

API Interaction with Plex

In the last conference we introduced a company VHS By Mail, who as the name suggests, distribute movies on VHS tapes to their customers. They are long time Plex users, and with the help of CM WebClient they were able to expand their operation to the web.

Business has been booming over the past 18 months and they are looking to expand their operations to allow customers to request movies and TV Shows that are not currently in their catalog.

It wasn't practical for the company to maintain a database of every movie ever made, but they learned about the Open Movie Database (OMDb API) and decided that accessing their database via a RESTful API was the obvious way to go.

In this workshop we'll follow the steps that VHS By Mail took to implement their vision.

Exercise 1: [Optional] Register for API Key

Introduction

The API we will be using for these exercises is the Open Move Database (OMDb API) which provides details about Movies and TV Shows. The API is free to use but requires an API Key parameter to be passed with the requests. The service is limited to 1,000 requests per day for a particular key. A couple of keys are available to use between all participants, so it's possible this limit may be reached. This exercise walks you through the steps to receive your own API key to use.

Create API Key

Visit <http://www.omdbapi.com/apikey.aspx> to access the API Key page.

Select the "FREE! (1,000 daily limit)" option and enter your details. Set the 'Use' to something like 'Learning about RESTful APIs'.

API Key

3/10/19 Disposable/Temporary Emails Purged! All keys associated with these emails have been removed. Friendly reminder, we do not sell or share your information with anyone else. ✕

Generate API Key

Account Type Patreon
 FREE! (1,000 daily limit)

Email

Name

Use

A short description of the application or website that will use this API.

A verification link will be sent to your email.

Here is your key: [REDACTED]

Please append it to all of your API requests,

OMDb API: [\[REDACTED\]](#)

Click the following URL to activate your key: [\[REDACTED\]](#)

If you did not make this request, please disregard this email.

Click on the activation link in the email to activate the key. You should see the following message in the browser: Your key is now activated!

Now you have your API Key, we can use it in the following exercises.

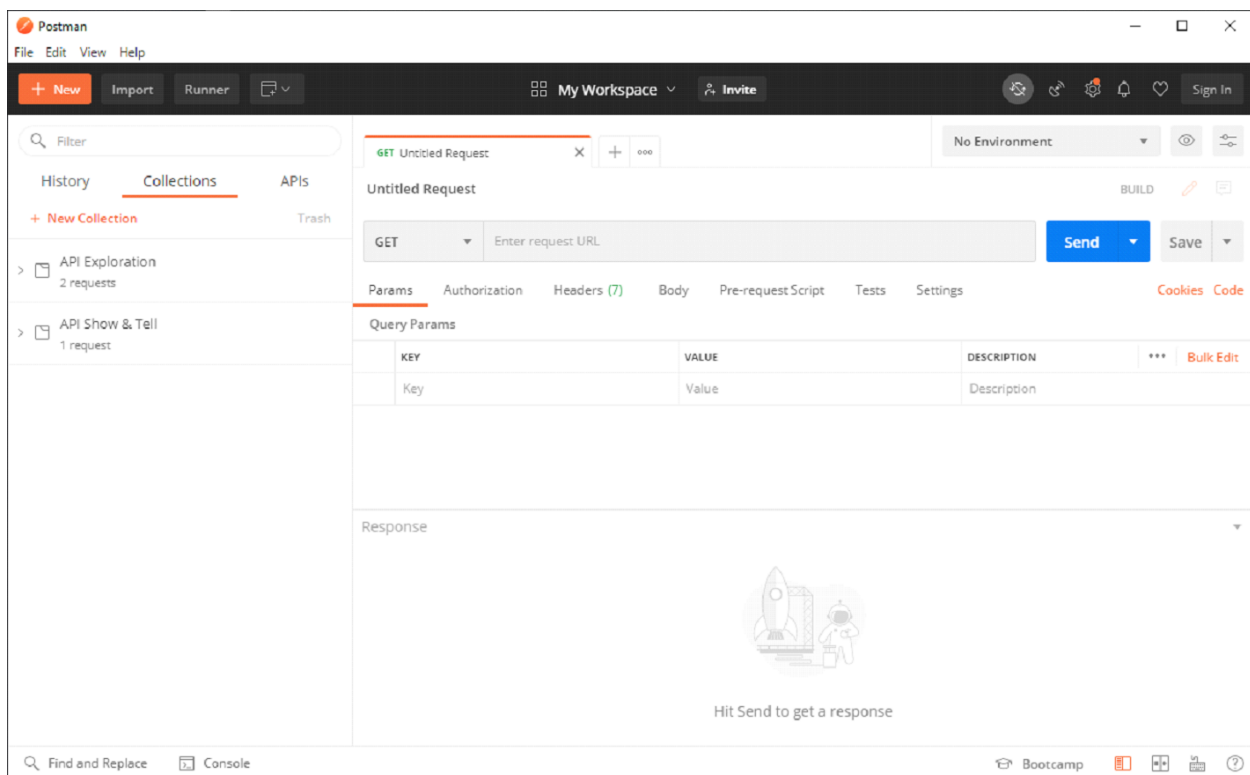
Exercise 2: Use Postman to explore the Open Movie Database API

Introduction

A useful tool for exploring APIs is Postman (<https://www.postman.com/product/rest-client/>). Postman is simple to use and lets you make API requests, view the output responses and has many other features. We'll explore some of the APIs we will use for the exercises.

Using Postman

Start Postman by clicking on the Postman icon on the



In the main area are the Request tabs where you can try out the Open Movie DB or other APIs. The documentation for the API (<http://www.omdbapi.com/>) shows that you need to make requests to the <http://www.omdbapi.com/> and pass in parameters to limit the search results. The 's' parameter is required to provide a title to search for:

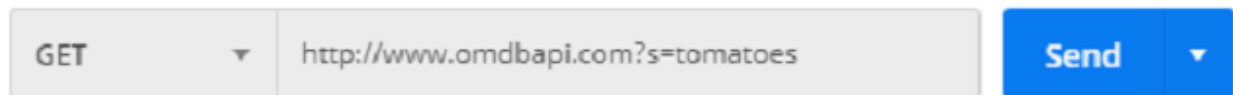
By Search

Parameter	Required	Valid options	Default Value	Description
s	Yes		<empty>	Movie title to search for.
type	No	movie, series, episode	<empty>	Type of result to return.
y	No		<empty>	Year of release.
r	No	json, xml	json	The data type to return.
page New!	No	1-100	1	Page number to return.
callback	No		<empty>	JSONP callback name.
v	No		1	API version (reserved for future use).

We'll start by performing a search for movies and shows with 'tomatoes' in the title, so we can format our request URL as

<http://www.omdbapi.com?s=tomatoes>

In Postman, make sure the request verb is set to 'GET' and enter the URL into the address area and press 'Send'.



Below the request area is the Response and we see that the request failed. Firstly, the response code is returned as '401 Unauthorized', and secondly the response returns a JSON object with two values:

Response: 'False', Error: 'No API key provided'.

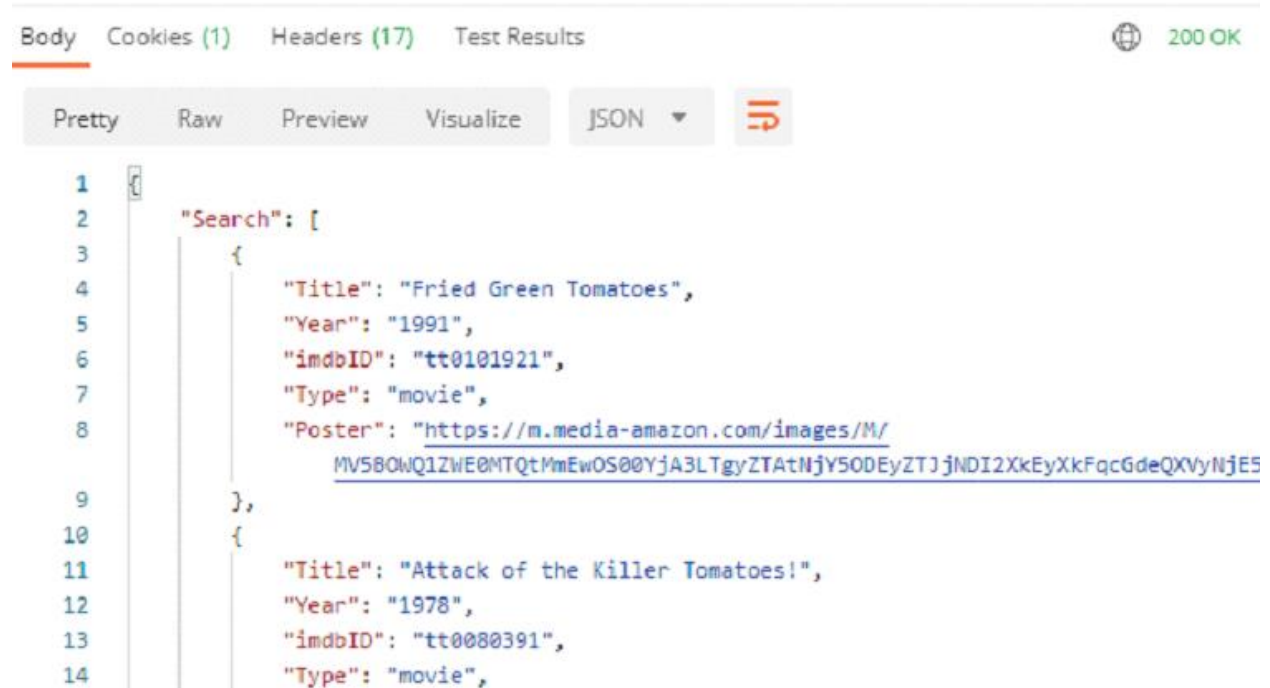


This is a reminder that requests are not guaranteed to be completed, so we should always validate the response code and response to ensure the request was successful. In this case, it's obvious that we didn't include the API Key with the request, so the service refused to process. Let's try again with an API Key parameter.

If you completed the first exercise, use the API key that you registered, otherwise use the key '67ecd31b' for the 'apikey' parameter. Each additional parameter is prefixed with a '&'. Note that the order of parameters doesn't matter.

<http://www.omdbapi.com?s=tomatoes&apikey=67ecd31b>

This request returns a status of 200 which means it was successful. The response is a JSON array returning up to 10 results. The ODb API returns results in pages of size 10, and there are other parameters that can be used to retrieve more results.



The screenshot shows a REST client interface with the following elements:

- Navigation tabs: Body (selected), Cookies (1), Headers (17), Test Results.
- Status: 200 OK.
- Response format: JSON.
- Response content (lines 1-14):

```
1 {
2   "Search": [
3     {
4       "Title": "Fried Green Tomatoes",
5       "Year": "1991",
6       "imdbID": "tt0101921",
7       "Type": "movie",
8       "Poster": "https://m.media-amazon.com/images/M/
9         MV5BOWQ1ZWE0MTQtMmEwOS00YjA3LTgyZTA0NjY5ODUyZTJjNDI2XkEyXkFqcGdeQXVyNjE5
10       },
11     {
12       "Title": "Attack of the Killer Tomatoes!",
13       "Year": "1978",
14       "imdbID": "tt0080391",
15       "Type": "movie",
```

Add the `page=2` parameter to return the next 10 results.

<http://www.omdbapi.com?s=tomatoes&apikey=67ecd31b&page=2>

Set the `type=series` parameter to only return TV shows in the results.

<http://www.omdbapi.com?s=tomatoes&apikey=67ecd31b&type=series>

Each movie or show has an 'imdbID' to uniquely identify it. We can use the 'i' parameter to return information about a particular show. "Attack of the Killer Tomatoes!" has an imdbID of 'tt0080391', so we can request its details with the following request.

<http://www.omdbapi.com?i=tt0080391&apikey=67ecd31b>

```
1 {
2   "Title": "Attack of the Killer Tomatoes!",
3   "Year": "1978",
4   "Rated": "PG",
5   "Released": "25 Dec 1978",
6   "Runtime": "83 min",
7   "Genre": "Adventure, Comedy, Horror, Musical, Sci-Fi",
8   "Director": "John De Bello",
9   "Writer": "Costa Dillon, John De Bello, J. Stephen Peace",
10  "Actors": "David Miller, George Wilson, Sharon Taylor, J. Stephen Peace",
11  "Plot": "A group of scientists band together to save the world from mutated killer tomatoes.",
12  "Language": "English",
13  "Country": "USA",
14  "Awards": "1 nomination.",
```

Exercise 3: Test Existing API Search Functionality

Introduction

VHS by Mail is expanding its functionality to allow customers to search for movies and TV shows not currently in their catalog. They have an existing Plex and CM WebClient application, so they have started to add the capability to access APIs from their Plex system.

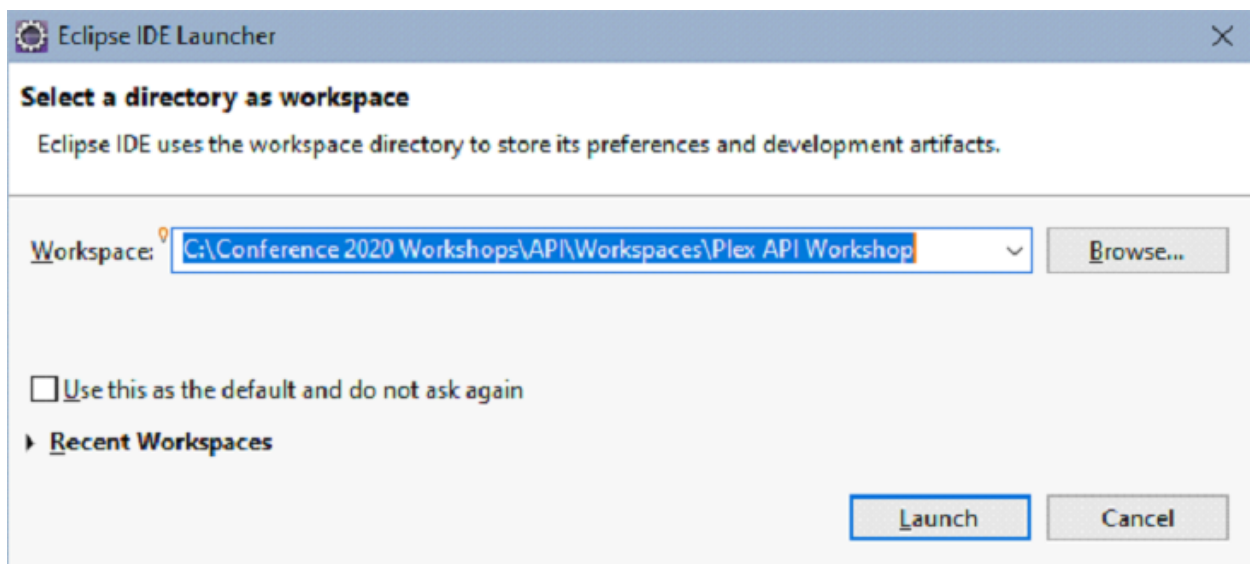
This exercise is to test the first version of their API Search WebClient application.

The Eclipse Environment

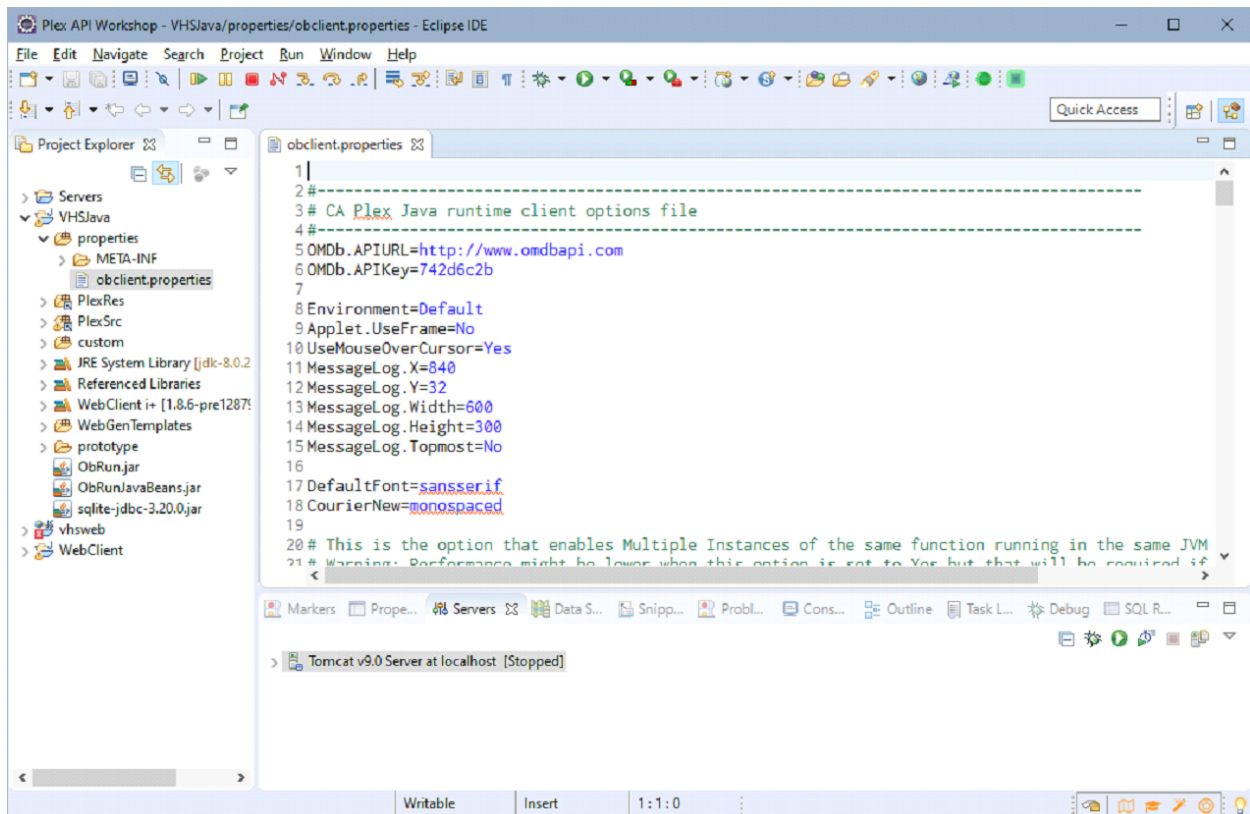
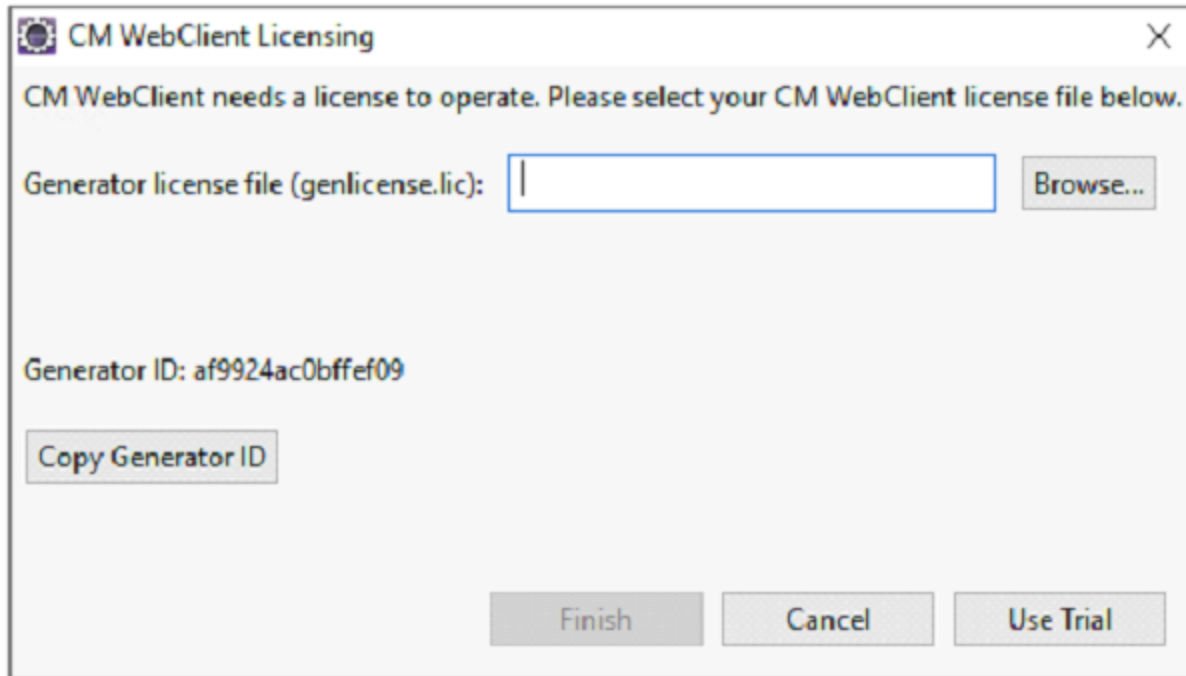
Open Eclipse by clicking on the desktop icon.



You will be asked to select a Workspace location. Make sure it is set to C:\Conference 2020 Workshops\API\Workspaces\Plex API Workshop



Note that the VM you are using has a trial version of WebClient. If you see this message in Eclipse, click the 'Use Trial' button.



If you're unfamiliar with Eclipse, the Workspace is a collection of Project folders. The Projects can be explored in the region on the left of the screen. The main Projects are:

- VHSJava – This project contains the generated Plex Java code, the obclient.properties file and custom templates for WebClient.

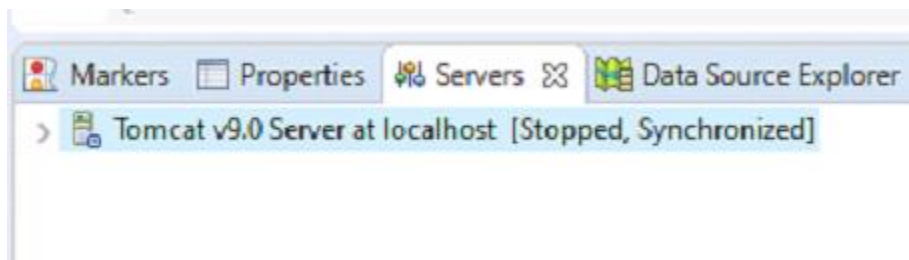
- vhsweb – This is the web project that contains the configuration files and resource files to be published on the web.
- WebClient – This contains system templates, the template generator, WebClient runtime and Plex runtime.

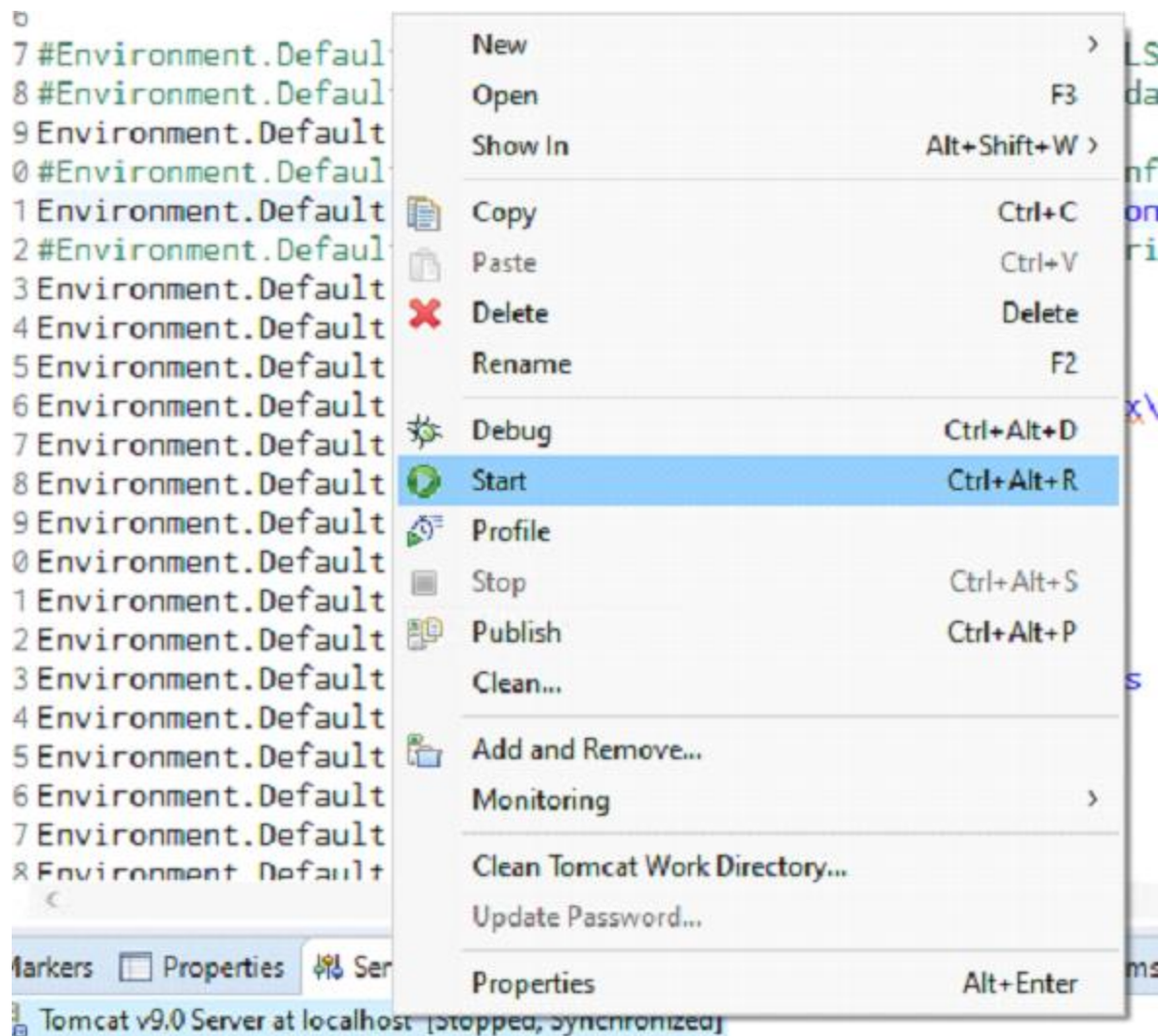
Double-clicking on a file within the project will open the file with its appropriate editor tab in the main region.

The bottom region contains several tools, each with their own tab. One of these is the Server tab which allows developers to publish the web application.

Publish and Test the Web Application

Open the Server Tab. Right-click on the Tomcat v9.0 Server





Wait for the web application to publish. When it's ready, the status of the server should display as [Started, Synchronized].

We can now run the application in a browser.

Open Chrome.



and enter <http://localhost:8080/vhsweb/wc> in the address bar.

The web application should launch. Enter some text in the Title field and press Search.

The screenshot shows a Plex interface with a search bar containing 'alien'. Below the search bar is a table of search results. The first result, 'Alien', is highlighted in green. To the right of the table is a detailed view for the selected item, 'Alien', showing its title, year (1979), and type (Movie), along with a movie poster image.

Title	YearMade	ShowType	ImageURL
Alien	1979	Movie	https://m.media-
Alien³	1992	Movie	https://m.media-
Alien: Covenant	2017	Movie	https://m.media-
Alien: Resurrection	1997	Movie	https://m.media-
Alien vs. Predator	2004	Movie	https://m.media-
My Stepmother Is an Alien	1988	Movie	https://m.media-
Alien Nation	1988	Movie	https://m.media-
Alien Raiders	2008	Movie	https://m.media-
Alien Abduction	2014	Movie	https://m.media-
Alien Autopsy	2006	Movie	https://m.media-

This is a regular Plex grid, but instead of reading data from a database it is using a custom BlockFetch function calling the OMDb API and returning the results in a Plex array. In the next exercise we'll look into how this was achieved.

VHS By Mail want to make a couple of changes to the search functionality.

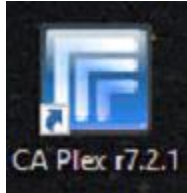
- Add an additional Search panel to allow searches by Movie, TV Show or Episode – Exercise Four.
- Change the poster display to something more visually appealing – Exercise Five.

Exercise 4: Add New Search Tab

The Plex Application

Let's take a look at how the Plex application calls the API.

Open Plex.



Open the model VHSByMail.mdl in C:\Conference 2020 Workshops\API\LocalModel

The model contains some basic patterns scoped under the entity named 'API' which contain some functionality for accessing APIs.

Note: These patterns are not complete, or fully tested. Use at your own risk!

Our main entity is 'OMDb API' which inherits from API.FetchResource. It inherits the view APIFetch which scopes a BlockFetch function which attempts to mimic the standard BlockFetch functionality, allowing it to work with Grids etc.

Plex doesn't have native support for API calls, so this is provided with Java source code objects although it could easily be expanded to use C# or even C++ source code.

The source code objects the BlockFetch uses are:

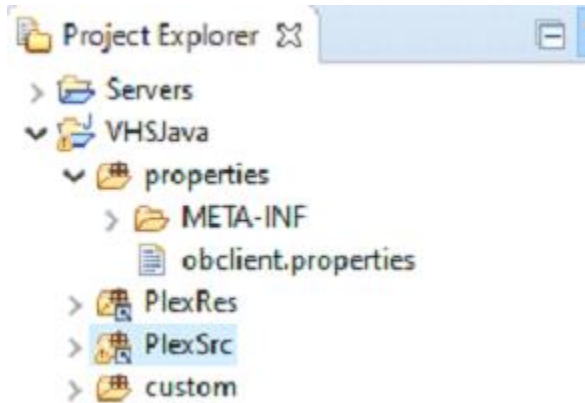
- APIRequest – This takes a server URL, a parameter string, and a request method (GET, POST, PUT etc.) and returns the JSON response in a Plex Object field.
- GetArrayInstance – Each API returns data in a different format, so this source code will need to be edited to identify the array part of the response to pass to an Array variable.
- MapJSONToFields – This takes the array instance and maps the JSON properties to Plex fields that will be returned by the BlockFetch
- CheckOMDbResponseStatus – This is a custom source code object written for this API to check the response is correct and report any error messages.

The APIRequest source code takes the server URL as a parameter, and it would be possible to hard-code a value for this in the application, but that would make it difficult to switch between a test API server and a production server. For this reason, this solution reads the API server as a run-time property from the obclient.properties file. For similar reasons the API key should be fetched from the obclient.properties file to allow it to be switched easily.

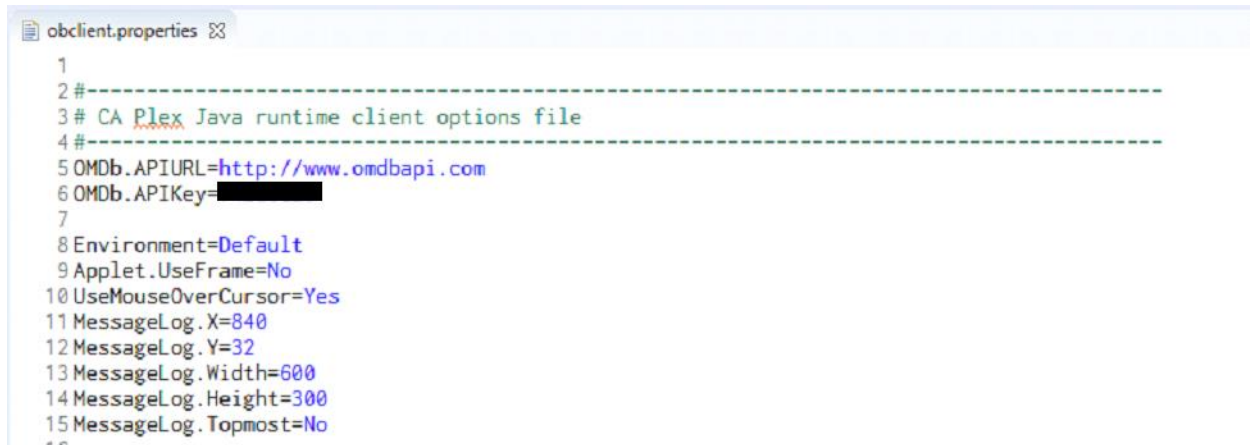
The function OMDb API.GetAPIConnectionProperties reads those properties and returns them to the calling function.

Open the obclient.properties file

In Eclipse, expand the VHSJava project in the Project Explorer on the left. Next, expand the 'properties' folder and double-click on the obclient.properties. The file will open up in an editor in the main area.



Scroll to the top of the file to see the properties for the OMDb API.

The image shows the Eclipse editor with the 'obclient.properties' file open. The text in the editor is as follows:

```
1
2 #-----
3 # CA Plex Java runtime client options file
4 #-----
5 OMDb.APIURL=http://www.omdbapi.com
6 OMDb.APIKey=[REDACTED]
7
8 Environment=Default
9 Applet.UseFrame=No
10 UseMouseOverCursor=Yes
11 MessageLog.X=840
12 MessageLog.Y=32
13 MessageLog.Width=600
14 MessageLog.Height=300
15 MessageLog.Topmost=No
```

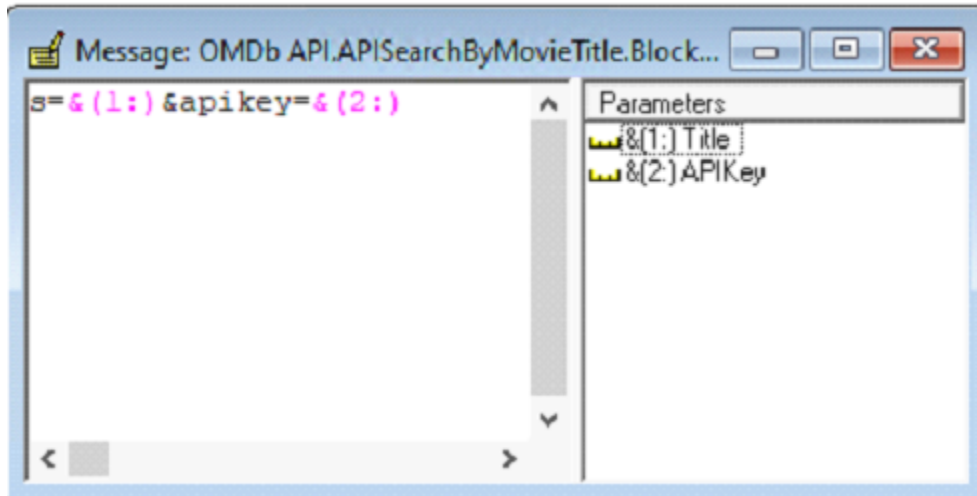
If you registered for the OMDb API key, you can enter that value in the OMDb.APIKey property.

Press Ctrl + S to save your changes.

View API Parameters

The APIRequest source code object takes a parameter string to pass to the API. This can be created in Plex with the use of Format Message 'APIParameters' scoped to the BlockFetch function.

In Plex, open the OMDb API.APISearchByMovieTitle.BlockFetch.APIParameters message object. It's pretty simple, it just takes the Title and APIKey fields and formats them in a way that the API expects.



Create a New Search Tab

We can use this information to add a new Search Tab to the API Search function. The new Search will allow us to filter the results to return only Movies or TV Shows.

The API provides a parameter 'type' to let us do exactly this, just set the value to 'movie' or 'series', so our request will look like:

<http://www.omdbapi.com/?s=banana&type=series&apikey=67ecd31b>

To achieve this in Plex we need to add a grid to the 2nd tab and create a new view with a BlockFetch to use with the grid. We can add the Title and Type fields to the panel to let the user enter their searches, and we need to format the parameters to pass to the API.

We can use inheritance to create the new view as it is very similar to the existing OMDb API.APISearchByMovieTitle view.

Create the New View

Add the triples:

OMDb API ENT view VW APISearchByMovieTitleAndType

OMDb API.APISearchByMovieTitleAndType VW is a VW OMDb API.APISearchByMovieTitle

The function we will be using already exists, so to use this new view we just need to add a replaces VW continuation triple:

OMDb API.SearchByType FNC replaces VW API.FetchResource.APIFetch

...TRP by VW OMDb API.APISearchByMovieTitleAndType

Add the Search Fields

We can now add our search fields to the panel. Add the following triples:

OMDb API.SearchByType.Panel PNL displays FLD Title

...TRP for VAR Selections

OMDb API.SearchByType.Panel PNL displays FLD ShowType

...TRP for VAR Selections

We also need to display the selected record in the DetailP region, so add the following triple:

OMDb API.SearchByType.Panel PNL displays view VW OMDb API.APISearchByMovieTitleAndType

...TRP for VAR DetailP

Clean Up the Panel Design

Spend a minute cleaning up the panel design so that it looks something like this.

The screenshot shows a web application interface with a search panel on the left and a detail panel on the right. The search panel has a text input field for 'Title', radio buttons for 'Movie' and 'Series', and a 'Search' button. Below this is a table with columns 'Title', 'YearMade', 'ShowType', and 'ImageURL'. The detail panel has an 'ExternalID' input field, a 'Title' input field, a 'YearMade' input field with the value '0', radio buttons for 'Movie' and 'Series', and an 'ImageURL' input field.

Hide the Selections<ShowType> label and move the values horizontally, set the width of Selections<Title> Edit control to 260. Set the width of the Edit controls for DetailP<Title> and DetailP<ImageURL> to 250. For the grid, set the width of the Title column to 200, and set the width of the rest of the columns to 100.

Add the New Search Parameter to the API Request

We are going to need a new BlockFetch for the new tab, so as everything is mostly in place in the existing BlockFetch we can use inheritance to create it.

OMDb API.APISearchByMovieTitle.BlockFetchByType FNC is a FNC

OMDb API.APISearchByMovieTitle.BlockFetch

OMDb API.APISearchByMovieTitle.BlockFetchByType FNC file name NME OMDbTypeBF

OMDb API.APISearchByMovieTitle.BlockFetchByType FNC impl name NME OMDbTypeBF

Now we can set up the Search tab to use the view and BlockFetch function

OMDb API.SearchByType FNC replaces VW API.FetchResource.APIFetch

...TRP by VW OMDb API.APISearchByMovieTitle

OMDb API.SearchByType FNC replaces FNC API.FetchResource.APIFetch.BlockFetch

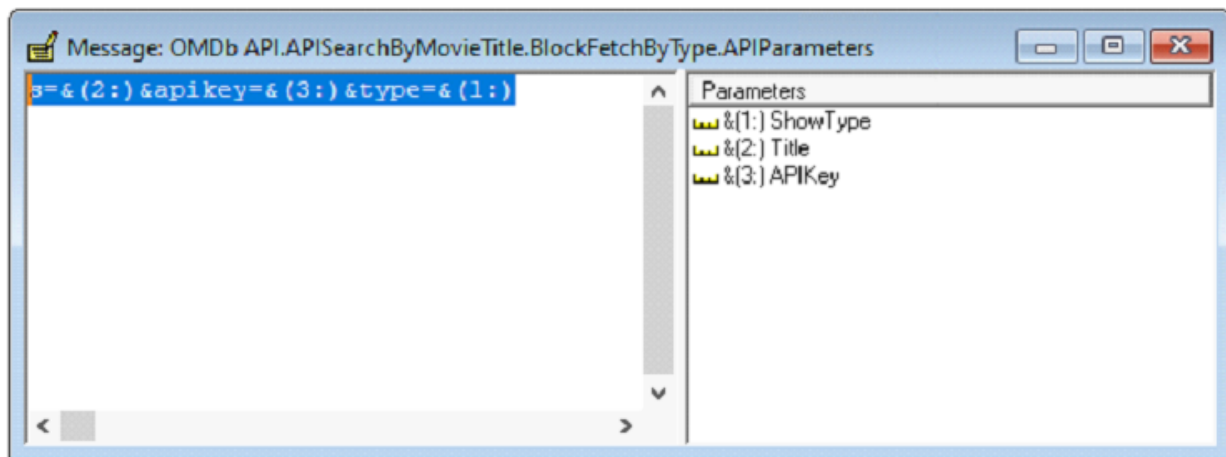
...TRP by FNC

OMDb API.APISearchByMovieTitle.BlockFetchByType

We still need to pass the Show Type to the API, so we can add a new parameter to OMDb API.APISearchByMovieTitle.BlockFetchByType.APIParameters. Note that this message object is inherited from the original BlockFetch, so it has inherited the Title and APIKey parameters, but if we add an explicit MSG Parameter FLD triple it will insert that parameter to position &(1:) so we'll need to reorder the other parameters.

Our new parameter string is:

s=&(2:)&apikey=&(3:)&type=&(1:)



Now we can retrieve the Show Type search field and use it to populate the ODb request.

Add the new Selection field to the BlockFetch:

OMDb API.APISearchByMovieTitle.BlockFetchByType FNC input FLD ShowType

...TRP for VAR/Selections

The APIParameters format message statement is set up to automatically map from the Selections variable, so there's nothing to do for that.

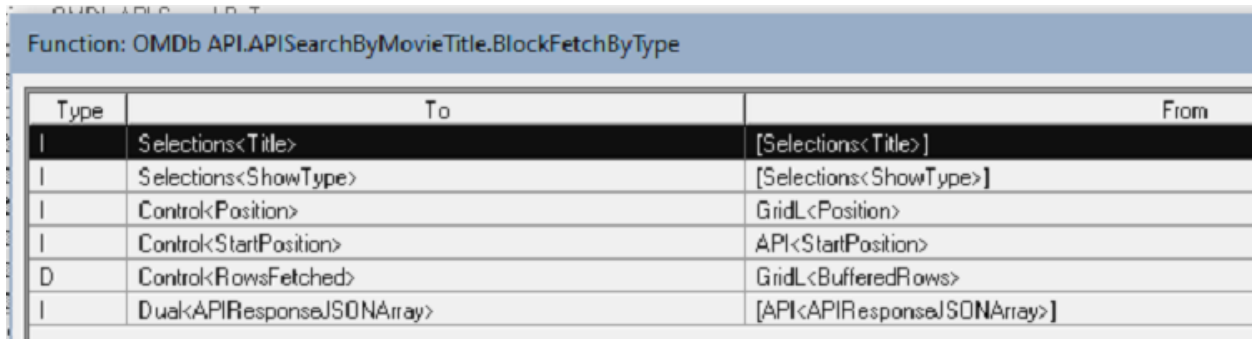
We now need to default the ShowType field from on the Search panel to 'Movie'.

Open OMDb API.SearchByType in the Action Diagram editor and add the following to the end of the Pre Point End Initialize collection point.

Set Selections<ShowType> = <ShowType.Movie>

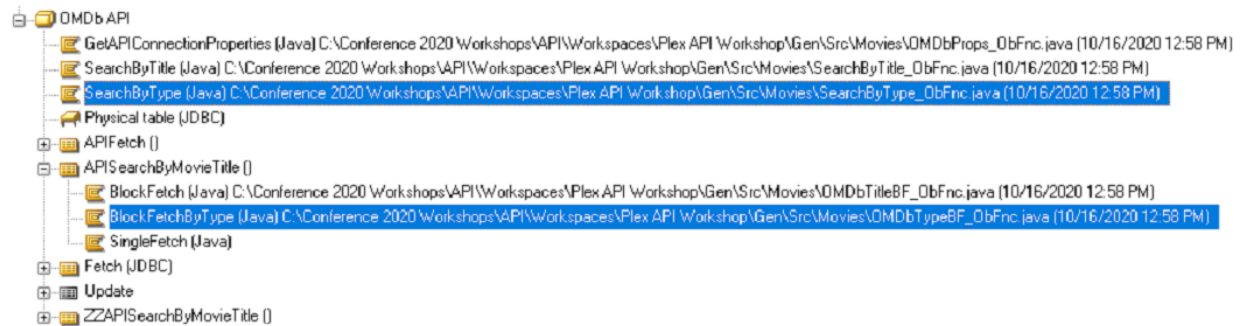
Put Selections

The call to the BlockFetch already maps the Selections region, so verify that it is being passed.



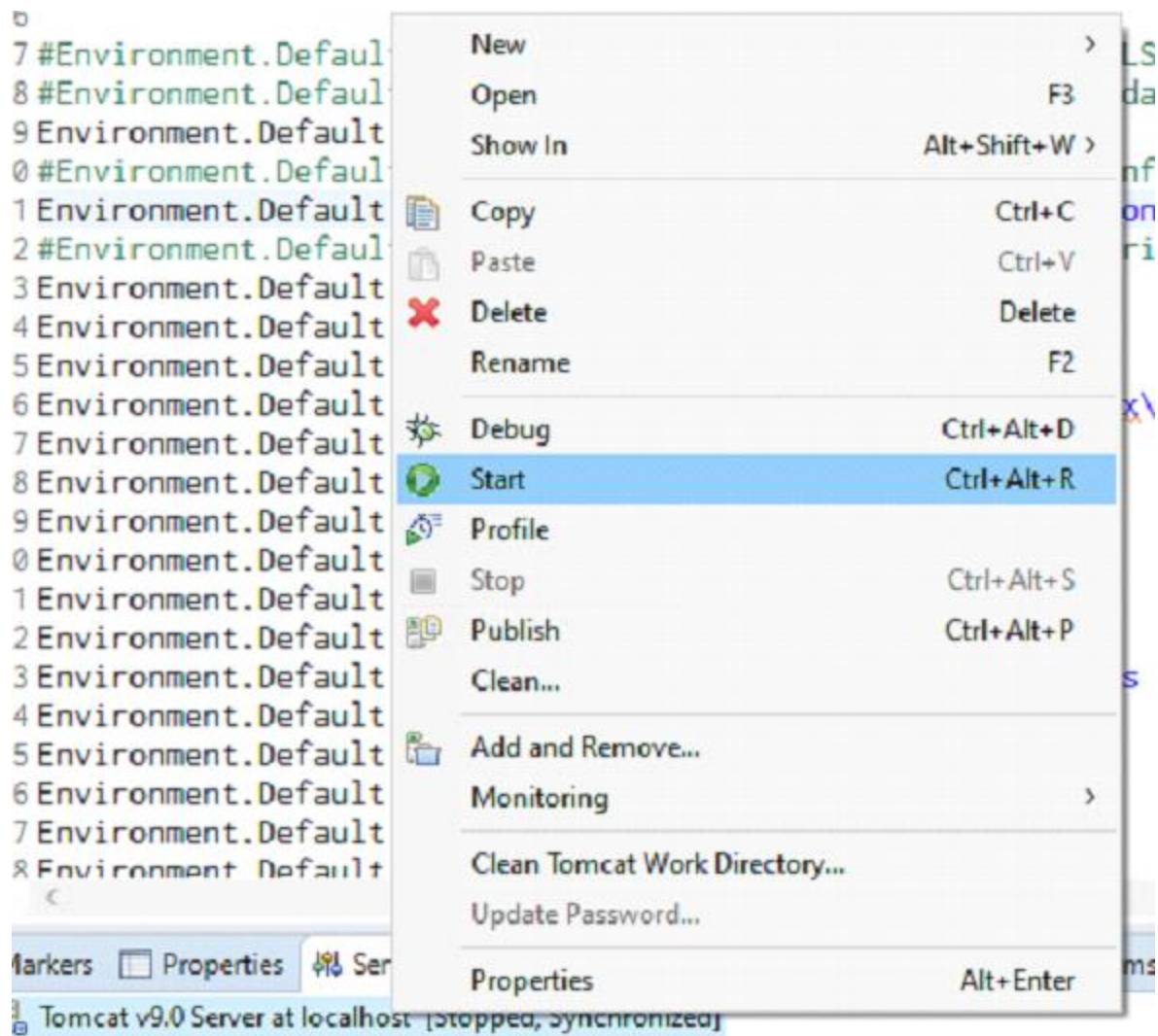
Type	To	From
I	Selections<Title>	[Selections<Title>]
I	Selections<ShowType>	[Selections<ShowType>]
I	Control<Position>	GridL<Position>
I	Control<StartPosition>	API<StartPosition>
D	Control<RowsFetched>	GridL<BufferedRows>
I	Dual<APIResponseJSONArray>	[API<APIResponseJSONArray>]

Now everything is in place to try it out. Drag the OMDb API.SearchByType and OMDb API.APISearchByMovieTitle.BlockFetchByType functions to the Gen & Build window, and Generate them (no need to build, we will do that in Eclipse).



Switch to Eclipse. Select the VHSJava project, right-click and select Refresh from the context menu. Eclipse will pick up the newly generated functions and the WebClient template builder will build them.

Now restart the Server on the Server tab by right-clicking on Tomcat v9.0 server in the Servers tab and selecting 'Start'.



Once the status is changed to [Started, Synchronized] we can open Chrome and access the application at <http://localhost:8080/vhsweb/wc>

You should now be able to click on the 'Shows By Type' tab and be able to filter your results by Movie or Series.

The screenshot shows a web application interface with a dark header bar at the top containing a small icon of a camera and an envelope. Below the header, there are two tabs: "Movies By Title" and "Shows By Type". A search bar contains the text "alien". To the right of the search bar are radio buttons for "Movie" and "Series", with "Series" selected, and a "Search" button with a magnifying glass icon. Below the search bar is a table with the following data:

Title	YearMade	ShowType	ImageURL
Ben 10: Alien Force	2008	Series	https://m.me
Alien Nation	1989	Series	https://m.me
Ben 10: Ultimate Alien	2010	Series	https://m.me
Unsealed: Alien Files	2012	Series	https://m.me
Benji, Zax & the Alien Prince	1983	Series	https://m.me
Alien News Desk	2019	Series	https://m.me
Alien Dawn	2013	Series	https://m.me
My Girlfriend is an Alien	2019	Series	https://m.me
Alien Encounters	2012	Series	https://m.me
Alien Highway	2019	Series	https://m.me

To the right of the table, there are several fields: "ExternalID" with the value "tt10874298", "Title" with the value "My Girlfriend is an Alien", "YearMade" with the value "2019", and "ShowType" with radio buttons for "Movie" and "Series", where "Series" is selected. Below these fields is the "ImageURL" field with the value "https://m.media-amazon.com/images/M/MV5BMzoxZ".

It looks good so far, but in the next exercise we will apply a new WebClient template that we can use to call the API directly from JavaScript.

Exercise 5: Apply Custom WebClient Template

Introduction

In this exercise, we'll take a look at another way of consuming APIs for WebClient applications. So far we've been using Java source code to consume the API and pass the returned values into Plex fields.

In this exercise we'll be using a WebClient control template with JavaScript code to consume the API and provide a visual representation of the data.

Despite the similarity in names, JavaScript is not actually related to Java. It was designed to run in a web browser and interact with the elements of the page. As it was designed for the web, it has inbuilt support for requesting internet resources and processing the responses.

Review Control Templates

A WebClient control template is used to provide a representation of a Plex control in a browser. By default, each control type, e.g. a Combo-box or a Grid is represented by a control template that offers the same functionality as the equivalent desktop control, so the web Grid will look and behave the same as a C++, Java or C# grid. Additionally, WebClient allows you to use a custom control template to override the default control template. We can see an example of this on the first tab where the image of the movie is displayed.

The screenshot shows a web application interface with two tabs: "Movies By Title" (selected) and "Shows By Type". A search bar contains the text "pulp" and a "Search" button. Below the search bar is a table with the following data:

Title	YearMade	ShowType	ImageURL
Pulp Fiction	1994	Movie	https://m.media-
Pulp	1972	Movie	https://m.media-
Pulp: A Film About Life, Death and	2014	Movie	https://m.media-
Marvel 75 Years: From Pulp to P	2014	Movie	https://m.media-
Pulp Fiction: The Facts	2002	Movie	https://m.media-
Pulp	2013	Movie	https://m.media-
Pulp Sport	2003	Series	https://m.media-
'Pulp Fiction' on a Dime: A 10th A	2004	Movie	https://m.media-
Pulp Comics: Louis C.K.'s Filthy S	1999	Movie	N/A
Pulp Diction	2003	Movie	N/A

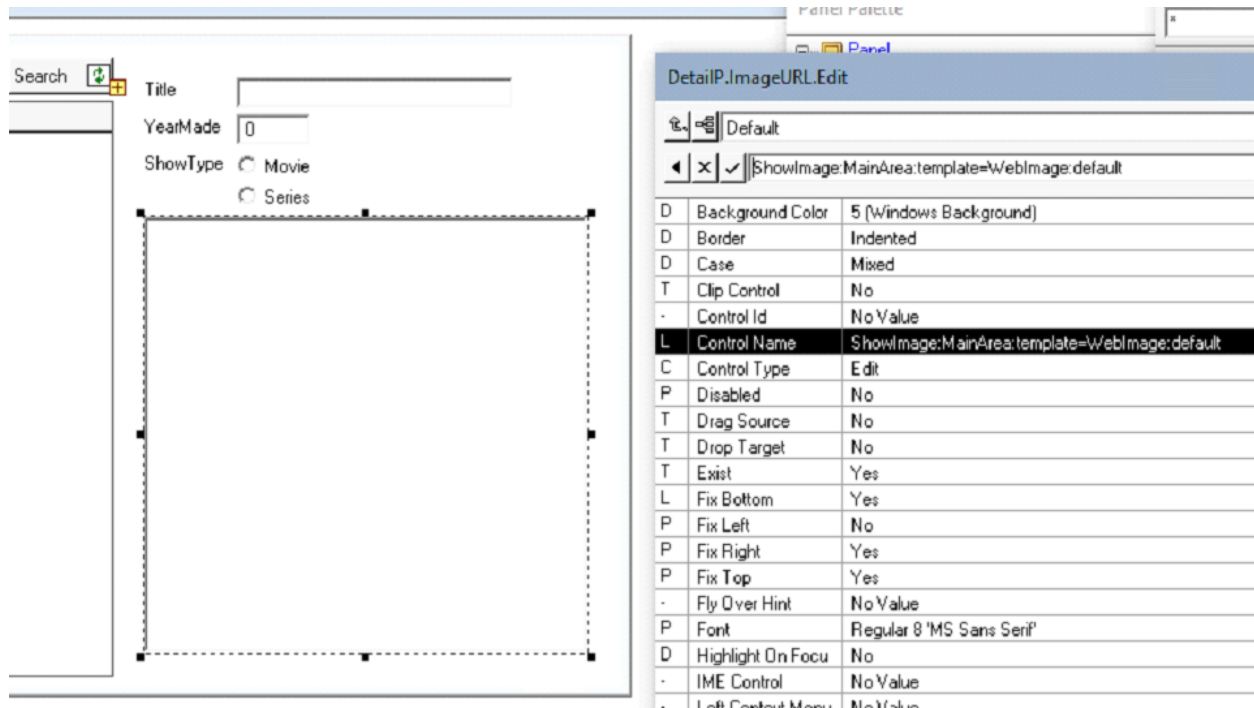
To the right of the table is a detail panel for "Pulp Fiction". It shows the title "Pulp Fiction", the year "1994", and the show type "Movie" (selected with a radio button). Below this is a movie poster for "Pulp Fiction" featuring Uma Thurman and John Travolta.

We can look at the panel design to see how this is implemented. Open up OMDb API.SearchByTitle.Panel in the panel editor.

The DetailP.ImageURL.Edit control has a control name of

ShowImage:MainArea:template=WebImage:default

The ShowImage part is a unique control name for the control, MainArea tells WebClient where to insert the control on the page, and template=WebImage directs WebClient to apply an alternate control template. The last part default tells WebClient to use size and position from the panel design.



In Eclipse, the WebImage.ctrl file is in the VHSJava/Custom folder. Essentially this template creates an HTML element with the src attribute set to the value of the field.

We'll be using another custom template RotatorView.ctrl on our new Search tab. This template uses HTML and CSS to provide a more advanced control which displays the movie poster, then rotates to show the description when the mouse hovers over the control. To retrieve the movie description, the control makes a request to the OMDb API, passing in the ID of the movie or show. We have the ID of the show stored in the DetailP<ExternalID> field, so we will use that field to assign the template to. In addition, the template needs the APIKey, so we can direct WebClient to reference the value of the DetailP<APIKey> field.

Before we do that, let's take a look at the template.

[Review RotatorView.ctrl Template](#)

In Eclipse, double-click on the VHSJava\Templates\RotatorView.ctrl template to open it in an editor.

We won't go into all the details, but essentially it provides a JavaScript function getMovieInfo() to call an API, then update HTML elements with the response details.

The API call is made with just a few lines. First we set the URL using \${variable} syntax to insert our parameters into the string.

```
const url = `http://www.omdbapi.com/?i=${OMDbID}&apikey=${movieAPIKey}`;
```

We then make the actual API request with the `await fetch()` method which performs a synchronous GET request. Usually requests are asynchronous, but for simplicity the `await` waits for the response before moving on.

```
const response = await fetch(url);
```

We then assign the response to a JavaScript variable `movieJSON`. From there we can populate the HTML elements with information about the movie or show.

```
const movieJSON = await response.json();
```

The `SetValue` action calls the `getMovieInfo()` every time the associated Plex control value is changed. The current value of the Plex field populates the `/(!This)` value.

```
/(!Action:SetValue)  
getMovieInfo(/(!This));  
/(!Action)
```

The last part to look at is how the `APIKey` is received.

```
var movieAPIKey = /(!MovieAPIKey);
```

This will populate the control with the value of another control with the control name “`MovieAPIKey`”.

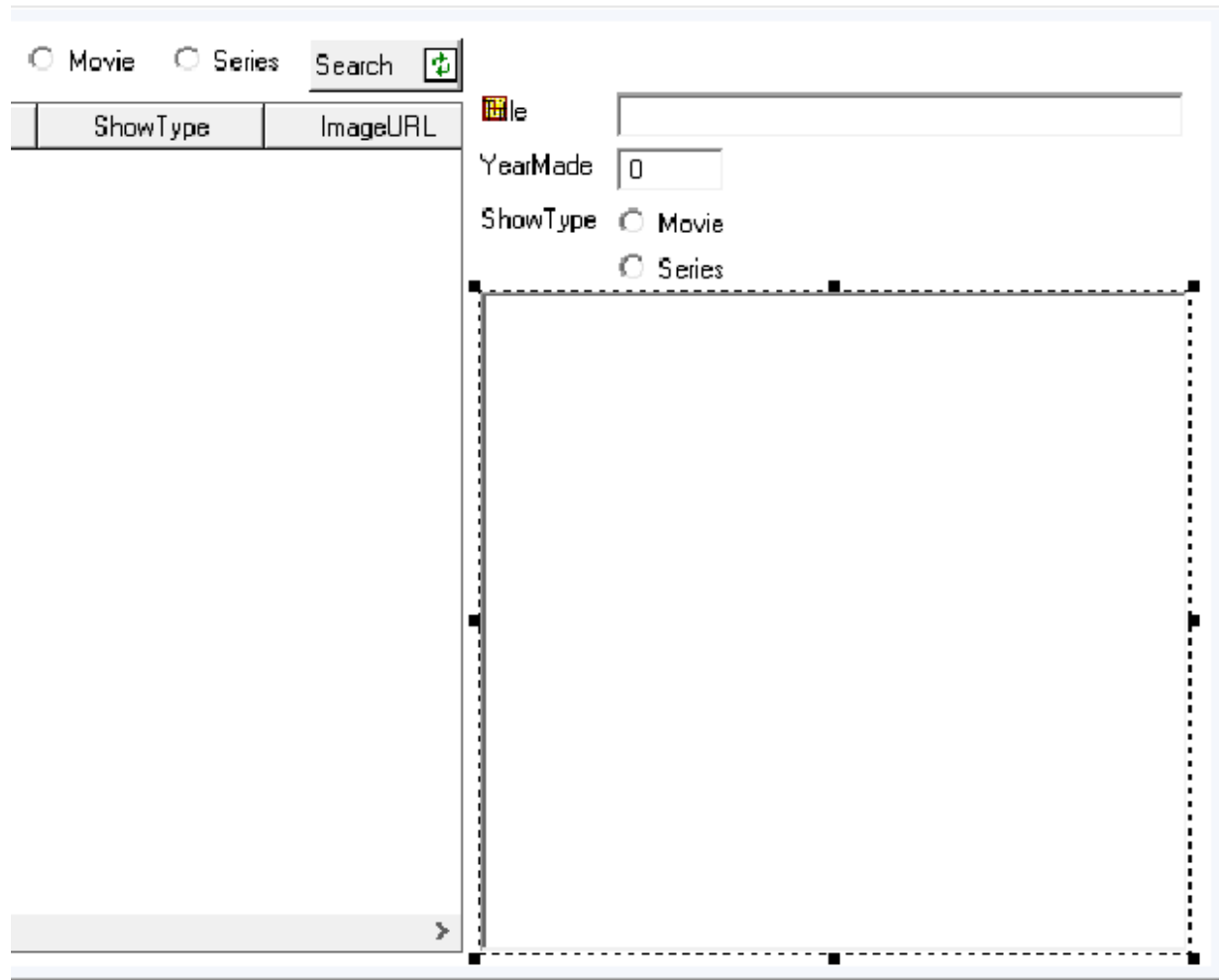
Let’s make the changes in Plex to implement this control template.

[Apply the control template to our Search Tab](#)

Open the panel `OMDb API.SearchByType.Panel` in the panel editor.

We don’t need the `DetailP<ImageURL>` field, so we can set the `Visible` property to `No`.

The `DetailP<ExternalID>` is the one we’ll be using for the template, so move the field below the others, then set the label for that field to `Visible = No`, and resize the `ExternalID.Edit` control to 312,292 to fill the rest of the region.



To associate the control with the template we can set the Control property of the ExternalID.Edit to
rotatorView:MainArea:template=RotatorView:default

DetailP.ExternalID.Edit		
Default		
rotatorView:MainArea:template=RotatorView:default		
D	Background Color	5 (Windows Background)
D	Border	Indented
D	Case	Mixed
T	Clip Control	No
·	Control Id	No Value
L	Control Name	rotatorView:MainArea:template=RotatorView:default
C	Control Type	Edit
P	Disabled	No
T	Drag Source	No
T	Drop Target	No

Finally for the panel we need to add the APIKey field so we can pass the value into the template.

Drag the APIKey field from the Object Browser to the DetailP region on the panel. We can hide the field as we don't need to see it by setting the Visible property to No, then select the APIKey.Edit control and set the Control name to `MovieAPIKey`

DetailP.APIKey.Edit		
Default		
MovieAPIKey		
D	Background Color	5 (Windows Background)
D	Border	Indented
D	Case	Mixed
T	Clip Control	No
·	Control Id	No Value
L	Control Name	MovieAPIKey
C	Control Type	Edit
P	Disabled	No
T	Drag Source	No

This will allow the value of this field to be retrieved in other templates. In the RotatorView.ctrl it is referenced on the line

```
var movieAPIKey = /(?!MovieAPIKey);
```

Now all we need to do is populate the DetailP<APIKey> in the action diagram.

Open the action diagram for function OMDb API.SearchByType and add the following code in the Post Point End Initialize collection point.

Set the API Key

Call `OMDb API.GetAPIConnectionProperties`

Set `DetailP<APIKey> = Output<APIKey>`

Put `DetailP`



Edit Point End initialize

Post Point

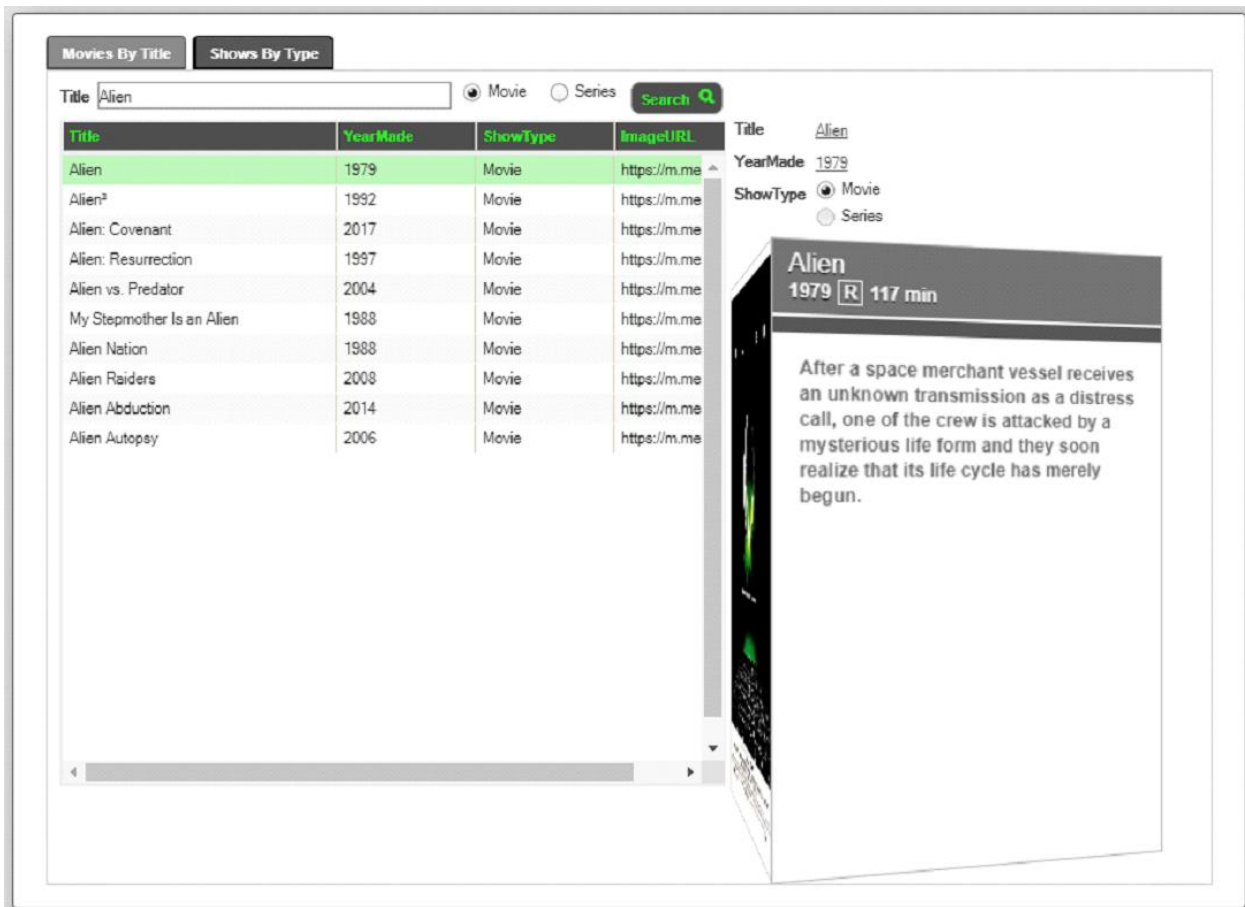
Save your model and regenerate the OMDb API.SearchByType function.

Switch to Eclipse. Select the VHSJava project, right-click and select Refresh from the context menu.

Eclipse will pick up the newly generated functions and the WebClient template builder will build them.

Now restart the Server on the Server tab by right-clicking on Tomcat v9.0 server in the Servers tab and selecting 'Start'.

Visit your web application at <http://localhost:8080/vhsweb/wc>



Search for a movies or TV show, then move your mouse over the poster image to reveal more details about the show.

Now you've seen a couple of ways to consume APIs from your Plex application you can start thinking of ways to expand your existing applications.

Exercise 6: Getting Stared with the Quarkus API Workspace

Introduction

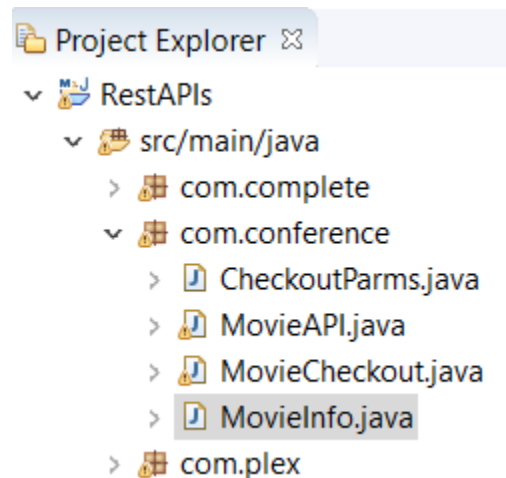
We will be using the Quarkus Java Library to easily create and deploy our APIs. Feel free to check out <https://quarkus.io/> to get more information on all Quarkus has to offer. There is a Quarkus plugin for Eclipse, which will allow us to develop and launch our APIs from our Eclipse workspace.

Workspace

Open Eclipse and open the “C:\Conference 2020 Workshops\API\APIWorkspace” workspace. Here we have 1 project (RestAPIs). This is already setup to reference the proper Quarkus libraries. If you expand “src/main/java” folder at the top, you will see several packages.

“com.complete” contains the completed version of the workshop. Feel free to use this as a reference if you get stuck, but it would be best to not just copy/paste our code from these files.

“com.conference” contains stub versions of all the Java Classes that we will be working with. All your code changes will be done in the files in this package.



Exercise 7: Creating our First API

Introduction

APIs through Quarkus need a Java Object to know what information to return from the API. Our API will return the movie's ID, Name, and Availability. So, we will create a class to house these return parameters. Then, we will write some Java code to call our Plex function (Movies.By Name.BlockFetchByName) and set its output to be returned by our API.

Creating the Class

Open the `MovieInfo.java` file in "com.conference". Here we will set the necessary attributes that we want to return, as well as the Java Constructors needed for the class (this is required by Quarkus). Add the following code to the `MovieInfo` Class:

```
public int movieId;
public String movieName;
public boolean available;

public MovieInfo() {
}

public MovieInfo(int movieId, String movieName, boolean available) {
    this.movieId = movieId;
    this.movieName = movieName;
    this.available = available;
}
```

We are setting up to store the Movie Id (integer), Movie Name (string), and Availability (boolean) in the `MovieInfo` object.

NOTE: You may notice a method already in the `MovieInfo` class called `getPropertyPath`. Please leave this here. This just makes our code cleaner when we call the Plex function.

Calling the Plex Function

When calling a Plex function from outside of Plex, the Plex runtime provides a method we can use: `ObApplication.obCallFunction`

This takes in the `ObUserApp`, the `Environment`, `Input`, and `Function Name`. Then, outputs a `String` array of the output fields.

The main issue with this is building the input string array. It is difficult to tell what order everything should be in.

To get around this we will use some other Plex runtime methods to build this for us.

We have also created our own Wrapper method to make using this easier: CallHandler.callPlexFunction

Creating our Method

Our method will be returning a list of the MovieInfo Objects. Start by putting this code in the MovieInfo class:

```
public static List<MovieInfo> blockFetchMovies(){
    List<MovieInfo> dataArray = new ArrayList<MovieInfo>();

    return dataArray;
}
```

We have created our method, initialized our return object and said it will return a List of MovieInfo Objects.

NOTE: These 2 lines will always be the first and last line of our method.

Setting up the Plex Objects

We need to setup our ObUserApp object. We will initialize it and point it to our Property File location. We will also setup a try/catch block to handle our calls. This will help us catch any errors and report it in the console. Please add the following code to our method:

```
//Call Movies.By Name.BlockFetchByName
ObUserApp app = new ObUserApp(MovieInfo.getPropertyPath());
String fncName = "Movies.AA16F";
try {

} catch (Exception e) {
    e.printStackTrace();
} finally {
    app.clearReferences();
}
```

NOTE: All of our remaining code will go in the "try" block of code.

Calling the Plex Function

We now need to make a new instance of the function object we want to call. Then, map the input parameters, similar to what you would see in Plex Generated Java. And finally, call the function.

Paste the following code into the "try" block:

```
AA16F_ObFnc fnc = new AA16F_ObFnc(app.m_callMgr);
ObVariableGroupX obIn = (ObVariableGroupX) fnc.getInVariable();
obIn.getVariable("AA16F_Control").getAsObCharFldField("S5trh2n").assign(new ObCharFld("Y"));
obIn.getVariable("AA16F_Control").getAsObLongFldField("S5trh30").assign(new ObLongFld(0));
obIn.getVariable("AA16F_Position").getAsObCharFldField("MovieName").assign(new
ObCharFld(""));
obIn.getVariable("AA16F_Selections").getAsObCharFldField("MovieName").assign(new
ObCharFld(""));
```

```
String[] outParms = CallHandler.callPlexFunction(app, fnc, obIn, fncName);
```

Here we create a new instance of the ObFnc for AA16F (which is our BlockFetch function).

Create a new instance of the Input Variable and map the input variable objects.

Then, call the Plex Function using our “callPlexFunction” wrapper method. Without this method, we would be manually building the Input String Array (which is a major pain to get right).

Parsing the Output

Now that our call is complete, we have our Output in a String Array. This is similar to the Input String Array we avoided manually creating, but it is much easier to deal with.

We know the Rows Fetched count is the 4th object in the Array.

NOTE: Java arrays start on index 0

We know our FetchedData starts on 6th object of the Array. And our FetchedData has 4 fields in it.

Using this information, we can create a loop to parse the Output array and create the List of MovieInfo Objects for us to output.

Put the following code after the callPlexFunction method call:

```
int rowsFetched = Integer.parseInt(outParms[3]);
int rowFieldCount = 4;
int startingPoint = 5;
for (int i = 0; i < rowsFetched; i++) {
    MovieInfo movie = new MovieInfo();
    movie.movieId = Integer.parseInt(outParms[startingPoint]);
    movie.movieName = outParms[startingPoint + 1];
    String availYN = outParms[startingPoint + 3];
    if(availYN.equals("A")) {
        movie.available = true;
    } else {
        movie.available = false;
    }
    dataArray.add(movie);
    startingPoint = startingPoint + rowFieldCount;
}
```

Here you can see our simple loop where we create a new instance of our MovieInfo Object. We set the ID and Name. Then, some simple code to set the Available boolean value.

With this, we have our method to call our Plex BlockFetch function and return the List of our MovieInfo Objects.

Just for reference, your blockFetchMovies method should look like this:

```
public static List<MovieInfo> blockFetchMovies(){
    List<MovieInfo> dataArray = new ArrayList<MovieInfo>();
```

```

//Call Movies.By Name.BlockFetchByName
ObUserApp app = new ObUserApp(MovieInfo.getPropertyPath());
String fncName = "Movies.AA16F";
try {
    AA16F_ObFnc fnc = new AA16F_ObFnc(app.m_callMgr);
    ObVariableGroupX oblIn = (ObVariableGroupX) fnc.getInVariable();
    oblIn.getVariable("AA16F_Control").getAsObCharFldField("S5trh2n").assign(new ObCharFld("Y"));
    oblIn.getVariable("AA16F_Control").getAsObLongFldField("S5trh30").assign(new ObLongFld(0));
    oblIn.getVariable("AA16F_Position").getAsObCharFldField("MovieName").assign(new
ObCharFld(""));
    oblIn.getVariable("AA16F_Selections").getAsObCharFldField("MovieName").assign(new
ObCharFld(""));
    String[] outParms = CallHandler.callPlexFunction(app, fnc, oblIn, fncName);
    int rowsFetched = Integer.parseInt(outParms[3]);
    int rowFieldCount = 4;
    int startingPoint = 5;
    for (int i = 0; i < rowsFetched; i++) {
        MovieInfo movie = new MovieInfo();
        movie.movieId = Integer.parseInt(outParms[startingPoint]);
        movie.movieName = outParms[startingPoint + 1];
        String availYN = outParms[startingPoint + 3];
        if(availYN.equals("A")) {
            movie.available = true;
        } else {
            movie.available = false;
        }
        dataArray.add(movie);
        startingPoint = startingPoint + rowFieldCount;
    }
} catch (Exception e) {
    e.printStackTrace();
} finally {
    app.clearReferences();
}

return dataArray;
}

```

Creating our API

Now that we have our class defined for what we will return, and our method to call our Plex function, we now need to create the API to expose this code.

Open the MovieAPI.java file in “com.conference”. You will see the following code already there:

```

@Path("/movies")
@Produces(MediaType.APPLICATION_JSON)

```



```
@Consumes(MediaType.APPLICATION_JSON)
```

This is the initialization of our URL path the API will use (which will be localhost:8080/movies) and the definition of the format of any body elements we will be receiving and returning (JSON).

In the MovieAPI class, add the following code:

```
@GET
public List<MovieInfo> callMovieBlockFetch() {
    return MovieInfo.blockFetchMovies();
}
```

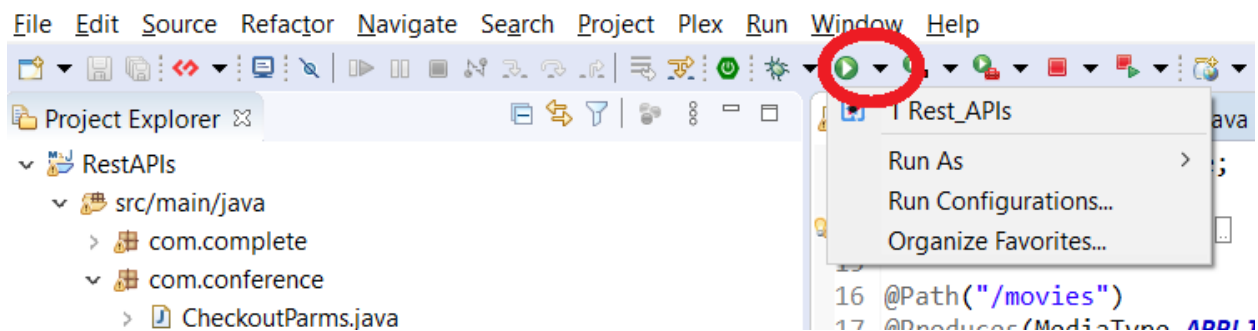
The “@GET” sets our API to use the GET HTML method.

The “List<MovieInfo>” in the definition tells the API that we will be returning a list of the MovieInfo object. This will all be interpreted by Quarkus to return a JSON array.

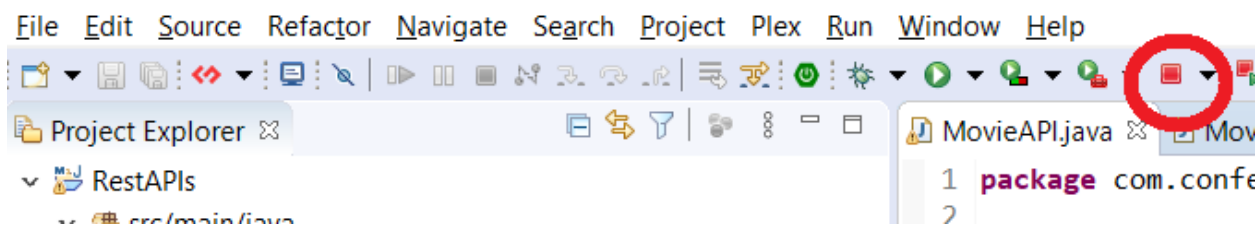
The final thing we do in our method is to call our blockFetchMovies method to get all our movies from the Plex function.

Deploying the API

To start the Quarkus API Service from Eclipse, click the dropdown arrow next to the Green Play button at the top toolbar. Then select the “RestAPIs” application. The Quarkus service will start. All of the APIs will be running on localhost:8080.



To Stop the Quarkus service in Eclipse, click the Red Stop Button in the main toolbar.



As we continue with the workshop, we will need to restart the Quarkus service. To do this, just stop the service and start it again using the same steps in this exercise.

Testing the API

Using PostMan, we can test our API. Just launch the postman application (there should be a shortcut on the desktop). Enter "localhost:8080/movies" into the URL and click Send. You should see a JSON array of our movies in the return body.

The screenshot shows the Postman application interface. At the top, a request is defined as a GET method to the URL localhost:8080/movies. The interface includes tabs for Params, Authorization, Headers (6), Body, Pre-request Script, Tests, and Settings. The Query Params section is currently active, showing a table with columns for KEY, VALUE, and DESCRIPTION. Below this, the Body tab is selected, displaying the response in JSON format. The response status is 200 OK, with a time of 1755 ms and a size of 683 B. The JSON response is a list of three movie objects.

KEY	VALUE	DESCRIPTION
Key	Value	Description

```
1 [
2   {
3     "available": false,
4     "movieId": 9,
5     "movieName": "Alien"
6   },
7   {
8     "available": true,
9     "movieId": 4,
10    "movieName": "Attack of the Killer Tomatoes"
11  },
12  {
13    "available": true,
14    "movieId": 10,
15    "movieName": "Beetlejuice"
16  },
17 ]
```

Exercise 8: Viewing our API in Action

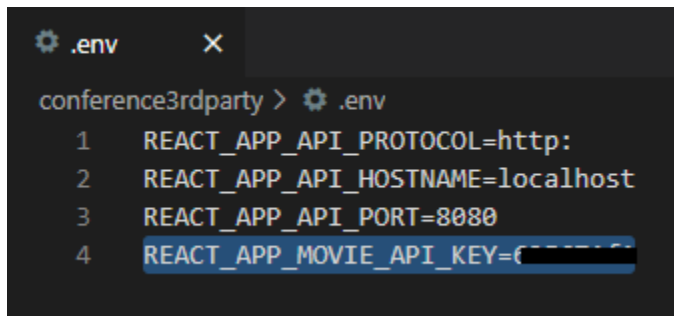
Introduction

An app has been developed using ReactJS. This app will be consuming our APIs. This allows us to see our Plex code being used in a completely different light. To run the application, we will only need to open VSCode and start the service.

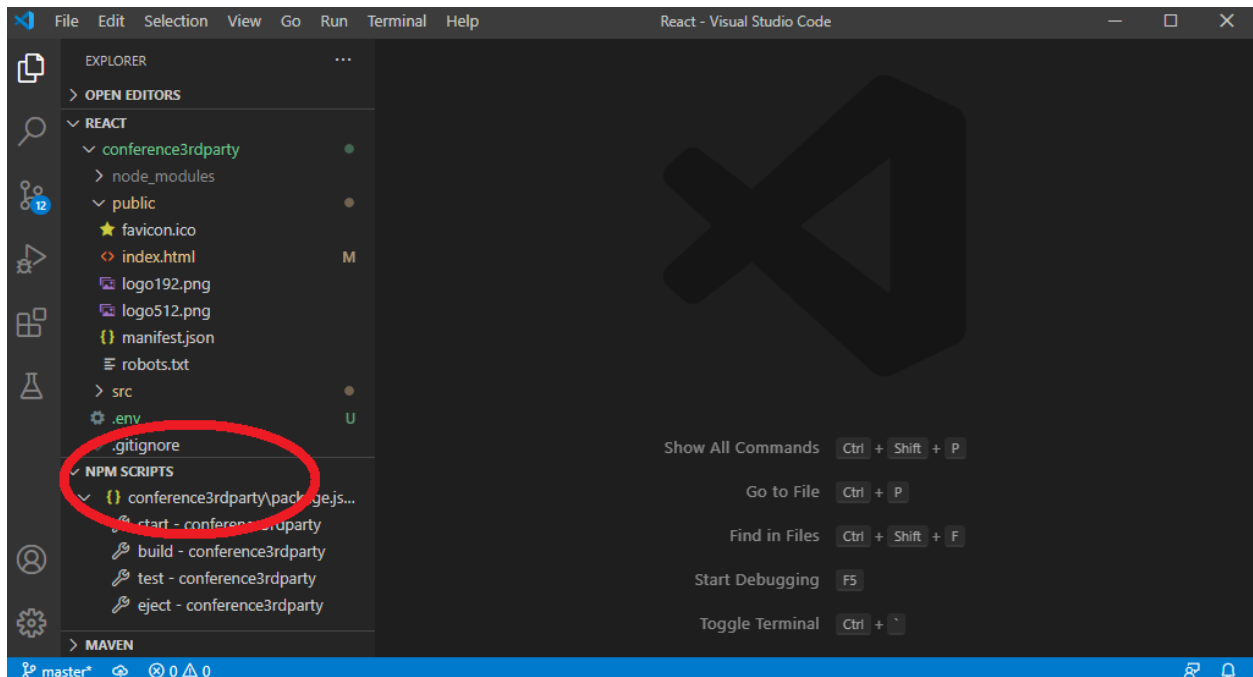
Starting VSCode

Open VSCode (a shortcut should be on the desktop). The workspace will open to the proper folder. In the Left area of the screen, there is a section called “NPM Scripts”. This is an easy way to launch NodeJS based applications.

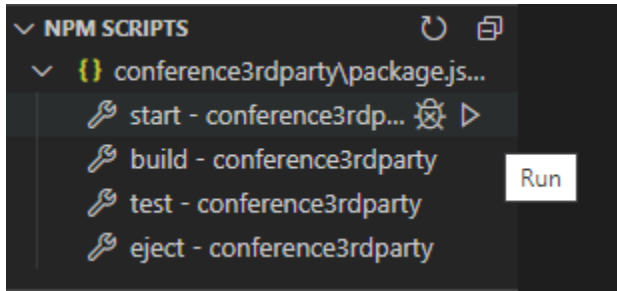
NOTE: Before you start the Server, if you created your own OMBD API Key, please enter it in the .env file. It should already be open, but if you need to search for it, you can use CTRL+P to search for the file.



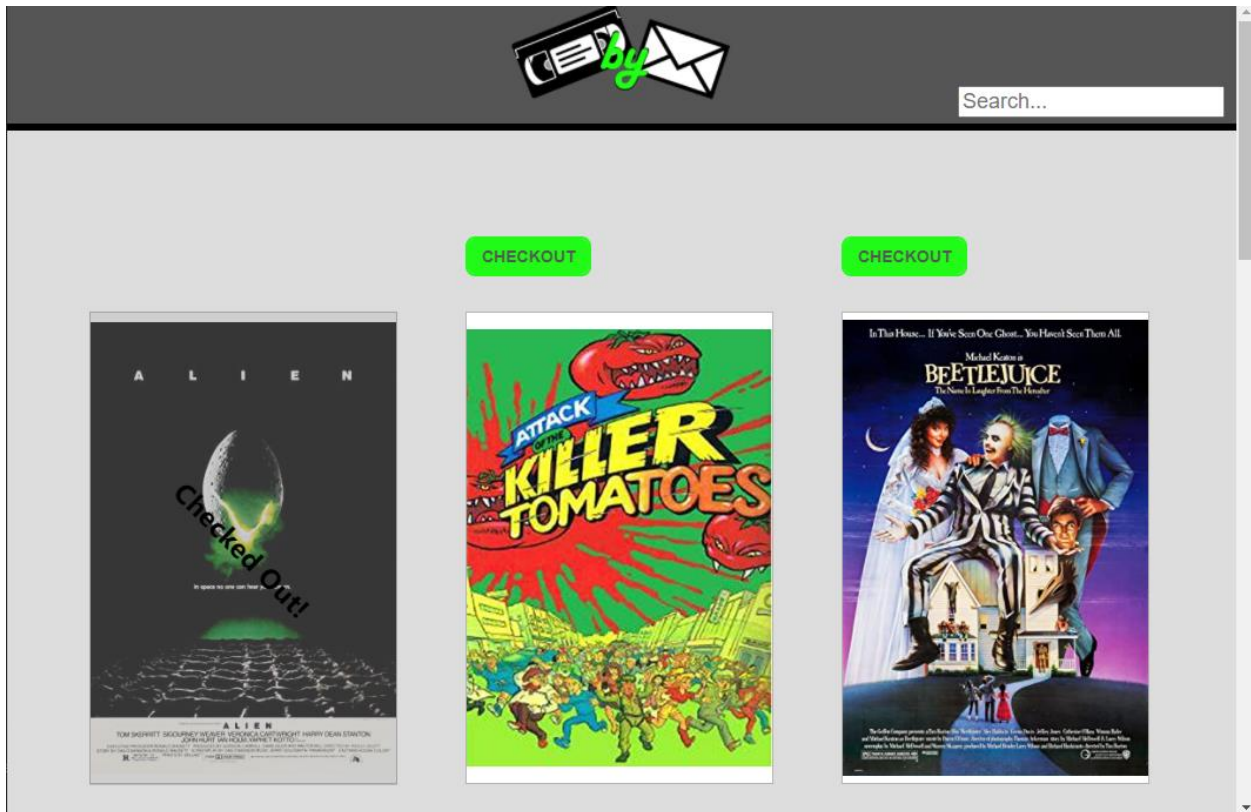
```
.env
conference3rdparty > .env
1  REACT_APP_API_PROTOCOL=http:
2  REACT_APP_API_HOSTNAME=localhost
3  REACT_APP_API_PORT=8080
4  REACT_APP_MOVIE_API_KEY=C
```



Hover over the “start” statement and you should see a play icon. Click that and it will start the React app.



A browser window should be automatically opened to localhost:3000 and you will see our app running.



Exercise 9: Adding a Parameter to your API

Introduction

In our React application, we would like to be able to search for movies based on the title. In our BlockFetch function, we already have the ability to do this. So, we just need to add the ability to include a search parameter in our API and pass it along to our BlockFetch call.

Modifying our BlockFetch Call

In our MovieInfo.java file, find the blockFetchMovies method we created. In the definition of the method, we should add "String movieName" to the parameters. The line should look like this:

```
public static List<MovieInfo> blockFetchMovies(String movieName){
```

We can now reference this parameter in our code. In the line where we set the input parameter for "Selections<MovieName>", instead of passing in "", we will pass in our movieName parameter. The line should look like this:

```
obIn.getVariable("AA16F_Selections").getAsObCharFldField("MovieName").assign(new  
ObCharFld(movieName));
```

Modifying our API

Our code is ready to accept the parameter. We now need to change our API code to handle this. In the MovieAPI.java file, we will make several changes. First, in the definition of the callMovieBlockFetch method, we will add a parameter. The code will look like this:

```
public List<MovieInfo> callMovieBlockFetch(@QueryParam("movie") String movie) {
```

Here we are telling Quarkus that we accept a parameter called "movie" in our request and to map the value into to a Java string called movie.

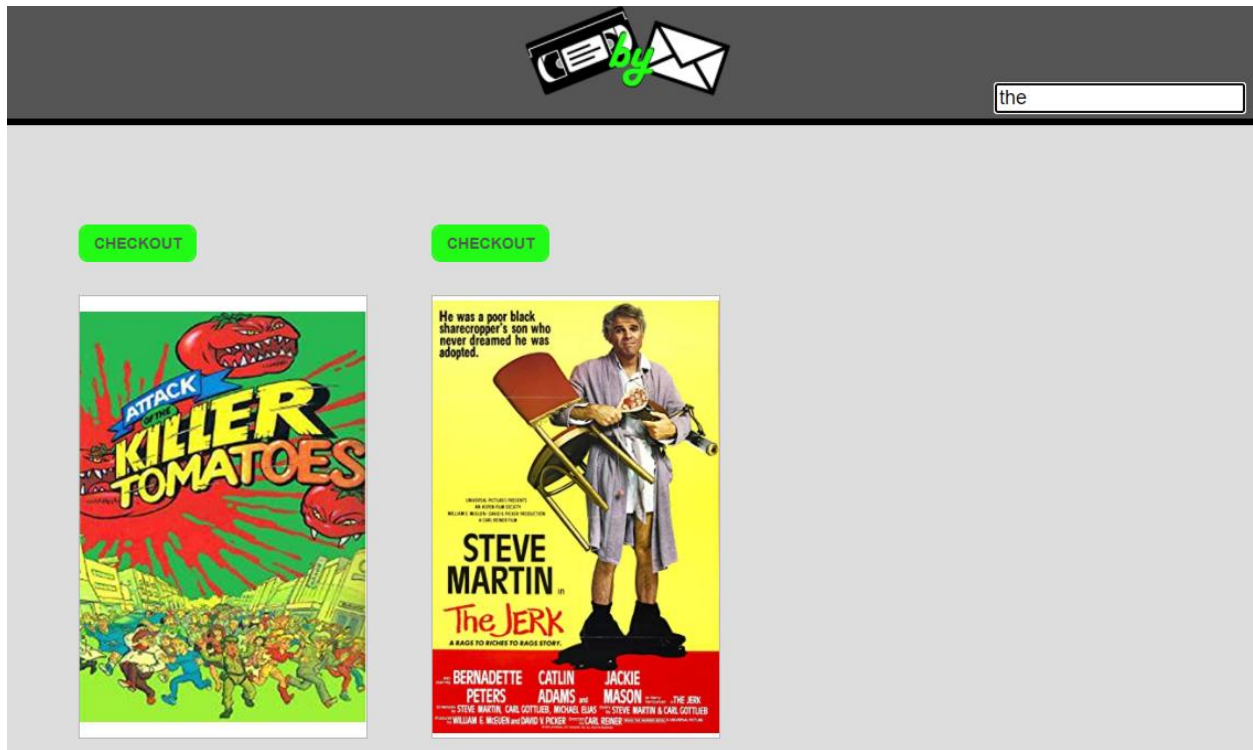
We just need to pass this into our call to the blockFetchMovies method. But, we should also add some code to handle if no parameter is passed in. Change the method to have the following code:

```
public List<MovieInfo> callMovieBlockFetch(@QueryParam("movie") String movie) {  
    String movieName = "";  
    if(movie != null) {  
        movieName = movie;  
    }  
    return MovieInfo.blockFetchMovies(movieName);  
}
```

Here we just make sure to handle the possibility of a NULL value and call our method.

Testing

Restart the Quarkus service and refresh our React app. You should now be able to use the Search Bar in the top right of the application.



Exercise 10: Using the Request Body with your API

Introduction

We need to expose the movie checkout process through an API. Since, this is changing data, this will need to be a PUT API call. For the parameters for the PUT, we will be using the body of the request. In Quarkus, the body parameters are represented by a Java Class. So, we will first create a new class to house our request body. Then, use it to call our Plex function (MovieCopies.CheckoutWrapper).

Create the Class

In the “com.conference” package, find the empty class called “CheckoutParm”. This class will have 2 integer attributes: movieId and customerId. We also need to create the proper constructors for our class. When you are finished, your class will contain the following code:

```
public int movieId;
public int customerId;

public CheckoutParms(){
}

public CheckoutParms(int movieId, int customerId){
    this.movieId = movieId;
    this.customerId = customerId;
}
```

API Setup

In our MovieAPI.java file, we are going to add a new item after the callMovieBlockFetch method. Please copy the following code:

```
@POST
@Path("/checkout")
public MovieCheckout callCheckout(CheckoutParms parms) {
    return MovieCheckout.checkoutMovie(parms.movieId, parms.customerId);
}
```

Please take note of the @POST. This tells Quarkus that our API will use the Post Method. Also notice the addition of the @Path. This appends “/checkout” to our existing path. So, the URL to access the API will be “localhost:8080/movies/checkout”.

Final note: we have not created the checkoutMovie method yet, so you will have a Java error, please ignore that for now.

Movie Checkout

We will now create the code to call our Plex function to process the movie checkout. Open the empty "MovieCheckout" class in "com.conference".

First, we need to setup the attributes that Quarkus will use for the return body of our API. Please put the following code in the MovieCheckout class:

```
public String returnCode;
public String returnMessage;

public MovieCheckout(){
}

public MovieCheckout(String returnCode, String returnMessage){
    this.returnCode = returnCode;
    this.returnMessage = returnMessage;
}
```

This will have Quarkus return a body containing 2 simple strings: a Return Code and a Return Message.

Now, we need to write our checkoutMovie method to run our Plex code.

Paste the following code into our MovieCheckout class:

```
public static MovieCheckout checkoutMovie(int movieId, int customerId) {
    MovieCheckout output = new MovieCheckout();

    //Call Movie Copies.CheckoutWrapper
    ObUserApp app = new ObUserApp(MovieInfo.getPropertyPath());
    String fncName = "MovieCopies.CheckOutWrapper";
    try {
        CheckOutWrapper_ObFnc fnc = new CheckOutWrapper_ObFnc(app.m_callMgr);
        ObVariableGroupX obIn = (ObVariableGroupX) fnc.getInVariable();
        obIn.getVariable("CheckOutWrapper_Input").getAsObLongFldField("MovieID").assign(new
ObLongFld(movieId));
        obIn.getVariable("CheckOutWrapper_Input").getAsObLongFldField("CustID").assign(new
ObLongFld(customerId));
        String[] outParms = CallHandler.callPlexFunction(app, fnc, obIn, fncName);
        output.returnCode = outParms[0];
        output.returnMessage = outParms[3];
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        app.clearReferences();
    }

    return output;
}
```


Similar to what we did before with our Plex call, we first initialize our output for the method. Then setup our Plex object and map the input parameters. Call our Plex function, and finally parse the output.

Viewing Our Work

Restart the Quarkus Service. The React app that we are running is already setup to use this API. Feel free to click the Checkout button for a movie. The React App does not currently show the number of copies remaining, but if you checkout a movie enough, the stock will eventually be depleted and that movie will show as Checked Out.

