# IRIS FOR HEALTH

## INTEROPERABILITY

### BUSINESS SERVICE

|  |  |
| --- | --- |
| * Main objective is to bring the data into the production, turn it into a message and send it to another business component for processing. * Receives data from outside of a production. * Creates first message. | Data entering a production via a business service. |
| * Usually uses an inbound adapter. * Does not require the use of an inbound adapter. * TCP inbound adapters are used often. | Data entering the business service via an inbound adapter |
| * Can send messages to a business process or a business operation. * Can accept a response from business processes and business operations. * Can return a response to calling application. | Business services respond to an app and send messages to a process and operation. |
| * Can send to more than one business component. | One business service sending messages to two business processes. |

### BUSINESS PROCESS

|  |  |
| --- | --- |
| * Main objectives are to control message flow within the production and orchestrate communication with other components that retrieve data from external applications and resources. * Does not access anything outside of a production. * Can accept, return and send response messages. | Business Process between a service and two operations. |
| * Accept a request message from a business service or a different business process. * Can coordinate requests and responses to multiple business components. * Can return responses to business processes or business services. | Business process accepting a request from another business process and responding to a business service and operation. |
| * Message router is a specialized business process. * A message router uses a set of rules to determine what to do with the incoming request. * Can send the request message as is to another business component. * Can transform the message before sending it to another business component. * Creating the rules used by a message router does not require writing code. * Message routers do not accept responses from other business components. | Message Router between a business process and business operation. |

### BUSINESS OPERATIONS

|  |  |
| --- | --- |
| * Main objective is to send data out of a production. * Connects to one application. * Typically uses an outbound adapter but not required. | Business operation sending data out of the production via an outbound adapter. |
| * Can query for data from an application and wait for a response. | Business operation waiting for a response from an application. |
| * Accept a request message most often from a business process but can also accept a request from a business service. | Business operations accepting requests from a business process and business service. |

### ADAPTERS

|  |  |
| --- | --- |
| * Handle the specific technology and protocol used to communicate with an application. * Know the low-level details of how to communicate with the application such as ports, message headers and so on. * Isolate the connection-specific logic from the rest of the production. * Some adapters are bi-directional, some are not. | Inbound and outbound adapters connect applications to a business service and operation. |
| * Two types of adapters:   + A business service can use an inbound adapter.   + A business operation can use an outbound adapter. * The product includes an extensive library of prebuilt adapters. * Examples include: TCP, file, SOAP, FTP and email. * It is possible to create a custom adapter. * For example, if an application uses a proprietary communication protocol then a custom adapter would be needed. | Inbound and outbound adapters connect the production to applications. |

### CONNECTIVITY OPTIONS

Recall that a production is able to receive and transmit messages using a variety of input and output types, including File, HTTP, TCP, FTP, and SOAP. In order to connect to InterSystems IRIS, these input and output types require corresponding adapters, each of which relies on certain business component settings (detailed in the table below). Additionally, some general fields should be completed for all adapters:

* **Comment:** description of components or any helpful explanatory text.
* **PoolSize:** the number of concurrent jobs. Usually, we keep the default value of 1 for this field to guarantee that messages are received in the correct order.

Besides the two common fields above, see the table below for some other important settings of the components for each adapter:

|  |  |  |
| --- | --- | --- |
| **Adapter** | **Component** | **Important settings to update** |
| **File** | Business Service | FilePath: input directory for data |
| FileSpec: wildcard filenames to look for |
| MessageSchemaCategory: specify incoming message schema |
| Business Operation | FilePath: output directory for data |
| FileName: name of the output file |
| **FTP** | Business Service | FilePath: directory for FTP Server |
| FileSpec: wildcard filenames to look for |
| MessageSchemaCategory: specify incoming message schema |
| FTPServer: name of the FTP Server |
| FTPPort: the FTP port number |
| Business Operation | FTPServer: name of the FTP Server |
| FTPPort: the FTP port number |
| FilePath: directory for FTP Server |
| FileName: name of the output file |
| **HTTP** | Business Service | Port: local port to listen to HTTP request |
| MessageSchemaCategory: specify incoming message schema |
| Business Operation | HTTPServer: IP address of the server to send request to |
| HTTPPort: TCP port of the server to send request to |
| Credentials: username and password to make the HTTP connection |
| SSLConfig: name of SSL to use |
| **SOAP** | Business Service | Port: local port to listen to SOAP request |
| MessageSchemaCategory: specify incoming message schema |
| Business Operation | WebServiceClientClass: Name of the Client Class that defines the Web Service |
| SOAPCredentials: Username and password to log in |
| Credentials: username and password to make the HTTP connection |
| SSLConfig: name of SSL to use |
| **TCP** | Business Service | port: TCP Port to connect to |
| MessageSchemaCategory: specify incoming message schema |
| Business Operation | IPAddress: IP address to make TCP connection to |
| port: TCP Port to connect to |
| SSLConfig: name of SSL to use |

## API – REST

In this lesson, you learned that REST is a way to share data between applications using a URL to GET, PUT, POST, and DELETE data. Within InterSystems products, data is passed from the client through a web application definition to a dispatch class that extends from [***%CSP.REST***](https://docs.intersystems.com/irislatest/csp/documatic/%25CSP.Documatic.cls?PAGE=CLASS&LIBRARY=%25SYS&CLASSNAME=%25CSP.REST). This class locates the implementation endpoint and handles errors. The endpoint can either query a database or, if further processing is needed, send data through a production. The dispatch class can then send a return message back to the client.

This section introduced the following terms, which will be useful to you throughout this course:

**Shared Resource**: An endpoint on a host server, accessible by clients on the host's network. An endpoint may contain data, a device, or a utility.

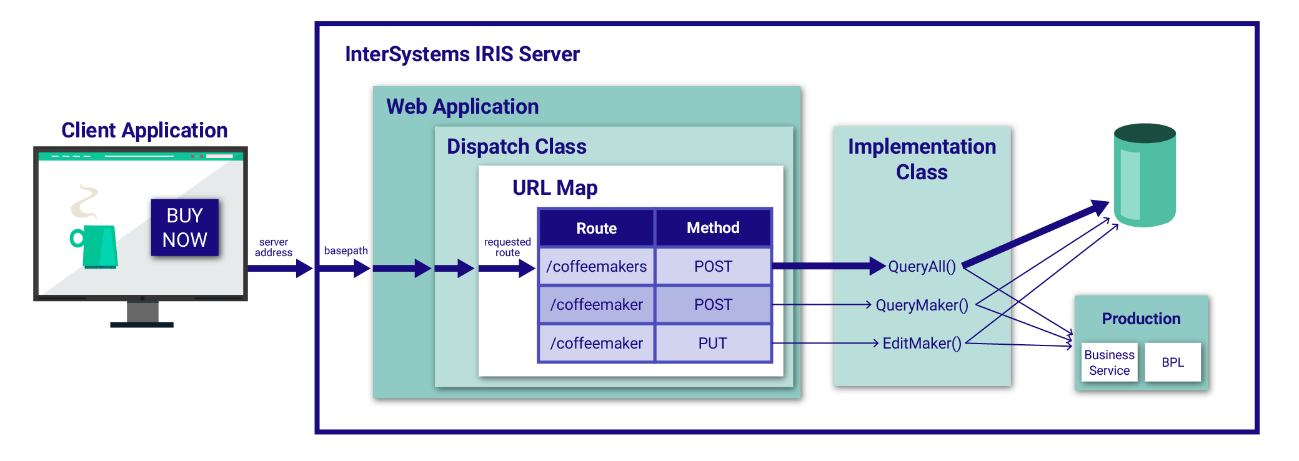
**REST:** An architecture for sharing resources using the HTTP protocol.

**HTTP method**: For each HTTP request, the HTTP method names the desired action to be taken on the requested resource. In this course, the common methods of GET, PUT, POST, and DELETE are used.

**Request Payload:** The request payload of an HTTP request may include meta-information about the request and a request body. Payloads are typically written in JSON.

Refer to the Glossary tab for a full list of terms in this course.

In the remainder of the course, you will create a RESTful server and application with the architecture shown below.



### CREATING A RESTful SERVICE

The spec-first workflow in InterSystems IRIS follows these steps:

Write a RESTful API specification based on desired functionality. Include the names of routes, the HTTP methods, the names of the endpoint functions, and any necessary parameters. The API specification should follow the [**Open API 2.0 standard**](https://swagger.io/specification/).

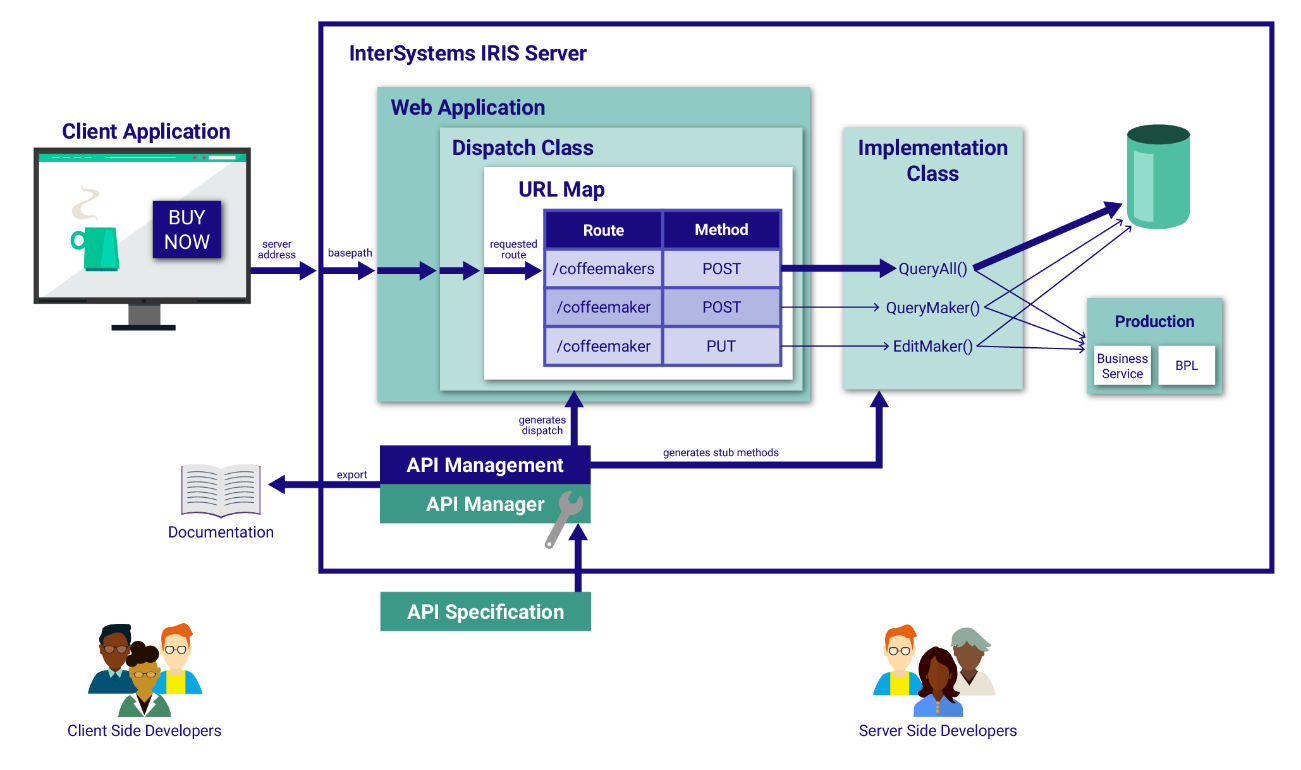
Post the API specification to InterSystems API Management for the instance in order to auto-generate classes on the server. These classes handle route dispatching and endpoint implementations. Multiple options exist in InterSystems IRIS for posting the spec, including an IRIS terminal method, an ObjectScript method, and an HTTP route. This course uses the HTTP route.

Create a web application via Management Portal. The web application receives messages sent to the URL’s basepath, and points to the dispatch class created in the previous step.

Multiple parties now work in parallel:

Client-side developers develop their web apps, utilizing the routes named in the API specification.

Server-side developers implement endpoints to handle API calls from client applications.



## Modifying the Implementation Class

Given a RESTful API specification, the API Management service generates stub methods in the implementation class, impl.cls. The API Management service creates one method for each route, naming each according to the operationId in the spec.  
  
For example, in the code below, the ObjectScript endpoint for PUT requests to /coffeemaker is called EditMaker, and it is used for updating existing coffeemaker information. Meanwhile, a POST request to /coffeemaker could lead to NewMaker instead of EditMaker.     
  
Generated stub methods act as empty endpoints until they are filled out. Stub methods come with the following suggested parts:

1. Executing business logic, including database calls and messages to integrated productions
2. Setting status codes, depending on the outcomes of the logic
3. Setting headers, depending on the needs of the response
4. Returning data of the type listed under produces in the specification

The endpoint below edits the information about a coffeemaker in the database. This ObjectScript code first checks to see if a coffeemaker with the given ID exists. If it exists, ObjectScript opens the object and imports the new data. Then the object is saved, exported as JSON, and sent to the client.

When called by the dispatch class (as in the typical data flow), the first argument, id, is taken from the URL (e.g. /rest/coffeemakerapp/coffeemaker?id=62). The second argument, body, is taken from the payload of the incoming request, in JSON format. The parameters, the body, and their respective types are all declared in the original API specification. They are handled first by the dispatch class, and then passed to the implementation methods, like the one shown below.

A well-written API specification names the possible responses and respective status codes for each route. Below, error codes are set for general server errors and for the specific case when the ID is not in the database. Set status codes using if statements and try/catch clauses.

/// Update existing coffeemaker coffeemaker given ID and data. Returns updated coffeemaker

/// The method arguments hold values for:

/// id, CoffeemakerID

/// body, coffeemaker info

ClassMethod EditMaker(id As %Integer, body As %Stream.Object) As %DynamicObject

{

try {

if '##class(cmAPI.coffeemaker).%ExistsId(id){

do ..%SetStatusCode("400")

return {"errormessage": "No coffeemaker with given ID"}

}

//open coffeemaker with given ID

set cm = ##class(cmAPI.coffeemaker).%OpenId(id)

//refill opened coffeemaker with data from message body

do cm.%JSONImport(body)

do cm.%Save()

do cm.%JSONExportToString(.cmOut)

return cmOut

} catch (ex) {

do ..%SetStatusCode("500")

set error = {"errormessage": "Server error"}

return error

}

}

## Modifying the Implementation Class to Call a Business Service

 Click to addBookmark this page

Now that you have seen an overview of the production, let’s go back to the REST implementation class and modify it to use the production.  
  
First, note that since these functions will be used by a rule within a business service, the class has to extend Ens.Rule.FunctionSet.

/// An API for coffee sales using InterSystems IRIS

/// Business logic class defined by OpenAPI in cmAPI.spec

/// Updated Oct 10, 2019 16:43:47

Class cmAPI.impl Extends (%REST.Impl, Ens.Rule.FunctionSet) [ ProcedureBlock ]

Since all of the endpoints in the implementation class will call a business service, the next step is to create a reusable class method that every endpoint can use. Within this method, we will complete the following steps:

1. Create a new instance of the business service.
2. Populate the request that will be sent to the business service.
3. Call the ProcessInput method from the service object.
4. Set the response message’s header and metadata info.
5. Write a response to the client.

// Helper method to send a message to the interface

ClassMethod CallInterface(URL As %String, Method As %String, Argument1 As %String = "", Payload As %DynamicObject = "") As %DynamicObject

{

//create business service instance

set service = ##class(Ens.Director).CreateBusinessService("cmAPI.RESTCoffeeMakerService",.tService)

//create and fill out request

set request = ##class(cmAPI.GenericCoffeeMakerRequest).%New()

set request.URL = URL

set request.Method = Method

set request.Argument1 = Argument1

set formatter = ##class(%JSON.Formatter).%New()

do formatter.FormatToString(Payload, .tmpPayload)

set request.Payload = tmpPayload

//send request and process input

do tService.ProcessInput(request, .output)

do ..%SetHeader("Content-Type","application/json")

//turn output into response

#Dim output As cmAPI.GenericCoffeeMakerResponse

return output.JSONResponse

}

The arguments of the method are URL, Method, Argument1, and Payload.

Inside the method, the following pieces should be implemented:

1. First create a new instance of cmAPI.RESTCoffeeMakerService.
2. Then create a request object that will be sent to the business service. The request is of type cmAPI.GenericCoffeeMakerRequest.
3. Populate the fields of the request object. Notice that since we are already fulfilling this route via the production, prod should not be passed as an argument here.
4. Call ProcessInput from the service object, passing in the request and retrieving any output.
5. A response header defines the response’s data type and other meta-information.
6. We set the output to be GenericCoffeemakerResponse.
7. Set the status code to OK and return the output.
8. The whole method is wrapped in a try/except clause to catch errors.

Next, each implementation method should be modified to have an optional parameter for triggering the business service. The parameter prod was built into the specification for these routes, but so far, it has not been used. See the example excerpt from QueryAll, below, to see how this can be done.

/// Returns all coffeemakers

/// The method arguments hold values for:

/// prod

ClassMethod QueryAll(prod As %Boolean = "") As %DynamicObject

{

//complete request through business process

if prod = 1 {

try {

return ..CallInterface("/coffeemakers","POST")

} catch (ex) {

do ..%SetStatusCode("500")

return {"errormessage": "Failed business process call"}

}

//complete request directly

} else {

//your previous code here

}

}

The default value of prod is a null string. This is what the class will see if the parameter is not included in the URL. If the value of prod is a null string or if it has been set to 0 via a URL parameter, the else clause will be active and the endpoint will use SQL to query the database.  
  
When the URL includes the parameter *?prod=1*, the endpoint calls the interface and returns the response from the helper method.