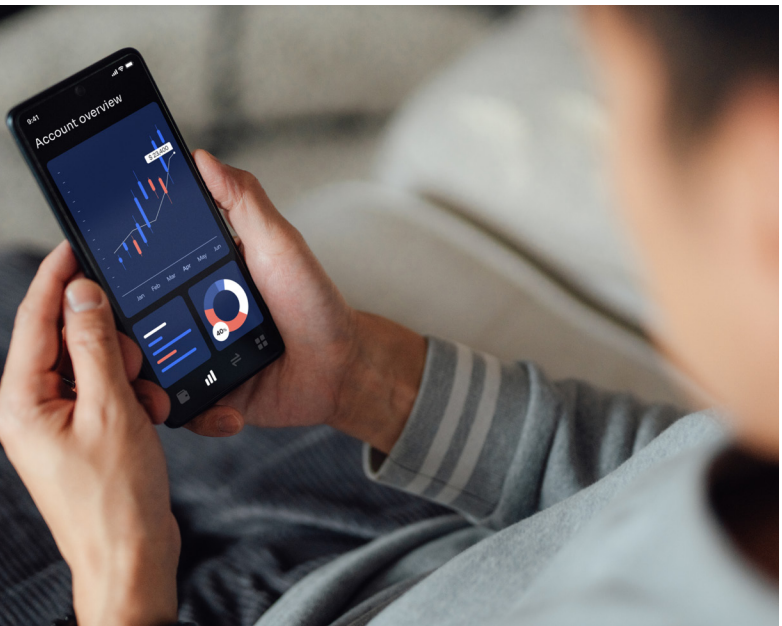
Micro frontends typically comprise of a host "shell" application which only serves to bring together multiple other applications stitching them together into one unified experience for the end users. The business functionalities are served by one or many micro applications that can be developed, deployed, and maintained separately, possibly by different teams.

Application shell

The shell loads first and establishes the main Angular application run-time on the client-side. Once the application has loaded successfully, it downloads the files related to each micro application as and when the user navigates to that specific route.

The routing in the main shell application looks as follows (Streyer, 2020):

```
@NgModule({
 imports: [
  RouterModule.forRoot([
   { path: '', component: HomeComponent, pathMatch: 'full' },
   {
    path: 'one',
    loadChildren: () =>   // These routes are configured to load the
    loadRemoteModule({  // micro app from their remotes
    type: 'module',
    remoteEntry: 'http://micro-host.com/mico-app-one.js'
    exposedModule: 'OneModule'
    }).then(m => m.AppModule)
   },
   {
    path: 'two',
    loadChildren: () =>
     loadRemoteModule({
      type: 'module',
      remoteEntry: 'http://micro-host.com/mico-app-two.js'
      exposedModule: 'TwoModule'
     }).then(m => m.AppModule)
   }
  ])
 ],
 exports: [RouterModule]
})
export class AppRoutingModule {}
```



Here, we use the loadRemoteModule utility function from **@angular-architects/module-federation** to dynamically download the micro application, using the location and filename on which that specific micro application exposes its interface, and load it into the Angular application scope.

The application shell is also the only application within the entire web application which must invoke RouterModule. forRoot().

The approach is akin to normal Angular lazy loading, except that the script being downloaded uses Webpack module federation to share common libraries with the shell.

Micro applications

Each micro application handles its own routing to pages that are part of it. However, these routes must be defined using RouterModule.forChild(), like how they would be defined in a typical lazy-loaded module.

The following is an example of a route configuration for a micro application:

```
RouterModule.forChild([
 {
   path: '',
   component: AppComponent,// Micro-app main page
   children: [          // All child routes within the application
    { path: 'child-one', component: ChildOneComponent },
    { path: 'child-two', component: ChildTwoComponent },
    { path: '', component: HomeComponent }
   ]
 }
];
```

The reason we mandate that the micro applications expose their routes using RouterModule.forChild() is because the micro application gets loaded into the same Angular context as the shell, which would have already invoked RouterModule.forRoot().

If the micro application also invokes forRoot(), Angular would throw an error at run-time when it tries to load the micro application. This is why micro applications must always use forChild() to define their routes.

However, this prevents the micro application from being run as a standalone application, effectively nullifying the benefit of building micro frontends in the first place. To get around this limitation, we wrap each micro application within another wrapper module.

```
@NgModule({
 declarations: [StandaloneAppComponent],
 imports: [
  BrowserModule,
  RouterModule.forRoot([
   {
    path: 'two',
    loadChildren: () => import('./app.module').then(m => m.AppModule)
   },
   {
    path: '',
    redirectTo: 'two',
    pathMatch: 'full'
   }
  ])
 ],
 bootstrap: [StandaloneAppComponent]
})
export class StandaloneAppModule {}
```

This wrapper module is only used when running the micro applications standalone, and can be used to do all the Angular startup activities that would otherwise be expected to be done within a main module of any Angular application, such as:

1. Importing BrowserModule, while the AppModule of the micro application will only import the CommonModule
2. Defining basic start-up routes using RouterModule.forRoot(), while the AppModule invokes RouterModule.forChild()
3. Any other application initialization processes that may be necessary for effective running of the micro application; this could include processes such as configuration initialization, authentication, etc.

The stand-alone bootstrap process of the micro applications can be updated to bootstrap the wrapper module instead of the AppModule.

```
platformBrowserDynamic()
 .bootstrapModule(StandaloneAppModule)
 .catch(err => console.error(err));
```

Building web components in angular

Web components are defined on the page using specific HTML5 syntax which allows to register the custom element in the CustomElementRegistry of the browser. After this, that custom element can be added to the web application's HTML code and the browser will be able to recognize the functionality that should be rendered in its place.

```
<html>
 <head>
  <title>Test Web Application</title>
  <script language="Javascript" src="/path/my-element.js" />
 </head>
 <body>
  <my-custom-element></my-custom-element>
 </body>
</html>
```

In the above example, <my-custom-element> is the custom element tag that is added to the HTML and that allows an entire functionality, maybe even an application, to be provided at that place.

Web components in Angular using Angular elements

Angular provides a powerful set of tools to package Angular applications and components as web components: Angular elements **(@angular/elements)**.

Using the Angular elements **createCustomElement**() API, it becomes trivial to package any Angular component or even whole Angular applications as a single Custom Element (Angular Team, n.d.).

References

Bachina, B., 2020. 6 Different Ways To Implement Micro-Frontends With Angular. [Online]
Available at: https://medium.com/bb-tutorials-and-thoughts/6-different-ways-to-implement-micro-frontends-with-angular-298bc8d79f6b

[Accessed 19 November 2020].

Bachina, B., 2020. How To Implement Micro-Frontend Architecture With Angular. [Online]

Available at: https://medium.com/bb-tutorials-and-thoughts/how-to-implement-micro-frontend-architecture-with-angular-e6828a0a049c

[Accessed 19 November 2020].

Mezzalira, L., 2019. YouTube: Micro Frontend Architecture - Luca Mezzalira, DAZN. [Online]

Available at: https://www.youtube.com/watch?v=BuRB3djraeM

[Accessed 12 May 2021].

Riches, D., 2018. A Micro Frontends Future: Using Angular with React and Vue in Enterprise apps. [Online]

Available at: https://www.youtube.com/watch?v=yPniBH5sjA4

[Accessed 12 May 2021].

Streyer, M., 2018. Micro Apps with Web Components using Angular Elements. [Online]

Available at: https://www.angulararchitects.io/aktuelles/micro-apps-with-web-components-using-angular-elements/

[Accessed 17 November 2020].

Streyer, M., 2019. 6 Steps to your Angular-based Microfrontend Shell. [Online]

Available at: https://www.angulararchitects.io/en/aktuelles/6-steps-to-your-angular-based-microfrontend-shell/

[Accessed 14 May 2021].

Streyer, M., 2020. Dynamic Module Federation with Angular. [Online]

Available at: https://www.angulararchitects.io/en/aktuelles/dynamic-module-federation-with-angular/

[Accessed 23 03 2022].

Streyer, M., 2020. The Microfrontend Revolution: Module Federation in Webpack 5. [Online]

Available at: https://www.angulararchitects.io/aktuelles/the-microfrontend-revolution-module-federation-in-webpack-5/

[Accessed 19 November 2020].

Streyer, M., 2020. The Microfrontend Revolution: Module Federation with Angular. [Online]

Available at: https://www.angulararchitects.io/en/aktuelles/the-microfrontend-revolution-part-2-module-federation-with-angular/

[Accessed 23 03 2022].

Streyer, M., 2021. Multi-Framework and -Version Micro Frontends with Module Federation: The Good, the Bad, the Ugly. [Online]

Available at: https://www.angulararchitects.io/en/aktuelles/multi-framework-and-version-micro-frontends-with-module-federation-the-good-the-bad-the-ugly/

[Accessed 24 03 2022].

Strumpflohner, J., 2018. Compile-time vs. Runtime configuration of your Angular App. [Online]

Available at: https://juristr.com/blog/2018/01/ng-app-runtime-config/

[Accessed 10 October 2019].

Trajan, T., 2019. The Best Way To Lazy Load Angular Elements. [Online]

Available at: https://medium.com/@tomastrajan/the-best-way-to-lazy-load-angular-elements-97a51a5c2007

[Accessed 23 November 2020].

About the Author

Kiran Janardhan Holla
Senior Project Manager, UI Architect

About the Mentor

Vishal Manchanda
Senior Principal Technology Architect

For more information, contact askus@infosys.com

Navigate your next