Communicating with the outside world

A look at the BEAM's External Term Format

Sasha Fonseca - Code BEAM Lite Berlin 2019

Intro

- Software Engineer in Lisbon, Portugal
- MSc. in S.E. by University of Lisbon
- Have been using Erlang/Elixir since 2015
- Involved in the community from the beginning
  - Organiser of Lisbon |> Elixir Meetup
- Very interested in learning BEAM internals

Lisbon |> Elixir

Motivation

- Noticed BEAMs communicating to other BEAMs via HTTP w/ JSON body
- Why use a notation from another language?
- BEAM nodes communicated with one another in some other way
- Found out about Erlang External Term Format

Motivation

- Elm enthusiast
- Could I squeeze out some extra performance from my BEAMs?
- JSON sometimes felt unfit for the Functional paradigm
- Elm offers good parsing and binary encoding/decoding libraries
- Never implemented a binary data format. Should be fun!

JSON (Javascript Object Notation)

- Data format widely used across the Web
  - HTTP payloads
  - Data representation
  - Data storage (document stores like MongoDB or CouchDB)
- Mirrors Javascript's (JS) object syntax
- Most languages nowadays have libraries for encoding/decoding

JSON (Javascript Object Notation)

- General advantages
  - Human-readable
  - Very optimised in the browser (e.g., JSON.parse())
  - More compact than some other formats like XML
  - Has some typing (e.g., integers, lists, dictionaries)

JSON (Javascript Object Notation)

- Disadvantages in the BEAM world
  - Human-readability unnecessary when machine-to-machine comm
  - Not native to the BEAM, probably not as optimised as the browser
  - < 100% fit for the Functional paradigm (e.g., lack of tuples)

External Term Format

- Implemented by Bjorn Gustavsson in 1997
- It is a binary data format
  - Formal spec to map terms from one language to a byte sequence
  - This mapping is called serialisation
  - ETF specifies how to serialise Erlang terms
- Evolves with the BEAM and offers some retro-compatibility
- Native to the BEAM and as such fits Erlang/Elixir perfectly

External Term Format

- What are Erlang Terms?
  - “A piece of data of any data type is called a **term**.”
  - 42
  - [1, 2, 3]
  - fun = fn(x) -> x + 1 end #Function<7.126501267/1 in :erl_eval.expr/5>
  - {:ok, {:user, 123, “Thomas”, “Johnson”}}

External Term Format

- How to use it?
  - erlang:term_to_binary/1,2
  - erlang:binary_to_term/1,2
  - Built-in Functions (BIFs)
    - Written in C and integrated in each Erlang release

External Term Format

- erlang:term_to_binary/1,2
  - Serialise any given term
  - Accept compression level and minor version
  - Compression is done via **zlib** with levels 0 (no compression), 1, 6 or 9

External Term Format

- erlang:binary_to_term/1,2
  - Deserialise a given binary into an Erlang term
  - Allows passing a ‘safe’ option to protect against
    - Direct dynamic creation of atoms
    - Indirect dynamic creation of atoms (nested structures)
    - External function references

External Term Format

- Use-cases:
  - Serialised terms can be stored in memory, disk or sent by the wire
  - BEAM-to-BEAM communication is easily freed of middleware protocols
  - Persistent or volatile memory (ETS/DETS/mnesia)
  - Build abstractions on top
    - GitHub’s BERT and BERT-RPC (http://bert-rpc.org)

Implementation

- Parser, encoder and decoder written in Elm
  - Functional and statically typed language (think a simpler Haskell)
    - Targets the browser, i.e., transpiles to Javascript
  - Follows the official elm/json package API
    - Migration can be as easy as changing the module name
  - Only uncompressed ETF due to no existing zlib package for Elm
  - URL: https://package.elm-lang.org/packages/sashaafm/eetf/latest/

Implementation

- Before implementing a formal reference is needed
  - http://erlang.org/doc/apps/erts/erl_ext_dist.html
- Implementation references can come in handy
  - Official implementation
    - https://github.com/erlang/otp/blob/master/erts/emulator/beam/external.c
  - Rust implementation
    - https://docs.rs/eetf/0.4.0/eetf/

Implementation

Encoded Term Format

1	1	N
131	Tag	Data

Implementation

Encoded Term Format

1	1	N
131	Tag	Data

Implementation

Encoded Term Format

1	1	N
131	Tag	Data

Implementation

Encoded Term Format

1	1	N
131	Tag	Data

Implementation

```
type Term
  = Integer Int
  | FloatingPoint Float
  | Atom String
  | Pid PidProperties
  | Port PortProperties
  | Reference ReferenceProperties
  | NewReference NewReferenceProperties
  | Tuple Elements
  | Map Pairs
  | Nil
  | ProperList Elements
  | ImproperList Elements Tail
  | Binary Bytes.Bytes
  | Export Mfa
```

Implementation

```
type Term
    = Integer Int
    | FloatingPoint Float
    | Atom String
    | Pid PidProperties
    | Port PortProperties
    | Reference ReferenceProperties
    | NewReference NewReferenceProperties
    | Tuple Elements
    | Map Pairs
    | Nil
    | ProperList Elements
    | ImproperList Elements Tail
    | Binary Bytes.Bytes
    | Export Mfa
```

Implementation

```
type Tag
  = SmallIntegerExt Int
  | IntegerExt Int
  | FloatExt Float
  | PortExt PortProperties
  | NewPortExt PortProperties
  | PidExt PidProperties
  | NewPidExt PidProperties
  | SmallTupleExt Elements
  | LargeTupleExt Elements
  | MapExt Pairs
  | NilExt
  | StringExt Elements
  | ListExt Elements
  | BinaryExt Bytes.Bytes
  | SmallBigExt Int
  | LargeBigExt Int
  | ReferenceExt ReferenceProperties
  | NewReferenceExt NewReferenceProperties
  | NewerReferenceExt NewReferenceProperties
  | FunExt Creation  Module Index Uniq FreeVars
  | NewFunExt NewFunProperties
  | ExportExt Mfa
  | BitBinaryExt Bytes.Bytes
  | NewFloatExt Float
  | AtomUtf8Ext String
  | SmallAtomUtf8Ext String
  | AtomExt String
  | SmallAtomExt String
```

Implementation

```
type Tag
  = SmallIntegerExt Int
  | IntegerExt Int
  | FloatExt Float
  | PortExt PortProperties
  | NewPortExt PortProperties
  | PidExt PidProperties
  | NewPidExt PidProperties
  | SmallTupleExt Elements
  | LargeTupleExt Elements
  | MapExt Pairs
  | NilExt
  | StringExt Elements
  | ListExt Elements
  | BinaryExt Bytes.Bytes
  | SmallBigExt Int
  | LargeBigExt Int
  | ReferenceExt ReferenceProperties
  | NewReferenceExt NewReferenceProperties
  | NewerReferenceExt NewReferenceProperties
  | FunExt Creation  Module Index Uniq FreeVars
  | NewFunExt NewFunProperties
  | ExportExt Mfa
  | BitBinaryExt Bytes.Bytes
  | NewFloatExt Float
  | AtomUtf8Ext String
  | SmallAtomUtf8Ext String
  | AtomExt String
  | SmallAtomExt String
```

Implementation

```
type Tag
  = SmallIntegerExt Int
  | IntegerExt Int
  | FloatExt Float
  | PortExt PortProperties
  | NewPortExt PortProperties
  | PidExt PidProperties
  | NewPidExt PidProperties
  | SmallTupleExt Elements
  | LargeTupleExt Elements
  | MapExt Pairs
  | NilExt
  | StringExt Elements
  | ListExt Elements
  | BinaryExt Bytes.Bytes
  | SmallBigExt Int
  | LargeBigExt Int
  | ReferenceExt ReferenceProperties
  | NewReferenceExt NewReferenceProperties
  | NewerReferenceExt NewReferenceProperties
  | FunExt Creation  Module Index Uniq FreeVars
  | NewFunExt NewFunProperties
  | ExportExt Mfa
  | BitBinaryExt Bytes.Bytes
  | NewFloatExt Float
  | AtomUtf8Ext String
  | SmallAtomUtf8Ext String
  | AtomExt String
  | SmallAtomExt String
```

Implementation

SmallIntegerExt

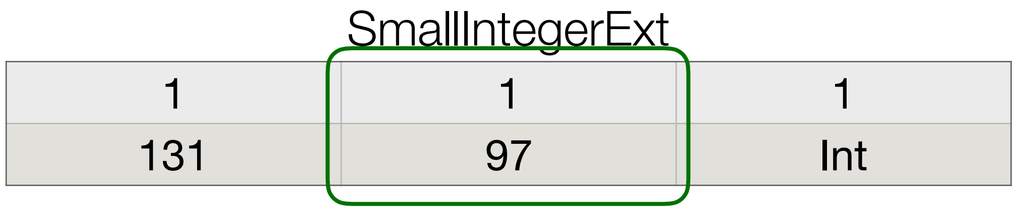
1	1	1
131	97	Int

Implementation

SmallIntegerExt

1	1	1
131	97	Int

Implementation



Implementation

SmallIntegerExt

1	1	1
131	97	Int

Implementation

```
decode : Decoder Term
decode = version |> andThen (\_ -> tag) |> andThen extractTerm

version : Decoder Version
version = map Version unsignedInt8

tag : Decoder Tag
tag = unsignedInt8 |> andThen pickTag

pickTag : Int -> Decoder Tag
pickTag tag_ =
  case tag_ of
    ...
    97 -> map SmallIntegerExt smallIntegerExt
    ...

smallIntegerExt : Decoder Int
smallIntegerExt = unsignedInt8

extractTerm : Tag -> Decoder Term
extractTerm tag_ =
  case tag_ of
    ...
    SmallIntegerExt integer -> succeed (Integer integer)
    ...
```

Implementation

```
decode : Decoder Term
decode = version |> andThen (\_ -> tag) |> andThen extractTerm

version : Decoder Version
version = map Version unsignedInt8

tag : Decoder Tag
tag = unsignedInt8 |> andThen pickTag

pickTag : Int -> Decoder Tag
pickTag tag_ =
  case tag_ of
    ...
    97 -> map SmallIntegerExt smallIntegerExt
    ...

smallIntegerExt : Decoder Int
smallIntegerExt = unsignedInt8

extractTerm : Tag -> Decoder Term
extractTerm tag_ =
  case tag_ of
    ...
    SmallIntegerExt integer -> succeed (Integer integer)
    ...
```

Implementation

```
decode : Decoder Term
decode = version |> andThen (\_ -> tag) |> andThen extractTerm

version : Decoder Version
version = map Version unsignedInt8

tag : Decoder Tag
tag = unsignedInt8 |> andThen pickTag

pickTag : Int -> Decoder Tag
pickTag tag_ =
  case tag_ of
    ...
    97 -> map SmallIntegerExt smallIntegerExt
    ...

smallIntegerExt : Decoder Int
smallIntegerExt = unsignedInt8

extractTerm : Tag -> Decoder Term
extractTerm tag_ =
  case tag_ of
    ...
    SmallIntegerExt integer -> succeed (Integer integer)
    ...
```

Implementation

```
decode : Decoder Term
decode = version |> andThen (\_ -> tag) |> andThen extractTerm

version : Decoder Version
version = map Version unsignedInt8

tag : Decoder Tag
tag = unsignedInt8 |> andThen pickTag

pickTag : Int -> Decoder Tag
pickTag tag_ =
  case tag_ of
    ...
    97 -> map SmallIntegerExt smallIntegerExt
    ...

smallIntegerExt : Decoder Int
smallIntegerExt = unsignedInt8

extractTerm : Tag -> Decoder Term
extractTerm tag_ =
  case tag_ of
    ...
    SmallIntegerExt integer -> succeed (Integer integer)
    ...
```

Implementation

```
decode : Decoder Term
decode = version |> andThen (\_ -> tag) |> andThen extractTerm

version : Decoder Version
version = map Version unsignedInt8

tag : Decoder Tag
tag = unsignedInt8 |> andThen pickTag

pickTag : Int -> Decoder Tag
pickTag tag_ =
  case tag_ of
    ...
    97 -> map SmallIntegerExt smallIntegerExt
    ...

smallIntegerExt : Decoder Int
smallIntegerExt = unsignedInt8

extractTerm : Tag -> Decoder Term
extractTerm tag_ =
  case tag_ of
    ...
    SmallIntegerExt integer -> succeed (Integer integer)
    ...
```

```
type Tag
  = SmallIntegerExt Int
  ...
```

Implementation

```
decode : Decoder Term
decode = version |> andThen (\_ -> tag) |> andThen extractTerm

version : Decoder Version
version = map Version unsignedInt8

tag : Decoder Tag
tag = unsignedInt8 |> andThen pickTag

pickTag : Int -> Decoder Tag
pickTag tag_ =
  case tag_ of
    ...
    97 -> map SmallIntegerExt smallIntegerExt
    ...

smallIntegerExt : Decoder Int
smallIntegerExt = unsignedInt8
```

```
extractTerm : Tag -> Decoder Term
extractTerm tag_ =
  case tag_ of
    ...
    SmallIntegerExt integer -> succeed (Integer integer)
    ...
```

Implementation

```
decode : Decoder Term
decode = version |> andThen (\_ -> tag) |> andThen extractTerm

version : Decoder Version
version = map Version unsignedInt8

tag : Decoder Tag
tag = unsignedInt8 |> andThen pickTag

pickTag : Int -> Decoder Tag
pickTag tag_ =
  case tag_ of
    ...
    97 -> map SmallIntegerExt smallIntegerExt
    ...

smallIntegerExt : Decoder Int
smallIntegerExt = unsignedInt8

extractTerm : Tag -> Decoder Term
extractTerm tag_ =
  case tag_ of
    ...
    SmallIntegerExt integer -> succeed (Integer integer)
    ...
```

Implementation

```
decode : Decoder Term
decode = version |> andThen (\_ -> tag) |> andThen extractTerm

version : Decoder Version
version = map Version unsignedInt8

tag : Decoder Tag
tag = unsignedInt8 |> andThen pickTag

pickTag : Int -> Decoder Tag
pickTag tag_ =
  case tag_ of
    ...
    97 -> map SmallIntegerExt smallIntegerExt
    ...

smallIntegerExt : Decoder Int
smallIntegerExt = unsignedInt8

extractTerm : Tag -> Decoder Term
extractTerm tag_ =
  case tag_ of
    ...
    SmallIntegerExt integer -> succeed (Integer integer)
    ...
```

```
type Term
  = Integer Int
  | FloatingPoint Float
  | Atom String
  | Pid PidProperties
  | Port PortProperties
  | Reference ReferenceProperties
  | NewReference NewReferenceProperties
  | Tuple Elements
  | Map Pairs
  | Nil
  | ProperList Elements
  | ImproperList Elements Tail
  | Binary Bytes.Bytes
  | Export Mfa
```

Implementation

- However, some shortcomings exist
- Elm’s bound to JS integers so BigNum arithmetic is unsupported
  - BigNum terms (smallBigExt/largeBigExt) implemented but will never execute
- Erlang/Elixir lists can have mixed types (e.g., [1, :hello, 5.2])
  - Elm’s static type system restricts lists to elements of the same type
- Maps only support String keys
  - Supporting other types is possible, but the API becomes cumbersome
  - Ruled as acceptable since the common use cases for maps has the keys as String

Benchmarks

- Three different approaches
  - Server-side using Elixir's benchee benchmarking tool
  - Client-side using Elm's elm-explorations/benchmark
  - Load testing using Tsung
- Three different scenarios
  - JSON
  - ETF compression level 0 (no compression)
  - ETF compression level 1
- URL: https://github.com/sashaafm/eetf_benchmarks

Benchmarks

- Server-side
  - Encoding/decoding with ETF 0 , ETF 1, Jason and Jiffy
  - Payload of 10.000 elements representing data about people
  - Only performance (benchee cannot measure BIF/NIF memory usage)

Benchmarks

Decoding (payload size 6.76 MB)

API	IPS	Avg	Dev	Median	99p
ETF 0	41.69	23.99 ms	±11.37%	23.90 ms	30.17 ms
ETF 1	29.49	33.91 ms	±7.81%	33.71 ms	40.01 ms
Jiffy	18.50	54.07 ms	±5.93%	54.12 ms	64.52 ms
Jason	6.61	151.29 ms	±3.34%	151.85 ms	164.40 ms

Benchmarks

Encoding (payload size 6.76 MB)

API	IPS	Avg	Dev	Median	99p
ETF 0	122.20	8.18 ms	±7.80%	8.12 ms	9.94 ms
Jiffy	24.06	41.57 ms	±3.11%	41.52 ms	45.75 ms
ETF 1	15.79	63.34 ms	±4.98%	63.36 ms	73.53 ms
Jason	5.43	184.21 ms	±4.60%	183.69 ms	207.42 ms

Benchmarks

Encoded size (original payload size 616 B)

API	ETF 0	ETF 1	JSON
ETF 0	325 B	< 20 %	> 11.37%
ETF 1	> 20%	260 B	< 10%
JSON	< 11.7%	< 20%	287 B

Benchmarks

- Take-aways
  - Encoding
    - ETF 0 ~5x faster than Jiffy and ~22x than Jason
    - ETF 1 incurs into an ~87x slow down compared to ETF 0
    - ETF 1 can still be ~66x faster than Jason

Benchmarks

- Take-aways
  - Encoded size
    - ETF 0 payload 20% smaller than ETF 1 but ~12% larger than JSON
  - Decoding
    - ETF 0 and ETF 1 faster than both Jiffy and Jason

Benchmarks

- Client-side
  - elm/json vs. sashaafm/eetf
  - Encoding/decoding performance
  - Payload representing data about a person

Benchmarks

Encoding

API	IPS	% Change
elm/json	126,426	-
sashaafm/eetf	18,014	-85.75%

Benchmarks

Decoding

API	IPS	% Change
elm/json	185,475	-
sashaafm/eetf	16,420	-91.15%

Benchmarks

- Take-aways
  - elm/json much much faster than eetf (> 85%) 😭
  - JSON.parse() is extremely optimised in browsers
  - A lot of branching?
  - Cannot go as low-level as the Elm Core Team

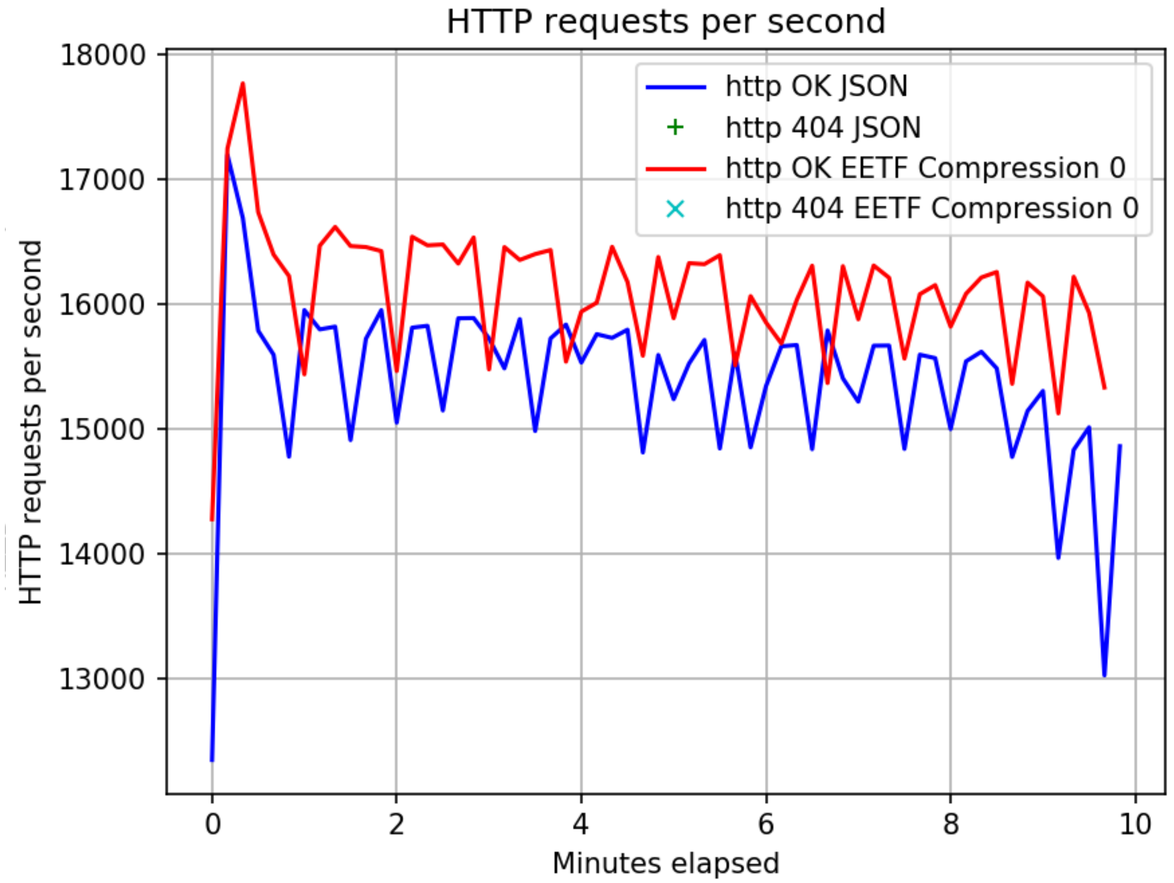
Benchmarks

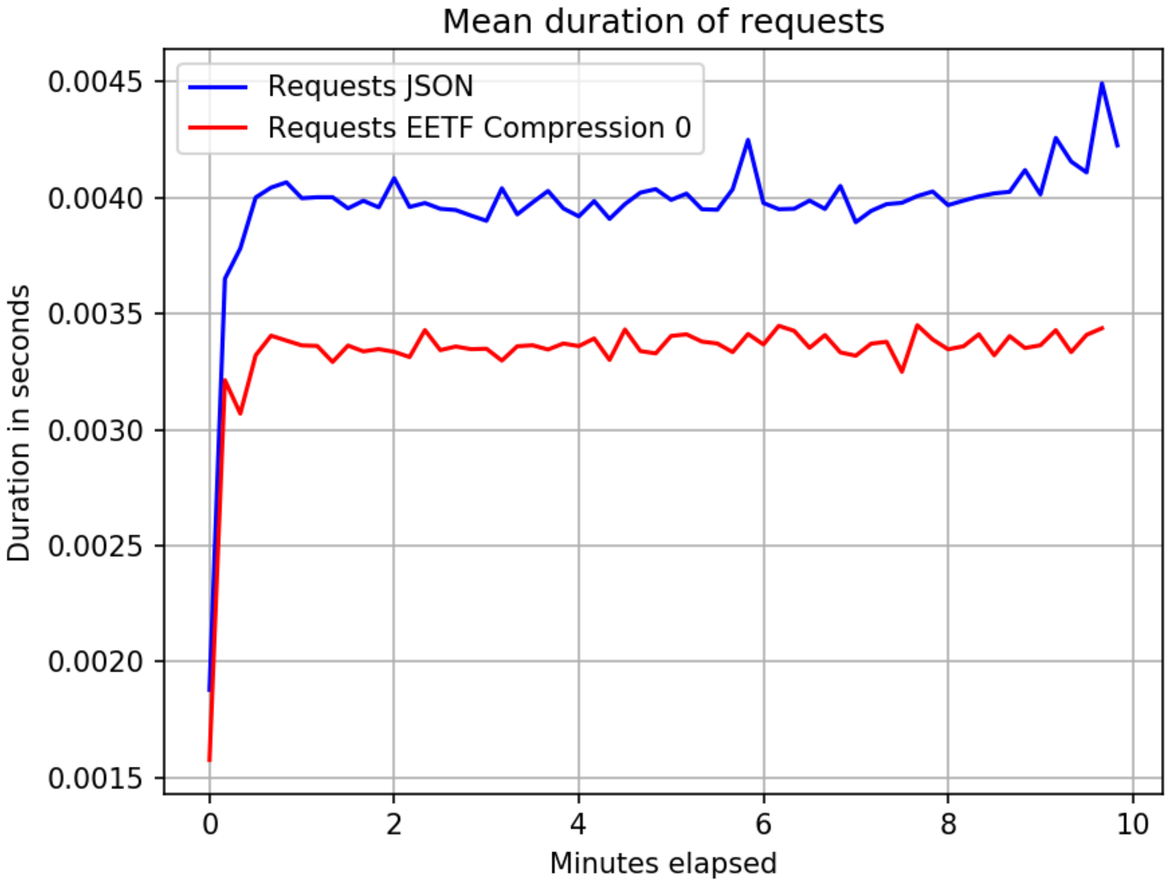
- Take-aways
  - However, this is the client-side
    - It’s best to reduce strain on the server
      - Less resource usage
    - Browsers are usually used by humans
      - Performance should still be negligible to human perception

Benchmarks

- Load testing
  - Using tsung (http://tsung.erlang-projects.org)
  - Simulate 100 concurrent users
    - Perform an HTTP request to a BEAM server to get data of a person
  - Measure ETF 0 vs. JSON and ETF 0 vs. ETF 1
    - Requests/sec
    - Mean duration of requests

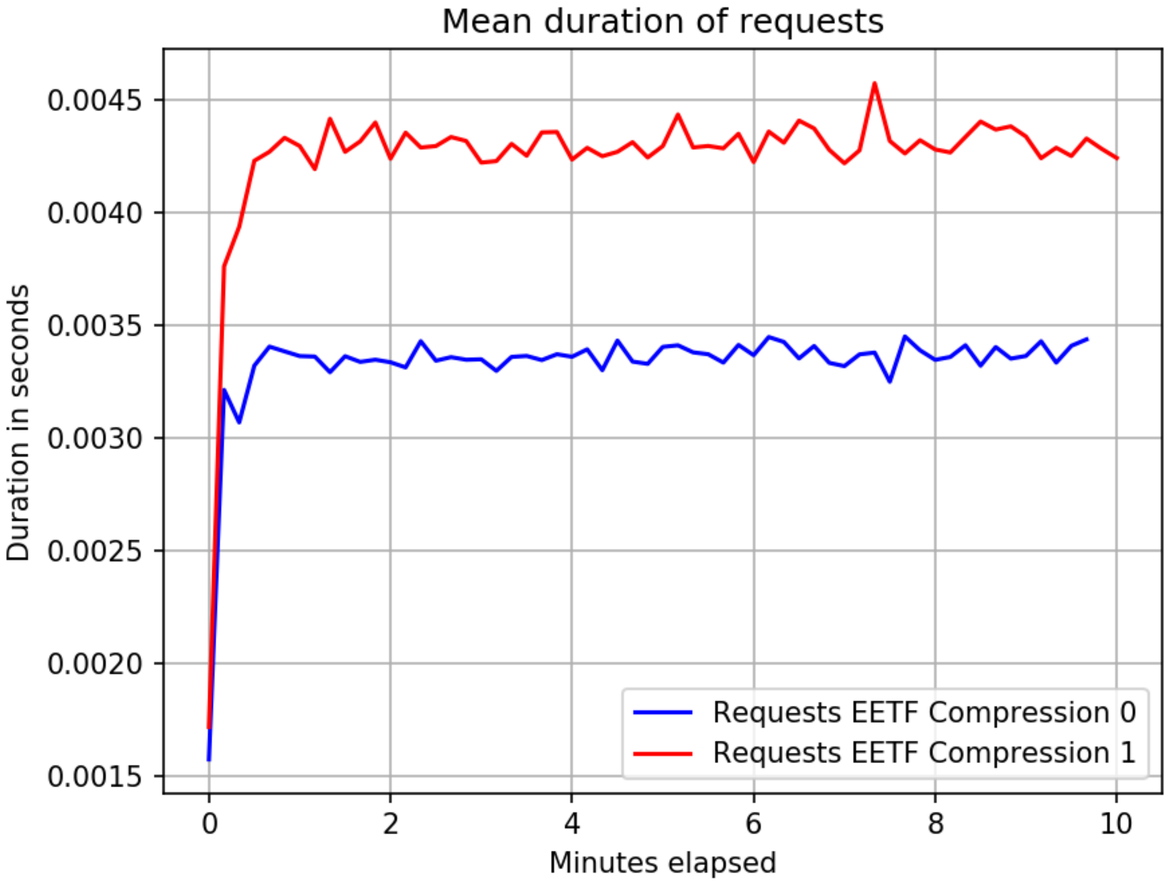
Benchmarks


HTTP requests per second
http OK JSON
http 404 JSON
http OK EETF Compression 0
http 404 EETF Compression 0
HTTP requests per second
Minutes elapsed
18000
17000
16000
15000
14000
13000
0
2
4
6
8
10


Mean duration of requests
Requests JSON
Requests EETF Compression 0
Duration in seconds
Minutes elapsed
0.0045
0.0040
0.0035
0.0030
0.0025
0.0020
0.0015
0
2
4
6
8
10

Benchmarks


HTTP requests per second
http OK EETF Compression 0
http 404 EETF Compression 0
http OK EETF Compression 1
http 404 EETF Compression 1
HTTP requests per second
Minutes elapsed
18000
17000
16000
15000
14000
13000
0
2
4
6
8
10


Mean duration of requests
Requests EETF Compression 0
Requests EETF Compression 1
Duration in seconds
Minutes elapsed
0.0045
0.0040
0.0035
0.0030
0.0025
0.0020
0.0015
0
2
4
6
8
10

Future Work

- Expand API
  - tupleIndex - Fetch tuple elements by index
  - tupleN - Support arity > 3 tuples by decoding them into a list
  - record - Support Erlang Records directly
- Make the internal parser API public?
- Support compressed ETF
  - Have to implement zlib for Elm first

Future Work

- Distribution protocol
  - http://erlang.org/doc/apps/erts/erl_dist_protocol.html
  - Would allow connecting the browser to a BEAM node?
  - Are there use-cases for it?
  - Would there be any benefits?
  - Security concerns?

Closing Words

- ETF provides good insights on some aspects of the BEAM
- Viable option to use instead of JSON unless size is the critical concern
  - ETF 0 and ETF 1 both faster or close to the most popular JSON libs
  - Payload size is mostly similar
  - Supports serialisation of Erlang-specific data like PIDs, Ports or functions
- JSON is ubiquitous and supported everywhere
  - However, this was very easy to implement (~10 hours)
  - Without any experience in this domain

Q & A