CLINICAL OFFICE: MPAGE EDITION

Version 3 Developer Training

Copyright © 2021 – Precision Healthcare Solutions

COURSE OVERVIEW

Over the next three days we will create a fully functional MPage using the following technologies:

- Angular v12
  - Robust JavaScript framework developed by Google.
  - https://angular.io
- Angular Material
  - Ready to use styles and components that follow the Material Design standards.
  - https://material.angular.io
- Clinical Office: MPage Edition (v3)
  - Custom Angular and CCL library designed to provide powerful functionality while greatly simplifying development.
  - https://www.clinicaloffice.com
- CCL
  - Cerner's proprietary programming language for collecting information from the Oracle database.

CLASS GOALS

- The primary goal of this course is to learn how to use Angular and Clinical Office: MPage Edition to be able to create and deploy a custom MPage.
- You will learn how to use common Angular features such as components, templates, services and routes.
- Connectivity through the Clinical Office MPage service will be taught along with several the custom services available in Clinical Office.
- We will cover some CCL during the development of our MPage however it is expected that you have a working knowledge of writing CCL scripts.

PATIENT HISTORY MPAGE

- During the next three days we will be building a Chart level MPage called Patient History.
- This MPage will show some basic patient information using some of the internal Clinical Office functionality and then move on to creating custom CCL to retrieve data specific to our project.
- We demonstrate Angular routing by offering multiple paths for viewing our patient history.
- Topics such as data binding will be covered to create interactive prompts. These techniques can be applied in later projects to build entire data entry systems if needed.

GETTING STARTED

The best way to learn is to do.

The format of this course will be tutorial based. Concepts will be taught as a narrative towards building a final product.

Before this class started, you should have completed the prerequisites.

The first step is to download your site copy of ie-mpage from GitHub.

GETTING STARTED

- Your first step in creating your MPage will be to clone the Clinical Office project template from GitHub.
- The Clinical Office project template contains everything you need to start building your MPage including references to Angular, Angular Material, Moment.js, Angular Flex Layout and Clinical Office: MPage Edition.
- Basic MPage project setup has been completed in the template including all the required Cerner meta-tags in the index.html file and any Internet Explorer polyfills. Later as MS Edge usage becomes the Cerner standard, an Edge based template will be available.
- Every MPage you create will start with a clone of the clinicaloffice/ie-mpage or your site-specific copy of the template project.

GETTING STARTED

- Along with the features mentioned on the last slide, the MPage template also contains instructions and the basic structure to use the development proxy server built into Angular.
- The proxy server will allow you to bypass normal CORS security and access the Cerner Web API from your development machine giving you the ability to perform live development of MPages.
- “Live Development” is a feature in Angular that lets you see the changes to your code immediately in real time without having to compile. The proxy server let’s you access the Cerner Discern MPages API during live development.

MPAGE TEMPLATE

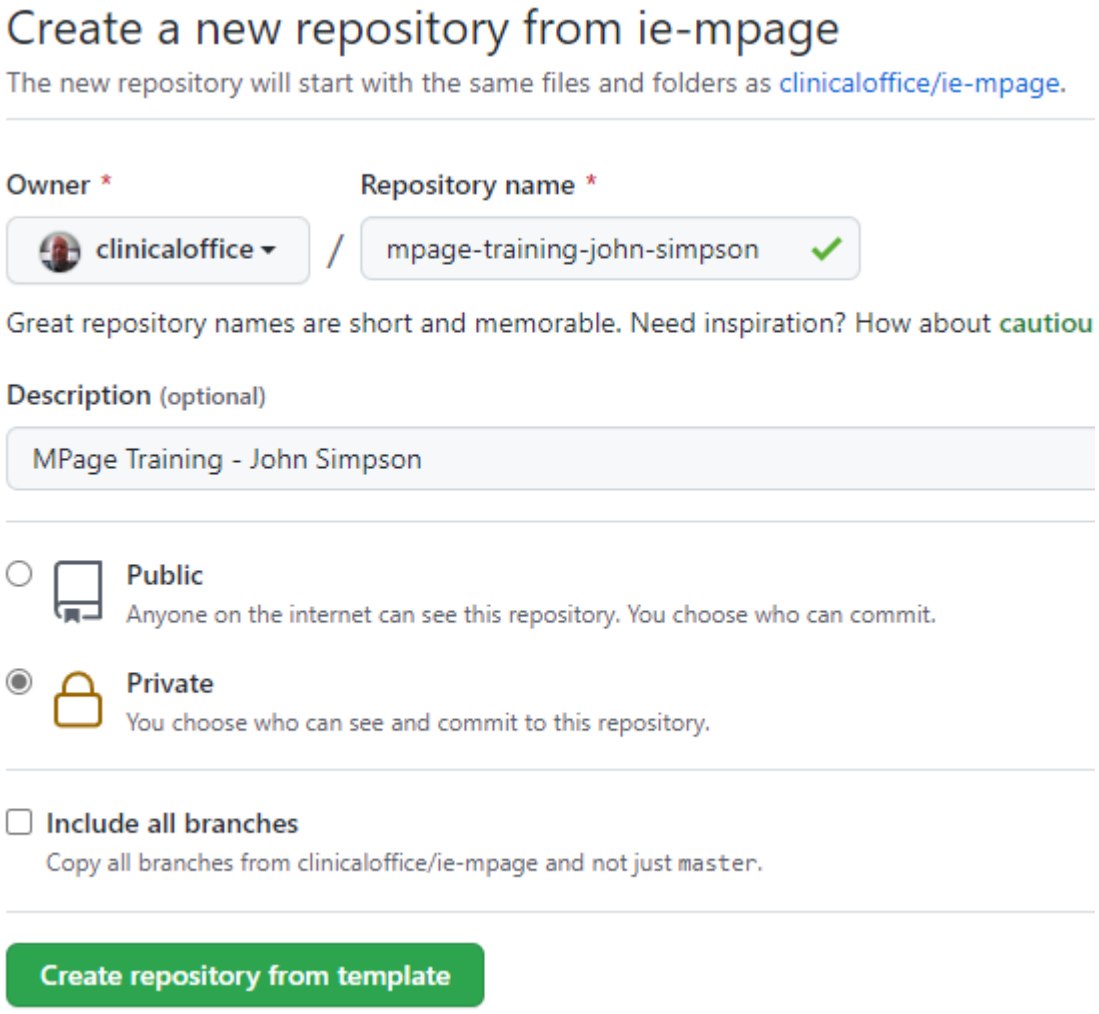
- From https://github.com search for clinicaloffice/ie-mpage or if you site already has its own customized version of the template, search for and use it instead.

YOUR SITE TEMPLATE NAME: cst-reporting/ie-mpage

- From the template page, click the green button labeled Use this template

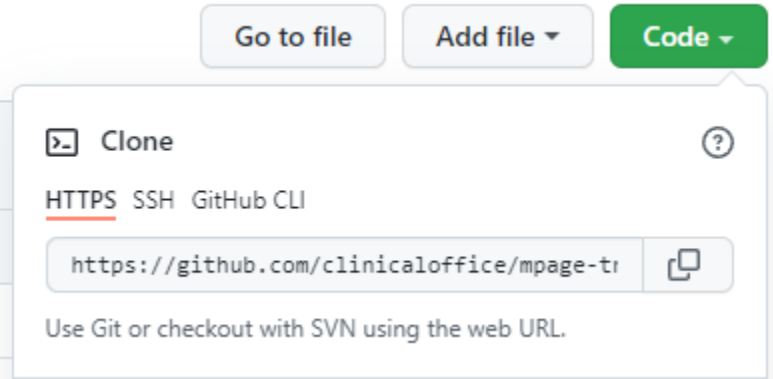
MPAGE TEMPLATE

- The next screen will ask for details on your project. Please fill in the "Owner" value and set it to your personal account. For a non-training MPage, you would normally choose your organization as the owner.
- For the "Repository name" field, enter the value mpage-training-{your name} where your name is replaced with your name.
- You can optionally complete the "Description"
- Unless you want the entire world to be able to access your MPage code, set the repository to be "Private"
- Finally, click the "Create repository from template" button.



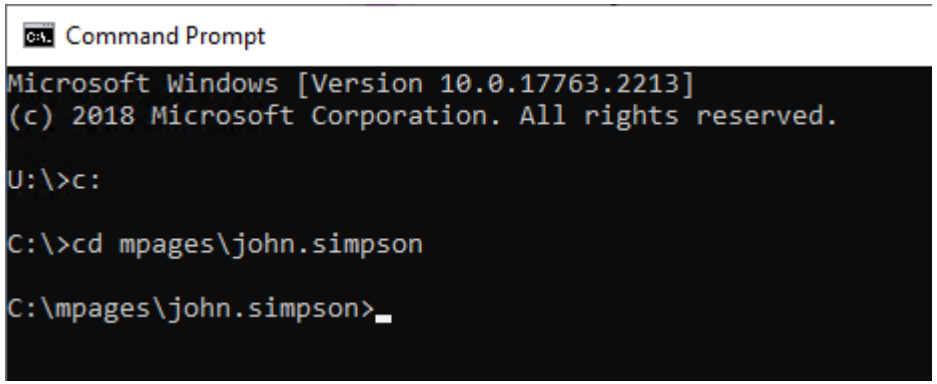
MPAGE TEMPLATE

- You should now be on your GitHub page for your project. Any changes you make to your project can be pushed back up to GitHub and you can provide access to other users. If this project was setup under an organization name and not your name, access is made available to other users automatically if they have owner privileges.
- To start working on your project, click the Code button to view the Clone information.
- From the Clone window, click the copy button. This step simply copy's the URL of your project into memory.



MPAGE TEMPLATE

- Open a command line prompt on your development machine and change folders to the location you wish to store your MPage source code.
- It is highly recommended that you store your working source code on your local hard drive as network speeds can significantly affect your development performance. Final versions of your source code should be pushed back up to GitHub when you have finished working on your project.
- If you are using a shared development machine, keeping your MPage source code in a folder that is easy to identify as yours is beneficial.



MPAGE TEMPLATE

- Type in git clone followed by pasting the URL from GitHub and press the enter key.

```
git clone https://github.com/clinicaloffice/mpage-training-john-simpson.git
```

- Change to the folder containing your project and view the contents of the folder.

```
cd mpage-training-john-simpson
dir
```

- Keep your command line window open.

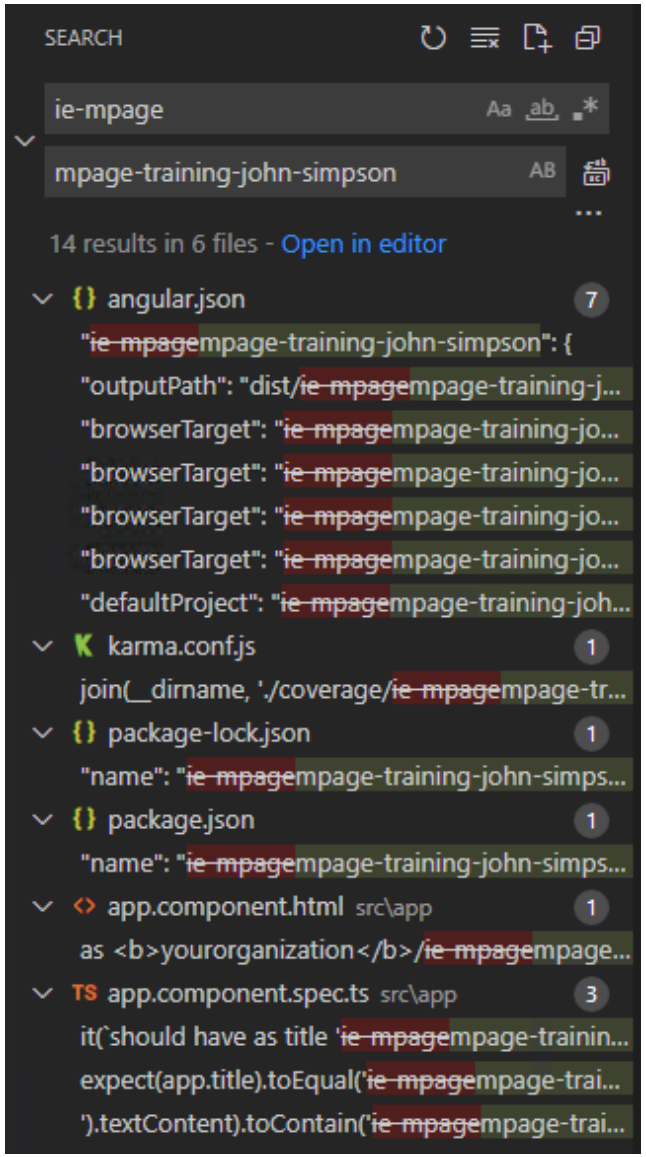
```
C:\mpages\john.simpson>cd mpage-training-john-simpson

C:\mpages\john.simpson\mpage-training-john-simpson>dir
 Volume in drive C is OS
 Volume Serial Number is BED4-C069

 Directory of C:\mpages\john.simpson\mpage-training-john-simpson

2021-11-23  11:19 AM    <DIR>          .
2021-11-23  11:19 AM    <DIR>          ..
2021-11-23  11:19 AM               623 .browserslistrc
2021-11-23  11:19 AM               290 .editorconfig
2021-11-23  11:19 AM               649 .gitignore
2021-11-23  11:19 AM             3,542 angular.json
2021-11-23  11:19 AM             1,469 karma.conf.js
2021-11-23  11:19 AM           568,904 package-lock.json
2021-11-23  11:19 AM             1,413 package.json
2021-11-23  11:19 AM             1,049 README.md
2021-11-23  11:19 AM    <DIR>          src
2021-11-23  11:19 AM               302 tsconfig.app.json
2021-11-23  11:19 AM               810 tsconfig.json
2021-11-23  11:19 AM               351 tsconfig.spec.json
              11 File(s)        579,402 bytes
```

MPAGE TEMPLATE

- Open your project folder in Visual Studio Code and from the Edit menu, select the "Replace in Files" option.
- Replace "ie-mpage" with the name of your project (e.g., "mpage-training-john-simpson")
- Return to your command line and type:

  npm update

- After a few moments, the project will be ready to work on.



FIRST RUN

- From the command prompt, type in ng serve. This will perform and initial compile and start your project as a standalone development server on your machine.

```
ng serve
```

Notes

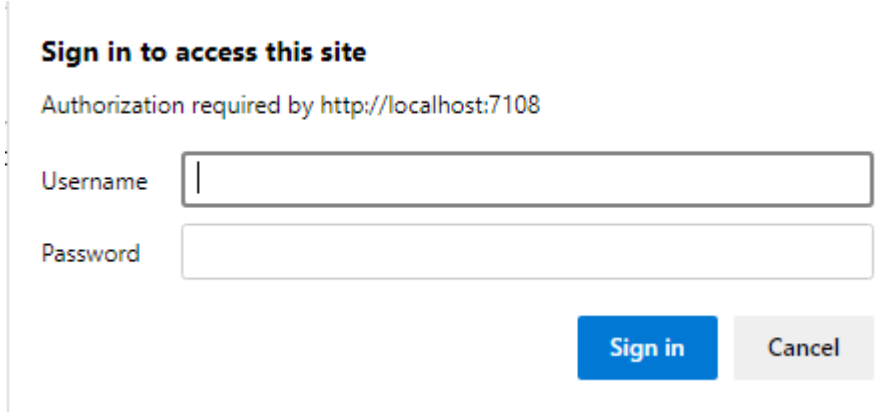
1. The first time will take a little while as everything needs to be compiled.
2. If you are running on a shared machine, you will need to specify the port number as only one application can be run from the same port at the same time. For the purpose of this class use your birth year and month as a port number (e.g., My birthday is August 1971 so my port would be 7108).

```
ng serve --port=7108
```

You should see a message stating your application has been compiled successfully.

FIRST RUN

- Open your web browser. This could be either MS Edge or Internet Explorer. I prefer using Edge over IE as the debugger in IE isn't as stable as Edge. I highly recommend not using other browsers such as Chrome as matching what you will see in Cerner may not be the same.
- In your web browser, open the URL http://localhost:4200 or http:localhost:{yourport} if you specified a port (e.g., http://localhost:7108)
- If you used an organization copy of the template with the proxy defined and if you are on the same network as the Cerner API, you will be prompted with the following login.

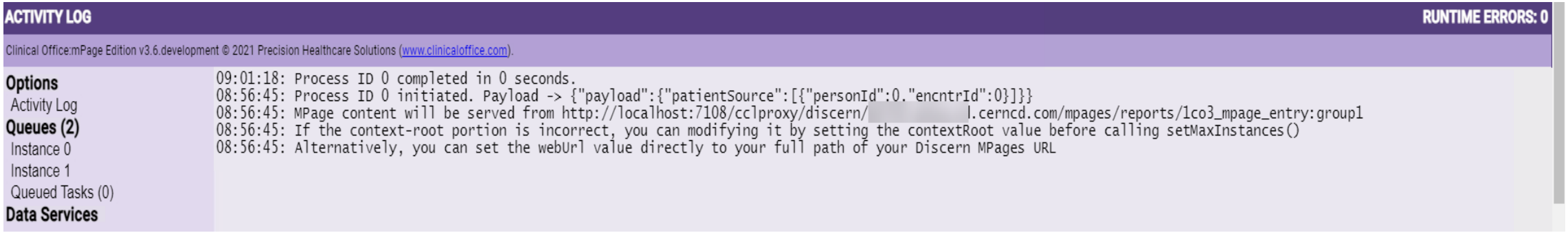


FIRST RUN

- Type in your Cerner username and password along with @{domain name} where {domain name} is the name of your Cerner domain (e.g., build, b1234, etc.).

  Note
  If you don't have your proxy setup you will not be prompted for the login however full instructions are provided on the template page you are viewing.

- Hold the CTRL+Z buttons to see the Clinical Office debugger terminal. You will see some activity information including proxy details.
- The debugger will be covered in detail later as we work further on our MPage.



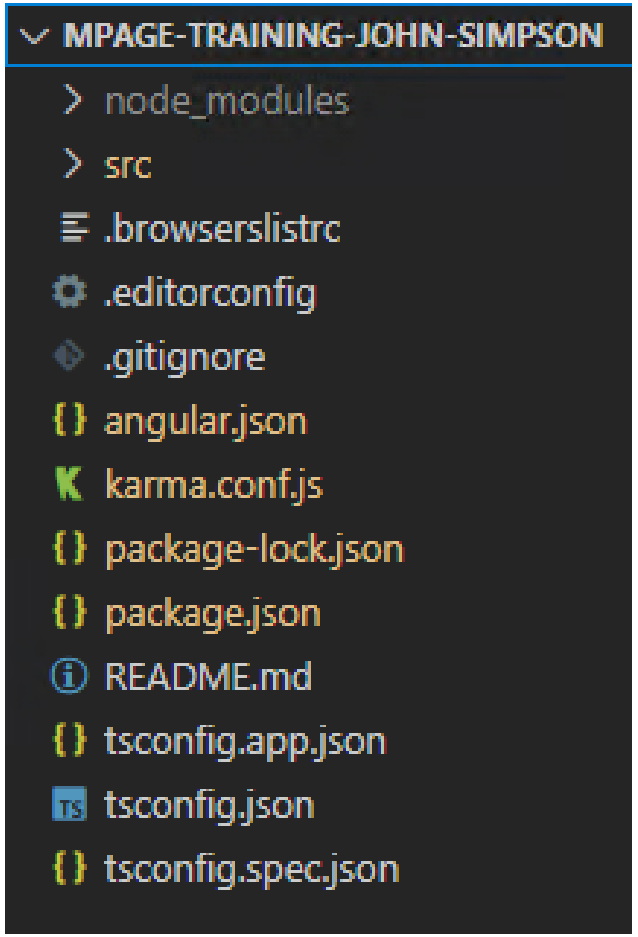
FILE TYPES

There are several file types you will see in a typical Angular application. The most common are.

- TypeScript (.ts) files contain program logic.
- Hyper Text Markup Language (.html) files store web page markup presentation code.
- Syntactically Awesome Style Sheet (.scss) defines colors, margins and other HTML decorators.
- JavaScript Object Notation (.json) files represent data in a file format. In your project source these files will typically contain project definitions however the data we work with when communicating to CCL is also in JSON format.

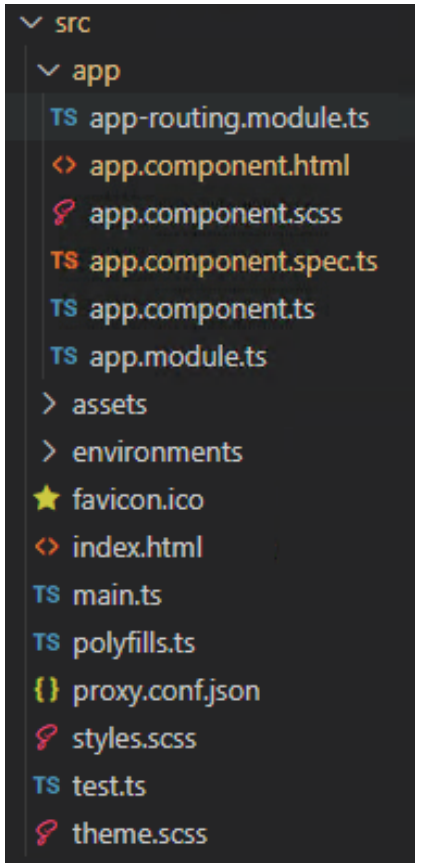
PROJECT FOLDER STRUCTURE

- All Angular projects have the same basic folder and file structure. While each folder and file has a purpose, we are only going to discuss those relevant to the work we will be doing.
- At the top level, we have the following folders and files:
  1. node_modules contains all the libraries that can be used in your project.
  2. src contains the project source code.
  3. .browserslistrc contains references to the browsers allowed to use the project.
  4. angular.json contains configuration information such as project filesize budgets and a reference to our proxy server.
  5. package.json is where the versions to all the libraries we use are defined. In addition, automated scripts can be defined here as well.
  6. README.md is a text file containing whatever message/documentation you desire. This will show on your GitHub project page.
  7. tsconfig.json defines the TypeScript settings. You will likely only ever edit the "target" value.



PROJECT FOLDER STRUCTURE - SRC

- The src folder contains several files and sub-folders.
  1. The app folder contains all your Angular source code.
  2. The assets folder is used to store any external assets you may have such as graphics.
- index.html is the starting point for any web page. It contains not only the Angular bootstrap code but also the Cerner required meta tags.
- pollyfills.ts is typically used for IE11 compatibility.
- proxy.conf.json contains the proxy server configuration.
- styles.css contains a link to theme.scss and is also the file where you place your custom application stylesheet information.
- theme.scss contains your global Angular Material theme for your project. You can modify this file manually or use the interactive tool at https://materialtheme.arcsine.dev to change your theme.



APP.MODULE.TS

- Located in the src/app folder, the app.module.ts file is responsible for importing your library source code into your application.
- There are two primary sections in the app.module.ts file that you will be concerned about. The first are the imports and the second is bootstrapping into NgModule.
- Importing libraries is as simple as issuing an import command followed by the name of the class and the location of the class file.

```
import {ClinicalOfficeMpageModule} from "@clinicaloffice/clinical-office-mpage";
```

- In the NgModule section, you will primarily bootstrap your components and libraries in the imports section however some classes may be defined as providers such as the ErrorHandler and DateAdapter.

APP.MODULE.TS

```
import {NgModule, ErrorHandler} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {NoopAnimationsModule} from "@angular/platform-browser/animations";
import {ClinicalOfficeMpageModule} from "@clinicaloffice/clinical-office-mpage";
import {MaterialModule} from "@clinicaloffice/clinical-office-mpage";
import {ErrorHandlerService} from "@clinicaloffice/clinical-office-mpage";
import {AppRoutingModule} from './app-routing.module';
import {AppComponent} from './app.component';
import {MatMomentDateModule, MomentDateAdapter} from '@angular/material-moment-adapter';
import {DateAdapter, MAT_DATE_FORMATS, MAT_DATE_LOCALE} from '@angular/material/core';
import {FlexLayoutModule} from "@angular/flex-layout";
```

APP.MODULE.TS

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    NoopAnimationsModule,
    ClinicalOfficeMpageModule,
    MaterialModule,
    AppRoutingModule,
    FlexLayoutModule,
    MatMomentDateModule
  ],
```

APP.MODULE.TS

```
providers: [
    {provide: ErrorHandler, useClass: ErrorHandlerService},
    {provide: DateAdapter, useClass: MomentDateAdapter, deps: [MAT_DATE_LOCALE]},
    {
      provide: MAT_DATE_FORMATS, useValue: {
        parse: {
          dateInput: ['l', 'LL'],
        },
        display: {
          dateInput: 'MM-DD-YYYY',
          monthYearLabel: 'MMM YYYY',
          dateA11yLabel: 'LL',
          monthYearA11yLabel: 'MMMM YYYY',
        }
      }
    }
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

ANGULAR COMPONENTS

- Components are the building blocks of every Angular application.
- Components can be stand-alone or embedded inside one or many other components.
- The primary component for your Angular application is app.component. Every component or service you use in your project originates from this component.
- Every component (including app.component) has 4 files including:
  - .html – Markup HTML of how your component will look.
  - .ts – TypeScript code to manipulate the HTML, load data, etc.
  - .scss – Component specific style sheet. Anything here will take precedence over the project styles.scss file.
  - .spec.ts. – Used by automated system testing utilities such as Karma. (These files are not covered in this class)

APP.COMPONENT.HTML

- The app.component.html file included in the ie-mpage template contains some template specific information such as configuration instructions for the proxy server.
- You will always remove the content of this file except for the <mpage-log-component></mpage-log-component> line at the bottom of the file.
- <mpage-log-component> is an Angular component developed as part of Clinical Office to offer advanced debugging features not available elsewhere.
- When running your MPage (either through the proxy server or in PowerChart), you can hold CTRL+Z to toggle the log component on or off.
- When developing your Angular application, any components you wish to use in your application will begin with an entry inside the app.component.html file.

APP.COMPONENT.TS

- All component files can be divided into three sections.
  1. Imports section where any components or classes used by the component need to be declared.
  2. The @Component decorator assigns the name of the new HTML element represented by your component and identifies the location of the .html and .scss templates.
  3. The class definition performs any initialization code and defines methods to be used by your component. This is where your program logic will live.
- The app.component.ts file includes imports for the Clinical Office mPageService service as well as initialization through the constructor and ngOnInit methods.

APP.COMPONENT.TS

```
import {Component, OnInit} from '@angular/core’;
import {mPageService} from "@clinicaloffice/clinical-office-mpage";


@Component({
  selector: 'app-root’,
  templateUrl: ’./app.component.html’,
  styleUrls: ['./app.component.scss’]
})
export class AppComponent implements OnInit {


  constructor(public mPage: mPageService) {
  }


  ngOnInit(): void {
    // Initialize MPage services with 2 queues, allow debugging and set to chart level
    this.mPage.setMaxInstances(2, true, 'CHART');

  }


}
```

APP.COMPONENT.TS

- Over the next few slides, we will explain the app.component.ts file in greater detail.
- The import statements are responsible for defining where your components must look for required code.

```
import {Component, OnInit} from '@angular/core’;
import {mPageService} from "@clinicaloffice/clinical-office-mpage";
```

- The @Component decorator defines the component selector name (app-root) as well as the location of the HTML template and component stylesheet.

```
@Component({
  selector: 'app-root’,
  templateUrl: ‘./app.component.html’,
  styleUrls: ['./app.component.scss’]
})
```

APP.COMPONENT.TS

```
export class AppComponent implements OnInit {

  constructor(public mPage: mPageService) {  }

  ngOnInit(): void {
    // Initialize MPage services with 2 queues, allow debugging and set to chart level
    this.mPage.setMaxInstances(2, true, 'CHART');
  }
}
```

- The class export requires at a minimum the constructor. In our ie-mpage template we inject the Clinical Office mPageService service into our component through the constructor. This gives us the ability to access the features available in our service from our component.
- In the ie-mpage template we have also declared an ngOnInit method. This code is called after the constructor and as the component is initializing.
- To implement ngOnInit you must first import it from @angular/core, add the implements OnInit to your class declaration and write an ngOnInit method.
- The ngOnInit method is where you would write your initialization code. In this example we are initializing the mPageService object (mPage).
- ngOnInit is a type of lifecycle hook. There are several other lifecycle hooks available which are all explained in detail at https://angular.io/guide/lifecycle-hooks.

MPAGE INITIALIZATION

- The ngOnInit method is where you would typically start up the MPage service. To initialize the MPage service you make a call to the setMaxInstances method.

```
ngOnInit(): void {
  // Initialize MPage services with 2 queues, allow debugging and set to chart level
  this.mPage.setMaxInstances(2, true, 'CHART');

}
```

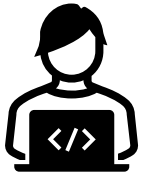
- The setMaxInstances method of the mPageService has three parameters. These parameters are:
  1. maxInstances represents the number of instances allowed to run at once. This lets you have your MPage call more than one CCL script at a time. Typically, 2 is a good number but if your MPage loads many scripts at once and requires results to be returned as fast as possible you can safely turn this number up to 4 (I typically get all my MPages working fine with a value of 2).
  2. enableLog can be set as either true or false. If set to true, the <mpage-log-component></mpage-log-component> can be used. If set to false, the log component will not display, and log results will not be captured in memory.
  3. mode can be set to either 'CHART' or 'ORG'. This setting should be based entirely on whether you are going to display your MPage at the Chart or Organizer level in PowerChart.
- If you are using the proxy server in development mode, you must complete the setup before calling setMaxInstances by setting this.mPage.enableProxy = true and by setting your this.mPage.contextRoot to your site value. Instructions are included in the ie-mpage template.

APPLICATION INITIALIZATION

- Open app.component.html and erase the content and replace it with the following code:

```
<h1>Patient History</h1>

<router-outlet></router-outlet>

<mpage-log-component></mpage-log-component>
```

- Viewing your MPage in your browser should now just show the title “Patient History”.
- We have two components we are running in our main application component. The first is a special Angular feature called the router-outlet. It is a container that handles routing different pages in your application. Since we have not built any routes yet, nothing shows here.
- The mpage-log-component is the Clinical Office Debug logger. This can be viewed at any time by pressing CTRL+Z in your browser.



PROXY SERVER VS HOSTING

- The absolute best way to develop your MPages is by using the Angular proxy host. The proxy host gives you the ability to develop your MPages and see any changes in real time. The proxy offers significant development time savings.
- Some clients however may have situations where staff cannot develop their MPages using the proxy server as they cannot access the Cerner Discern MPage API from their development machine. This is typical for remote workers who are developing their code locally.
- If you are in a position where you cannot use the proxy server, you have two solutions available.
  1. Host your MPage on a web server available through the Cerner/Citrix firewall.
  2. Deploy your MPage to the custom_mpage_content folder on Cerner and refresh the WebSphere server.

HOSTED MPAGE

- All hosted Angular MPages need to be compiled before deployment. We are going to do this now as it is also required to deploy your completed MPage's to end users.
- Open a terminal window in either Visual Studio code or from windows and from your project folder, type in:

```
ng build
```

- Your application will start it's first compile. The first compile can take a few minutes and the speed is dependent entirely on the capabilities of your development PC.
- When the compile has completed, you will see a new folder in your project called "dist". This folder contains the compiled ready-to-run version of your MPage.

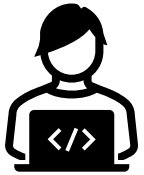
```
Initial Chunk Files                    | Names         |      Size
main.3751eb724c3d7436df7f.js           | main          | 922.21 kB
styles.066e1476cbe22f051f90.css        | styles        | 142.00 kB
polyfills-es5.d7b47112499fa1a2d4b6.js  | polyfills-es5 | 138.16 kB
polyfills.484de0fc26f72fe50a11.js      | polyfills     |  42.60 kB
runtime.b7466179984b81ec2429.js        | runtime       |   1.31 kB

                                       | Initial Total |   1.22 MB

Build at: 2021-11-25T15:09:53.340Z - Hash: 3ba81e38582c9ea8efa3 - Time: 161876ms
```

HOSTED MPAGE

- Copy the content of your dist folder to either your external web server or if you are hosting inside of Cerner on WebSphere you can copy to I:\Winintel\Static_Content\custom_mpage_content.
- If you have copied your distribution to the custom_mpage_content folder you need to take an additional step to refresh the content on WebSphere.
- You can determine the location of the manager page to refresh custom_mpage_content by running the following CCL statement.

```
select build(info_char,"/manager")
from dm_info
where info_domain = "INS" and info_name = "CONTENT_SERVICE_URL"
```

- Copy the value returned by the statement into the navigation bar of any of your open Cerner folders to launch the page from within the Cerner Citrix environment.



HOSTED MPAGE

- You should now see the MPages Static Content Management Page. On this page will be a directory called “custom_mpage_content”. Click the Refresh button on the same line to start the refresh.

MPages Static Content Management Page

This page provides status information for all the cached folders. The column 'Primed' will show if the folder has been cached. Clicking on the Refresh Button refreshes the cache for the folder. The 'Additional information' column provides any error messages. If the folder is refreshing the column 'Is Refreshing?' will show 'yes'.

The static content base path is:/media/codewarehouse/b0783/Winintel/Static_Content

Directory	Group	Refresh	Pending Refresh Request?	Server ID (set randomly)	Can Read?	Is Primed?	Is Refreshing?	Refresh Start Date Time	Refresh End Date Time	Additional Information
care-pathways-build-tool	CARE_PATH_BUILD_TOOL_BASE	Refresh	No	ba5ade73-d234-46b7-b2b9-92c1296b7d16	Yes	Yes	No	--	--	
			No	fde1dd8b-0209-452d-98b0-20dac8a4d12d	Yes	Yes	No	--	--	
CareMgmtContent	Not Applicable	Refresh	No	ba5ade73-d234-46b7-b2b9-92c1296b7d16	Yes	Yes	No	--	--	
			No	fde1dd8b-0209-452d-98b0-20dac8a4d12d	Yes	Yes	No	--	--	
CareTeamBuildTool	Not Applicable	Refresh	No	fde1dd8b-0209-452d-98b0-20dac8a4d12d	Yes	Yes	No	--	--	
ChargesBuildTool	Not Applicable	Refresh	No	fde1dd8b-0209-452d-98b0-20dac8a4d12d	Yes	Yes	No	--	--	
configuration-mgr	BED_MP_CONFIG_MGR	Refresh	No	ba5ade73-d234-46b7-b2b9-92c1296b7d16	Yes	Yes	No	--	--	
			No	fde1dd8b-0209-452d-98b0-20dac8a4d12d	Yes	Yes	No	--	--	
cp-unified-build-tool	CP_UNIFIED_BUILD_TOOL_BASE	Refresh	No	ba5ade73-d234-46b7-b2b9-92c1296b7d16	Yes	Yes	No	--	--	
			No	fde1dd8b-0209-452d-98b0-20dac8a4d12d	Yes	Yes	No	--	--	
custom_mpage_content	Not Applicable	Refresh	No	fde1dd8b-0209-452d-98b0-20dac8a4d12d	Yes	Yes	No	--	--	

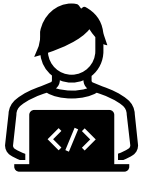
HOSTED MPAGE

- Regardless of if you are hosting your MPage on an external web server or through the Cerner WebSphere server, you can use the same CCL script in Discern Visual Developer to test your MPage.
- Open discernvisualdeveloper.exe and from the Build menu choose Run Prompt Program. Alternatively, you can press CTRL+R to access the same screen.
- The program to run value should be set to **1co_mpage_redirect:group1**
- Parameters should be ^MINE^,^page location^ where page location is the name of your distribution folder if hosted on WebSphere.

```
^MINE^,^mpage-training-john-simpson^
```

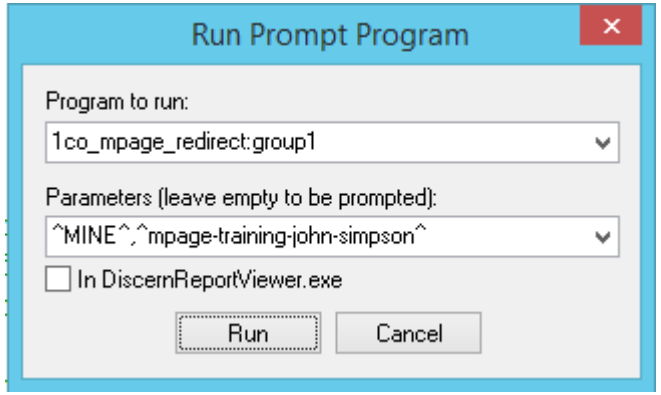
- If you are hosting on your own server, you need to include the full path to the page.

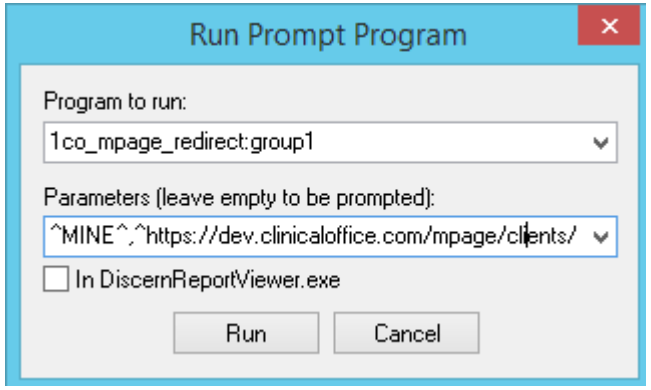
```
^MINE^,^https://dev.clinicaloffice.com/mpage/clients/mpage-training-john-simpson/index.html^
```

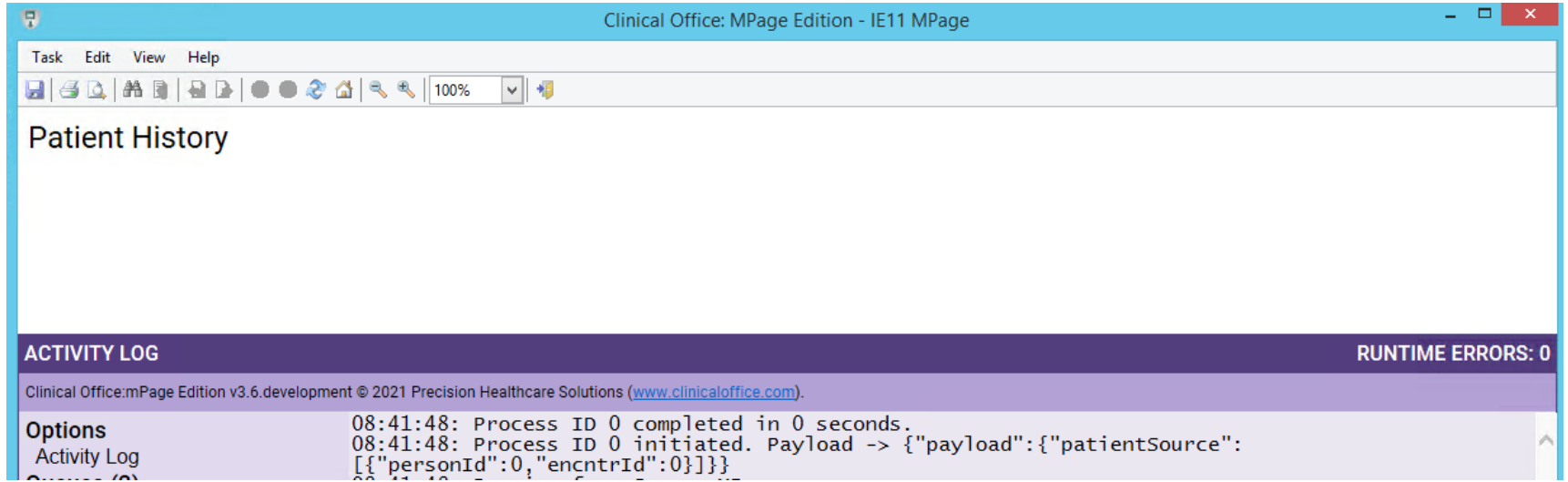


HOSTED MPAGE

- Click the Run button and you should see your MPage appear.

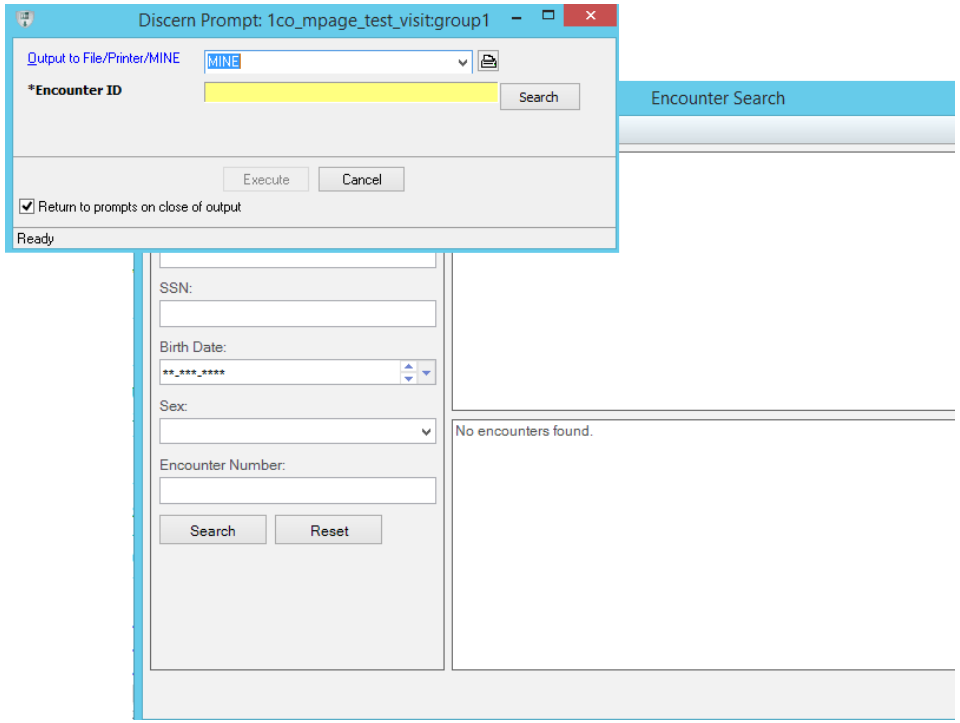






TESTING WITH PATIENT CONTEXT

- Our MPage should now have a successful connection to Cerner no matter if you have run it with the proxy server or through Discern Visual Developer.
- If you were running this MPage through PowerChart it would have a patient context associated to it as you would have opened a patient chart to view the MPage.
- You can force patient context with the CCL script `1co_mpage_test_visit:group1`. Simply run this script from either Discern Visual Developer or add it to DA2 if you have a developer folder setup. The source name for this script is 1co_mpage_test_visit.prg.
- When run, simply click the Search button to see the standard Cerner visit search dialog. Fill in your search, select Ok and click the Execute button on the original parameter screen. You will see a report showing your patient has been assigned.



TESTING WITH PATIENT CONTEXT

- From now on, whenever you open any Clinical Office MPage that requires patient context from anywhere other than the patient chart, you will be using your chosen visit.
- If you open your MPage from a patient chart in PowerChart, the value in PowerChart will be the value used on the MPage.
- You can change your test patient any time by running the script again.
- The test patient is tied to you alone and other developers will need to set their own values.
- Patient context is only needed for chart level MPages. Organization level pages render data for more than one patient at a time and are either run from the organizer on PowerChart or from other tools such as DA2.

CLINICAL OFFICE SERVICES

Angular can expose functionality and data using services.

All Clinical Office data retrieval is performed through custom Angular services.

The <mpage-log-component> element we used in our MPage to display the activity log is an example of an Angular component.

During this course we will be building our own components to display information.

First however we need to understand how to access data from Cerner with Clinical Office services.

MPAGESERVICE

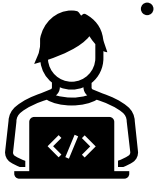
- The mPageService contains the core data retrieval functionality of Clinical Office. All other Clinical Office services extend mPageService.
- mPageService uses specific payload format.
- You can perform one or more data retrieval tasks in a single payload.
- Payload data is only accessible from the core service it is derived from. For example, calling the “person” payload option will load person specific data but you can only access that data from the PersonService.
- The activity log component uses all of the available MPage services to display data from each type of payload tag.

MPAGE SERVICE

- Add the following code to app.component.ts in the ngOnInit() method immediately below the setMaxInstances line to load basic person level information for our selected patient.

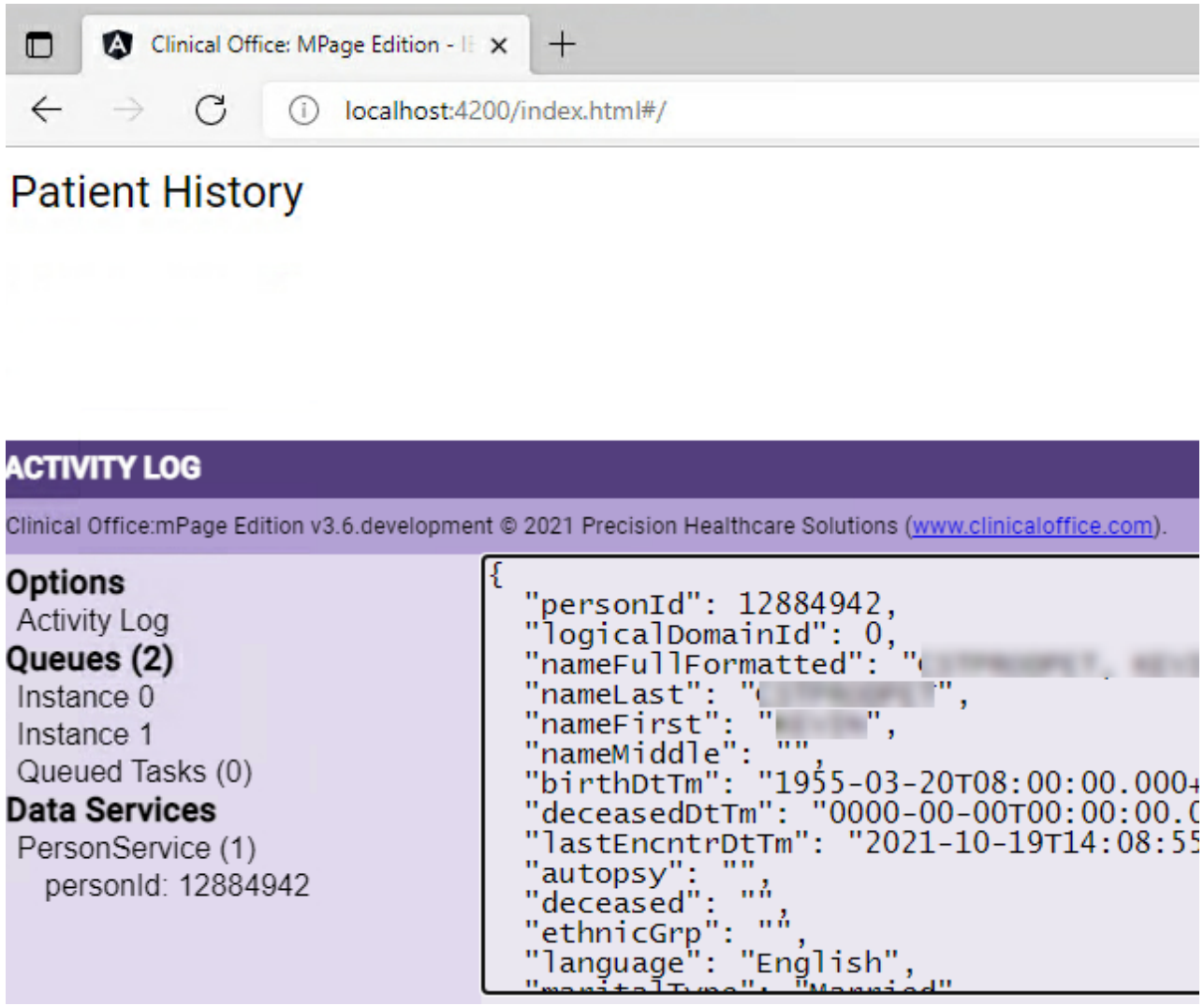
```
this.mPage.executeCCL({
  payload: {
    patientSource: [{personId: 0, encntrId: 0}],
    person: {}
  }
});
```

- The executeCCL method sends a payload to the CCL script 1co3_mpage_entry which is responsible for parsing the payload and calling any necessary scripts to collect data.
- In Chart mode, if you pass 0 for the personId and encntrId, the values for the current person/encounter will be used.
- If you pass different personId/encntrId values as the patientSource, information for that person/encounter will be used.
- Adding person to the payload with empty properties uses default settings which returns basic person information.



MPAGE SERVICE

- After updating your code, open your MPage in your browser and press CTRL+Z to open your debugger.
- You should now see an entry under Data Services called PersonService. The PersonService indicates that 1 record has been loaded into memory.
- Click on “PersonService” to expand the section. You will see the personId for your patient appear.
- Click on the “personId” to view the data loaded for your patient.
- This data is part of the PersonService service. To display this information on your Mpage you will need to add PersonService to any component that displays or uses the information.



MPAGE SERVICE

- Many payload options have filters available that can customize the data being retrieved.
- Let's add person_patient data as well as person_alias data to our payload to see it in action.
- Modify your payload in your executeCCL tag as follows:

```
this.mPage.executeCCL({
  payload: {
    patientSource: [{personId: 0, encntrId: 0}],
    person: {
      patient: true,
      aliases: true
    }
  }
});
```

- Recompile/Deploy/Refresh to see the differences.

NAVIGATING THE ACTIVITY LOG

- The activity log is a great place to see how your MPage interacts with CCL. It is opened and closed by holding CTRL+Z.
- The activity log is divided into three sections which are:

1. Options – Contains the Activity Log which is a history of when requests are made to CCL as well as how long the request took to run.

2. Queues – Lists each queue and a Queued Tasks section. Each instance shows the data that was most recently retrieved, and the queued tasks shows the requests that still must be sent to a queue when available.

3. Data Services – Any data service that has data can be viewed here. So far, we have only accessed data in the PatientService. As other services are called data will appear.

DISPLAYING DATA ON OUR MPAGE

- At this point, we are now in a good place to start talking about how we are going to put our MPage together.
- Currently we have a single component called app that displays our title, displays routing content and finally displays our activity log.
- We could theoretically put all MPage content inside the app component. This however is bad programming practice and loses some of the incredible functionality Angular offers with custom components.
- Instead, we will be building a series of single task components that each have their own purpose and responsibility.
- These components will be self-contained and can be copied and used in other MPages.

DISPLAYING DATA ON OUR MPAGE

- While building our MPage, we will create Angular components that display the following information:
  - Patient name, MRN, birth date, gender and PCP
  - Allergies
  - Problems
  - Diagnosis
  - Encounter history
  - Appointment history

BASIC PATIENT DEMOGRAPHICS COMPONENT

- From the terminal in Visual Studio Code, create a new component called demographics with the following command.

```
ng generate component demographics
```

- This command will create a new folder in your src/app folder called demographics.
- In this folder you will have four files representing your component with the extensions html, ts, scss and spec.ts.
- We can safely delete the .scss file provided we remove styleUrls from the .ts file.
- If you don't feel like typing the full ng generate component demographics, you could use the short cut:

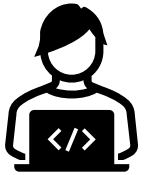
```
ng g c demographics
```

DISABLING COMPONENT STYLE SHEET CREATION

- If you do not plan on using any component styles, you can configure your application to prevent the creation of style sheets during the creation of new components.
- To do this, simply add the “inlineStyle”: true value to the “@schematics/angular:component” section in angular.json as shown below.

```
      "schematics": {
        "@schematics/angular:component": {
          "style": "scss",
          "inlineStyle": true
        },
```

- You can still use component specific styles after turning this setting off however you will need to manually create the .scss file and add the styleUrls parameter to your @Component decorator.

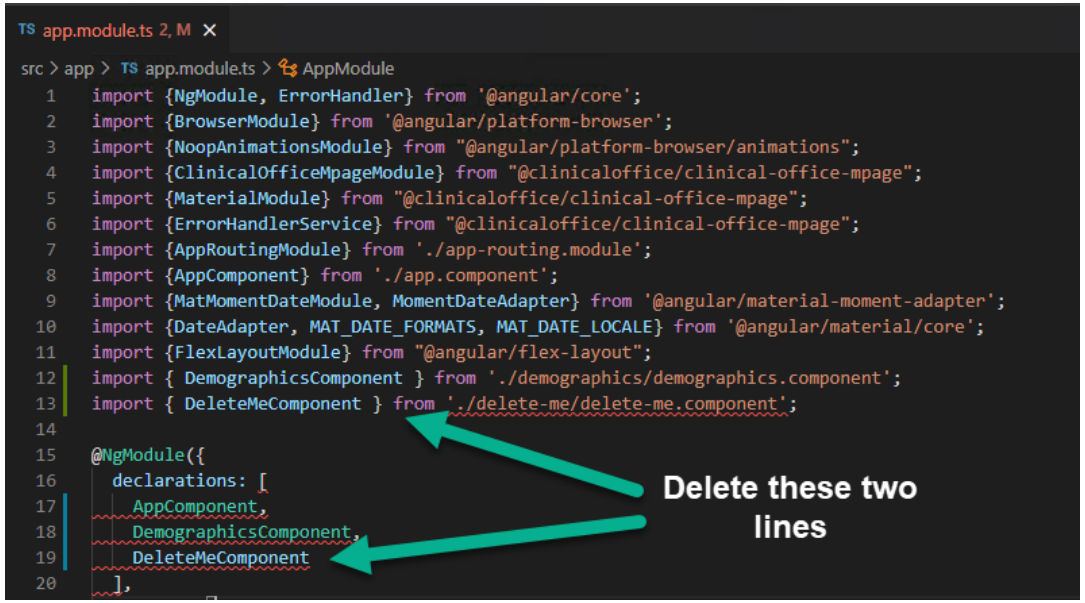


DELETING AN UNWANTED COMPONENT

- Sometimes you want to permanently remove a component from your Angular application.
- Let's step through how you would do this by creating the following component. Type in:

  ng generate component delete-me

- You should now see a new component folder called delete-me.
- First, right click on the folder and choose delete. This removes the files from your project.
- Next, you need to open app.module.ts and remove both the import line and declaration line for your DeleteMeComponent.



BASIC PATIENT DEMOGRAPHICS COMPONENT

- Our component needs to be called in order to use it. This is done by simply adding it to our app.component.html file.
- Replace the <h1>Patient History</h1> element with <app-demographics></app-demographics>.
- Your app.component.html should appear as follows.

```
<app-demographics></app-demographics>

<router-outlet></router-outlet>

<mpage-log-component></mpage-log-component>
```

- If you compile/deploy/refresh your MPage you should now see the words "demographics works!" where you previously saw the text "Patient History"

BINDING VARIABLES

- Angular has incredible capabilities for binding variables. By simply enclosing a variable or method name with a binding tag {{ }} you will always have the most current results showing on screen.
- This type of activity is part of what is referred to as reactive web development. The idea is that your webpage is in a constant living space that adjusts and changes to data and user actions.
- Reactive web development isn't a tough concept once you understand what it really is. Just like other highly reactive applications such as video games, your program runs in one big loop waiting for things to happen.
- This is how data is refreshed instantly on screen.

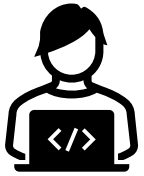
BINDING VARIABLES

- Let's do a quick test to demonstrate.
- Open demographics.component.html and add the following code.

```
<p>demographics works!</p>

{{ test }}
```

- Next, open demographics.component.ts and add the following lines of code right before your constructor statement.

```
export class DemographicsComponent implements OnInit {
  loopValue = 0;

  get test(): number {
    return this.loopValue++;
  }

  constructor() { }
```



BINDING VARIABLES

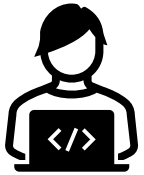
- When you view your output, you should see the words “demographics works!” followed by a number increasing in value.
- The reason for this is that our get test() method increments the value of this.loopValue by one right before it returns the value to our page.

```
get test(): number {
  return this.loopValue++;
}
```

- As the page progresses through its constant loop, the test() method is run every iteration of that loop increasing the value of this.loopValue.
- This should be something you always take note of as it is one of the biggest reasons for performance issues that can happen in your Angular application. **Always make sure that any methods you call from your HTML performs as little as possible to ensure optimal performance.**

PATIENT DEMOGRAPHICS

- Let's put some actual patient data on our component.
- Remove the loopValue variable and test methods from demographics.component.ts and all the content from demographics.component.html.
- To display information from the PersonService service, we must inject the service into our component.
- This is done by importing the PersonService and injecting the service as an object in the constructor. From there we can use the service anywhere in our component.



PATIENT DEMOGRAPHICS

- Modify your demographics.component.ts file to match the following code:

```
import { Component, OnInit } from '@angular/core';
import { PersonService } from '@clinicaloffice/clinical-office-mpage';

@Component({
  selector: 'app-demographics',
  templateUrl: './demographics.component.html'
})
export class DemographicsComponent implements OnInit {

  constructor(public personService: PersonService) { }

  ngOnInit(): void {
    this.personService.load('PERSON_PATIENT');
  }

}
```

PATIENT DEMOGRAPHICS

- The import { PersonService } line is required to allow our component the ability to use the PersonService code and data.

```
import { PersonService } from '@clinicaloffice/clinical-office-mpage';
```

- In the constructor, we inject the PersonService as a new public object called personService. As a public object, the personService object is available everywhere in our component.

```
  constructor(public personService: PersonService) { }
```

- In the ngOnInit method, we make use of the personService object and trigger the load method to load our current patient.

```
  ngOnInit(): void {
    this.personService.load('PERSON_PATIENT');
  }
```

PATIENT DEMOGRAPHICS

- If you view your patient in the debugger, you will see much more information available than our previous load.
- The additional data is due to the use of a payload tag called “PERSON_PATIENT”.
- Payload tags are available on many of the Clinical Office services and are designed to give you default options for common tasks. For example, the PersonService offers the payload tags “PERSON_MIN” and “PERSON_PATIENT”. The Clinical Office API documentation found at www.clinicaloffice.com covers all data service configuration options.
- The “PERSON_PATIENT” payload tag is the same as including the following parameters.

```
person: {
  includeCodeValues: true,
  aliases: true,
  patient: true,
  names: true,
  personInfo: true,
  prsnlReltn: true,
  personReltn: true,
  orgReltn: true
}
```

PATIENT DEMOGRAPHICS

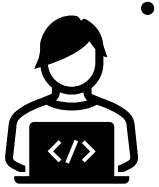
- Open the activity log.
- In the PersonService section you will see a single entry for our patient. It now includes additional information such as physician names and previous names.
- If you look at the activity section in the activity log you will see that we have triggered the load of PersonService for the same patient twice. The first time in our app.component.ts file and the second in demographics.component.ts.

```
07:40:06: Process ID 0 completed in 0 seconds.
07:40:05: Process ID 0 initiated. Payload -> {"payload":{"patientSource":[{"personId":0,"encntrId":0}],"person":
{"includeCodeValues":true,"aliases":true,"patient":true,"names":true,"personInfo":true,"prsnlReltn":true,"personReltn":true,"orgReltn":true}}}
07:40:05: Process ID 1 completed in 0 seconds.
07:40:05: Process ID 0 completed in 0 seconds.
07:40:05: No open process instances available. You should think of setting a higher maxInstances value or bulk loading your payload.
07:40:05: Process ID 1 initiated. Payload -> {"payload":{"patientSource":[{"personId":0,"encntrId":0}],"person":{"patient":true,"aliases":true}}}
```

- This is not only inefficient, but it can also lead to bugs in our code as the PersonService object stores information by person_id and if the first run took longer than the second more descriptive run, information would be overwritten.

PATIENT DEMOGRAPHICS

- The solution is to remove one of the calls to the PersonService.
- While designing our MPage, we need to consider the best time and place to load our services. If the information for the service is going to be used in a single place (such as our demographics component), placing the code to load the service in the component makes the most sense.
- If the data in a service is going to be used across multiple components but never changed (e.g., pre-loading code sets), the best place would be in app.component.ts or an application specific service.
- If your service is needed in multiple components and will be refreshed or new data loaded, you should create your own application specific service that can be shared between your components.
- The MPage we are building does not need an application specific service however we will cover how to create a custom service at the end of this course.
- Open your app.component.ts file and remove the call to this.mPage.executeCCL.



PATIENT DEMOGRAPHICS

- Open demographics.component.html and add the following code:

```
<ng-container *ngIf="personService.isLoaded()">
    <div class="callout">
        <h2>{{ personService.get().nameFullFormatted }}</h2>
        <div class="sideways-list">
            <ul>
                <li><strong>MRN:</strong> {{ personService.getAlias("MRN").aliasFormatted }}</li>
                <li><strong>DOB:</strong> {{ personService.get().birthDtTm | date : 'MM/dd/yyyy' }}</li>
                <li><strong>Gender:</strong> {{ personService.get().sex }}</li>
                <li><strong>PCP:</strong> {{ personService.getPrsnlReltn("PCP").nameFullFormatted }}</li>
            </ul>
        </div>
    </div>
</ng-container>
```

PATIENT DEMOGRAPHICS

- Most of the code you just typed in is standard HTML.
- There are however a few Angular specific things going on.
- <ng-container> is an Angular specific HTML element that does not render anything on the page but is instead used to create conditional logic blocks for displaying data within the opening and closing <ng-container> tags.
- In our example, we use the *ngIf directive to return a boolean value from the PersonService isLoaded() method.

```
<ng-container *ngIf="personService.isLoaded()">
```

- If our patient has been loaded, any content inside the <ng-container> element is rendered on the page.

PATIENT DEMOGRAPHICS

- The curly braces {{ }} allow you the ability to output the results of variables and methods available within your component.
- In our example we call a number of the available methods in the PersonService to display values such as nameFullFormatted, MRN, DOB, gender and the name of the Primary Care Provider.

```
<h2>{{ personService.get().nameFullFormatted }}</h2>
```

- Angular pipes are special filters that format data. We use the standard date pipe to format our DOB.

```
<li><strong>DOB:</strong> {{ personService.get().birthDtTm | date : 'MM/dd/yyyy' }}</li>
```

PATIENT DEMOGRAPHICS

- Refresh your MPage to see the changes to your output.

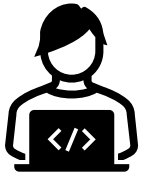
- This output does the job of displaying our information, but it isn't very attractive and wastes a fair bit of screen real estate.
- We used a standard <h2> Header 2 tag for our patient's name and put the remaining information in a <ul> unordered list.
- You may have noticed that we assigned a few CSS class names to our div tags. These CSS names don't exist yet and will be responsible for improving the visual appearance of our component.

PATIENT DEMOGRAPHICS

- Modify the styles.css file in your src folder as follows:

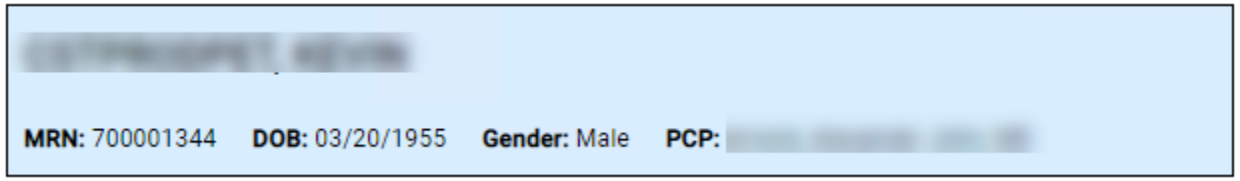
```
// * Import theme from theme generator *//
@use '~@angular/material' as mat;
@import "theme";

// * Custom CSS *//
.callout {
    background-color: #d8eeff;
    border: 1px solid black;
    padding: 0.5rem;
}
```

```
.sideways-list ul {
    list-style-type: none;
    margin: 0;
    padding: 0;
}

.sideways-list li {
    display: inline-block;
    padding-right: 1.5em;
}
```



PATIENT DEMOGRAPHICS

- When deployed, your component will now appear in the same format as the sample below.



- The “callout” class is responsible for setting the background color, border and padding between the border and content.
- The “sideways-list ul” class changes the behavior of the unordered list, so the bullets are dropped off and all padding and margins are removed.
- Finally, the “sideways-list li” class changes the display mode to inline-block forcing all elements to appear horizontally instead of vertically. The right-side padding of 1.5em is applied to space the list items apart.

PATIENT DEMOGRAPHICS

- Our basic patient demographics component is now complete.
- Although this example is a simple component, it does demonstrate how few lines of code are needed to build MPage content in Angular with Clinical Office.
- In our demographics.component.ts file, we were able to edit the standard generated Angular component by adding/modifying 3 lines of code to accomplish our data collection needs.
- Our demographics.component.html file is primarily standard HTML with a few Angular directives that make calls to our Clinical Office data service. In total the HTML file is 13 lines of code.
- Finally, we created three CSS style tags that can be used for any HTML elements in our application. Our styles.scss file currently sits at 21 lines of code.
- Other than the CSS, our demographics component is completely self contained and can be dropped in any Clinical Office Angular application. We could have put the CSS inside demographics.component.scss to make it 100% self-contained.

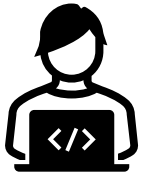
ALLERGIES COMPONENT

- Create a new component called allergies by typing the following from your command line:

  ng g c allergies

- If you turned off the generation of CSS files you will see three new files in a folder called allergies. If you did not turn off CSS creation you will see 4 files.
- Your app.module.ts file also now has the imports defined for AllergiesComponent.
- Your new allergies component will be used to display allergy information in a table format.
- Before your component can be used, it must be added to your application. Open app.component.html and add the allergies component as follows.

```
<app-demographics></app-demographics>
<app-allergies></app-allergies>

<router-outlet></router-outlet>

<mpage-log-component></mpage-log-component>
```



ALLERGIES COMPONENT

- Open allergies.component.ts and modify your code as follows:

```
import { Component, OnInit } from '@angular/core';
import { AllergyService  } from '@clinicaloffice/clinical-office-mpage';

@Component({
  selector: 'app-allergies',
  templateUrl: './allergies.component.html'
})
export class AllergiesComponent implements OnInit {

  constructor(public allergyService: AllergyService) { }

  ngOnInit(): void {
    this.allergyService.load();
  }

}
```

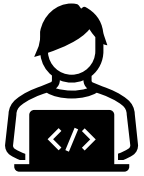
ALLERGIES COMPONENT

- Open allergies.component.html and modify your code as follows:

```
<div class="component-container">
    <h2>Allergies</h2>
    <p>{{ allergyService.toString() }}</p>
</div>
```

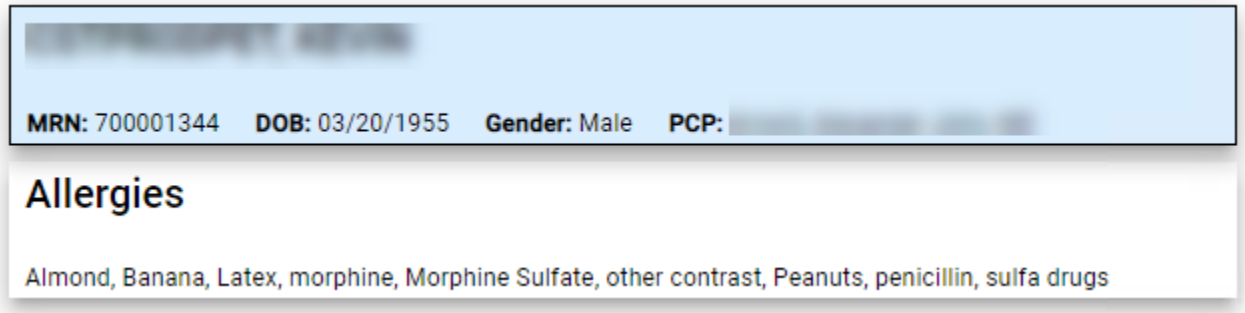
- Open styles.css and change the .callout section to appear as follows:

```
.callout {
    background-color: #d8eeff;
    border: 1px solid black;
}

.component-container, .callout {
    @include mat-elevation(8);
    margin-top: 0.5rem;
    padding: 0 0.5rem;
}
```



ALLERGIES COMPONENT

- Our page should now appear as follows:



- Including mat-elevation(8) is an Angular Material style that creates the shadow effect we see on our page. Changing the value from 8 to lower or higher values will affect the appearance of the shadow.
- CSS styles can be grouped together (.component-container, .callout) to reduce the amount of code you type for values that are the same.

ALLERGIES COMPONENT

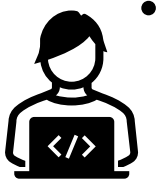
- The allergy component does clearly show our patient allergies with the toString method, but we can do much better.

```
<p>{{ allergyService.toString() }}</p>
```

- Delete the line above from your component and create the following table.

```
<table>
    <tr><th>Type</th><th>Substance</th></tr>
    <tr *ngFor="let allergy of allergyService.get()">
        <td>{{ allergy["substanceType"] }}</td>
        <td>{{ allergy["substance"] }}</td>
    </tr>
</table>
```

- Refresh your page to see the results.



ALLERGIES COMPONENT

- We now have a basic HTML table with our allergies that you could add CSS style properties to.
- Our table row <tr> element makes use of the Angular *ngFor directive. The *ngFor directive is incredibly powerful and will be used in all but the most trivial Angular applications.
- *ngFor loops through any type of iterator object (e.g., array), and repeats the HTML element for that object. For example, if our patient has 5 allergies, 5 <tr> elements would be created.
- Any content inside of an *ngFor statement has access to the singular item (variable or object), derived in the loop.
- This is how our two <td> elements know which allergy we are displaying substance type and substance information for.

ALLERGIES COMPONENT

- The *ngFor directive made it easy to loop through our table data however Angular Material has a table control that offers a consistent look and feel as well as functionality such as sorting and pagination.
- Delete or comment out the following block of code from your MPage.

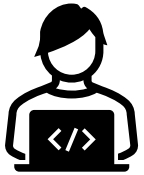
```
<table>
    <tr><th>Type</th><th>Substance</th></tr>
    <tr *ngFor="let allergy of allergyService.get()">
        <td>{{ allergy["substanceType"] }}</td>
        <td>{{ allergy["substance"] }}</td>
    </tr>
</table>
```

ALLERGIES COMPONENT

- Return to allergies.component.ts
- Ensure your imports have the following code:

```
import { Component, OnInit } from '@angular/core';
import { AllergyService, IAllergy } from '@clinicaloffice/clinical-office-mpage';
import { MatTableDataSource } from '@angular/material/table';
```

- IAllergy is an interface that describes how an allergy object should appear. We could get away with using the any type however, when possible, it is good practice to use a proper interface as it helps in debugging.
- The MatTableDataSource is an object that is required by the Angular Material Table component to store the data for our table.

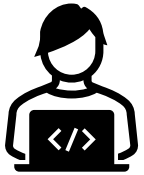


ALLERGIES COMPONENT

- Add the following code to your class right before the constructor statement.

```
export class AllergiesComponent implements OnInit {
  isReady = false;
  displayedColumns: string[] = ['substance_type', 'substance', 'substance_identifier'];
  dataSource: MatTableDataSource<IAllergy> = new MatTableDataSource();
```

- isReady is a variable we will be using to determine our data has been assigned to the Angular Material data source.
- displayedColumns is an array that is required by the Angular Material table to indicate identifiers for the columns that will be displayed.
- dataSource is an object of a MatTableDataSource type using the interface of IAllergy to define the field types inside the data source.

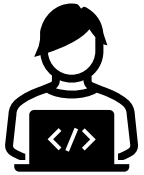


ALLERGIES COMPONENT

- Add the following method to your class.

```
// Determine if data has loaded and assign to data source
get ready(): boolean {
  if (!this.isReady) {
    const allergies = this.allergyService.get();
    if (allergies[0].allergyId > 0) {
      this.dataSource.data = allergies;
      this.isReady = true;
    }
  }
  return this.isReady;
}
```

- This method is a get accessor method. As a get accessor it cannot accept any parameter values and must return a value. We will be using it later in our HTML to determine if the data source is ready. If it is not ready and the CCL has returned our allergy data, it will populate the data source.



ALLERGIES COMPONENT

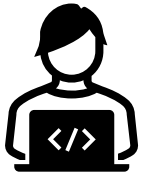
- Return to allergies.component.html and add the following code.

```
<!-- Display the allergy table if loaded -->
<ng-container *ngIf="ready">
    <table mat-table [dataSource]="dataSource">
        <!-- Define table columns -->
        <ng-container matColumnDef="substance_type">
            <th mat-header-cell *matHeaderCellDef>Substance Type</th>
            <td mat-cell *matCellDef="let element">{{ element["substanceType"] }}</td>
        </ng-container>
        <ng-container matColumnDef="substance">
            <th mat-header-cell *matHeaderCellDef>Substance</th>
            <td mat-cell *matCellDef="let element">{{ element["substance"] }}</td>
        </ng-container>
        <ng-container matColumnDef="substance_identifier">
            <th mat-header-cell *matHeaderCellDef>Substance ID</th>
            <td mat-cell *matCellDef="let element">{{ element["substanceIdentifier"] }}</td>
        </ng-container>
```

ALLERGIES COMPONENT

```
        <!-- Define rows -->
        <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
        <tr mat-row *matRowDef="let row; columns: displayedColumns"></tr>
    </table>
</ng-container>
```

- The syntax above uses some concepts we have already covered such as <ng-container> however many of the Angular Material Table component elements have other attributes and directives we have not encountered before.
- In Angular, you can create your own directives that let you extend the functionality of your components. This is an advanced technique outside the scope of this course. You can however still use these directives without knowing how the Angular Material team built them.



ALLERGIES COMPONENT

- Let's step through how this works starting with the first ng-container.

```
<ng-container *ngIf="ready">
    Do some code here
</ng-container>
```

- Every iteration of our application loop will run this code and changes will appear as they happen. If the value returned from our ready get method returns true, anything between the <ng-container></ng-container> elements is executed.
- We could display a loading message on the component by performing a check for *ngIf="!ready" followed by a message inside the ng-container tags.

ALLERGIES COMPONENT

- At its core, the Angular Material Table still renders as an HTML table and requires the HTML table elements on the page.

```
<table mat-table [dataSource]="dataSource">
```

- Our table statement uses an Angular Material directive called mat-table which is interpreted by Angular at runtime to generate the table through Angular Material Table.
- The [dataSource] input gives you the ability to specify which data source is used on our table. You created a Material Table Data Source objected called dataSource in the allergies.component.ts file. This is the value passed to the [dataSource] input defined above.

```
dataSource: MatTableDataSource<IAllergy> = new MatTableDataSource();
```

ALLERGIES COMPONENT

- Inside our <table> element, we break from the usual HTML standard and define our column headers and cells outside of their row definitions. This is an Angular Material Table design decision.
- Each cell is defined inside an <ng-container> and has a matColumnDef assignment that matches the values inside our displayedColumns array we created in the .ts file.

```
<ng-container matColumnDef="substance_type">
    <th mat-header-cell *matHeaderCellDef>Substance Type</th>
    <td mat-cell *matCellDef="let element">{{ element["substanceType"] }}</td>
</ng-container>
```

- Our <td> element has a *matCellDef assignment that states “let element”. This assigns the content of each row to an object called element which we can use the same way we did in our *ngFor demonstration earlier.

ALLERGIES COMPONENT

- Our final step is to define the two different types of table rows. The first is our header row and it is declared with the mat-header-row directive. The second row renders our table data using the mat-row directive and a special *matRowDef attribute that functions the same as our *ngFor from earlier.

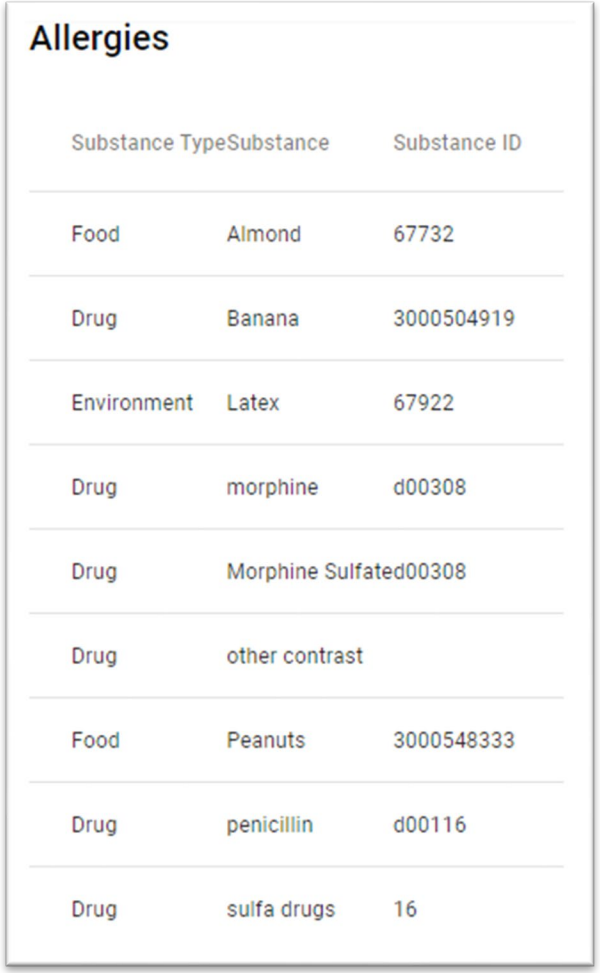
```
<tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
<tr mat-row *matRowDef="let row; columns: displayedColumns"></tr>
```

- In each of the definitions above you must specify which columns will be used. These of course are contained in our displayedColumns array.
- A more advanced table could have different arrays used to display different columns of data.

ALLERGIES COMPONENT

- Viewing our table shows that the table is now using Angular Material however it still needs some work to look great.
- Our table isn't very wide, and the Allergies title has a large amount of space below it.
- Add the following CSS to styles.scss to fix these issues.

```
table {
    width: 100%;
}

h2 {
    padding-bottom: 0;
    margin-bottom: 0 !important;
}
```



Allergies

Substance Type	Substance	Substance ID
Food	Almond	67732
Drug	Banana	3000504919
Environment	Latex	67922
Drug	morphine	d00308
Drug	Morphine Sulfated	00308
Drug	other contrast	
Food	Peanuts	3000548333
Drug	penicillin	d00116
Drug	sulfa drugs	16

ALLERGIES COMPONENT

- Currently our table will display every row it has data for. If your patient has 100 allergies the screen will be filled with allergies and be difficult to view other components we create.
- A paginator is an Angular Material component that lets you define a specific number of table rows that can be displayed at a single time and offers navigation buttons to scroll through multiple pages of data.
- Add the following paginator code to allergies.component.html immediately after the closing ng-container element to add a paginator with the name #pageinator. ****** NOTE *** This must be outside of the <ng-container> as the element name paginator must always exist on the component.***

```
        </table>
    </ng-container>

    <!-- Paginator -->
    <mat-paginator #paginator [pageSizeOptions]="[5, 10, 25]" showFirstLastButtons>
    </mat-paginator>
```

ALLERGIES COMPONENT

- Open allergies.component.ts and modify the imports to include ViewChild and MatPaginator as shown below:

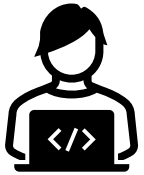
```
import { Component, OnInit, ViewChild } from '@angular/core';
import { AllergyService, IAllergy } from '@clinicaloffice/clinical-office-mpage';
import { MatTableDataSource } from '@angular/material/table';
import { MatPaginator } from '@angular/material/paginator'
```

- Directly below your declaration for dataSource, add the following line of code:

```
@ViewChild('paginator', {static: true}) paginator!: MatPaginator;
```

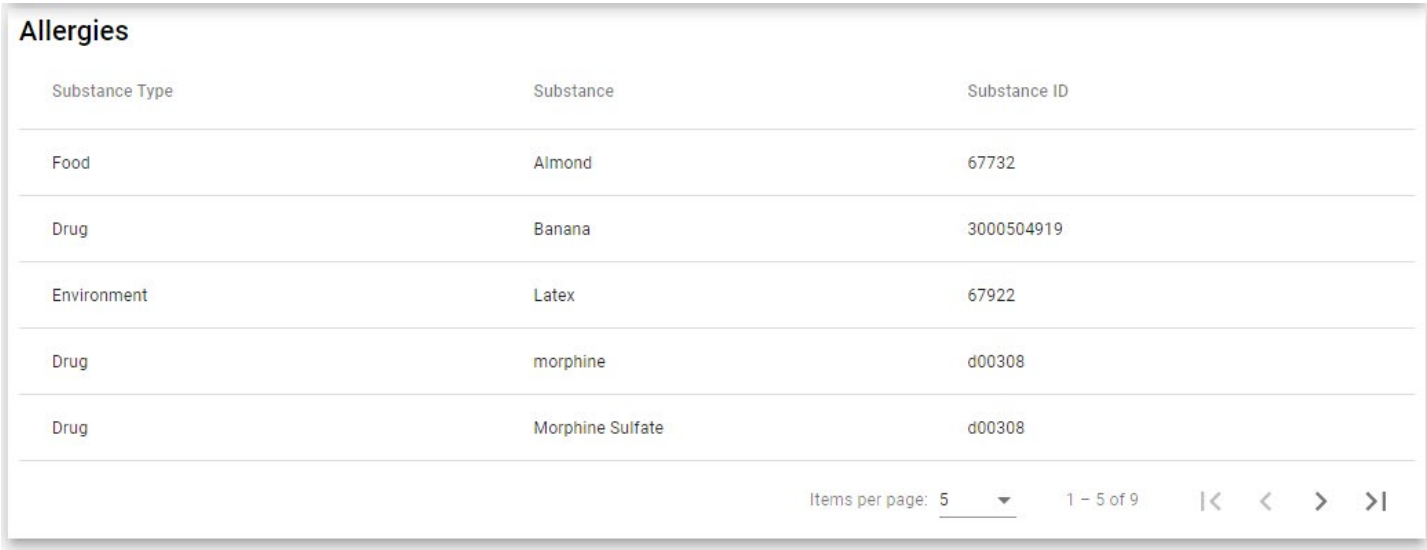
- Finally, modify your ngOnInit method to appear as follows:

```
ngOnInit(): void {
    this.allergyService.load();
    this.dataSource.paginator = this.paginator;
}
```



ALLERGIES COMPONENT

- Your allergies component should now show paginator buttons on the bottom right defaulting to 5 rows of allergies on the table at once.



Allergies

Substance Type	Substance	Substance ID
Food	Almond	67732
Drug	Banana	3000504919
Environment	Latex	67922
Drug	morphine	d00308
Drug	Morphine Sulfate	d00308

Items per page: 5 1 – 5 of 9

- This looks great however your users will want to be able to sort the table by clicking the header row.

ALLERGIES COMPONENT

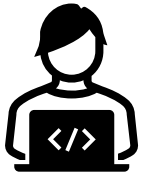
- In allergies.component.html, modify the <table> element so it includes the matSort directive and set the output for matSortChange to a new method called sortData as follows:

```
<table mat-table [dataSource]="dataSource" matSort (matSortChange)="sortData($event)">
```

- Your editor will underline sortData and mark it as an error as we have not yet created this method.
- On each of your three <th> elements, add the mat-sort-header directive.

```
<th mat-header-cell *matHeaderCellDef mat-sort-header>Substance Type</th>
```

- Save your source and open allergies.component.ts to complete the sort logic.



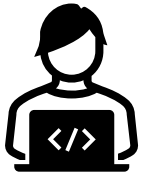
ALLERGIES COMPONENT

- Add imports for Sort and UtilityService in allergies.component.ts.

```
import { Sort } from '@angular/material/sort';
import { UtilityService } from '@clinicaloffice/clinical-office-mpage';
```

- Modify your constructor to inject the UtilityService as a new object called util.

```
constructor(public allergyService: AllergyService, public util: UtilityService) { }
```



ALLERGIES COMPONENT

- Create a new method called sortData with the following code:

```
// Sort the table
sortData(sort: Sort) {
    this.dataSource.data = this.dataSource.data.sort((a: IAllergy, b: IAllergy) => {
        const isAsc = sort.direction === 'asc';
        switch (sort.active) {
            case 'substance_type’:
                return this.util.compare(a.substanceType, b.substanceType, isAsc);
            case 'substance’:
                return this.util.compare(a.substance.toUpperCase, b.substance.toUpperCase, isAsc);
            case 'substance_identifier’:
                return this.util.compare(a.substanceIdentifier, b.substanceIdentifier, isAsc);
            default:
                return 0;
        }
    });
}
```

ARROW FUNCTIONS

- Our sort method is using something we haven't seen before called TypeScript arrow functions.

```
this.dataSource.data = this.dataSource.data.sort((a: IAllergy, b: IAllergy) => {
    Do some code
});
```

- Arrow functions let you pass code expressions to methods that support them such as array sorting and filtering.
- Values can be passed to the arrow function and used inside the function.
- In this example, the sort method passes a copy of two IAllergy records to the arrow function to compare the values against each other.
- The format of arrow functions is:

```
(param1: type, param2: type, etc.) => { code }
```

ALLERGIES COMPONENT

- Our allergy component is almost complete. The only thing remains is to add the ability to let users filter allergies by allergy type (e.g., Drug, Environment, Food).
- Add the following code in allergies.component.html directly below the <h2>Allergies</h2> element.

```
<!-- Filters -->
<div>
    <mat-checkbox [(ngModel)]="drugFilter" (change)="refreshDataSource()">Drug</mat-checkbox>
    <mat-checkbox [(ngModel)]="envFilter" (change)="refreshDataSource()">
        Environment
    </mat-checkbox>
    <mat-checkbox [(ngModel)]="foodFilter" (change)="refreshDataSource()">Food</mat-checkbox>
    <mat-checkbox [(ngModel)]="otherFilter" (change)="refreshDataSource()">
        Other
    </mat-checkbox>
</div>
```

DATA BINDING

- The code you just entered introduces a new concept called two-way data binding.

```
<mat-checkbox [(ngModel)]="drugFilter" (change)="refreshDataSource()">Drug</mat-checkbox>
```

- When you see [(ngModel)] you are telling Angular that the form control you are using needs to read and write from the variable assigned. In the sample above the model is a variable called drugFilter.
- ngModel can also be bound in a single direction. Using [ ] will allow you to write to a variable and ( ) performs a read from the variable.
- For ngModel to work, we must add the FormsModule and ReactiveFormsModule imports from @angular/forms to app.module.ts.
- We also must define the variables that our model is bound to.
- Finally, the (change) method above calls our refreshDataSource() method which we will create in a moment.

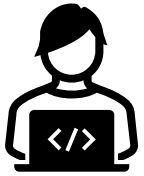
ALLERGIES COMPONENT

- Add the following variables to your AllergiesComponent class.

```
// Filter variables
drugFilter = true;
envFilter = false;
foodFilter = false;
otherFilter = false;
```

- Change your ready method to the following as we want to have only one place that refreshes the data source.

```
get ready(): boolean {
    if (!this.isReady) {
        if (this.allergyService.get()[0].allergyId > 0) {
            this.isReady = true;
            this.refreshDataSource();
        }
    }
    return this.isReady;
}
```



ALLERGIES COMPONENT

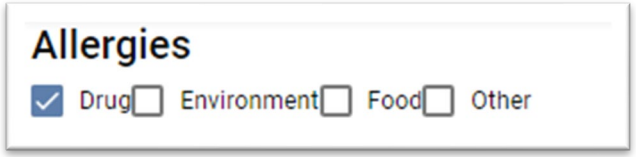
- Create a new method called refreshDataSource(). This method will be called from our ready method as well as when the check box values are changed.

```
// Refresh the data source
refreshDataSource() {
  if (this.isReady) {
    this.dataSource.data = this.allergyService.get().filter((e: IAllergy) => {
      return (
        (e.substanceType === 'Drug' && this.drugFilter) ||
        (e.substanceType === 'Environment' && this.envFilter) ||
        (e.substanceType === 'Food' && this.foodFilter) ||
        (e.substanceType === 'Other' && this.otherFilter)
      )
    });
  }
}
```

- This method uses an arrow function in the filter statement to filter out check-box values we don't want to see.

ALLERGIES COMPONENT

- Everything now functions in our component except the check-box filters don't look right.



- Add the following code to your styles.scss file to include padding to the right of mat-checkbox labels.

```
.mat-checkbox label {
    padding-right: .5rem;
}
```

ALLERGIES COMPONENT

- Your MPage should now look like the screen shot below.

MRN: 700001344 **DOB:** 03/20/1955 **Gender:** Male **PCP:**

Allergies

☑ Drug ☐ Environment ☐ Food ☐ Other

Substance Type	Substance	Substance ID
Drug	Banana	3000504919
Drug	morphine	d00308
Drug	Morphine Sulfate	d00308
Drug	other contrast	
Drug	penicillin	d00116

Items per page: 5 1 – 5 of 6

TWO DOWN, FOUR TO GO!

- We now have two components down out of the 6 needed.
  1. Demographics **(Complete)**
  2. Allergies **(Complete)**
  3. Problems
  4. Diagnosis
  5. Visit History
  6. Appointments
- The remaining components will roughly follow the same pattern as the Allergies component. Problems and Diagnosis will be simple tables with a paginator and sorting.
- Visit History will have filters for date ranges and encounter class.
- Appointments will also have date filters and be the first script to use custom CCL.

PROBLEMS & DIAGNOSIS

- Having to create the virtually the same component we just built would be cruel and un-necessary.
- Instead, we are going to copy the problems and diagnosis components from another project.
- From a new command line, navigate to a folder of your choice and type in:

  git clone https://github.com/clinicaloffice/mpage-training.git

- From the File menu in Visual Studio Code, select “Add Folder to Workspace” and select the new “mpage-training” folder.

PROBLEMS & DIAGNOSIS

- Expand the src/app folder in mpage-training.
- Select the diagnosis and problems folder and press CTRL+C to copy the folder reference into memory.
- Navigate to the src/app folder in your MPage and left click the word app. Press CTRL+V to paste a copy of the two source folders into your application.
- Although the source code is now in your project, Angular won't do anything with the code until you import into app.module.ts.

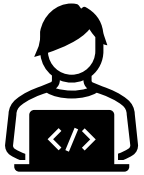
PROBLEMS & DIAGNOSIS

- Open your app.module.ts file and add the import statements into your code as follows.

```
import { ProblemsComponent } from './problems/problems.component';
import { DiagnosisComponent } from './diagnosis/diagnosis.component';

@NgModule({
  declarations: [
    AppComponent,
    DemographicsComponent,
    AllergiesComponent,
    ProblemsComponent,
    DiagnosisComponent
  ],
```

- Finally, add <app-problems> and <app-diagnosis> to your app.component.html file.

```
<app-demographics></app-demographics>
<app-allergies></app-allergies>
<app-problems></app-problems>
<app-diagnosis></app-diagnosis>

<router-outlet></router-outlet>

<mpage-log-component></mpage-log-component>
```



FOUR DOWN, TWO TO GO!

- We now have four components down out of the 6 needed.
  1. Demographics **(Complete)**
  2. Allergies **(Complete)**
  3. Problems **(Complete)**
  4. Diagnosis **(Complete)**
  5. Visit History
  6. Appointments

VISIT HISTORY

- From the command line in your project folder, create a new component called visit-history.

  ng g c visit-history

- Add the component to your app.component.html file.

```
<app-demographics></app-demographics>
<app-allergies></app-allergies>
<app-problems></app-problems>
<app-diagnosis></app-diagnosis>
<app-visit-history></app-visit-history>

<router-outlet></router-outlet>

<mpage-log-component></mpage-log-component>
```

VISIT HISTORY

- Start by setting up some of the basic imports we will need.

```
import { Component, OnInit } from '@angular/core';
import { EncounterService, UtilityService } from '@clinicaloffice/clinical-office-mpage';
```

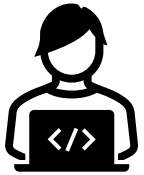
- Add the two services to your constructor.

```
constructor(public encounterService: EncounterService, public util: UtilityService) { }
```

- Our encounter component will offer date prompts to choose the date range of the encounters. Add two variables called fromDate and toDate.

```
export class VisitHistoryComponent implements OnInit {
  fromDate!: Date;
  toDate!: Date;
```

- The exclamation mark (!) at the end of the variable name indicates that while the value has not been assigned yet, we promise it will be before it is used.



MOMENT.JS

- Working with dates in JavaScript can be challenging. Moment.js was developed to ease the task of date manipulation and calculations.
- We are going to use Moment.js in our Visit History panel to calculate our default encounter date range of 90 days in the past.
- Moment.js has been marked as obsolete however while developing MPages for Internet Explorer it is still a valuable tool for date calculations. If you are using Edge, Moment.js is still valid working code however there are more efficient alternatives available.
- The ie-mpage template has Moment.js included in the package.json file so the only thing you need to do is import it in your component source.

```
import * as moment from 'moment';
```

- The Moment.js website (https://momentjs.com) offers extensive documentation on all the methods and formats available.

VISIT HISTORY

- In the ngOnInit function, add the following two lines to default the fromDate variable to 90 days ago and toDate to the end of today.

```
ngOnInit(): void {
  this.fromDate = moment(this.fromDate).subtract(90, 'days').startOf('day').toDate();
  this.toDate = moment().endOf('day').toDate();
}
```

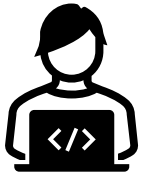
- Let's do a quick test to see if our dates are defaulting correctly.
- In visit-history.component.html, replace the code with the following:

```
<p>From Date: {{ fromDate | date: 'short' }}</p>
<p>To Date: {{ toDate | date: 'short' }}</p>
```

- When you view your MPage, you should see something like the following.

From Date: 8/31/21, 12:00 AM

To Date: 11/29/21, 11:59 PM



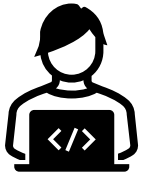
VISIT HISTORY

- Add a call to a new method called refreshEncounters in your ngOnInit method.

```
  ngOnInit(): void {
    this.fromDate = moment(this.fromDate).subtract(90, 'days').startOf('day').toDate();
    this.toDate = moment().endOf('day').toDate();

    this.refreshEncounters();
  }
```

- Add the following refreshEncounters method to your code.

```
  // Refresh the encounter data from CCL
  refreshEncounters() {
    if (moment(this.toDate).isBefore(this.fromDate)) {
      alert('Your To Date value must be greater than the From Date. Please check your date
values.');
    } else {
```



VISIT HISTORY

```
      this.encounterService.loadList('1CO_MPAGE_ENC_LIST:GROUP1',
      {
        dateField: 'REG_DT_TM',
        fromDate: this.fromDate,
        toDate: this.toDate
      },
      false,
      'VISIT_HISTORY',
      [{codeSet: 0, type: '', typeCd: 0}],
      {encounter: {
        aliases: true,
        prsnlReltn: true
      }}
      );
    }
  }
```

VISIT HISTORY

- The if statement uses Moment.js to check to see if our toDate is greater than or fromDate variable. On initialization this should never happen, but later when our users are changing dates with a drop-down calendar, they may pick the wrong dates. This code alerts the user to this problem and prevents the loading of incorrect data.

```
    if (moment(this.toDate).isBefore(this.fromDate)) {
      alert('Your To Date value must be greater than the From Date. Please check your date
values.');
```

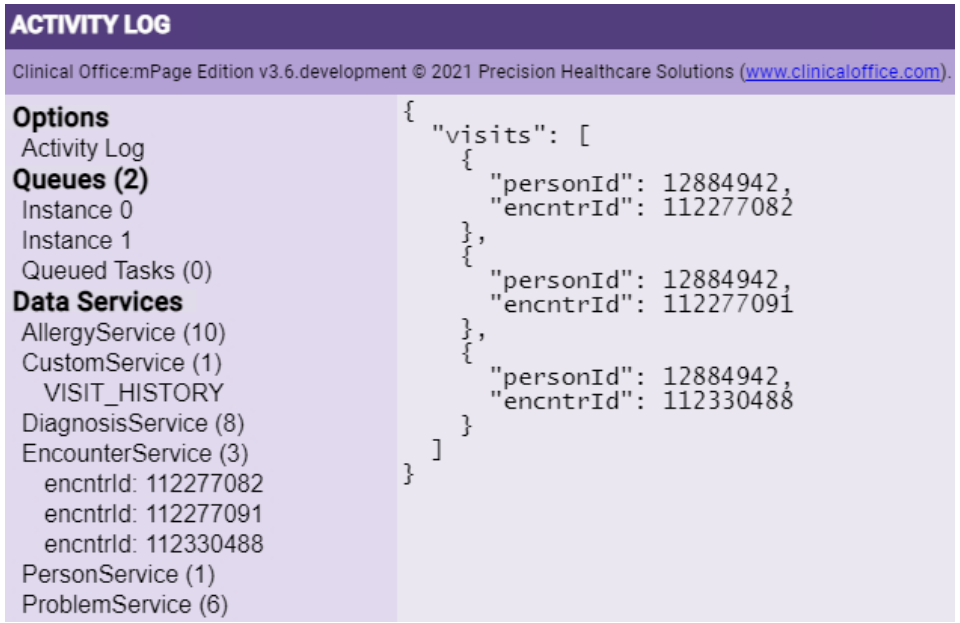
VISIT HISTORY

- The EncounterService offers a method called loadList that allows running a CCL script that is responsible for loading multiple encounters. This functionality makes use of the CustomService as well as the EncounterService.
- When 1CO_MPAGE_ENC_LIST has finished running, the encounters returned in it are then passed to the standard payload. In this case, the encounter payload has been run.
- We could also add other payload values such as “person”.
- If clearPatientSource had been set to true, the current encounter would have been cleared and all encounters for all patients in the past 90 days would be loaded.
- You can create your own loadList CCL scripts.

```
this.encounterService.loadList(
     '1CO_MPAGE_ENC_LIST:GROUP1',
      {
        dateField: 'REG_DT_TM',
        fromDate: this.fromDate,
        toDate: this.toDate
      },
      false,
      'VISIT_HISTORY',
      [{codeSet: 0, type: '', typeCd: 0}],
      {encounter: {
        aliases: true,
        prsnlReltn: true
      }}
      );
```

VISIT HISTORY

- Refresh your MPage and view a patient that has more than one visit.
- Expand your activity log and you should see your EncounterService with multiple encounters and your CustomService with an entry called “VISIT_HISTORY”.
- The name “VISIT_HISTORY” is the name you assigned in your loadList call.
- EncounterService contains all encounters currently loaded in memory. This includes the encounter currently open in PowerChart. If you had opened a visit from two years ago, EncounterService would contain not only the last 90 days of visits but also the currently opened visit.
- This will lead to a problem where you will later see visits that are loaded but not part of the date range you selected.
- The CustomService “VISIT_HISTORY” however only contains a list of encounters retrieved during the last execute of our loadList.



VISIT HISTORY

- Currently we have a situation where we have a list of encounters we want to display stored in a CustomService entry and our encounter details are stored in EncounterService.
- Ideally, we would like to display only the visits chosen in the date range selected on the MPage.
- In addition to displaying only our selected data, we later want to be able to further filter the data by encounter type class and additionally sort the data.
- To do this, we are going to create a new array containing only the data we wish to display on our MPage.

VISIT HISTORY

- Modify the import statement in visit-history-component.ts to include CustomService.

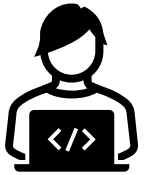
```
import { Component, OnInit } from '@angular/core';
import { EncounterService, CustomService, UtilityService } from '@clinicaloffice/clinical-
office-mpage';
import * as moment from 'moment';
```

- Include CustomService in the constructor.

```
constructor( public encounterService: EncounterService,
             public customService: CustomService,
             public util: UtilityService) { }
```

- Add the following variables to your class.

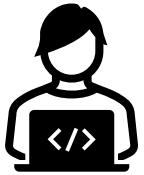
```
  isReady = false;
  encounterData: any[] = new Array();
```



VISIT HISTORY

- We are going to create a ready() method that loops through all the entries in the “VISIT_HISTORY” custom service object and with an arrow function, retrieve all qualified encounters and push them to a custom array called encounterData.
- Create a ready get method with the following code.

```
// Determine if data has been loaded
get ready(): boolean {
  if (!this.isReady && this.customService.isLoaded('VISIT_HISTORY')) {
    // Loop through VISIT_HISTORY to populate the encounterData array
    this.customService.get('VISIT_HISTORY').visits.forEach((e: any) => {
      const enc = this.encounterService.get(e.encntrId);  // Retrieve the encounter
```



VISIT HISTORY

```
        this.encounterData.push(
          {
            regDtTm: enc.regDtTm,
            dischDtTm: enc.dischDtTm,
            fin: this.encounterService.getAlias('FIN NBR', e.encntrId).aliasFormatted,
            encntrTypeClass: enc.encntrTypeClass,
            encntrType: enc.encntrType,
            location: enc.locFacility + ' ' + enc.locNurseUnit,
            medService: enc.medService,
            attending: this.encounterService.getPrsnlReltn('ATTENDDOC',
e.encntrId).nameFullFormatted
          }
        );
      });

      this.isReady = true;
    }
    return this.isReady;
  }
```

VISIT HISTORY

- There is quite a bit happening in the ready() method.
- First, we do a simple check to see if the isReady variable is true and if “VISIT_HISTORY” data has been loaded in the CustomService. If it has, we continue to the logic block that copies data from the EncounterService to our encounterData array.
- If the “VISIT_HISTORY” data has not been loaded yet, we simply return false.
- Later, in our HTML we are going to add code that uses the date derived from the ready() function to display a table of our data.

VISIT HISTORY

- After checking if “VISIT_HISTORY” has been loaded, we do some interesting things.
- The following line uses the forEach method to loop through every entry retrieved in our “VISIT_HISTORY” array.

```
this.customService.get('VISIT_HISTORY').visits.forEach((e: any) => {
```

- The code block inside the forEach assigns the variable e an iteration of the object returned. In this example, the object contains the variables encntrId and personId. We are then able to reference the encntrId as e.encntrId.

VISIT HISTORY

- Next, we assign a temporary variable an encounter retrieved from the Encounter Service. Each iteration of the forEach will return an encounter from EncounterService.

```
const enc = this.encounterService.get(e.encntrId);  // Retrieve the encounter
```

- We can then reference all the data for a specific encounter with the enc variable (e.g., enc.regDtTm, enc.encntrType, etc.)
- Certain functionality such as the getAlias and getPrsnlReltn methods must be called directly from the EncounterService by passing an encntrId. This is easily handled as our e variable contains the encntrId we are looking for (e.encntrId)

VISIT HISTORY

- The array push method adds a new entry to an existing array. In our case, we define a new object with the brackets { }. We can then later use this new object to display our data.

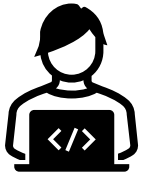
```
this.encounterData.push(
          {
            regDtTm: enc.regDtTm,
            dischDtTm: enc.dischDtTm,
            fin: this.encounterService.getAlias('FIN NBR', e.encntrId).aliasFormatted,
            encntrTypeClass: enc.encntrTypeClass,
            encntrType: enc.encntrType,
            location: enc.locFacility + ' ' + enc.locNurseUnit,
            medService: enc.medService,
            attending: this.encounterService.getPrsnlReltn('ATTENDDOC',
e.encntrId).nameFullFormatted
          }
```

VISIT HISTORY

- Before we move on to adding our encounterData array to a Material Table, lets test our data.
- Replace the contents of visit-history.html with the following code and refresh your MPage.

```
<div class="component-container">
    <h2>Visit History</h2>

    <!-- Display the problems table if loaded -->
    <ng-container *ngIf="ready">
        {{ encounterData | json }}
    </ng-container>
</div>
```

- The json pipe shown above will display an object in JSON format. It can be very useful when testing.



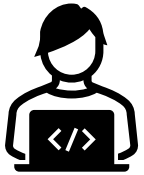
VISIT HISTORY

- We are now going to apply our data to a Material Table in a similar fashion to our other components.
- Start with the import statements.

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { EncounterService, CustomService, UtilityService } from '@clinicaloffice/clinical-
office-mpage';
import { MatTableDataSource } from '@angular/material/table';
import { MatPaginator } from '@angular/material/paginator'
import { Sort } from '@angular/material/sort';
import * as moment from 'moment';
```

- Add the required variables for the table and paginator.

```
  displayedColumns: string[] = ['reg_date', 'disch_date', 'fin', 'encntr_type', 'location',
'med_service', 'attending'];
  dataSource: MatTableDataSource<any> = new MatTableDataSource();
  @ViewChild('paginator', {static: true}) paginator!: MatPaginator;
```



VISIT HISTORY

- Add the paginator code to the ngOnInit method.

```
this.dataSource.paginator = this.paginator;
```

- Assign the encounterData array to the dataSource object in the ready() method immediately before the this.isReady = true; statement.

```
      });

      this.dataSource.data = this.encounterData;
      this.isReady = true;
    }
    return this.isReady;
```

VISIT HISTORY

- Add a sortData method to your code.

```
// Sort table
sortData(sort: Sort) {
  this.dataSource.data = this.dataSource.data.sort((a: any, b: any) => {
    const isAsc = sort.direction === 'asc';
    switch (sort.active) {
      case 'reg_date': return this.util.compare(a.regDtTm, b.regDtTm, isAsc);
      case 'disch_date': return this.util.compare(a.dischDtTm, b.dischDtTm, isAsc);
      case 'fin': return this.util.compare(a.fin, b.fin, isAsc);
      case 'encntr_type': return this.util.compare(a.encntrType, b.encntrType, isAsc);
      case 'location': return this.util.compare(a.location, b.location, isAsc);
      case 'med_service': return this.util.compare(a.medService, b.medService, isAsc);
      case 'attending': return this.util.compare(a.attending, b.attending, isAsc);
      default:
        return 0;
    }
  });
}
```

VISIT HISTORY

- Replace visit-history.component.html with the following code to view the table.

```
<div class="component-container">
    <h2>Visit History</h2>

    <!-- Display the visits table if loaded -->
    <ng-container *ngIf="ready">
        <table mat-table [dataSource]="dataSource" matSort (matSortChange)="sortData($event)">
            <!-- Define columns -->
            <ng-container matColumnDef="reg_date">
                <th mat-header-cell *matHeaderCellDef mat-sort-header>Registration Date</th>
                <td mat-cell *matCellDef="let element">{{ element.regDtTm | date: 'MM/dd/yyyy'
}}</td>
            </ng-container>
```

VISIT HISTORY

```
            <ng-container matColumnDef="disch_date">
                <th mat-header-cell *matHeaderCellDef mat-sort-header>Discharge Date</th>
                <td mat-cell *matCellDef="let element">{{ element.dischDtTm | date : 'MM/dd/yyyy'
}}</td>
            </ng-container>

            <ng-container matColumnDef="fin">
                <th mat-header-cell *matHeaderCellDef mat-sort-header>FIN #</th>
                <td mat-cell *matCellDef="let element">{{ element.fin }}</td>
            </ng-container>

            <ng-container matColumnDef="encntr_type">
                <th mat-header-cell *matHeaderCellDef mat-sort-header>Type</th>
                <td mat-cell *matCellDef="let element">{{ element.encntrType }}</td>
            </ng-container>

            <ng-container matColumnDef="location">
                <th mat-header-cell *matHeaderCellDef mat-sort-header>Location</th>
                <td mat-cell *matCellDef="let element">{{ element.location }}</td>
            </ng-container>
```

VISIT HISTORY

```
            <ng-container matColumnDef="med_service">
                <th mat-header-cell *matHeaderCellDef mat-sort-header>Service</th>
                <td mat-cell *matCellDef="let element">{{ element.medService }}</td>
            </ng-container>

            <ng-container matColumnDef="attending">
                <th mat-header-cell *matHeaderCellDef mat-sort-header>Attending</th>
                <td mat-cell *matCellDef="let element">{{ element.attending }}</td>
            </ng-container>

            <!-- Define rows -->
            <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
            <tr mat-row *matRowDef="let row; columns: displayedColumns"></tr>
        </table>
    </ng-container>

    <!-- Paginator -->
    <mat-paginator #paginator [pageSizeOptions]="[5, 10, 25]" showFirstLastButtons></mat-
paginator>
</div>
```

VISIT HISTORY

- If you view your MPage you will now see the table rendered correctly.
- Our next step is to create date prompts to control the date range of our table. Add the following to visit-history.component.html after the <h2>Visit History</h2> element.

```
<div class="form-filters">
    <!-- From Date -->
    <mat-form-field>
    <input matInput [matDatepicker]="iVisitFromDate"
            placeholder="From Date" [value]="fromDate"
            (dateChange)="setDate('FROM', $event)">
        <mat-datepicker-toggle matSuffix [for]="iVisitFromDate"></mat-datepicker-toggle>
        <mat-datepicker #iVisitFromDate [calendarHeaderComponent]="datePickerHeader">
        </mat-datepicker>
    </mat-form-field>
```

VISIT HISTORY

```
    <!-- To Date -->
    <mat-form-field>
        <input matInput [matDatepicker]="iVisitToDate"
                placeholder="To Date" [value]="toDate" (dateChange)="setDate('TO', $event)">
            <mat-datepicker-toggle matSuffix [for]="iVisitToDate"></mat-datepicker-toggle>
            <mat-datepicker #iVisitToDate [calendarHeaderComponent]="datePickerHeader">
            </mat-datepicker>
    </mat-form-field>

    <!-- Refresh Button -->
    <button mat-flat-button color="primary" (click)="refreshEncounters()">Refresh</button>
</div>
```

- This code will render a from and to date calendar control along with a Refresh button on your page.

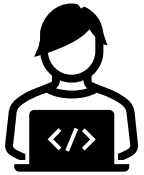
VISIT HISTORY

- In the visit-history.component.ts file modify your imports to include the DatepickerHeaderComponent and the MatDatepickerInputEvent directive.

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { EncounterService, CustomService, UtilityService, DatepickerHeaderComponent } from
'@clinicaloffice/clinical-office-mpage';
import { MatTableDataSource } from '@angular/material/table';
import { MatPaginator } from '@angular/material/paginator'
import { Sort } from '@angular/material/sort';
import { MatDatepickerInputEvent } from '@angular/material/datepicker';
import * as moment from 'moment';
```

- Add a new variable for our datePickerHeader component.

```
datePickerHeader = DatepickerHeaderComponent;
```



VISIT HISTORY

- Since our data can be re-loaded with the Refresh button, it is vital that we initialize our variables on every call to refreshEncounters. Add the following code immediately before the loadList call.

```
this.isReady = false;                          // Set the isLoaded flag to false
this.customService.clear('VISIT_HISTORY');     // Clear any custom visit history entries
this.encounterData.length = 0;                 // Truncate the encounterData array
this.dataSource.data = new Array();            // Clear the data source with a new array

this.encounterService.loadList('1CO_MPAGE_ENC_LIST:GROUP1’,
    {
```

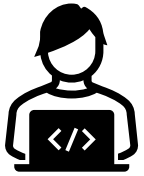
VISIT HISTORY

- The final step is to create the setDate function which takes the dates entered in our prompts and assigns the value back to either fromDate or toDate.

```
  // Set the date from a prompt
  setDate(dateField: string, event: MatDatepickerInputEvent<any>) {
    if (dateField === 'FROM') {
      this.fromDate = event.value;
    } else {
      this.toDate = event.value;
    }
  }
```

- You can add the following to styles.scss to improve visibility of the prompts.

```
.mat-form-field {
    padding-right: .5rem;
}

.form-filters {
    background-color: #d6d6d6;
}
```



VISIT HISTORY

- The final step in our Visit History is to setup a prompt for the encounter type class and set a default to only load inpatient visits.
- There are many ways to accomplish this, however for this example we are going to make use of the typeList object in Clinical Office.
- We could have alternatively loaded all the encounters and filtered the results after the load but that wouldn’t be as educational as what we are about to do.

VISIT HISTORY

- Our encounter type drop-down box is going to use code values from code set 69.
- As mentioned earlier in the course, reference data that doesn't change should be loaded in app.component.ts.
- Add CodeValueService to the import and constructor of app.component.ts.
- Next, immediately after the call to mPage.setMaxInstances code, initiate the code value load.

```
    // Load code set 69
    this.codeValueService.load(69);
```

VISIT HISTORY

- Import the CodeValueService and TypeList to your Clinical Office imports.
- Your final import block for visit-history.component.ts should be as follows:

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { EncounterService, CustomService, UtilityService,
DatepickerHeaderComponent, CodeValueService, TypeList } from
'@clinicaloffice/clinical-office-mpage';
import { MatTableDataSource } from '@angular/material/table';
import { MatPaginator } from '@angular/material/paginator'
import { Sort } from '@angular/material/sort';
import { MatDatepickerInputEvent } from '@angular/material/datepicker';
import * as moment from 'moment';    {
```

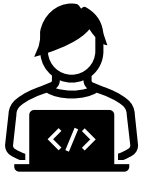
VISIT HISTORY

- Add a new array to your class variables called encTypeClass and a new Boolean value called encDefaultSet.

```
encTypeClass: any[] = new Array();
encDefaultSet = false;
```

- Ensure that your constructor includes the CodeValueService.

```
  constructor(public encounterService: EncounterService,
              public customService: CustomService,
              public codeValueService: CodeValueService,
              public util: UtilityService) { }
```



VISIT HISTORY

- We are going to add a new prompt in front of the Refresh button in visit-history.component.html.
- This prompt will use the mat-select component to create a multiple-selection drop-down that assigns checked values to our new encTypeClass array from code set 69.

```
<!-- Multi-Select Encounter Type Class -->
<mat-form-field>
    <mat-select [(value)]="encTypeClass" multiple placeholder="Encounter Type Class">
        <mat-option *ngFor="let cv of codeValueService.getCodeSet(69)" [value]="cv.codeValue">
                    {{ cv.display }}
        </mat-option>
    </mat-select>
</mat-form-field>
```

VISIT HISTORY

- Go ahead and run your code. You should see a new prompt that when viewed will show you all the values from code set 69.
- There is one problem however and that is the default value for the control is not set. We can take care of this with the following code added to our ready method which will set the default to Inpatient.

```
// Update the default encounter type checkbox
if (!this.encDefaultSet && this.codeValueService.getCodeSet(69).length > 0) {

  this.encTypeClass.push(this.codeValueService.getCodeSet(69).filter((cv: any) => {
    return cv.displayKey === 'INPATIENT';
  })[0].codeValue);

  this.encDefaultSet = true;
}
```

VISIT HISTORY

- There are several things going on in this block so let's break it down.
- The value binding on our encTypeClass mat-select control expects an array of one or more values to be assigned in the format of [num, num, …]
- The getCodeSet method in CodeValueService returns an array of all code value objects for code set 69.
- We use an Angular filter to only return objects where the displayKey is equal to 'INPATIENT'. There should only be one qualified value but it comes back as an array of objects.
- We only want the code value of the first returned object which is what [0].codeValue represents and that single number is what is stored in the setValue([ ]) array.

VISIT HISTORY

- It's now time to use our filter in our data selection.
- Return to the refreshEncounters() method and scroll down to the typeList section.

```
[{ codeSet: 0, type: '', typeCd: 0 }],
```

- Most of the Clinical Office services offer some form of type list filtering. The documentation for each service identifies what is available.
- Type list filtering lets you specify that only specific types of data can be retrieved during the data load. For example, the following block of code would allow only INPATIENT and OUTPATIENT encounter type classes to be loaded.

```
[
    { codeSet: 69, type: 'INPATIENT', typeCd: 0 },
    { codeSet: 69, type: 'OUTPATIENT', typeCd: 0 }
],
```

VISIT HISTORY

- The typeList array contains an object with three parts which are:
- codeSet – Number representing the code set to filter.
- type – Display key or CDF Meaning text value to filter by
- typeCd – Code value to filter by
- You can either filter by type or typeCd. If you specify a non-empty string for type (e.g., ‘INPATIENT’), the type field is used in data selection.
- Entering a typeCd value will result in filtering by the code value (e.g. {codeSet: 69, ‘’, 329}.

VISIT HISTORY

- Our code example is a bit more advanced than just passing a static typeList. We want our typeList to be dynamic based on the values chosen in our prompt.
- To do this, we are going to replace the existing typeList with a call to a new method called typeList() that we are going to create.
- Replace the line:

```
[{ codeSet: 0, type: '', typeCd: 0 }],
```

- with:

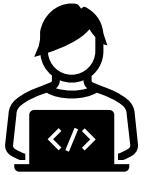
```
this.typeList(),
```

VISIT HISTORY

- Add the following new method to visit-history.component.ts.

```
// Returns typelist values or default
typeList(): TypeList[] {
  const tl: TypeList[] = new Array();
  if (this.encTypeClass.length > 0) {
    this.encTypeClass.forEach((e: any) => {
      tl.push({codeSet: 69, type: '', typeCd: e});
    });
    return tl;
  }
  return [{codeSet: 69, type: 'INPATIENT', typeCd: 0}];
}
```

- This code will return a TypeList array using the values selected in the encTypeClass array. Since the values in the array are in the format of [num, num, num], we build out the proper TypeList structure in our push statement.
- If nothing is selected in the encClassTypeList object, we simply force a class of 'INPATIENT' which prevents a bad data pull and additionally loads our default type of inpatient.



VISIT HISTORY

- Our visit history component is now complete however there is one bug in the code that I'll leave you with to solve on your own.
- If you de-select all the encounter class prompt values, you will still load inpatient visits even though the prompt has none selected.
- You could correct this in several ways including forcing the prompt to always have at least one value selected.

ONE MORE TO GO!

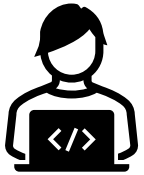
- We now have five components down out of the 6 needed.
    1. Demographics **(Complete)**
    2. Allergies **(Complete)**
    3. Problems **(Complete)**
    4. Diagnosis **(Complete)**
    5. Visit History **(Complete)**
    6. Appointments

APPOINTMENTS COMPONENT

- Our MPage is nearly complete, and the only outstanding component is our Appointment History component.
- Up until now, we have developed our entire MPage without having to write a single line of CCL code.
- Many of the MPages we have developed for clients using Clinical Office have not required the use of any additional CCL. The standard CCL library has covered our needs.
- Our Appointment Component will however make use of a custom CCL script that we are going to write.
- Data from this custom CCL script will be loaded with the CustomService data service.

APPOINTMENTS COMPONENT

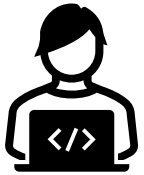
- We are going to start development of the Appointment History component by writing our CCL data collection script. This script will be called "1trn_train_appt_hist.prg" and will be compiled as GROUP1.
- I will develop this script in DiscernVisualDeveloper and you can choose to use the script I develop or follow along by writing your own copy replacing _train_ in the file/object name with your first initial and 4 characters of your last name (e.g. John Simpson -> jsimp -> 1trn_jsimp_appt_hist.prg).
- Your Clinical Office CCL library contains a script called 1co_mpage_template.prg. Open it now in Discern Visual Developer and save it using the filename convention mentioned above. Make sure to accept changing the object name.
- This script can be used as a starting point for all custom scripts that will be used by the CustomService data service.



APPOINTMENTS COMPONENT

- The empty template mainly contains comments designed to help you understand how to use it.
- The Special Instructions section of the comments gives an overview of the sample payload you would include in your Angular application to make your script run.
- Please take note of the clearPatientSource option. If set to true, any patient information sent via the MPage is cleared and not available for your script. The intention is that you may want to use your script to populate the patient source (e.g. Hospital Census).
- The comment block immediately following your drop and create program statements discuss how to access any parameters you may send in your script.
- In our component we will be passing both a fromDate and toDate value to allow date filtering of appointments. In the CCL script, these would be accessible as:

```
payload->customscript->script[nScript]->parameters.fromdate
payload->customscript->script[nScript]->parameters.todate
```

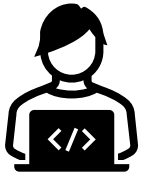


APPOINTMENTS COMPONENT

- The first real section of code in the script is the following if statement:

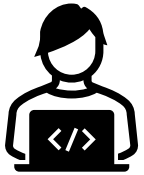
```
if (validate(payload->customscript->clearpatientsource, 0) = 0)
    if (size(patient_source->patients, 5) = 0)
        go to end_program
    endif
endif
```

- This code simply checks to see if you have cleared the patient source and if not ensures that there are patient records in the patient source to run the script with.
- If everything passes in this check the script continues otherwise it aborts the script.



APPOINTMENTS COMPONENT

- Next up you will see a section marked “BEGIN YOUR CUSTOM CODE HERE”. This is where you will write all of your CCL code.
- Typically, you will write a CCL script that populates a record structure. To assist your efforts, a placeholder record structure called rCustom has been added to the code.
- We recommend that you stick with the name rCustom however you are free to change the name to whatever you like; just make sure you update the record structure name at the bottom of your script in the add_custom_output subroutine call.



APPOINTMENTS COMPONENT

- Replace the rCustom definition with the following code:

```
free record rCustom
record rCustom (
    1 appointments[*]
        2 beg_dt_tm              = dq8
        2 appt_type              = vc
        2 resource               = vc
        2 location               = vc
        2 sch_state              = vc
)
```

- Declare a custom variable called nNum to be used by the CCL EXPAND function.

```
declare nNum = i4           ; Used by expand
```

APPOINTMENTS COMPONENT

- Add the following code to your script.

```
; Collect appointments
select into "nl:"
from sch_appt          sa,
     sch_event         se,
     sch_appt          sa2
plan sa
     where expand(nNum, 1, size(patient_source->patients, 5),
                   sa.person_id, patient_source->patients[nNum].person_id)
     and sa.beg_dt_tm between
         cnvtdatetime(payload->customscript->script[nscript]->parameters.fromdate) and
         cnvtdatetime(payload->customscript->script[nscript]->parameters.todate)
     and sa.role_meaning = "PATIENT"
     and sa.state_meaning in ("CONFIRMED", "CHECKED IN", "CHECKED OUT")
     and sa.version_dt_tm > sysdate
     and sa.active_ind = 1
     and sa.end_effective_dt_tm > sysdate
join se
     where se.sch_event_id = sa.sch_event_id
     and se.version_dt_tm > sysdate
     and se.active_ind = 1
     and se.end_effective_dt_tm > sysdate
```

APPOINTMENTS COMPONENT

- Add the following code to your script.

```
join sa2
    where sa2.sch_event_id = se.sch_event_id
    and sa2.role_meaning = "RESOURCE"
    and sa2.state_meaning in ("CONFIRMED", "CHECKED IN", "CHECKED OUT")
    and sa2.version_dt_tm > sysdate
    and sa2.active_ind = 1
    and sa2.end_effective_dt_tm > sysdate
order sa.beg_dt_tm
head report
    nCount = 0
detail
    nCount = nCount + 1
    stat = alterlist(rCustom->appointments, nCount)

    rCustom->appointments[nCount].beg_dt_tm = sa.beg_dt_tm
    rCustom->appointments[nCount].appt_type = uar_get_code_display(se.appt_type_cd)
    rCustom->appointments[nCount].resource = uar_get_code_display(sa2.resource_cd)
    rCustom->appointments[nCount].location = uar_get_code_display(sa.appt_location_cd)
    rCustom->appointments[nCount].sch_state = uar_get_code_display(sa.sch_state_cd)
with expand=0
```

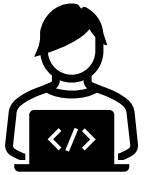
APPOINTMENTS COMPONENT

- The CCL script we created was nothing more than a simple three table join that collects appointments for a specific person_id and date range.
- Once collected the data is stored in our rCustom record structure which is then converted and added to our JSON stream in the add_custom_output subroutine at the bottom of the script.
- Using the available person_id in our patient_source structure we can use the CCL EXPAND function to search the sch_appt table.

```
where expand(nNum, 1, size(patient_source->patients, 5), sa.person_id,
                          patient_source->patients[nNum].person_id)
```

- We also make use of our custom fromDate and toDate parameters included in our JSON payload.

```
and sa.beg_dt_tm between
        cnvtdatetime(payload->customscript->script[nscript]->parameters.fromdate) and
        cnvtdatetime(payload->customscript->script[nscript]->parameters.todate)
```



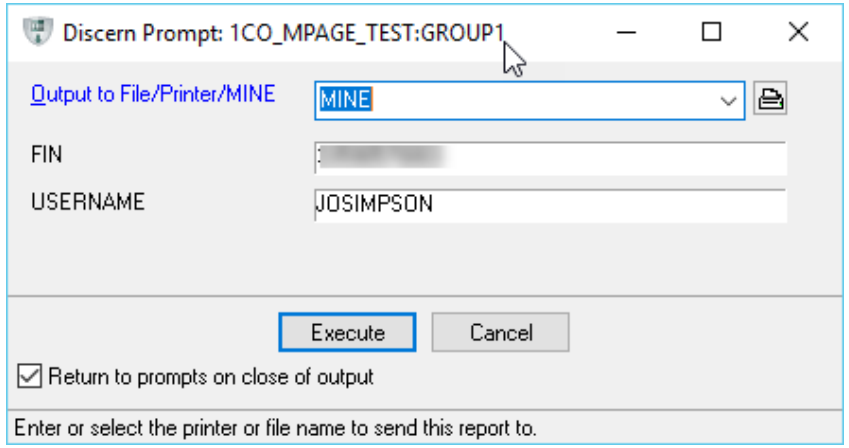
APPOINTMENTS COMPONENT

- We can perform a test inside CCL to ensure our script is working. This is done by modifying the script 1co_mpage_test.prg so our JSON payload is included.
- Near the bottom of the script is a line starting with SEQ REQUEST->BLOB_IN =.
- Modify this line to include your payload as a single string. The CONCAT function is used to make it easier to view. For example, use the following code (you may need to change the dates):

```
SET REQUEST->BLOB_IN = CONCAT(^{"payload": {^,
    ^"customScript" : {^,
        ^"script":[^,
            ^{"name":"1trn_train_appt_hist:group1", ^,
            ^"run":"pre", "id":"APPOINTMENT_HISTORY", ^,
            ^"parameters":{^,
                ^"fromDate": "2019-10-01T14:44:51.000+00:00",^,
                ^"toDate": "2020-10-11T14:44:51.000+00:00"^,
                ^}^,
            ^}^,
        ^],^,
        ^"clearPatientSource": false,^,
    ^}}}^)
```

APPOINTMENTS COMPONENT

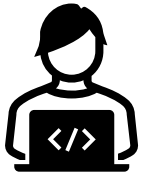
- Compile and execute 1co_mpage_test. You will be prompted for an output device, FIN # and username.



- This script will work on the front end or back-end. If you run on the back-end however you will see more information if there are errors in your script.
- Your final output will show as JSON on screen.

APPOINTMENTS COMPONENT

- Now that our CCL script has been completed, let's write the Angular code to complete the component.
- The majority of what we need is in our visit history component and if you feel like challenging yourself, please go ahead and copy the pieces you need to make the appointment history work.
- For everyone else, please copy the content of the appointment-history folder into your src/app folder and modify your app.module.ts file to add the component.
- The most significant difference in the code will be how you access your data.
- Our CCL script presents the data as a simple array that we can use directly in our data source which means the forEach loop used with custom data mapping isn't needed and we can assign our data directly to the data source.
- Compile and deploy your completed MPage. Be sure to peek at your activity log to see how data is returned through your custom CCL script.



PUTTING IT ALL TOGETHER

- Our components are now complete.
  1. Demographics **(Complete)**
  2. Allergies **(Complete)**
  3. Problems **(Complete)**
  4. Diagnosis **(Complete)**
  5. Visit History **(Complete)**
  6. Appointments **(Complete)**

- One item remains which is to use Angular routing to divide our single cluttered page into a user-friendly experience.

ROUTING

- In our MPage, we are going to create the following routes for the following pages of data.
- Allergies
- Problems & Diagnosis
- Visit History
- Appointment History
- When the MPage is loaded, the user will always see our current patient demographics banner, followed by a menu and the content of the currently chosen route.
- The default route will show the patient allergies.

ROUTING

- Modify your app.component.html file so it only contains the app-demographics component, router-outlet and mpage-log-component as shown below.

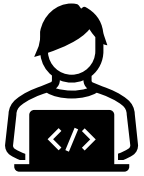
```
<app-demographics></app-demographics>
<router-outlet></router-outlet>

<mpage-log-component></mpage-log-component>
```

- If you were to compile now you would no longer see anything other than the demographics bar and log component.

ROUTING

- Angular routing lets you assign a component to a route path. To assign a component you need to import that component and add it to the routes object in your app-routing.module.ts file.
- Modify app-routing.module.ts so it appears as follows.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AllergiesComponent } from './allergies/allergies.component';

const routes: Routes = [
  {path: 'allergies', component: AllergiesComponent},
  {path: '**', redirectTo: 'allergies'}
];

@NgModule({
  imports: [RouterModule.forRoot(routes, {useHash: true})],
  exports: [RouterModule]
})
export class AppRoutingModule {
}
```



ROUTING

- The routes object is where we define what our routes will be.
- In the previous slide, we created a route called allergies and assigned it to the AllergyComponent.
- We also created a path called “**” which is a catch-all that redirects to our allergies route. This definition will ensure that the root (or default) route will display allergies as well as an invalid routes that may get run from our page.

ROUTING

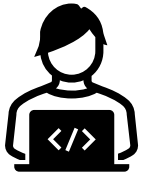
- As discussed earlier, our second route will display the Problems & Diagnosis components.
- We can only assign a single component to a route so the only way to show two components on the same route is to create a new component that displays both our desired components.
- From the command line, create a new route called problems-diagnosis with the following command:
  ng g c problems-diagnosis
- In the newly created problems-diagnosis.component.html file include the following lines of code.

```
<app-problems></app-problems>
<app-diagnosis></app-diagnosis>
```

ROUTING

- Return to your app-routing.module.ts file and import the ProblemsDiagnosisComponent and create a path for it in the routes object.

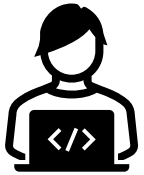
```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AllergiesComponent } from './allergies/allergies.component';
import { ProblemsDiagnosisComponent } from './problems-diagnosis/problems-diagnosis.component';

const routes: Routes = [
  { path: 'allergies', component: AllergiesComponent },
  { path: 'problemsdiagnosis', component: ProblemsDiagnosisComponent},
  { path: '**', redirectTo: 'allergies' }
];
```



ROUTING

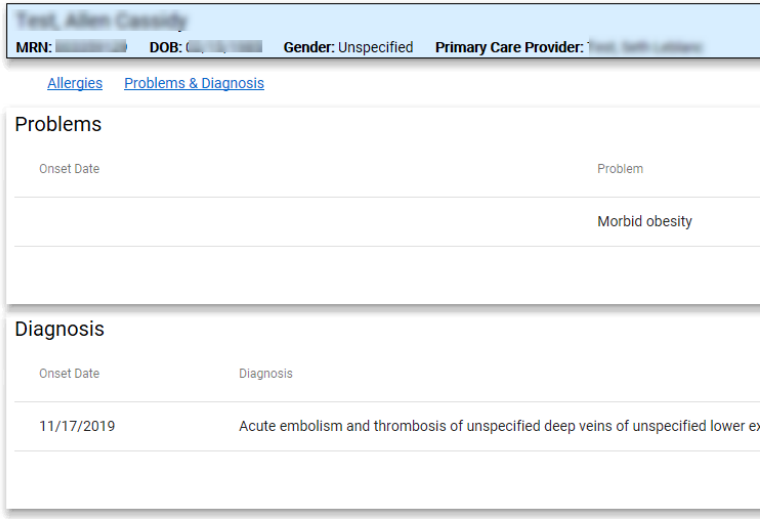
- At this point, our MPage will show use the allergies route however we have no way of pointing our page to the problemsdiagnosis route.
- Later on, we are going to use an Angular Material tabbed nav bar to display a menu of routes, but for now we are going to create a simple link to test our routes.
- In your app.component.html file, add the following code:

```
<app-demographics></app-demographics>

<ul class="sideways-list">
    <li><a [routerLink]="['/allergies']">Allergies</a></li>
    <li><a [routerLink]="['/problemsdiagnosis']">Problems & Diagnosis</a></li>
</ul>

<router-outlet></router-outlet>

<mpage-log-component></mpage-log-component>
```



ROUTING

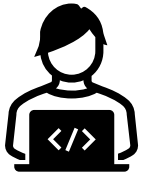
- If you compile, deploy and refresh your MPage you will now see links for both Allergies and the Problems & Diagnosis routes. Clicking these will refresh the content below the menu.
- In the example we are using simple HTML links however you can create a menu system as advanced as you desire with the tools you have available.
- On the next slide we will be replacing our menu with the Angular Material tabbed nav bar.



ROUTING

- Open app.component.ts and create an array called navLinks and assign it the following values.

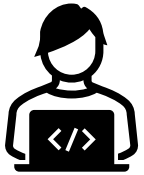
```
export class AppComponent implements OnInit {
  navLinks = [
    {label: 'Allergies', link: '/allergies', index: 0},
    {label: 'Problems & Diagnosis', link: '/problemsdiagnosis', index: 1},
    {label: 'Visit History', link: '/visithistory', index: 2},
    {label: 'Appointment History', link: '/appointmenthistory', index: 3}
  ];
```



ROUTING

- The tabbed nav bar control has a very specific format for populating its content. It uses an *ngFor command to iterate through each of the items stored in the navLinks array we just created.
- At its core however, it still presents the user with a list of valid [routerLink] options.
- Your final app.component.html code should appear as follows:

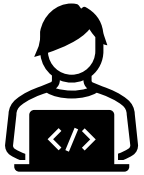
```
<app-demographics></app-demographics>

<nav mat-tab-nav-bar>
    <a mat-tab-link *ngFor="let link of navLinks"
        [routerLink]="link.link" routerLinkActive #rla="routerLinkActive"
        [active]="rla.isActive">
        {{link.label}}
    </a>
</nav>
<router-outlet></router-outlet>

<mpage-log-component></mpage-log-component>
```



ROUTING

- The last thing to do is add the routes for visit and appointment history to app-routing.module.ts.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AllergiesComponent } from './allergies/allergies.component';
import { ProblemsDiagnosisComponent } from './problems-diagnosis/problems-diagnosis.component';
import { VisitHistoryComponent } from './visit-history/visit-history.component';
import { AppointmentHistoryComponent } from './appointment-history/appointment-history.component';

const routes: Routes = [
  { path: 'allergies', component: AllergiesComponent },
  { path: 'problemsdiagnosis', component: ProblemsDiagnosisComponent},
  { path: 'visithistory', component: VisitHistoryComponent },
  { path: 'appointmenthistory', component: AppointmentHistoryComponent },
  { path: '**', redirectTo: 'allergies' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes, { useHash: true })],
  exports: [RouterModule]
})
export class AppRoutingModule {
}
```



READY TO DEPLOY

- Our MPage is complete and ready to deploy to PowerChart.

MRN: 700001344 **DOB:** 03/20/1955 **Gender:** Male **PCP:**

Allergies | Problems & Diagnosis | Visit History | Appointment History

Appointment History

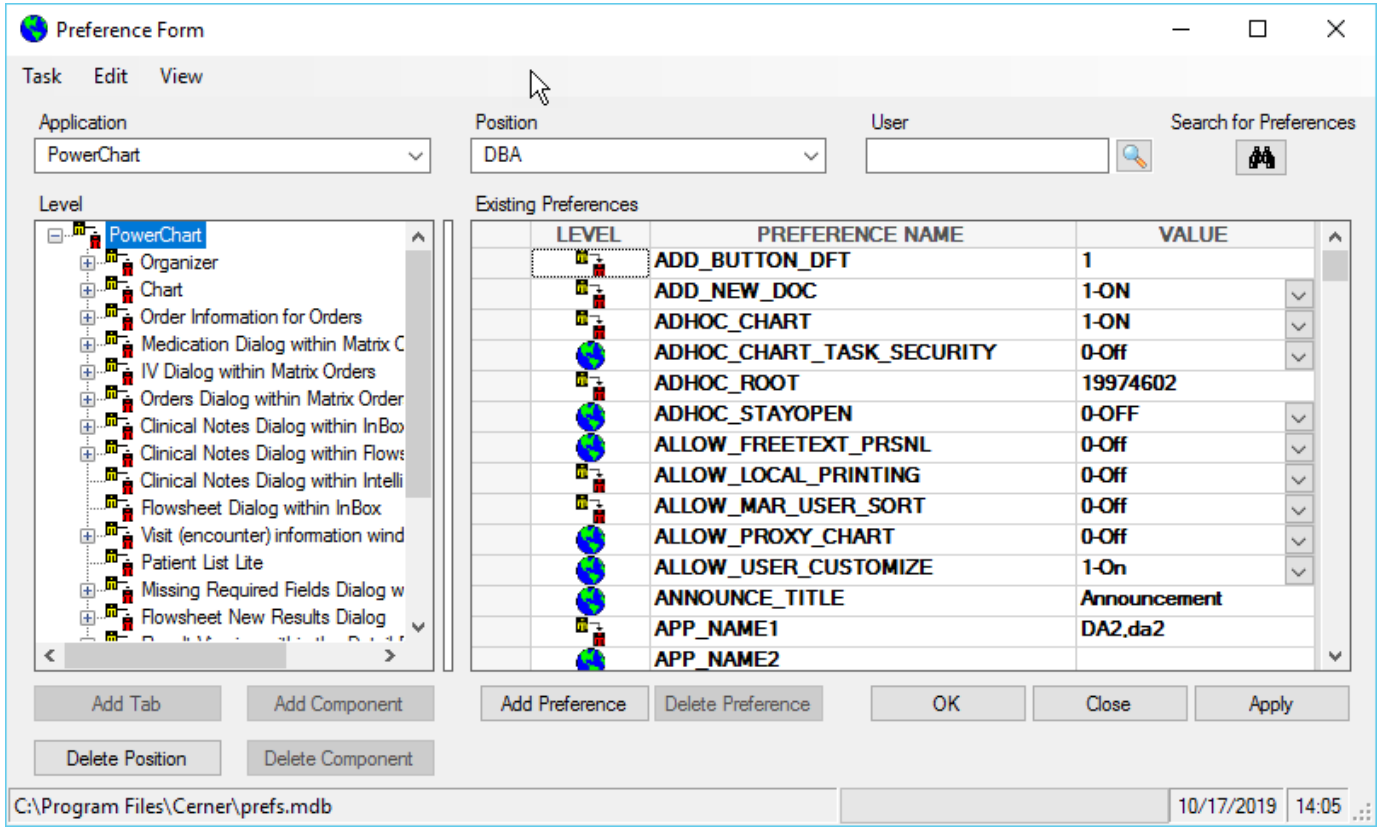
From Date 08-31-2021 To Date 11-29-2021 Refresh

Appointment Date	Type	Resource	Location	State
09/13/2021	Chloride Sweat			Checked In
10/19/2021	Post-COVID F/Up - RN			Checked In
11/03/2021	Cardiac Rehab New Consult			Confirmed
11/03/2021	Cardiac Rehab New Consult			Confirmed
11/17/2021	Post-COVID F/Up - 18 month			Confirmed

Items per page: 5 1 – 5 of 5

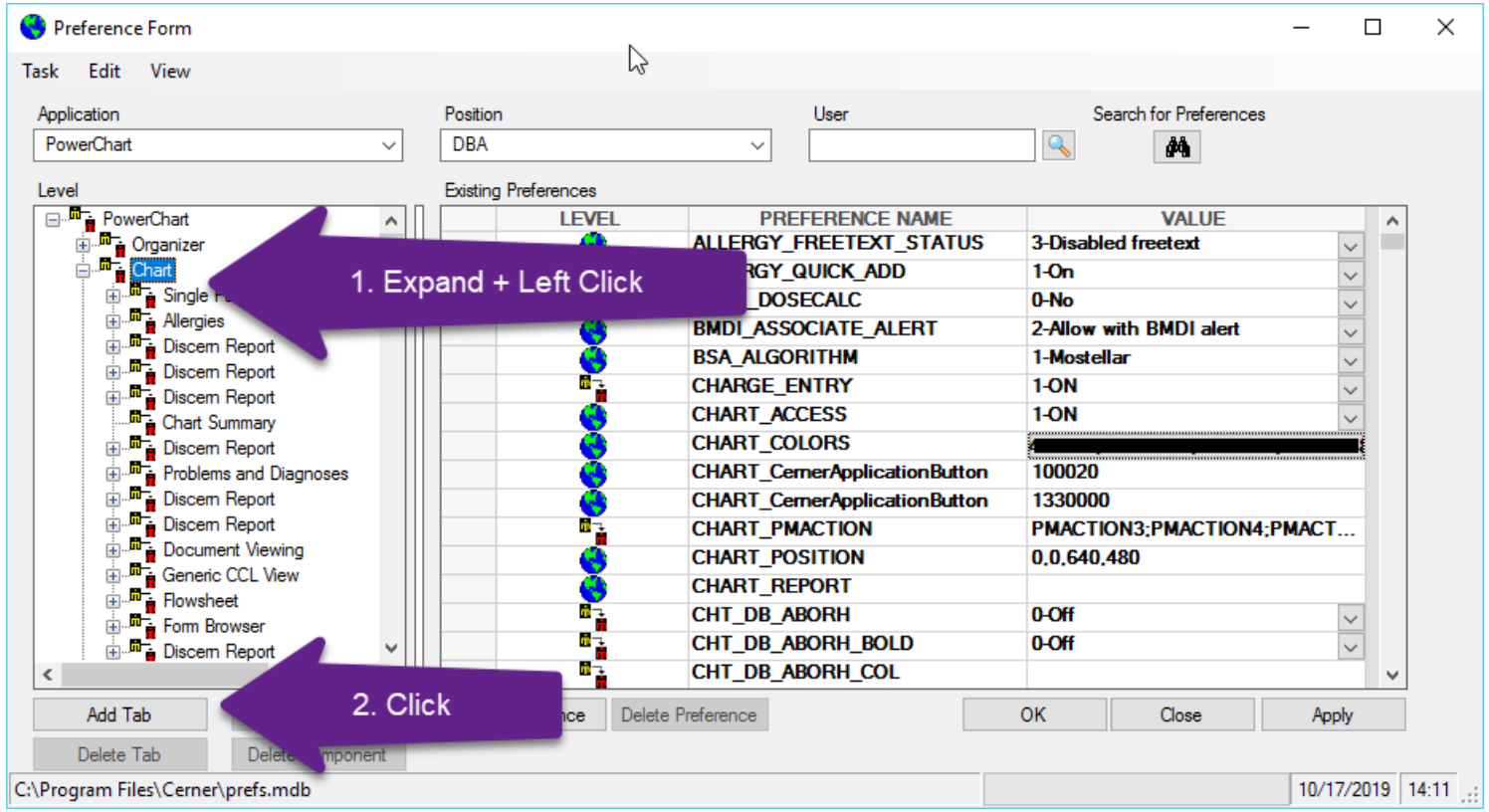
PREFMAINT SETTINGS

- The next step is to make our MPage available in Cerner.
- Compile your MPage and migrate the code to WebSphere as described on slides 33-35.
- Launch prefmaint.
- When loaded, select PowerChart in the Application drop down, DBA (or your position) from the Position menu and double-click on the Search for Preferences label in the Level window.



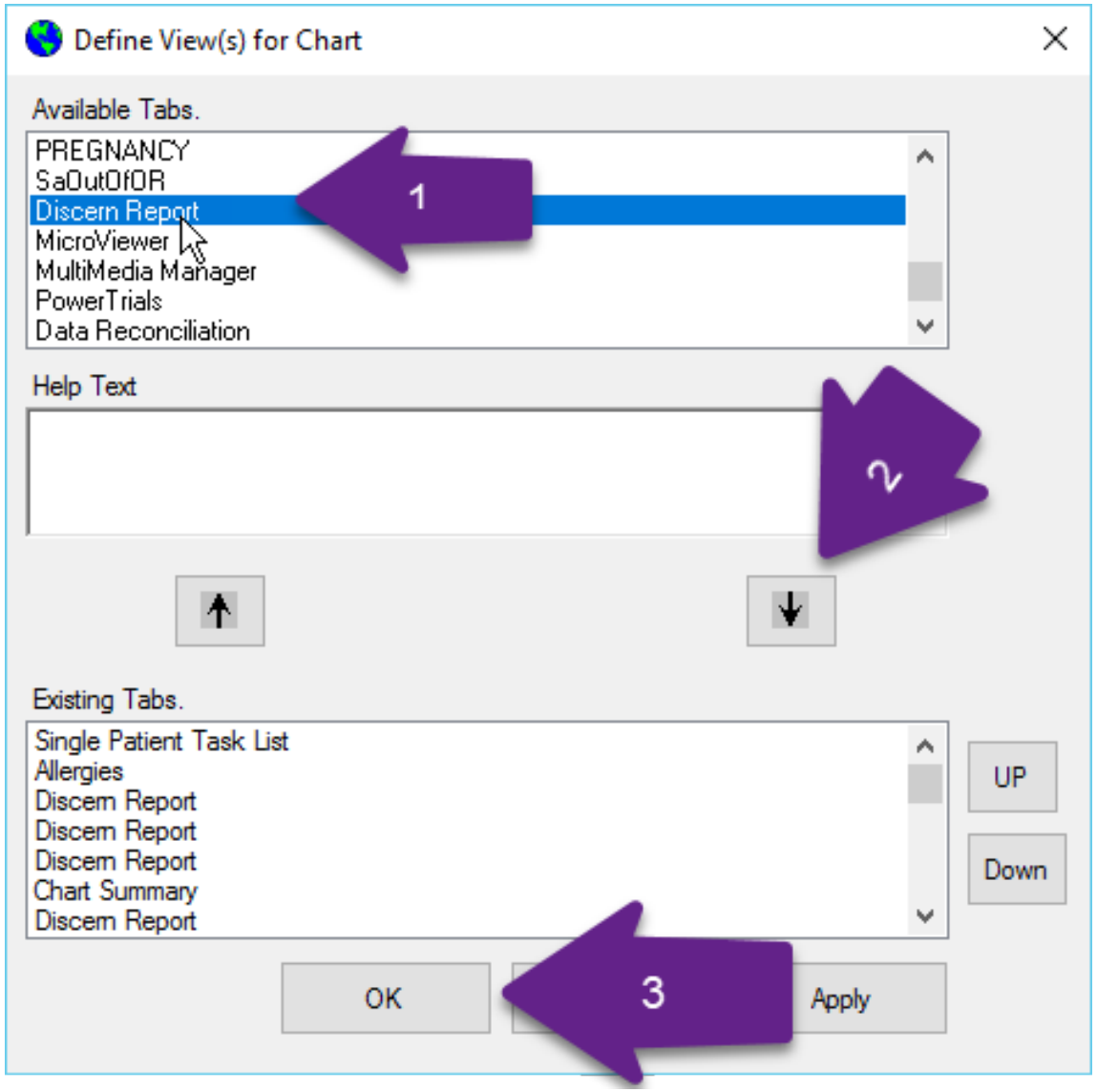
PREFMAINT SETTINGS

- At this point we need to open the level in which our MPage will be accessed. If your MPage is not patient-specific you should put it in the Organizer level. For our Patient History MPage we will be working at the patient or Chart level.
- Expand the Chart branch in the Level window and click on the word Chart.
- Click the Add Tab button.



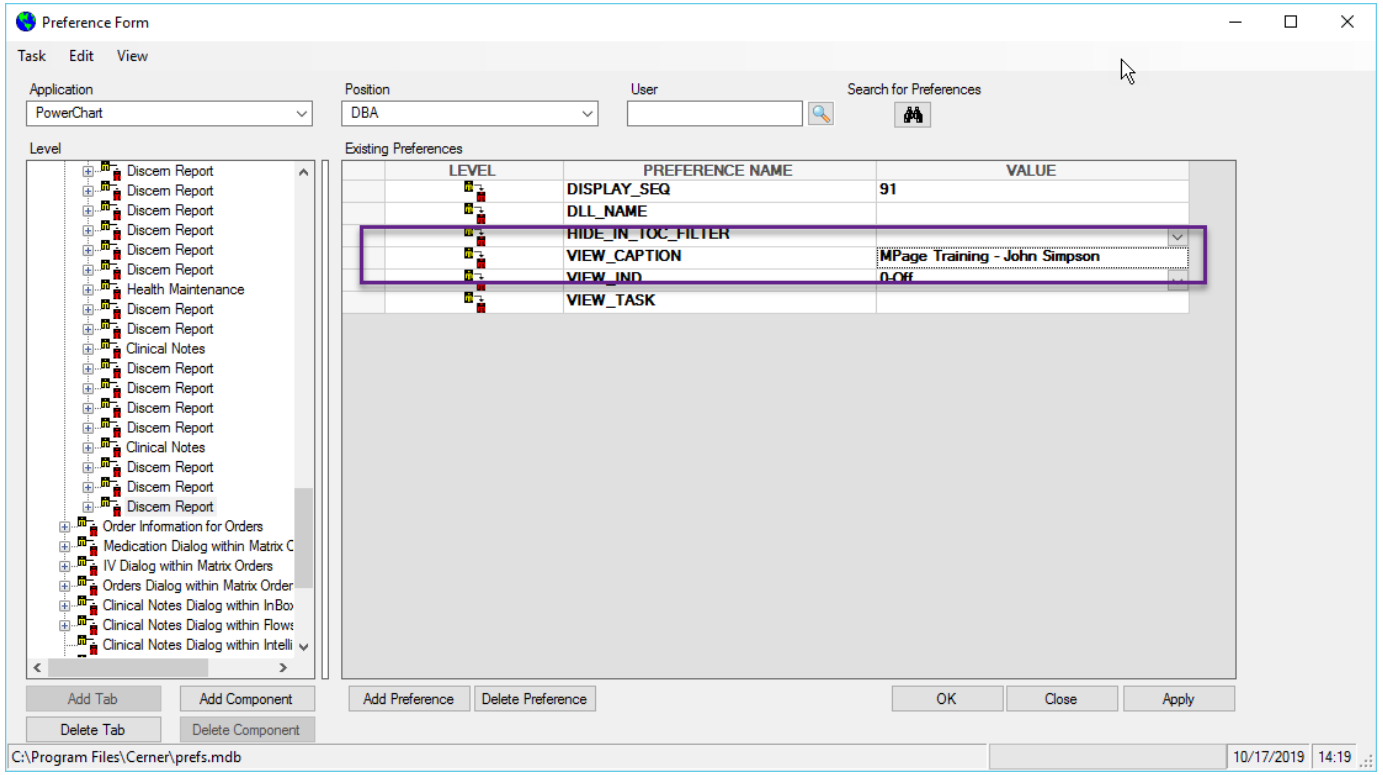
PREFMAINT SETTINGS

1. On the Available Tabs window, find and left click on "Discern Report".
2. Click the arrow that points downward to move the tab to the Existing Tabs window.
3. Click ok to add the Discern Report tab to the Chart view.



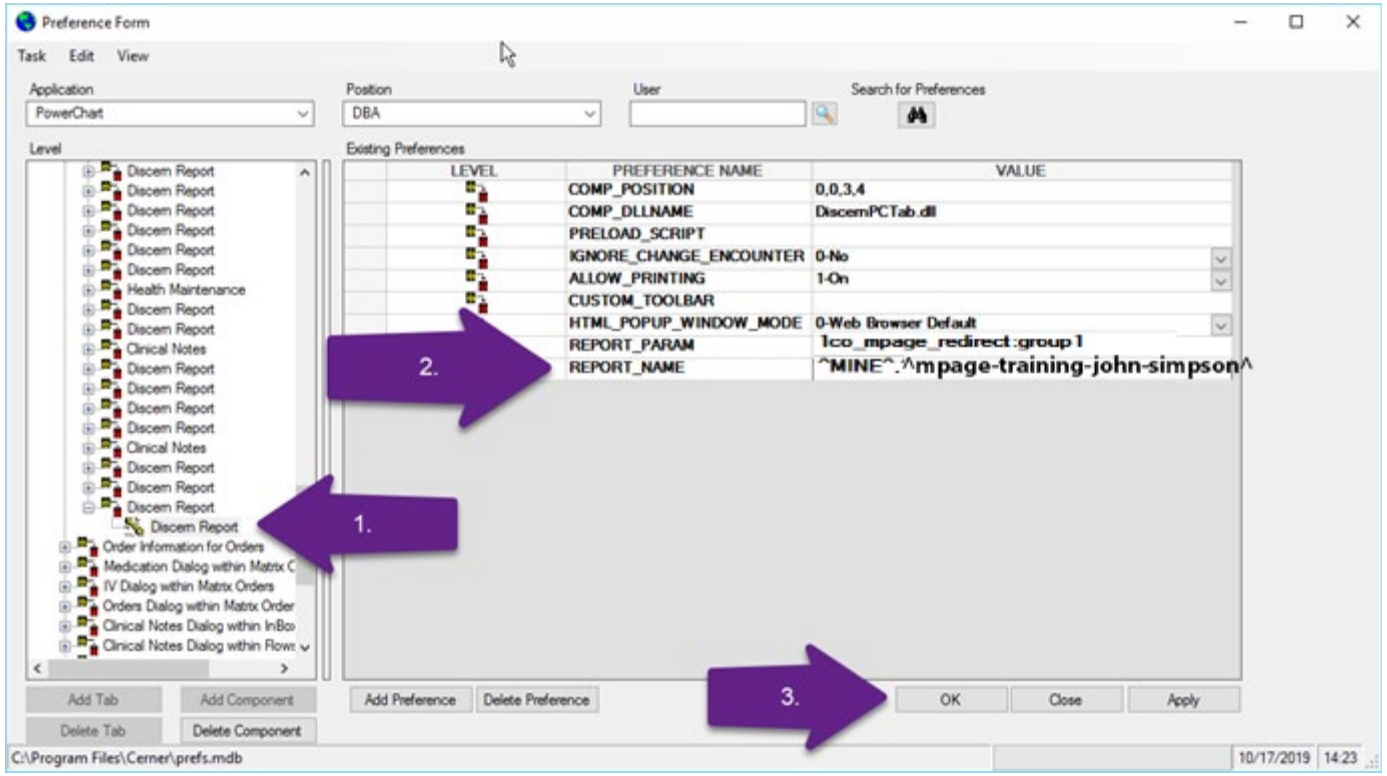
PREFMAINT SETTINGS

- Navigate to the last entry in the Chart view to find your new Discern Report entry. Left click on it and change the VIEW_CAPTION to the name you wish users to see in PowerChart.
- If you see a title called PVC_NAME with descriptions such as "The display name of the tab", simply choose "Display PVC_Name" from the View menu.



PREFMAINT SETTINGS

1. Expand your Discern Report in the Level window to see the next branch of settings.
2. In the REPORT_NAME preference, type in 1co_mpage_redirect_group1
3. The REPORT_PARAM preference should be set to ^MINE^,^mpage-training-john-simpson^
4. Click the OK button and save when prompted.



CUSTOM SERVICES

- Early in the course we discussed custom services. We have been using custom services throughout the entire class.
- Creating a custom service is simply a matter of typing in:

  ```
  ng generate service service-name
  ```

  Where service-name is the name of the service you wish to create.
- Creating a new service puts the file in the src/app folder but unlike our components the file does not get stored in its own sub-folder.
- The filename created is in the format of service-name.service.ts

CUSTOM SERVICES

- Creating a new service called test will offer the following code.

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class TestService {

  constructor() { }
}
```

- The key piece is the Injectable import which allows you the ability to inject your service into other components.
- Any variables or methods you create will be available to any component that uses the service.

FINAL THOUGHTS

- Your MPage is now complete although you still have much to learn about Angular, TypeScript and CSS.
- The materials covered in this course have given you enough information to write some powerful MPages with Angular and Clinical Office.
- As new features become available, upgraded versions of the Clinical Office:MPage framework will be made available to all clients. Documentation for these features can be found on the Clinical Office website.
- You are encouraged to expand your learning by visiting the following sites and experimenting with different code samples and tutorials.
  - https://www.clinicaloffice.com
  - https://www.angular.io
  - https://material.angular.io
  - https://blog.angular-university.io/