# Authentication

TODO

# Elements

ESP/Elements is a server-side render page environment supporting arbitrary scripting capability through a Controller/View arrangement.

The in-page scripting language is provided by Apache Velocity, a well-known and industry-standard templating language which includes complex flow control as well as the ability to execute Java code.

Support for controllers is supported through arbitrary Java classes which can be registered to be run as filters before the template is executed. They are free to perform any process they like, and may communicate with the view by setting up data in the request.

A standardized request and response object is also provided by the Elements environment which both the view and the controller can interact with. This works similarly to the request variables which are passed into a PHP page.

# Transaction Processing System

## Overview

ESP/Transaction Processing System (TPS) is an architectural component of ESP which provides a transport-agnostic transaction processing capability to untrusted clients.

## Message Specification

TPS messages are divided into a request message and a response message. For each request message there will be a single response message.

The request message dictates the contents of the transaction. Both the request & response messages are broken down into segments, where each segment contains either message control information (such as transaction begin & end) or operations (such as execute function).

Every type of message segment has two common parameters; the operation, which indicates which type of segment it is, and, a sequence id. The sequence id is used to correlate request segments and response segments.

Segments optionally contain other control fields which direct the specific behavior of the segment. For example, the `CALLS` segment contains fields containing the class & function name to call.

The segment may also contain data fields. The contents of the data fields exist outside of the TPS protocol specification, the meaning of which are defined by individual TPS applications.

## Request

### Header & Footer

The request message always starts with a `TXNB` segment and ends with a `TXNE` segment. The purpose of the `TEXNE` segment is to indicate to the server that it has received the entire message.

The TXNB segment contains the identifier for a partition, which uniquely identifies the execution context for this transaction. Transactions cannot span partitions, so all

DRAFT

segments will be executed in the given partition.

### Variables

The client can allocate up to 64 named variables in the transaction using the `VAR` segment. These variables can be used to hold primitive values (not objects). The number of variables in the table is limited to prevent memory exhaustion by malicious clients.

The most typical use for the variables is to hold the ID of objects which are created during the transaction, since these IDs are not known ahead of time by the client. The IDs can then be referenced by subsequent operations within the transaction.

Values can be moved into a variable from a preceding result object by using the `MOV` segment. Once any segment other than `MOV` has executed, the previous contents of the result object is lost to the transaction processor, and no data from it can be referenced elsewhere in the transaction.

### Function Calling

There are two forms of function call; a static call and a dynamic call, represented by the `CALLS` and the `CALLD` segments.

`CALLS` is the simpler of the two calls, and invokes a static named function on a given class. Since the function is static, is it not required that the client have a particular object instance in mind.

The function call can also contain an arbitrary number of primitive arguments, which are passed in via the data object on the call. Function calling arguments do not support complex types, such as objects or other collections.

As an example, `CALLS` would typically be used to create an object, since at that point the client does not have an ID to an existing object, since it has not been created yet.

`CALLD` works similarly to `CALLS` except it calls a function on a particular object instance. This requires the client to know the ID of the object to call the function on.

The ID could be provided to `CALLD` either directly from the client, if it knows about the ID from some means, or, could be a variable which was previously allocated with `VAR` and loaded with `MOV`.

## Response

### Header & Footer

The response message always starts with a `RESB` segment and ends with a `RESE` segment. The purpose of the `RESE` message is to indicate to the client that it has received the entire response.

Between `RESB` and `RESE`, the response consists of an arbitrary number of other segments which are correlated with the segments in the request.

For each segment in the request there will be at least one segment in the response which indicates the result of the request segment. In some cases, a single request segment can produce multiple response segments.

The request segment is correlated to the response segments through the Sequence ID (SID). The SID can be any value provided by the client, but must be a unique value for each request segment.

DRAFT

The suggested convention is to use incrementing numbers for the SID, for example, 0, 1, 2, ... etc.

### Response Segments

Request segments can produce one of three basic segment types:

The `OK` response segment indicates that the correlating request segment operated properly, but otherwise did not produce any output.

The `ERR` response segment indicates that the correlating request failed, and details as to the failure are provided within the segment. An `ERR` result may or may not end the transaction at that point; if the `ERR` does end the transaction, the next segment will be `RESE`.

CALLS and CALLD request segments which return values will produce, at minimum, an OBJ segment. This contains the root level of an object whose primitive data is contained within the segment attributes.

For example, if the function call returns an `address` object, the data might be `first=John,last=Smith`.

If the object contains collections, they are not contained within the segment. Rather, they are provided as additional response segments.

For each collection, a `STMB` segment will be provided which indicates the collection name relative to the preceding object. After `STMB`, one or more `OBJ` segments will be sent, each object representing one of the objects in the collection. Once all the objects have been transmitted, the `STME` segment will be sent.

It is possible that a child object itself can contain a collection. If this is the case, then a nested `STMB` and `STME` pair will be provided directly after the child object.

## Authentication & Authorization

The TPS message specification provides no facility for authentication; any necessary authentication must happen on the transport. For example, HTTP-based transports can choose to use any of the many available schemes for authenticating a HTTP request.

Likewise, the TPS message contains no native facility for authorization. The TPS server will perform authorization checks for transaction segments based on application-level rules. Applications which fail transaction processing due to authorization failures will return appropriate errors in the transaction response.

## Example Request & Response

### Request

The following example request creates a new application object representing a person, adds two addresses to it, updates a value on the object, and then reads it back.

In this example, the request & response messages are presented in psueud-code. For actual encoding formats, consult the specification for the TPS transport the client & server are communicating with.

```
TXNB {partition=12345,seq=1}

VAR {name=lastid,seq=2}
```

DRAFT

```
CALLS
{class=lc.example.person,func
=create,seq=3}
{first=John,last=Smith}

MOV {from=id,to=lastid}

CALLD
{class=lc.example.person,func
=addr_add,id=$pid,seq=4}
{type=home,addr1=123 Main St}

CALLD
{class=lc.example.person,func
=addr_add,id=$pid,seq=5}
{type=work,addr1=567 Factory
Ln}

CALLD
{class=lc.example.person,func
t=update,id=$pid,seq=6}
{first=Jon}

CALLD
{class=lc.example.person,
func=query,id=$pid,seq=7}

TXNE {seq=8}
```

**Response**

```
RESB {seq=1}

OK {seq=2}

OBJ {seq=3} {id=12345}

OK {seq=4}

OBJ{seq=5}{id=8000}

OK {seq=6}

OBJ{seq=7,class=lc.example.pe
rson} {first=Jon,last=Smith}

STMB {seq=7,name=addresses}

OBJ{seq=7,class=lc.example.ad
dress} {type=home, 123 Main
St}

OBJ{seq=7,class=lc.example.ad
dress} {type=work,addr1=567
Factory Ln}

STME {seq=7}

RESE {seq=8}
```

## Request Segments

### TXNB

Indicates the beginning of a transaction.

- `partition`: The partition to execute the transaction in.
- `seq`: The segment sequence number.

This segment has no data.

### TXNE

Indicates the end of a transaction.

- `seq`: The segment sequence number.

This segment has no data.

### VAR

Allocate a named variable.

- `name`: The name of the variable.
- `seq`: The segment sequence number.

This segment has no data.

### MOV

Set a value into a variable. The variable most previously has been created with VAR. There must have been a last-seen object, and, it must contain the data attribute specified.

- `from`: The name of the data attribute in the last-seen object to read the value from.

DRAFT

- `to`: The name of the variable to write the value to.

This segment has no data.

**CALLS**
Call a static function on a class. This does not require having a reference to an object.

- `class`: The name of the class.
- `func`: The name of the function.
- `seq`: The segment sequence number.

If an argument name is specified with a dollar sign, such as "`$foo`", then the value of the argument specifies a variable name whose value will be substituted on the call.

The variable must previously have been created with VAR and populated with MOV.

**CALLD**
Call a dynamic function on an object. This requires having a reference to the object.

- `class`: The name of the class.
- `func`: The name of the function.
- `id`: The object id.
- `seq`: The segment sequence number.

The data for the segment consists of arguments to the function call.

If an argument name is specified with a dollar sign, such as "`$foo`", then the value of the argument specifies a variable name whose value will be substituted on the call.

The variable must previously have been created with `VAR` and populated with `MOV`.

## Response Segments

**RESB**
Begin a response.

- `seq`: The segment sequence.

This segment has no data.

**RESE**
End a response.

- `seq`: The segment sequence.

This segment has no data.

**OBJ**
This segment contains key/value pairs for primitive types.

- `seq`: The segment sequence.
- `class`: The class for the object.

The data for the object contains name/value pairs representing all of the non-collection object attributes. The values may be primitive types, as supported by the particular transport.

**STMB**
Begin an object stream, representing the contents of an array collection belonging to the preceding object. Object streams may be nested.

- `seq`: The segment sequence.
- `collection`: The name of the collection this stream represents, relative to the last seen object (the last OBJ segment).

This segment has no data.

**STME**
End the current object stream which was previously started with `STMB`.

- `seq`: The segment sequence.

DRAFT

This segment has no data.

## TPS Transports

### Design Considerations

TPS Messages are designed to be transport agnostic. That is to say, they can be sent via any medium; HTTP, message queues such as Kafka and ActiveMQ, plain-text files, even email. As a result of this, the message format should be entirely self-contained.

A TPS message fundamentally consists of an operator (`MOV`, `CALLS`, etc), a map of attributes, and aa map of user-data. Consequently, a TPS message can be naturally represented in any system which supports a string and two maps of primitives.

The entire message specification is designed to support untrusted clients making streaming requests. Many of the message formation characteristics are a direct result of these requirements.

- Object attributes can only contain primitives. If they were allowed to contain collections, they could become arbitrarily large, which would impact the memory required to deserialize them.

  By splitting collection objects into multiple distinct segments, they can be processed one-at-a-time by the client & server, using minimal RAM.
- No conditional cases, loops, etc. are provided. The message format is not a programming language and is not intended to be used as such. Consequently, untrusted clients will not be able to construct small messages which produce an asymmetrically large expense on the server.
- Variables are supported, but the server limits variable to containing primitives, and, the message specification limits the client to 64 variables.

### Austere Environments

For environments which are austere, or lack complex types, it is possible to combine everything into a single map. To do this, you can adopt an approach of leaving the data attribute names unchanged, prefixing all the TPS attributes names with _, and underscoring the operator twice, such as__.

For example:

```
{
__: "TXB"
_partition: 12345
}
{
__: "CALLD"
_class: "lc.example.widget"
_func: "do_thing"
first: "John"
last: "Smith"
}
{
__: "TXE"
}
```

Or, in a JSON minified form:

DRAFT

```
[{"__":"TXB","_partition":123
45},{"__":"CALLD","_class":"l
c.example.widget","_func":"do
_thing","first":"John","last"
:"Smith"},{"__":"TXE"}]
```

Although not necessarily practical examples, these extreme examples show the flexibility of the TPS message format and illustrate how they can be used in unusual environments while preserving full fidelity of the information.

**OpenAPI**

ESP mk22 provides a TPS transport implemented using the OpenAPI 3.0 specification. This transport is ideal for web-based clients are interested in using TypeScript generated clients.

OpenAPI 3.1.0 was not able to be supported at the time of TPS mk22 development due to lack of support from the relevant open source projects.[i]

Access to the OpenAPI endpoint is made through a single door, which is present at the well-known URL `/tps/mk22/door` although other URLs are possible depending on the individual server configuration.

ESP mk22 supports OpenID Connect (OIDC) authentication flows with foreign Identity Provider (IdP) implementations, including ADFS, Azure AD, Okta, Google, etc.

Authentication to the API with OIDC or other means happens external to the TPS implementation, as TPS does not have authentication or authorization defined within the message format, nor is it necessary.

For more information on the authentication subsystem in ESP, see the Authentication section.

The OpenAPI TPS door receives its parameters in the request body; HTTP parameters in the URL or in a posted form are ignored.

Likewise, the OpenAPI TPS door provides its response a JSON array.

Although many JSON implementations will try to serialize & deserialize an entire array at once, it is possible to read one object at a time using a JSON streaming parser, and to create it using a producer.

This is because the opening [ can be consumed, and then the message segments, each of which are a JSON object, can be read one at a time, comma delimited.

[i] https://github.com/OpenAPITools/openapi-generator/issues/9083