CAS WEBINARS

EQUINOX TRACK 2: MAKING PERL WORK FOR YOU IN EVERGREEN

JUNE 11, 2020

CAPTIONING PROVIDED BY:

CAPTIONACCESS

contact@captionaccess.com

www.captionaccess.com

>> Good afternoon, everyone. We are going to get started with this program. This track is sponsored by NC Cardinal with captioning made possible by Equinox Open Library Initiative. And we would like to thank our captioner.
We would also like to thank the other conference sponsors for making this event possible. Mobius, Bibliomation, and Evergreen Indiana. This event is being recorded and will be available on YouTube following the conclusion of the conference. We would like to encourage anyone to use the chat window to pose questions. The facilitators will be collecting questions along the way and passing them to the presenter at the end of the session or whenever they ask for questions. Caps off your presented for this session is Galen Charlton and Mike Rylander from eco- knocks. They will be talking about making pearl work for you in Evergreen.
>> Okay. Good morning and good afternoon, everybody. Thank you for this session about using pearl to write in Evergreen. I will start by talking about a couple of housekeeping items. Then hand it over to Mike.
During this presentation, we will be grounding our example in a specific and timely case, namely the project we took on to -- features in Evergreen to -- services. This feature includes a lot of curbside APIs. So we will be talking about the way open SRF messages connect with pearl services, and then we'll start talking about the nitty-gritty, showing some of the boilerplate you need to start up a Perl service, writing attention, during the mechanics of actually getting that service up and running, to that streaming versus atomic metaresponses and can discussing common tasks and tools. This will your pretty fast-paced presentation. We had a lot of slides and we only have 40 minutes. So we will try to make a tiny bit of time to answer questions, but we may need to follow up after the presentation.
With that, I will turn it over to Mike Rylander to talk about the story of curbside.
>> All right. Hello, everybody. Thank you, Galen. And I apologize in advance to our captioner. This will be pretty quick. So as Galen said, we are going to use curbside pickup as an example. For how to create the service for Evergreen. On top of the OpenSRF infrastructure.
To start off, we received a few questions in early May from ivories asking if there was some way to facilitate curbside pickup, in Evergreen, and there was not. So we surveyed what was out there today, and we took a look at how five different libraries were doing curbside, in five different ILS player. Sally Fortin did that over the course of a couple of days in early May and one of the processes did involve using a bullhorn at the library.

So we identified some critical properties that we wanted to, if we are going to take this on, create some sort of curbside service facilitation in Evergreen. We need to make sure that this service would fulfill. So we needed to make sure that we didn't preclude any particular physical workflow or prescribed any physical workflow. We wanted to make sure that we did not change how holder relations work in Evergreen. We wanted to make sure that we are being as thin as possible so we chose staff just what is needed by patrons right now compared and we shall patrons just what they need in order to facilitate the pickup of items. That they have requested a curbside appointment for.

We also want to make sure that this is entirely optional so that you did not have to turn on globally. Each library can make it on decision about how to make use of this and to what degree. And that's true with any consortium, not just Evergreen. We want to make this as reportable as possible so we can see how effective curbside pickup was effective for libraries.

And of course, bullhorns had to be supported.

So we did a feasibility assessment. We need to make sure that we can get this then quickly because some states are starting to open up, and others are even starting to do in person Now, we don't know how long the full curbside will be useful. And we needed to make sure that as libraries were starting to open up, they would have something, they wouldn't have to re-architect their curbside pickup process halfway through the reopening.

It took me about 3.5 hours to do a first cut of the core business logic on a Saturday afternoon, and that was based on the initial assessment we had done to see how people were trying to do curbside in the world already. We needed to make sure that we could, in fact, avoid touching hold answer collation is this logic as it exists today, and we convinced ourselves that we could do that. And we need to make sure that we would be able to do this as seamlessly as possible for patrons. And how all three of those -- all three of those criteria in the feasibility assessment, we convince ourselves that we can succeed.

So on May 16 was when I did the initial proof of concept, some of the core APIs, not all of them. On the 25th, the staff UI was substantially greater, and at the same time we were able to pull together a patron UI that does not get in the way of any existing futures, and we can push patrons to where they need to go. As a bonus, one pain point, in the my OPAC area was there are tables that each role really wants to be a form. And I found a mechanism to make that possible in HTML 5 tables using CSS. So now there is a way to much more easily create forms and tables.

So we have some code available for you now. There is a git branch. The number is embedded in the name. We do have a concerto based test server available for community testing. You can go there pretending to be a patron or a staff member and see how it works. And there's a set of testing documentation for community testing at that URL.
That has not only some testing notes, but also staff and patron login information for testing purposes as well as documentation for part of the documentation we want to provide are some ways that you can make use of this with different physical processes. Some small libraries, may have different needs from large libraries, so we wanted to outline a few different usage scenarios so that everybody would be able to have a starting point to design their curbside service around.
So we have gotten some feedback on the launch as well as from internal testing. And we have ideas for future enhancement. We want to add the ability for staff to claim appointments so that multiple staff can coordinate more easily in a single building even if they are spread out further than just one service desk. We want to create receipt triggering functionality right now. The curbside delivery does not generate a receipt. But you can go from the curbside interface to the patron interface to print out the items, a list if the patron like a put printed receipt. But we also will be triggering email receipts.
Long-term, we want to create a way to let patrons signal they want to do curbside at the time that the place holds so when staff call the patient to inform them that their items are available for pickup, they can ask of the paper would like to set a curbside time. Right Now, curbside is based on the curbside times available to patients are based on the hours of operations for the library. Given that the hours of operation should reflect the times during the day when patrons can interact with the library. But we do want to create a mechanism for a separate schedule for curbside availability. So that you can more easily do in person pickup and curbside pickup.
And we want to enhance the date and time restrictions so that patrons can choose a date after the card has expired or after the holds on the hold shelf have expired from the shelf.
And now we move on to how we actually did some of this. So we actually start off some of this these types of how these Perl in every with a discussion of open serve and how it works and how all the pieces fit together. That really, 99% of the time, you do not have to care about.
And people always say, wait, I want to know about this or what, it's very big and scary. And there are lots of pieces to it. And understanding all of

the moving parts of open serve can be beneficial, especially if you are writing a particularly applicator set of services. But what in reality, you really ever need to know about any of that beyond the basics of how to write some business logic and the few things you need to do to tell OpenSRFabout your business logic.
So you do have to have some basic boilerplate code. This is code -- this is Perl code that tells OpenSRF what the allocation is, gives you some information and some tools to interact with the rest of your application the under single function that you are writing in this case.
So going through these, the first line there, the package line tells OpenSRF the name of the Perl module that should be loading in order to find the set of Perl code that it should run in response to requests from OpenSRF clients. It is a policy to always use strict and use warnings when you are running an OpenSRF application in Evergreen service, because that will allow Perl to tell you when you have made a syntax error or if you have forgotten to use the proper SQL or variable or things like that.
In order to be able to talk to the rest of the application, you need to be able to create OpenSRF sessions. So you use OpenSRF app session, and in a to be a service, you need to make your package, make your new service child service of the open ILS application, and then you open ILS application, and then use base to tell Perl that this module, this package, is an open ILS package application.
The next three use minds are useful utilities that you will need in almost all OpenSRF applications that exist inside of Evergreen. So it's a good idea to go ahead and put those in place. And the next line, my $U, saves you from having to type that long name over and over again. Anywhere you need to get to, in a – utils function, it will give you the on package name.
The same basic idea happens with the OpenSRF chills logger. It creates a longer variable that you can use to write long lines out. They are useful for debugging, useful for providing information about when things go wrong so you can go back and fix it.
And then the last line in the package, the last line of the file, is going to be a truth eval. Just put a 1 and 8; there. That is a requirement of Perl. The last one need to return true, or Perl want to get is compiled correctly.
So writing a function. Writing a function that is going to be used as an OpenSRF API call. You need to be able to take input. There is a special convention, you don't have to understand why, you just have to know it's always this way, when you write an OpenSRF method, you're going to have to account for two magic inputs that you're going to get. The first one is the info can't deftly $self. That variable is passed in and tells the

function how it was called. OpenSRF provides information that was the -- we will see later about what that name was, whether or not this method is a streaming method, and in fact, you can give it extra information about the context of that API call and get a bit more -- you can make a bit more informed decision about how the method should act based on how exactly how it was called.

The connection object, generally called $conn in Evergreen applications is a way to talk to who called us. It responds in several different ways. After that, if the parameters that the caller passed in the request when they originally made it.

The parameters at this level are just regular Perl variables, related strings of numbers, or Perl objects with -- types. Your just want to get parameters like you would in any Perl function after the invocation.

So this is how you use those. As you see the self parameter is an object that has an API named method on it. And in this particular case, we were creating it for the curbside appointment, whether we were called with an API name that includes the string update.

The connection object, or how it talks back to the client in this case, just responding to the client.

Enter the parameters, most public services in Evergreen require an authtoken, we'll talk a little bit more about the utility functions and we can quickly check whether or not the color is authenticated. Ask which workstation you are at. They let you test who the person is that is logged in at that station. Things like that.

So we wrote some business logic. We used the parameters to decide on what you want to do and how you want to do it. And I have something to return to the caller, you can just return it like any Perl function, you can return it and it will go back to where it was that you're sub was called. This is the simplest way to return something in OpenSRF service. And it's what you want to use when you had exactly one thing to return, and nothing more after you have sent the data back to whoever asked for it.

But if you have more than one thing, to return to a caller, such as when you are returning a list of all of the potential curbside appointment that a user might have, you can return more than one by calling the respondent method on the conn object over and over again. In other services were you may have a bit more, you need to do in the middle of -- for each of the pieces of data you want to return, this can significantly reduce the amount of time that it takes to get the first result back to the caller. Because the data is sent back to the OpenSRF client immediately when you call respond. There are some caveats, but we will talk about those later.

So you can have a complicated method that say, such as a hold and looks up a bunch of auxiliary information and calculates the position of the hold in the hold list, and while it's doing that for the second one, it has Artie sent the first one after the client and can be processing it for display or requesting an update of the hold or other things. And this is where you can get some parallelism between the client and server by having the server and your business logic respond as quickly as possible to a series of data.
Now, there may be cases where you want to respond as quickly as you can. But you may have other things you want to do. So you can't return the data using the Perl standard return because you have stuff you want to do after that and returning exits the sub.
So as we do when we are creating curbside appointment here, we respond complete with the curbside object as it currently exists in the database and then we go on and talk to the Action Trigger service and receive the we say we need to send them a new confirmation message. So we can send the caller the object and then we can proceed to make the trigger call after we have said that. -- sent that. And we have got the data back to the caller as quickly as possible while still being able to go out and do other processes.
This is used in other services in Evergreen quite a bit. We and up parallelizing a lot of searches by doing this kind of thing.
So we have a function, and the sub, and we need to tell the world about it. They don't know what is that everything that is in a package, because some of them might be support functions. So when you have a sub, Perl package that is with an appropriate boilerplate, you need to tell OpenSRF about it by using the register method call. So you call register method and you pass the set of parameters, and register method is a part of the OpenSRF structure that tells the rest of the system about methods that should be published as -- calls.
Anti-node as you tell register method which of the subs you have written are, in fact, the business logic you want to expose as an open serve API or new Evergreen API, utility name of the sub with the method parameter. You will recall that create data point what was the name of the sub in the package that we saw as one of the earlier examples.
And did you want to give it an API name. You give it an API name because there is no reason that multiple packages can't have subs with the same name. So we want to be able to name our API clerk severally from the sub in the package. This also separates the way that Perl things about the names of things from the way that OpenSRF does. So OpenSRF doesn't have to be -- we have -- services. We could conceivably have Python or

Java services. Different languages name things in different ways. So this is the language specific part of that.
And the convention in Evergreen is to use an API name that starts with the name of the service. So this service is called open ILS.curbside. We have an API name called open ils.curbside.update.appointment.
And then -- and this is very important. You want to provide a signature. Now, there's nothing by default that the signature does actively on the curb itself. But it is very important to provide a signature for a few different reasons. And the way you provide the signature is the signature parameter is a hash with one key of that hash being param, and that is an array of patches and that describe what the parameter is. So we will call the first parameter passed the education parameter. We type a string, because it is essentially a -- hash. In the patient ID is a number, the date and time are strings, which we don't have special types at the Perl level for dates and times, specifically. And then the library ID, which defaults to where the workstation was.
And all of those typed and described parameters can be used in ways which we will see.
And finally, you want to tell OpenSRF about what is going to come back from your function. In this case, you do that using the return parameter, or the return key to the signature parameter. In this case we just described, we will come back and it will be a curbside object, curbside appointment object. If everything went well, if there was an error that we could detect, we would return that ever. And if we got some bad data, then we return nothing.
So why do we do the signature? The most important reason is really that it's being kind to other developers that are going to come along and look at your code. Because having in-line documentation the structure is very helpful. It makes it quick to look at how much you want to use the functions.
Second is that there is a documentation generator. And it works for any public OpenSRF application. If the excess XSL did work, but some browsers have broken it. So I will be looking at how to on break it going forward, hopefully.
And lastly, and it doesn't apply to every facility, because we haven't made use of it, but OpenSRF does supply a strict mode that will check the parameters coming in on request from a client to make sure there are enough of them, to make sure they are the correct type. And if not, we will reject the request upfront without ever going to code. It's a way to protect your code from bad data coming in -- malicious data.

And finally, we talked earlier about the respond method on the set object. If you're going to call it more than once, and you want to be able -- you want the clients to be able to decide whether they want to wait for everything or get the responses as quickly as possible, you should market the method as stream. If you do that, stream two, and 1 is true in Perl, the register method will actually call itself, creating a second registration for this API, and it will change the name of the API so that it ends in .atomic. We'll be seeing more of that in a minute, but tablet the caller decide what the best way to talk to your API is for their particular situation.
And with that out of the way, I will pass it back to Galen for a big chunk of the remainder.
>> Okay. Thank you, Mike. So if you have been following along, with Mike, but you now have is a Perl code that does something.
The next step is to tilt the application, in this case, Evergreen, that it exists. So I will go through some of the mechanics of taking your fancy new service and hooking it up.
The first thing to do is in the configuration file, OpenSRF.XML, adding to the service and the action.
So OpenSRF for XML will expect an element, a convention that is given the name, as service, open ILS.efficacy.curbside, they're saying what it is and where to find it.
So the part that is highlighted, the implementation, is the name of the Perl module that we have just created. And we know it is Perl, because the language element looks like Perl. So if this were a different language, the implementation would be pointing to a different path.
Some other settings of interest, I will go through all of them Now, include things like max requests, which is saying how long from requests -- four curbside process before it exits? This can be useful for dealing with potential leaks. Max and min children, and in spiritual and max spiritual and, saying how many of these will the server need? In this case, curbside wouldn't be intermittently -- service, so min and max should apply to most instances. If you have max children, set it higher.
So we have defined a new Perl service. Then left the you also have a decision whether to make this a service the accessible through the public OpenSRF router. Making it public has a couple of implications. A public service can be accessed through things like the web socket, JavaScript or others. And a public service, needs to ensure patient of its methods of requiring an off token, or if the version check, it doesn't matter if any random client looks at API -- but most public services will require authentication.

If you don't get service public, it goes only through the private router. And for private services, it can be invoked by other services. They could be accessed -- but otherwise not directly callable from the outside world.
So services in this case that we want to be publicly accessible, there is a bit more to write.
Then the mechanical question. Really you put your modules? It would go to open ILS source, Perl mods, and then in this case,/application, and added content p.m.
Where you're going to the process of developing the surface -- service, you get yourself in the Perl mods directly and install, and anytime you need to update the service, you can. Then to actually start the service, you can see the OpenSRF dog OpenSRF serve user to start, stop or restart the service. And if the Perl code is actually a TPAC, you would use a Patty 2 to use the latest and greatest open ILS code.
So because I like to live dangerously, we will take a pause from the slides to do a live demo. Please cross your fingers.
So what I'm going to do is to demonstrate the example of streaming versus atomic methods. So I have opened up a session, and go ahead and log in.
I should point out if you are used to just login username and password, there is a longer form where you can say in addition to the password, the type of log in, in this case a staff, the org unit and the workstation.
So now that we have an off token, I will use it to call one of the curbside methods in a streaming fashion and what you're seeing is it is receiving the data one object at a time.
However, if I tack on atomic to the name of the method, it receives data only once. And that's because the atomic word is stopping the streamed output and just taking into right away. So rather than getting a sequence of objects, we would be getting a single object that is an array.
So now that we have survived the live demo, we'll jump back to the slides. And we will ignore the backup slide.
And we will talk about some of the things that you would do in the call method. What would want to do is call in method in the service. So the top example is a one-shot request. We are using at session, we are creating the connection to the ILS trigger service, and passing a request, in this case, which method, so we look at the parameters that there are no pedestal parameters. So that's a one-shot request.
We can also do multiple requests. So the example below, we are using app session to connect, and that makes a persistent connection to open ILS.circ, and what we are doing here is for each hold in our holds, we are

saying let's go ahead and make an open ILS circ check out net request to actually check out the item associated with that holder. And two, as far as possible, ignore conditions like pickup check out.
For each of them, for their costs, responses, we turn in the entire block of checkout responses to our client. And then before we forget, we will disconnect from open ILS.circ.
Now, sometimes the method you want to call is in the same service you are writing code for. So this is a circ example. Calling self method lookup says please find a get me a reference to the sub – implement side.
With that, I can invoke run, which will passing the same self connection as well as any other parameters specified. Why would you do this?
Sometimes is just a matter of XMPP, and sometimes you need to guarantee that actions take place in the same process. With the example of open ILS storage transaction commit, being a good one. Since each open ILS storage pop -- if you want to do a transaction on an open ILS storage, you have to make sure that the process that issues the transaction to begin -- otherwise, the database will be very confused.
Another aspect is events. An event is just a way of passing data about an awareness of a condition. Any bad structure of the debt -- that the Evergreen clients recognize. And these are defined in ILS.XML. If you run into something where you cannot proceed further, you can go ahead and return an open ILS event object, either directly or through Cstore editor and there will be information about the exception that we will hopefully know how to deal with.
I mentioned the app UtilsAt details is a package that has a function like is true pickup rings together all of the diverse ways that you can say true, including T and F. The quickest way to do this is the simple -- method, which you pass the name of the service, any parameters, and then it takes care of making the request, gathering the responses, and sending it back.
So that brings me to Cstore editor. Cstore editor is open ILS utils, Cstore editor. This helps you not to have to put together manual Cstore requests. You specify education, indicate if you want to run everything in a single transaction. Has utility methods like check off, to verify whether your request is valid, but what makes it Cstore is it comes with a lot of methods from the IDL with transactions. In this case, such action curbside table, based on parameters, and get results.
Field matter, takes the IDL and it makes it possible for you to say -- SQL, object sent to the database row, and updated. This case with the Cstore calls. You'll see a lot of field matter in curbside as well as helpers in client-side.

So this is -- there was a lot in this presentation. So a very quick recap. We talked about a specific example. We talked briefly about OpenSRF and how Perl services fit into it. And then we went through the process of setting up a new Perl Evergreen service. Writing the code, registering the methods, deploying the methods, calling other methods, and then discussing the utilities for it.
So this takes us to 50 minutes. Thank you very much for your time and attention. I see one question from Tiffany that I can expand upon. In my demo, what did I do to specify the login type and workstation? That because the open air curbside methods for the most part assume that you have a workstation. Because that tells it that curbside is enabled at that library. And if you log in without specifying a workstation, the login session that that creates has no workstation. And most open ILS curbside methods would support because of a lack of a workstation.
So thank you for this program. We are at our time limit. But I would be happy to take questions. So thanks again.
>> Thank you, Galen.
>> Up next, let me take this back and I will show you the next. So up next, we will have if this then that, and talking about action triggers in just a few more minutes.