DUMPSARENA

MuleSoft Certified Developer - Level 1 (Mule 4) DELTA

Mulesoft MCD-Level1-Delta

Version Demo

Total Demo Questions: 20

Total Premium Questions: 201

Buy Premium PDF

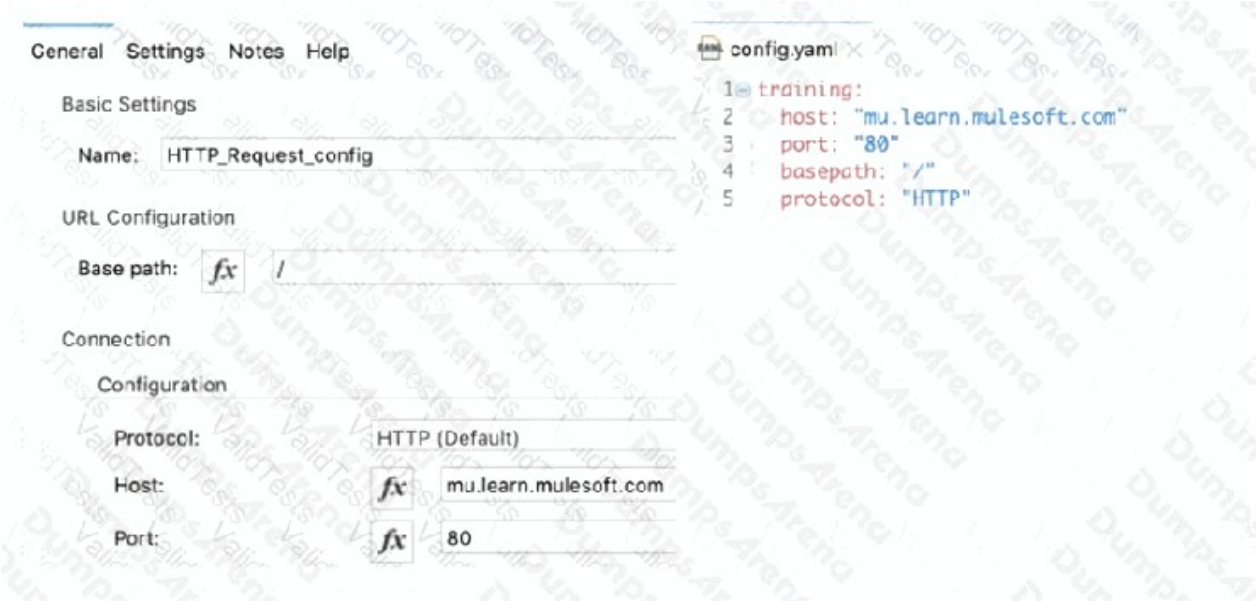
https://dumpsarena.co

sales@dumpsarena.co

sales@dumpsarena.co
dumpsarena.co

QUESTION NO: 1

Refer to the exhibits.



A Mule application has an HTTP Request that is configured with hardcoded values. To change this, the Mule application is configured to use a properties file named config.yaml.

what valid expression can the HTTP Request host value be set to so that it is no longer hardcoded?

A. ${training.host}

B. ${training:host}

C. #[training:host]

D. #[training.host]

ANSWER: A

Explanation:

`${training.host}` is the valid configuration-property placeholder for resolving the HTTP Request host from `config.yaml`. Mule applications can load YAML property files through the Configuration Properties mechanism, and hierarchical YAML keys are accessed with dot notation. Thus, when the YAML structure defines a `training` section containing a `host` value, Mule exposes that value under the key `training.host`. The `${...}` syntax instructs Mule to resolve the value during application configuration, before the HTTP Request operation uses its host setting. This makes the endpoint externalized and allows its host to be changed by editing configuration rather than modifying the flow XML. It also supports environment-specific configuration practices, where deployment environments provide different property values while the application logic remains unchanged. Configuration-property placeholders are valid in connector configuration fields such as the HTTP Request host, port, and protocol. Mule documents property placeholders and the use of YAML configuration files in its configuration-properties guidance: Configuring Properties. The HTTP Request Connector configuration reference also shows that connection settings, including host, can be configured through Mule application properties: HTTP Connector Documentation.

QUESTION NO: 2

Refer to the exhibits.



What is valid text to set the field in the Database connector configuration to the username value specified in the config.yaml file?

A. ${db.username}

B. #[db.username]

C. #[db:username]

D. ${db:username}

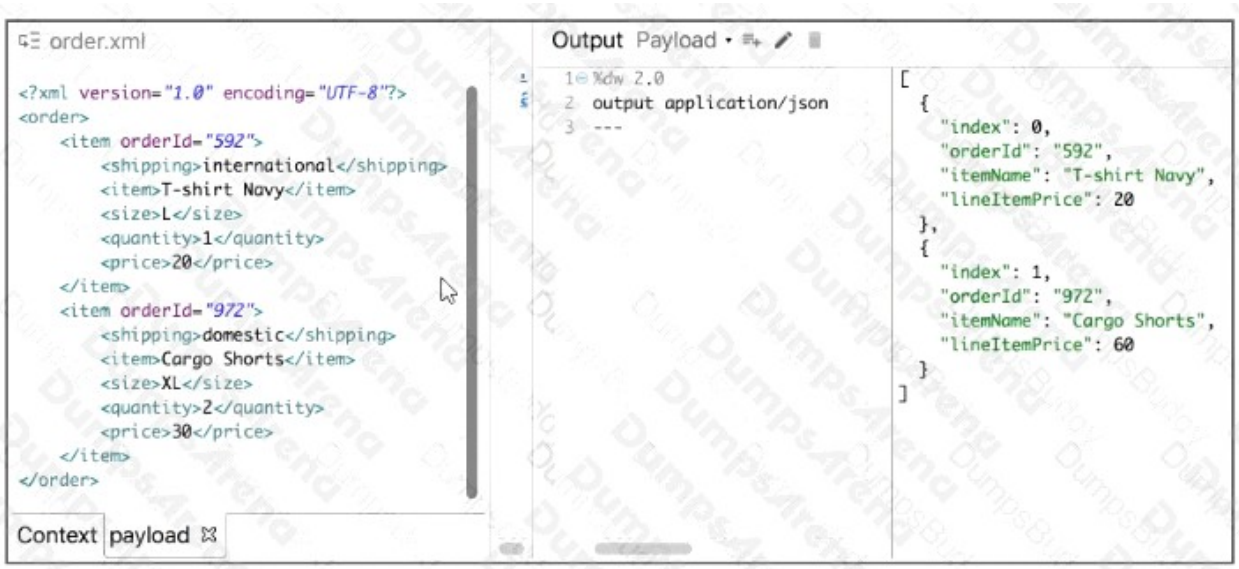
ANSWER: A

Explanation:

`${db.username}` is the valid property-placeholder expression for supplying the Database connector's username from a value loaded from `config.yaml`. Mule applications commonly load YAML configuration through the Configuration Properties module; YAML's nested structure is accessed using dot notation. Thus, when the YAML file defines a `username` key under the `db` section, Mule exposes that value with the property key `db.username`. Placing that key inside the `${...}` placeholder syntax instructs Mule to resolve it during application configuration, before the Database connector establishes its connection. This lets the connector configuration remain environment-independent: the connector refers to the logical property name, while the actual username is maintained outside the XML flow/configuration. MuleSoft documents that configuration properties can be referenced with the `${propertyName}` syntax and that YAML properties use dot-separated paths for nested values. See the Mule configuration properties documentation and the YAML configuration properties documentation.

QUESTION NO: 3

Refer to the exhibit.



What Database expression transforms the input to the output?

A)

```
payload.order.*item map ( (value,index) ->  {
    index: index,
    orderId: value.orderId,
    itemName: value.item,
    lineItemPrice: (value.price as :number) * (value.quantity as :number)
})
```

B)

```
payload.order.*item map ( (value,index) ->  {
    index: index,
    orderId: value.@orderId,
    itemName: value.item,
    lineItemPrice: (value.price as Number) * (value.quantity as Number)
})
```

C)

```
payload.order.*item map ( (value,index) ->  {
    index: index,
    orderId: value.@orderId,
    itemName: value.item,
    lineItemPrice: (value.price as :number) * (value.quantity as :number)
})
```

D)

```
payload.order.*item map( (value,index) ->  {
    index: index,
    orderId: value.orderId,
    itemName: value.item,
    lineItemPrice: (value.price as Number) * (value.quantity as Number)
})
```

A. Option A

B. Option B

C. Option C

D. Option D

ANSWER: B

Explanation:

The selected Database expression is correct because it produces the output structure shown in the exhibit while preserving the required relationship between the incoming values and the values supplied to the database operation. In Mule applications, a Database connector operation evaluates its input expressions using DataWeave. This lets the application derive SQL parameters, query text, or the parameter map/list directly from the current Mule event, including the payload and variables. The expression must therefore return the exact data shape expected by the configured database operation, rather than merely returning a textual representation of the input. When multiple rows or parameter sets are involved, the resulting value must also retain the required collection structure so that the connector can bind values correctly for execution. This follows MuleSoft's model in which connector fields can use DataWeave expressions to transform event data at runtime, and Database connector operations use parameterized values to pass data safely and consistently to SQL statements. See the Database Connector documentation and the DataWeave documentation for the expression and transformation model used by Mule 4.

QUESTION NO: 4

By default, what happens to a file after it is read using an FTP connector Read operation?

A. The file is deleted from the folder

B. The file is moved to a different folder

C. The file stays in the same folder unchanged

D. The file is renamed in the same folder

ANSWER: C

Explanation:

The file stays in the same folder unchanged is correct. The FTP connector's Read operation retrieves the content of the file at the specified remote path and makes that content available to the Mule flow as the operation output. Reading is non-destructive by default: it does not alter the remote file or its location. This behavior is important when a flow needs to access a file without changing the source FTP server, such as when the file must remain available for another process, be retained for audit purposes, or be read again later.

In Mule applications, file lifecycle actions are explicit. If a use case requires a remote FTP file to be removed, relocated, or renamed after its contents are processed, the application must invoke the appropriate FTP operation as a separate, deliberate step. The Read operation itself is focused solely on obtaining file content. MuleSoft's FTP connector documentation describes Read as an operation that reads a file from an FTP server, while the connector reference separately documents operations for actions such as deleting, moving, and renaming remote files. See the FTP Connector documentation and the FTP Connector Reference.

QUESTION NO: 5

Refer to the exhibits.




```
<flow name="logPayload" >
    <scheduler doc:name="Scheduler" >
        <scheduling-strategy >
            <fixed-frequency />
        </scheduling-strategy>
    </scheduler>
    <set-payload doc:name="Set to JSON" value='#[{
          "accounts": {
            "account": {
              "accountName": "ABC Widgets",
              "type": "New Customer",
              "stage": "Qualification"
            }
          }
        }]' />
    <logger level="INFO" doc:name="typeOf(payload)" message='#[typeOf(payload)]'/>
</flow>
```

A JSON payload is set in the Set Payload transformer.

What is logged by the Logger?

A. "String"

B. "Object"

C. "Array"

D. "JSON"

ANSWER: D

Explanation:

"JSON" is logged because the Set Payload operation assigns the payload a JSON media type, and the Logger expression shown in the exhibit accesses the media type's subtype. In Mule 4, a message payload has associated metadata, including its `MediaType`. A JSON payload uses the media type `application/json`; its primary type is `application` and its subtype is `json`. DataWeave represents this subtype as the value `JSON`, which is rendered by the Logger as `"JSON"`. This evaluates payload metadata rather than inspecting the payload's runtime data structure or converting the complete payload to text. Therefore, the logged value identifies the payload format declared by its media type. Mule's Set Payload component can set both the payload content and its MIME type, allowing downstream components and DataWeave expressions to correctly interpret the data. MuleSoft documents payload metadata and DataWeave selector behavior in its Mule Message Structure documentation and describes MIME/media-type handling in the DataWeave Formats documentation.

QUESTION NO: 6

A Utility.dwl file is located in a Mule project at src/main/resources/modules. The Utility.dwl hie defines a function named pascalize that reformats strings to pascal case.

What is the correct DataWeave to call the pascalize function in a Transform Message component?

A)

```
%dw 2.0
output application/json
import modules.Utility
---
pascalize( "max mule" )
```

B)

```
%dw 2.0
output application/json
import modules::Utility
---
pascalize( "max mule" )
```

C)

```
%dw 2.0
output application/json
import modules::Utility
---
Utility::pascalize( "max mule" )
```

D)

```
%dw 2.0
output application/json
import modules.Utility
---
Utility.pascalize( "max mule" )
```

A. Option A

B. Option B

C. Option C

D. Option D

ANSWER: B

Explanation:

A DataWeave module stored at `src/main/resources/modules/Utility.dwl` is imported using its resource-relative module path, `modules::Utility`, without the `.dwl` extension. To make the module's public functions available without a namespace prefix, the Transform Message script uses `import * from modules::Utility`. Once imported, the function can be invoked directly as `pascalize(...)`, passing the string or expression that must be reformatted. This is the standard approach for sharing reusable DataWeave functions across Mule application transforms. The module must export the function, for example with `fun pascalize(value) = ...`; public declarations in a DataWeave module are then available to consuming scripts through the import statement. Mule applications resolve resources in `src/main/resources` at runtime, and the `modules` directory is conventionally used to organize reusable DataWeave libraries. MuleSoft's documentation describes creating and importing custom DataWeave modules at Create a Custom DataWeave Module and documents the `import` directive at DataWeave Import Directive.

QUESTION NO: 7

What execution model is used by For Each and Batch Job scopes?

A. For Each is single-threaded and Batch Job is multi-threaded

B. Both are single-threaded

C. Both are multi-threaded

D. Batch Job is single-threaded and For Each Is multi-threaded

ANSWER: A

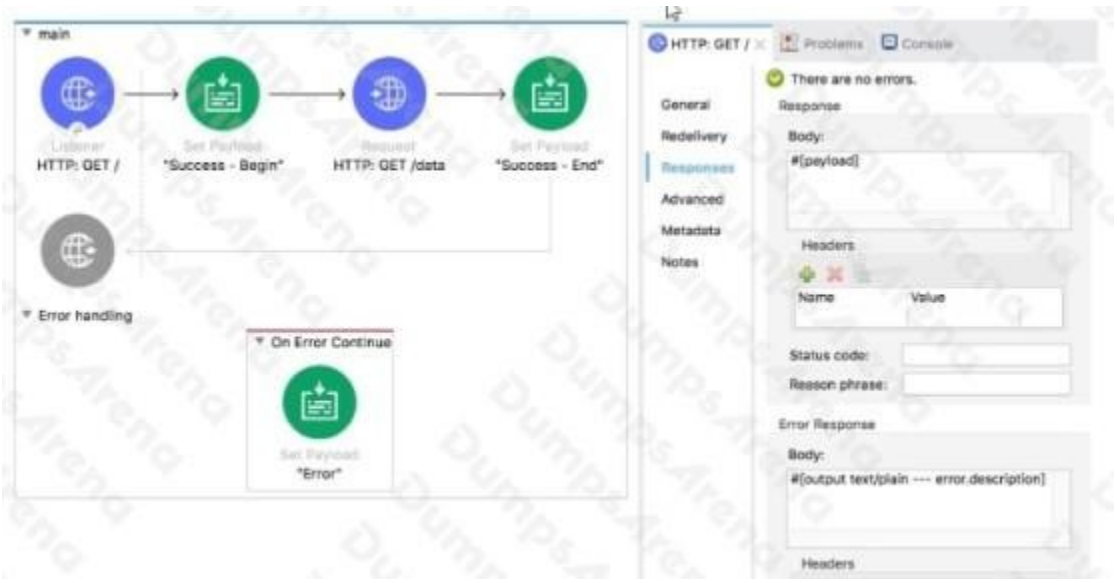
Explanation:

For Each is single-threaded and Batch Job is multi-threaded. A For Each scope iterates through the elements of a collection in sequence, executing the processors in its route once per element before moving to the next one. This sequential execution is useful when the order of items matters or when each iteration must complete before processing continues. The scope also preserves the context required to continue processing the collection as a single flow operation.

A Batch Job is designed for handling large collections asynchronously and efficiently. After records are loaded and dispatched, Mule can process multiple batch records concurrently during the batch step phase. Its concurrency is configurable through settings such as `maxConcurrency`, allowing the application to balance throughput with downstream-system capacity. Batch processing also provides record-level handling and reporting suited to bulk workloads. MuleSoft documents that the For Each scope processes collection elements sequentially, while batch processing distributes records for concurrent execution. See the For Each Scope documentation and the Batch Processing documentation.

QUESTION NO: 8

Refer to the exhibits.




```
<flow name="main">
    <http:listener doc:name="HTTP: GET /"  config-ref="HTTP_Listener_config" path="/" />
    <set-payload value="Success - Begin" doc:name="'Success - Begin'" />
    <http:request method="GET" doc:name="HTTP: GET /data" url="http://mu.learn.mulesoft.com/data"/>
    <set-payload value="Success - End mainFlow" doc:name="'Success - End'" />
    <error-handler >
        <on-error-continue enableNotifications="true" logException="true" doc:name="On Error Continue">
            <set-payload value="Error" doc:name="'Error'"  />
        </on-error-continue>
    </error-handler>
</flow>
```

A web client submits a request to the HTTP Listener and the HTTP Request throws an error.

What payload and status code are returned to the web client?

Refer to the exhibits. A web client submits a request to the HTTP Listener and the HTTP

Request throws an error.

What payload and status code are returned to the web client?

A. Response body: "Error" Default response status code: 200

B. Response body: "Success - Begin* Default response status code: 200

C. Error response body: error, description Default error response status code: 500

D. Response body: "Success - End" Default response status code: 200

ANSWER: A

Explanation:

When the HTTP Request operation raises an error, Mule transfers control to the flow's error handler. The exhibited handling logic uses an *On Error Continue* scope and sets the payload to `"Error"`. An On Error Continue scope handles the failure and allows the flow to complete successfully from the perspective of its owner, rather than propagating the error to the HTTP Listener source. Therefore, the payload produced by that error-handling route becomes the response body returned to the web client.

The HTTP Listener returns a successful response status unless its response configuration explicitly sets another status code. Because the error has been handled rather than propagated, the listener's default HTTP response status remains `200`. Thus, the web client receives the body `"Error"` with status `200`. MuleSoft documents that On Error Continue treats an error as handled and that processing continues successfully for the owning flow, which is the key behavior determining the listener response here. See MuleSoft On-Error scope documentation and HTTP Listener configuration reference.

QUESTION NO: 9

A Mule flow has three Set Variable transformers. What global data structure can be used to access the variables?

A. Mule event attributes

B. Mule event message

C. Mule application properties

D. Mule event

ANSWER: D

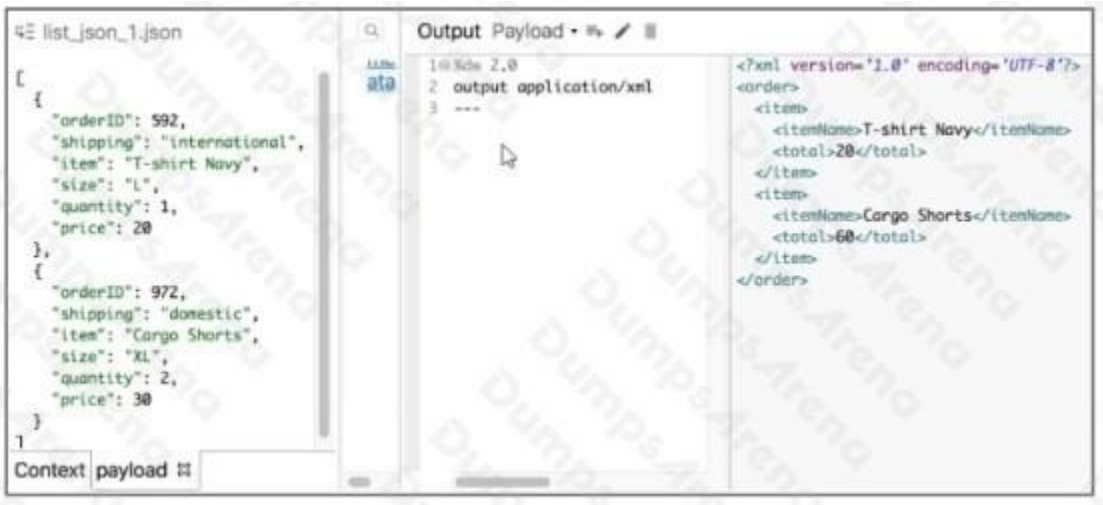
Explanation:

Mule event is correct because variables created by Set Variable components are stored in the variables portion of the Mule event. As a message moves through a flow, each Set Variable operation adds or updates a named value associated with the current event. Subsequent components can access those values using DataWeave expressions such as `vars.customerId`, `vars.status`, or `vars.myVariable`. This provides a shared event-scoped structure for carrying temporary contextual data between processors during that event's flow execution.

The Mule event is the runtime object that carries the message, attributes, and variables through an application flow. Variables are therefore available to later processors handling the same event, including routing, transformation, logging, and connector operations. They are not intended as application-wide persistent state; their scope is tied to the event as it travels through the flow. MuleSoft's documentation describes variables as information stored for use within a flow and accessed through the `vars` object, while the event documentation identifies variables as one of the event's core parts. See Mule event documentation and Transform Data with DataWeave.

QUESTION NO: 10

Refer to the exhibit.



What Database expression transforms the input to the output?

A)

```
{(
    payload map ( (value, index) ->
        order: {
            item: {
                itemName: value.item,
                total: value.price * value.quantity
            }
        }
    )
)}
```

B)

```
order:
    payload map ( (value,index) ->
        item: {
            itemName: value.item,
            total: value.price * value.quantity
        }
    )
```

C)

```
payload map ( (value, index) ->
    order: {
        item: {
            itemName: value.item,
            total: value.price * value.quantity
        }
    }
)
```

D)

```
order:
((
    payload map ( (value,index) ->
        item: {
            itemName: value.item,
            total: value.price * value.quantity
        }
    )
))
```

A. Option A

B. Option B

C. Option C

D. Option D

ANSWER: C

Explanation:

The selected Database expression is correct because it produces the output structure and values represented in the exhibit while using DataWeave's database-oriented transformation syntax. In Mule applications, DataWeave expressions can read the incoming payload, navigate its fields and collections, and construct a new output value by mapping source data into the required target shape. This lets a Database operation receive parameter values in the structure expected by its SQL statement, including values for named parameters or collections used in bulk operations.

The important point is that the expression must preserve the intended relationship between each input record and its corresponding output record. DataWeave evaluates the expression against the current Mule event, so the payload and its nested content are available through the expression context. The resulting value is then supplied to the Database connector as operation input. MuleSoft documents how Database connector operations use DataWeave expressions for query parameters and how DataWeave transforms data structures in Mule flows. See the Database Connector query documentation and the DataWeave language introduction.

QUESTION NO: 11

A web client submits a request to What is the correct DataWeave expression to access the firstName parameter?

A. #[attributes.queryParams.firstName]

B. #[message.queryParams.hrstName]

C. #[message.inboundProperties.'http.query.params'.firstName]

D. #[attributes.'http.query.params'.firstName]

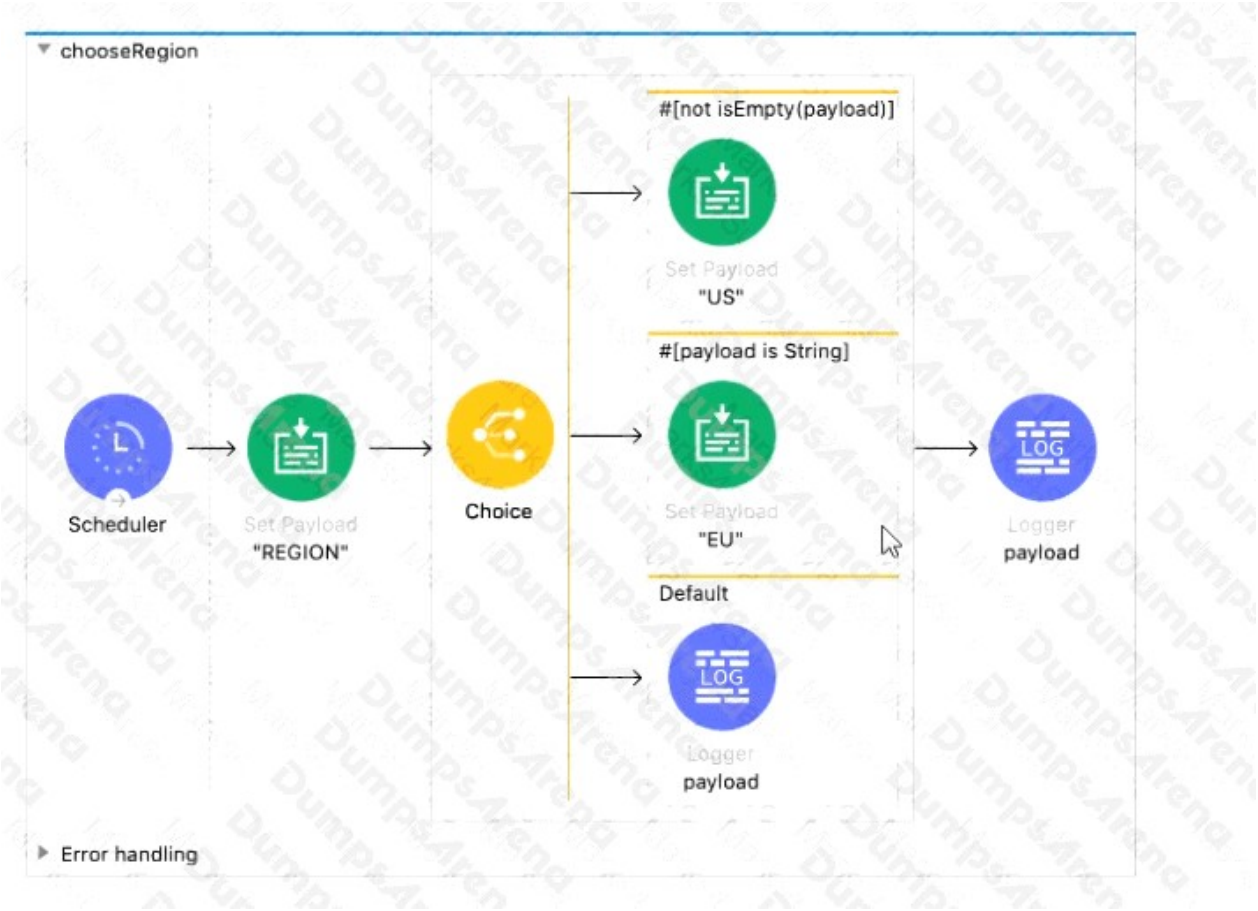
ANSWER: A

Explanation:

`#[attributes.queryParams.firstName]` is the correct DataWeave expression because Mule 4 stores HTTP request metadata separately from the message payload in the event's `attributes`. For an inbound HTTP request handled by an HTTP Listener, the listener provides HTTP request attributes, including query parameters, headers, URI parameters, request path, method, and other request details. The `queryParams` attribute is a map-like structure containing values supplied in the request URL after the question mark. Therefore, for a request such as `/customers?firstName=Ana`, accessing `attributes.queryParams.firstName` retrieves the value associated with the `firstName` query parameter. The expression is enclosed in `#[...]`, which tells Mule to evaluate it as a DataWeave expression when used in a component configuration or expression-enabled field. This reflects Mule 4's event model, where inbound connector-specific metadata is exposed through `attributes`, rather than the Mule 3 message and inbound-properties model. MuleSoft documents HTTP Listener request metadata and its available attributes in the HTTP Listener reference; the general Mule event structure is described in the Mule event documentation.

QUESTION NO: 12

Refer to the exhibits.



```
<flow name="chooseRegion" >
    <scheduler doc:name="Scheduler" >
        <scheduling-strategy >
            <fixed-frequency frequency="5000"/>
        </scheduling-strategy>
    </scheduler>
    <set-payload value='#["REGION"]' doc:name='"REGION"' />
    <choice doc:name="Choice" >
        <when expression='#[not isEmpty(payload)]'>
            <set-payload value='#["US"]' doc:name='"US"'  />
        </when>
        <when expression='#[payload is String]'>
            <set-payload value='#["EU"]' doc:name='"EU"' />
        </when>
        <otherwise>
            <logger level="INFO" doc:name="payload" message="#[payload]"/>
        </otherwise>
    </choice>
    <logger level="INFO" doc:name="payload" message="#[payload]"/>
</flow>
```

A Mule application contains a Choice router. What is logged when the flow completes?

A. EU

B. US

C. " REGION "

D. [" US " , " EU "]

ANSWER: B

Explanation:

`US` is logged. A Choice router evaluates its routing expressions in their configured order and sends the event to the first route whose expression evaluates to `true`. The routing operation does not itself transform the payload or concatenate values; it simply selects the matching message processor path. In the exhibited flow, the value available to the Choice router satisfies the condition for the route that logs `US`. That route is therefore executed, and its Logger outputs `US` when processing reaches it. Mule events retain their current data unless a component explicitly changes it, but the observable result here is determined by the selected Choice route and its logger message. MuleSoft documents that Choice is a content-based router that evaluates conditions and executes the first matching route; if no condition matches, the otherwise route is used when configured. See the Mule Choice Router documentation and the Mule event documentation.

QUESTION NO: 13

Refer to the exhibit.

The main flow is configured with their error handlers. A web client submit a request to the HTTP Listener and the HTTP Request throws an HTTP:NOT_FOUND error.

What response message is returned?''

What response message is returned?

A. APP: API RESOURCE NOT FOUND

B. HTTP: NOT FOUND

C. other error

D. success - main flow

ANSWER: A

Explanation:

APP: API RESOURCE NOT FOUND is returned because the flow's configured error handling handles the `HTTP:NOT_FOUND` error raised by the HTTP Request operation and creates the application-specific response payload. In Mule, an error handler evaluates its error-handling components when an error occurs in its owning flow. An `On Error Propagate` or `On Error Continue` scope can match a particular error type, such as `HTTP:NOT_FOUND`, and use a Set Payload operation to replace the message payload with a custom API error message. Therefore, the client receives the payload produced by that matching error-handler path: `APP: API RESOURCE NOT FOUND`.

Error types are hierarchical and can be explicitly targeted in an error handler, enabling an application to translate connector or protocol errors into its own consistent API response body. The HTTP Listener then returns the resulting Mule message to the calling web client, subject to any configured listener response or error-response settings. MuleSoft documents the behavior and ownership of flow error handlers in its On-Error Scope documentation, and describes Mule error types, including HTTP-related errors, in the Mule Errors documentation.

QUESTION NO: 14

Refer to the exhibits. A web client sends a POST request to the HTTP Listener and the Validation component in the Try scope throws an error.

What response message is returned to the web client?



A. Validation Error

B. "END"

C. "ERROR1"

D. "ERROR2"

E. Validation Error

F. "END"

G. "ERROR1"

H. "ERROR2"

ANSWER: D

Explanation:

"ERROR2" is returned to the web client. The Validation component throws a Mule error while execution is inside the Try scope. The Try scope's configured error handling processes that error and propagates it to the containing flow's error

handler. The flow-level handler then sets the message payload to **"ERROR2"** and handles the error using its configured error strategy. Because the HTTP Listener uses the event payload produced by the flow as the normal response body, the client receives that final payload as the HTTP response message.

Mule error handling is hierarchical: an error raised in a scope can be handled locally, and an error that is propagated remains available to be handled by an enclosing owner, such as the flow. An error handler that continues processing makes the resulting event available for the response rather than returning the original unhandled validation failure. This is why the payload assigned by the applicable enclosing error handler is the client-visible result. MuleSoft documents the propagation and handling behavior of Mule error handling and explains how Try scopes use their own error handlers before an error reaches an owning flow.

QUESTION NO: 15

A flow uses an Object Store Store operation with a key of `customer-123` and a value of a customer object.

What operation retrieves the stored customer object later in the flow?

A. Object Store Retrieve

B. Object Store Remove

C. Object Store Contains

D. Object Store Clear

ANSWER: A

Explanation:

The Object Store Retrieve operation is correct. Object Store stores values under keys, and Retrieve obtains the value associated with a specified key. In this case, specifying `customer-123` retrieves the customer object that was previously stored with that key.

Object Store is commonly used for state that must be available beyond a single message processor, such as idempotency data, tokens, or temporary application state. The key used by Retrieve must match the key used when the value was stored.

See the Object Store Connector documentation.

QUESTION NO: 16

Refer to the exhibit.



What should be changed to fix the 415 error?

A. set the response Content-Type header to text/plain

B. set the response Content-Type header to application/json

C. Set the request Content-Type header to application/json

D. set the request Content-Type header to text/plain

ANSWER: C

Explanation:

Set the request Content-Type header to `application/json`. An HTTP 415 Unsupported Media Type response means that the receiving endpoint cannot process the representation format declared for the incoming request. When the Mule application sends a JSON payload, the outbound HTTP request must explicitly identify that payload format using the `Content-Type: application/json` request header. This enables the target API to select its JSON-capable message reader, validate the request against an API contract that consumes JSON, and deserialize the body correctly. In Mule, headers configured on an HTTP Request operation are sent to the remote service as request headers; consequently, this is the appropriate place to correct the media type of the payload being submitted. The response Content-Type describes the media type returned by the server and does not define the format of the body that the target endpoint is being asked to consume. MuleSoft's HTTP Request documentation describes configuring outbound request headers, while the HTTP specification defines 415 as rejection of an unsupported request-content format. See MuleSoft HTTP Request Reference and MDN: HTTP 415 Unsupported Media Type.

QUESTION NO: 17

What valid RAML retrieves details on a specific by its orderId as a URL parameter?

A)

```
/orders:
    /{orderId}:
        get:
```

B)

```
/orders:
    /orderId:
        get:
```

C)

```
/orders:
    get:
        /{orderId}:
```

D)



A. Option A

B. Option B

C. Option C

D. Option D

ANSWER: B

Explanation:

Option B is correct because a RAML resource for retrieving one order by its identifier must model the identifier as a URI parameter in the resource path, such as `/orders/{orderId}`, and define a `get` method beneath that resource. Curly braces indicate a URI parameter placeholder in RAML. At runtime, a client supplies a concrete value in that location, for example `GET /orders/12345`, allowing the API to identify the single requested order. This path structure expresses the resource hierarchy clearly: `orders` is the collection and `{orderId}` identifies an individual member of that collection.

RAML defines resources using URI paths and nests HTTP methods under those resources. URI parameters can also be documented with a `uriParameters` section, including details such as the parameter type and whether it is required. This produces an API contract that accurately represents a request for a specific order and can be used by API tooling to validate and document the endpoint. See the RAML 1.0 specification's resource and URI-parameter syntax at RAML 1.0 specification and MuleSoft's RAML API design documentation.

QUESTION NO: 18

Refer to the exhibits. The Set Payload transformer in the addItem child flow uses

DataWeave to create an order object.

What is the correct DataWeave code for the Set Payload transformer in the createOrder flow to use the addItem child flow to add a router call with the price of 100 to the order?



A. lookup("addItern", { price: "100", item: "router", itemType: "cable" })

B. addItem({ payload: { price: "100", item: "router", itemType: "cable" > })

C. lookup("addItem", { payload: { price: "100", item: "router", itemType: "cable" } >)

D. addItem({ price: "100", item: "router", itemType: "cable" })

ANSWER: A

Explanation:

`lookup( "addltern", { price: "100", item: "router", itemType: "cable" } )` is the appropriate DataWeave expression because `lookup` invokes another Mule flow synchronously from a DataWeave script. Its first argument identifies the flow to execute, and its second argument becomes the payload supplied to that flow. Therefore, the object containing the price, item name, and item type is available as the payload in the child flow, where the child flow's Set Payload transformation can construct the corresponding order-item structure. The value returned by `lookup` is the payload produced by the invoked flow, so it can be used directly as the result of the Set Payload transformation in the calling flow. This is useful when a reusable child flow encapsulates a transformation and the calling DataWeave expression needs its computed result immediately. MuleSoft documents that `lookup` runs a flow using a specified payload and returns the resulting payload; the invoked flow must not contain a source component. See the DataWeave lookup function reference and the Mule flow invocation documentation.

QUESTION NO: 19

A Mule application contains an ActiveMQ JMS dependency. The Mule application was developed in Anypoint Studio and runs successfully in Anypoint Studio.

The Mule application must now be exported from Anypoint Studio and shared with another developer.

What export options create the smallest JAR file that can be imported into the other developer ' s Anypoint Studio and run successfully?

A. Attach project sourcesDo not include project modules and dependencies

B. Attach project sourcesInclude project modules and dependencies

C. Do not attach project sourcesDo not include project modules and dependencies

D. Do not attach project sourcesInclude project modules and dependencies

ANSWER: A

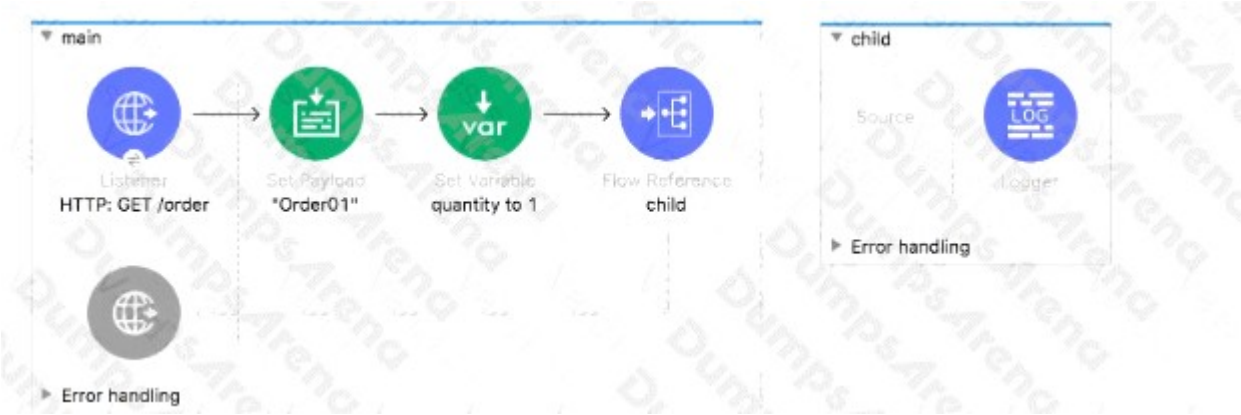
Explanation:

Attach project sources; do not include project modules and dependencies creates the smallest archive suitable for another developer to import into Anypoint Studio. Attaching the project sources packages the Mule project content needed for Studio to recreate the project, including its configuration and Maven project metadata. When the recipient imports the project, Studio uses the Maven configuration to resolve the declared ActiveMQ JMS dependency from the configured Maven repositories, then builds and runs the application in that developer's local Studio environment.

Because the dependency is declared by the project rather than needing to be embedded for standalone deployment, packaging all project modules and dependency JARs is unnecessary for this collaboration scenario and increases the exported archive size. The key distinction is that the archive is intended for import and development in Anypoint Studio, where Maven dependency resolution is available, rather than as a self-contained deployment artifact. Mule applications use Maven to manage connector and library dependencies, while Studio's import workflow restores the project from its source and build descriptors. See MuleSoft's Exporting and Importing Mule Projects documentation and its Maven Reference for the Maven-based dependency model.

QUESTION NO: 20

Refer to the exhibit.



The main flow contains a Flow Reference for the child flow.

What values are accessible in the child flow after a web client submits a request to http://localhost:8Q81/order? color=red?

A. payload

B. quantity var

C. color query param

D. quantity var color query param

ANSWER: D

Explanation:

quantity var color query param is correct because a Flow Reference invokes the referenced child flow using the same Mule event that is present in the calling flow. The Mule event includes the message payload, attributes, and variables. Therefore, a variable named `quantity` that was created in the main flow is available to processors in the child flow without needing to be passed explicitly. The HTTP Listener also creates request attributes containing HTTP request metadata. Those attributes expose query parameters through `attributes.queryParams`, so the submitted `color` query parameter is accessible in the child flow as well, for example with `attributes.queryParams.color`. This behavior lets a referenced flow use context established by the caller while keeping the processing logic modular. MuleSoft documents that the Flow Reference component passes the current event to the referenced flow, and the HTTP Listener request attributes include query parameters. See MuleSoft Flow Reference documentation and HTTP Listener reference.