NATS Server Error Codes

Metadata	Value
Date	2021-05-12
Author	@ripienaar
Status	Partially Implemented
Tags	server, client, jetstream

Status

Partially Implemented in #1811 and #2409

The current focus is JetStream APIs, we will as a followup do a refactor and generalization and move onto other areas of the server.

Context

When a developer performs a Consumer Info API request she might get a 404 error, there is no way to know if this is a 404 due to the Stream not existing or the Consumer not existing. The only way is to parse the returned error description like `consumer not found`. Further several other error situations might arise which due to our code design would be surfaced as a 404 when in fact it's more like a 5xx error - I/O errors and such.

If users are parsing our error strings it means our error text form part of the Public API, we can never improve errors, fix spelling errors or translate errors into other languages.

This ADR describes an additional `error_code` that provides deeper context into the underlying cause of the 404.

Design

We will adopt a numbering system for our errors where every error has a unique number within a range that indicates the subsystem it belongs to.

Range	Description
1xxxx	JetStream related errors
2xxxx	MQTT related errors

The JetStream API error will be adjusted like this initially with later work turning this into a more generic error usable in other parts of the NATS Server code base.

```
// ApiError is included in all responses if there was an error.
type ApiError struct {
```

```
    Code        int    `json:"code"`
    ErrCode     int    `json:"err_code,omitempty"`
    Description string `json:"description,omitempty"`
    URL         string `json:"-"`
    Help        string `json:"-"`
}
```

Here the `code` and `error_code` is what we'll consider part of the Public API with `description` being specifically out of scope for SemVer protection and changes to these will not be considered a breaking change.

The `ApiError` type will implement `error` and whenever it will be logged will append the code to the log line, for example:

```
stream not found (10059)
```

The `nats` CLI will have a lookup system like `nats error 10059` that will show details of this error including help, urls and such. It will also assist in listing and searching errors. The same could be surfaced later in documentation and other areas.

Using in code

Raising an error

Here we raise a `stream not found` error without providing any additional context to the user, the constant is `JSStreamNotFoundErr` from which you can guess it takes no printf style interpolations vs one that does which would end in `...ErrF`:

The go doc for this function would also include the content of the error to assist via intellisense in your IDE.

```
err = doThing()
if err != nil {
    return NewJSStreamNotFoundError()
}
```

Some errors require string interpolation of tokens, eg. the `JSStreamRestoreErrF` has the body `restore failed: {err}`, the `NewJSStreamRestoreError(err error)`will use the arguments to in the tokens, the argument name matches the token. Some tokens have specific type meaning for example the token `{err}` will always expect a `error` type and `{seq}` always `uint64` to help a bit with sanity checks at compile time.

Note these errors that have tokens are new instances of the ApiError so normal compare of `err == ApiErrors[x]` will fail, the `IsNatsError()` helper should generally always be used to compare errors.

```
err = doRestore()
if err != nil {
```

```
    return NewJSStreamRestoreError(err)
}
```

If we had to handle an error that may be an `ApiError` or a traditional go error we can use the `Unless` ErrorOption. This example will look at the result from `lookupConsumer()`, if it's an `ApiError` that error will be set else `JSConsumerNotFoundErr` be returned with the error message replaced in the `{err}` token.

Essentially the `lookupConsumer()` would return a `JSStreamNotFoundErr` if the stream does not exist else a `JSConsumerNotFoundErr` or go error on I/O failure for example.

```
var resp = JSApiConsumerCreateResponse{ApiResponse: ApiResponse{Type:
JSApiStreamCreateResponseType}}

_, err = lookupConsumer(stream, consumer)
if err != nil {
    resp.Error = NewJSConsumerNotFoundError(err, Unless(err))
}
```

Testing Errors

Should you need to determine if a error is of a specific kind (error code) this can be done using the `IsNatsErr()` function:

```
err = doThing()
if IsNatsErr(err, JSStreamNotFoundErr, JSConsumerNotFoundErr) {
    // create the stream and consumer
} else if err !=nil{
    // other critical failure
}
```

Maintaining the errors

The file `server/errors.json` holds the data used to generate the error constants, lists etc. This is JSON versions of `server.ErrorsData`.

```
[
  {
    "constant": "JSClusterPeerNotMemberErr",
    "code": 400,
    "error_code": 10040,
    "description": "peer not a member"
  },
  {
    "constant": "JSNotEnabledErr",
    "code": 503,
    "error_code": 10039,
```

```
    "description": "JetStream not enabled for account",
    "help": "This error indicates that JetStream is not enabled either at a global
level or at global and account level",
    "url": "https://docs.nats.io/jetstream"
  }
]
```

The `nats` CLI allow you to edit, add and view these files using the `nats errors` command, use the `--errors` flag to view your local file during development.

After editing this file run `go generate` in the top of the `nats-server` repo, and it will update the needed files. Check in the result.

When run this will verify that the `error_code` and `constant` is unique in each error