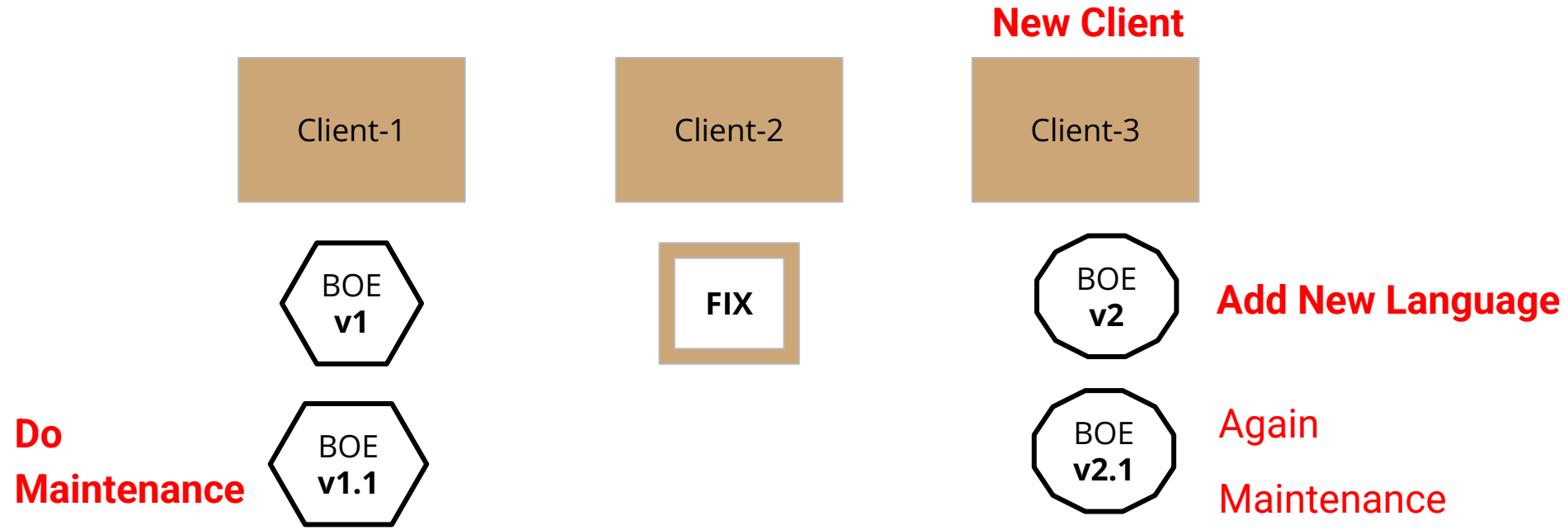
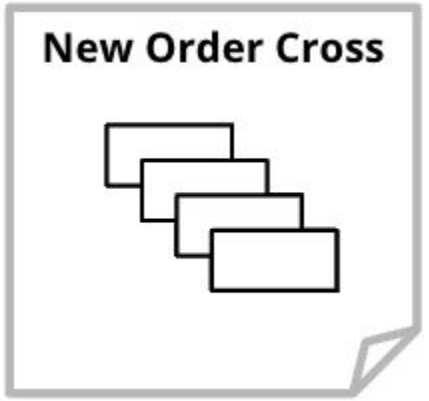
The Problem



Each change needs 1 month developer hours!



Binary Order Entry Protocol

- A **special language** to trade stocks super fast.
- **Customized** buying and selling of securities.

"Buy 10 AAPL for $10" = *0011010010011101*

Language is detailed and complex!

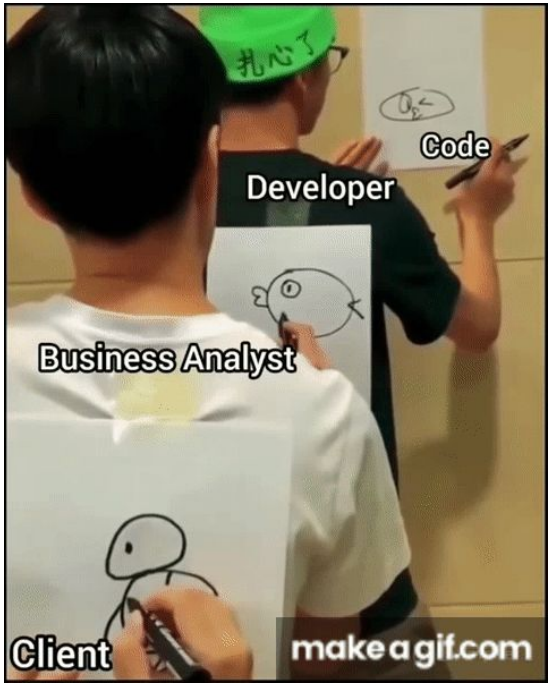
M+RE with L_SS

A POC
To Bridge Business and Development

Current Approach

the Gap between Business and Development

- Business Analysts gather changes
- Developers manually implement them
- **slow, back-and-forth** communication
  - for understanding and testing

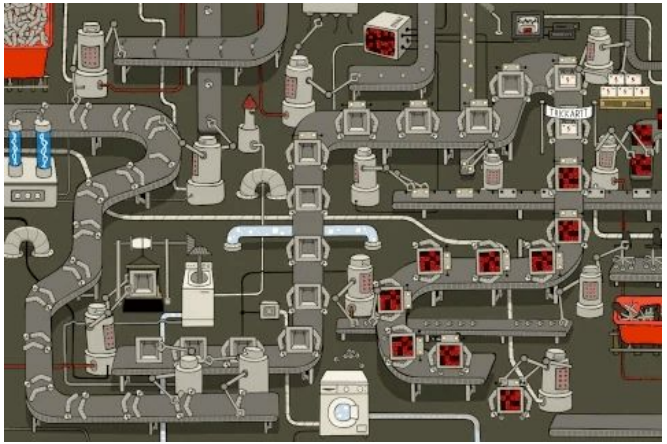


The Solution

MA Genie the code generator

No Code

- Analysts define changes in blueprint
- Code generator generates code



Developers maintain the factory, not language!

Business Perspective

100+ exchanges, 27+ protocols, 5+ versions

1 Month to 1 Week, Time is Money!

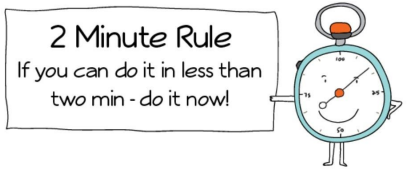
The Learnings

Compiler Optimization

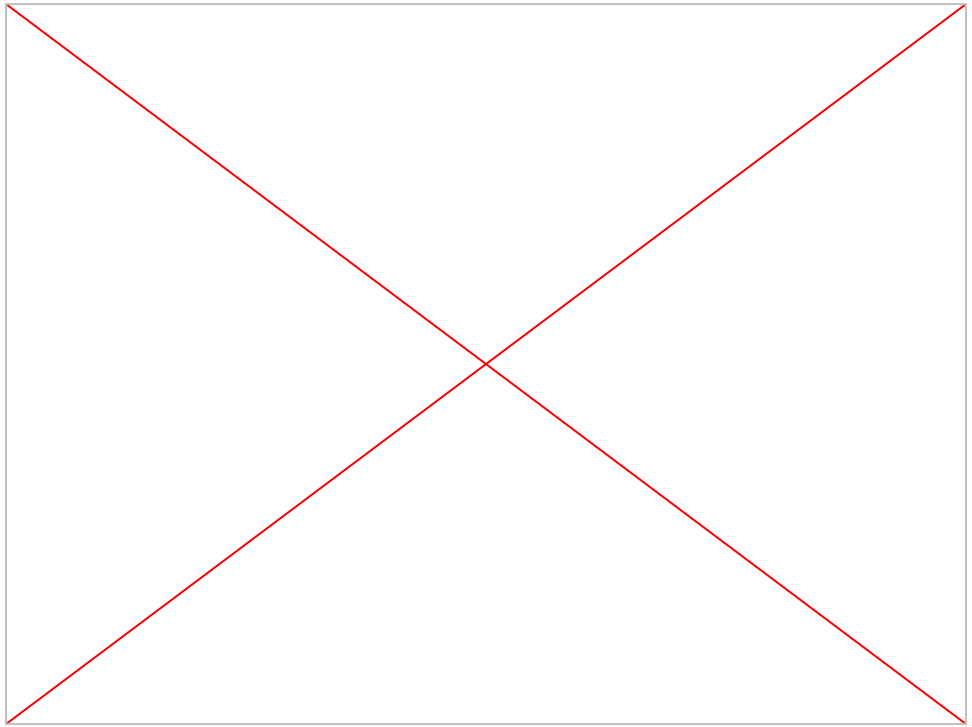
Stakeholder Management

The Learnings Beyond

Technical and Non-Technical



The Demo



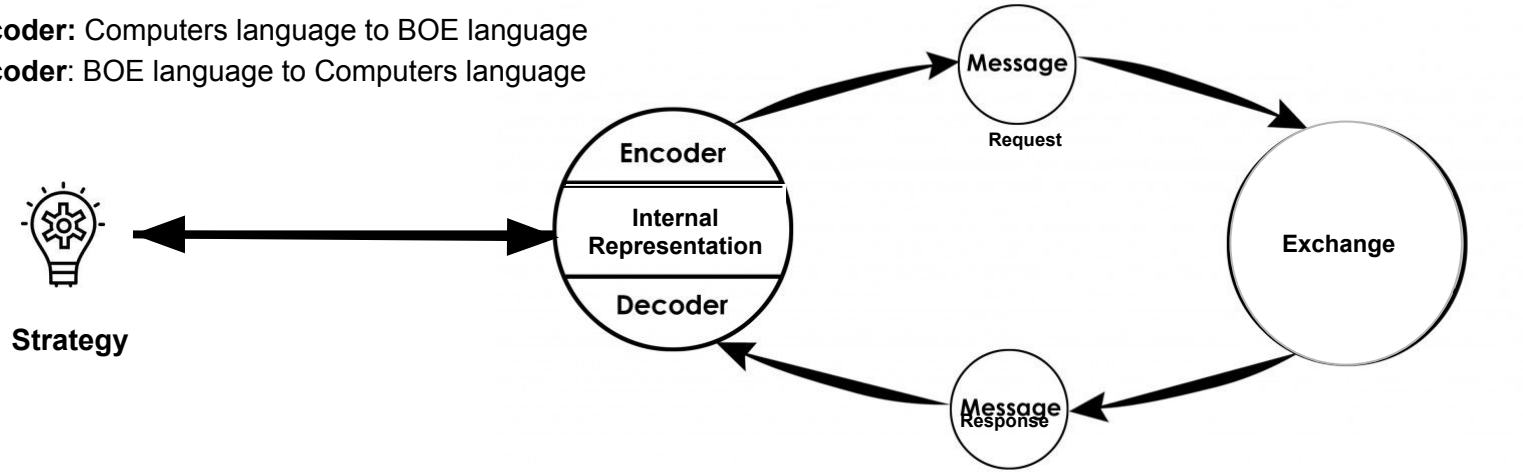
Thank You!

Overview

Strategy: Analyze market and decide action.

Translators:

- **Encoder:** Computers language to BOE language
- **Decoder**: BOE language to Computers language



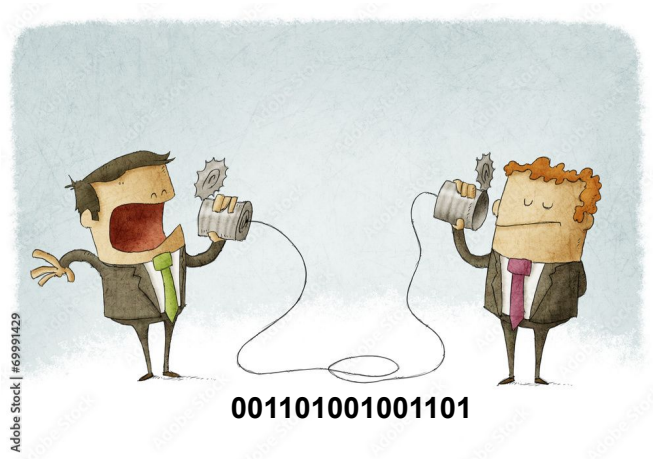
Encoder and Decoder depend on the Language!

Binary Order Entry Protocol

A **special language** to trade stocks super fast!

Ordering involves sending and receiving messages.

Instead of BOE protocol, use client1 client2 client3 using different protocols.



"Buy 10 AAPL for $10"

Each Language has specific rules!

"Buy 10 AAPL for $10" = *001101001001101*

Complexity of Ordering securities through BOE

BOE allows

- **Customized** buying and selling of securities.

Language is very detailed and complex!

Tell them what I’m going to do in the presentation.

Give a grasp of what i’m going to do.

Good Morning Everyone, I’m Subhash, I’m a C++ developer, introduction.

Thank you everyone for attending this meeting.

In this presentation, I will be going to

Put past projects and technical details in presentation.

How to achieve

Hey Subhash, tell me about yourself?
Touch up on all things, but don't tell everything.
Don't go into depth first half
What is your career goal?
Just have a structure
What did you do at arista?

-

What did you do during internship?

-

What excites you to work at MA Captial?

- Culture, Growth, and challenges.

Why do you wanna work there?

- It allows me to constantly push and improve myself both technically and non-technically. It starts with rapid prototyping to delivering the optimized and efficient code. I still remember my first conversation with balaji, "Your contribution matters". I

Don't have too much fancy ppt?

the Complexity of Ordering Pizza

Universe of Pizza and Drinks online orders only aliens need food every alternate hour/minute

Rule for Order: Name Address Spice-Option Extra-Cheese-Option Urgency-Option N-Items Items-Descriptions (Urgency-Level)

Rule for Pizza-Description: Pizza Size N-Toppings Topping-Type (Spice-Level) (Extra-Cheese-Type)

Rule for Drink-Description: Drink Drink-Flavour Drink-Size

Ash 305 YES NO YES 4 Pizza Large 2 Cheese Mushroom High Pizza Medium 1 Olives Normal Drink Orange 200 Drink Coke 500 infinity

any update in Rules need update in Robot.

Key Areas

- **Group, Param-Group, Nested Groups**
  - **Binary message sometimes have repeating set of bits, whose bits itself are repeating**
- **Optional fields and bit fields**
  - **Binary message can have flags which determine which indicate whether fields are present**

Solution

"Built for Flexibility, Streamlined for Maintenance, Optimized for Performance"

Motivation

Handwritten code per version

- New version, New code
- (BOEv-1.x, code-1) (BOEv-2.x, code-2) ... (BOEv-n.x, code-n)

Handwritten code with inheritance/C++ features.

- base-code and extending it with inheritance to support BOEv-version.x
- reduced handwritten work and flexibility at the cost of latency.

Code generation

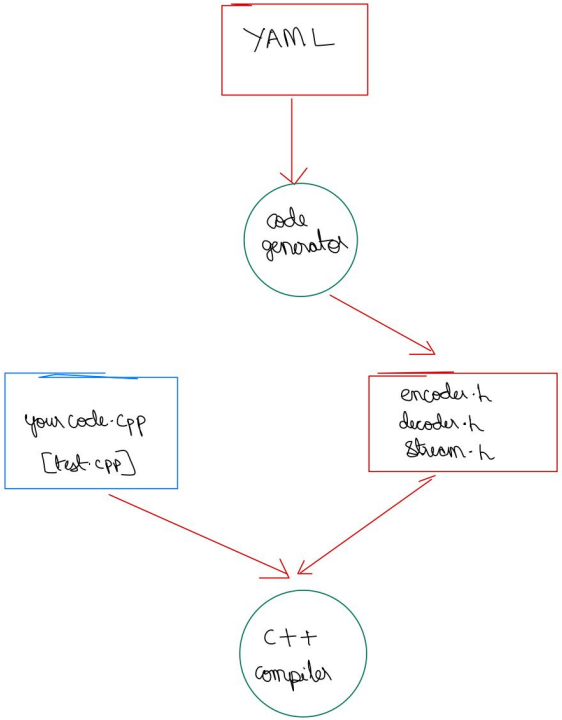
- spec driven code generation of type safe C++ code.
- spec driven run-time message validation.

Approach	Manual per Version	Inheritance-based	Code Generator Approach
Description	Separate code for each BOE version	Base code extended with inheritance for each new version	Code generator handles versions automatically
Flexibility (New protocol-version)	Zero, new code	Moderate, but extensions can be complex	High, only YAML config changes needed
Latency	High	Moderate	High
Maintenance (Error Management or New C++ feature)	Hard, version-specific code edits	Complex, maintenance for each extension	Centralized in YAML & templates
Consistency	Proportional to manual efforts	Partial	Automated

BOE Code generator

A python program

- takes YAML spec file for BOE protocol BOEv-x.y
- generates efficient C++ code for BOEv-x.y
  - BOE_Msgs.h
  - BOE_Encoder.h
  - BOE_Decoder.h
  - BOE_Stream.h
  - Common file: BOE_Handler.h, BOE_Common.h
  - test.cpp for unit testing



Code-generator

Code overview

Generated code

- BOE_Msgs.h
  - Has generated class definitions for all messages and groups along with accessors, constructors, reset member functions according to the spec file.
- BOE_Encoder.h
  - Has generated Encoder class definition, having multiple overloaded public encode functions which takes message class object and converts it into payload using multiple private encode functions.
- BOE_Decoder.h
  - Has generated Decoder class definition, having decode function which takes the input payload, populates corresponding message class object.
    - It allows user to access the decoded message class object.
    - It automatically calls handler on the decoded message object.
- BOE_Stream.h
  - Has generated Ostream operator for all the messages and groups, which pretty print the message class object.

Static code

- BOE_Handler.h
  - The handler class definition has multiple functions, which needs to be completed by the user.
  - By default it calls the custom ostream operator on the message object.
- BOE_Common.h
  - Contains helper classes and functions which could be used across different protocols.
  - Status class is defined which will be populated during the encoder and decoder if any errors happen during their tasks like buffer overflow, corrupted payload, invalid message type, etc.
- test.cpp
  - Contains unit tests

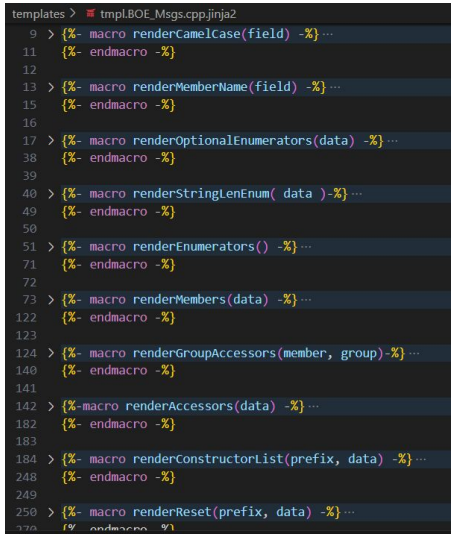
YAML overview

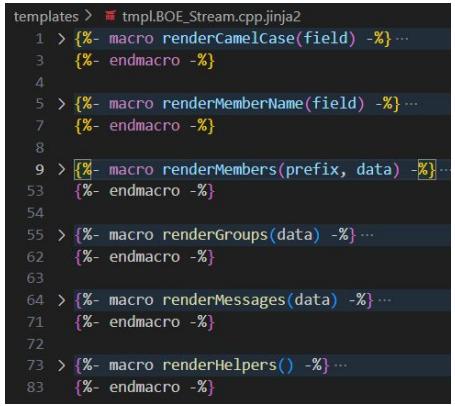
Modular code structure to easily identify changes required.

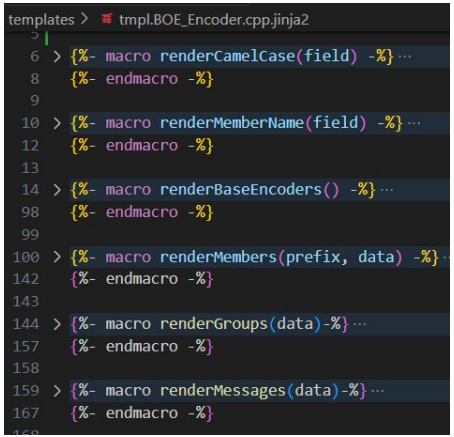
- tmpl.BOE_Msgs.h.jinja2
  - renderStringLenEnum
  - renderOptionalEnumerators
  - renderEnumerators
  - renderMembers
  - renderGroupAccessors
  - renderAccessors
  - renderConstructorList
  - renderEnumeratorOptions
  - renderReset
  - renderGroups
  - renderMessages
- tmpl.BOE_Encoder.h.jinja2
  - renderBaseEncoders
  - renderMembers
  - renderGroups
  - renderMessages
- tmpl.BOE_Decoder.h.jinja2
  - renderHelperMembers
  - renderHelpers
  - renderMembers
  - renderGroups
  - renderMessages
- tmpl.BOE_Stream.h.jinja2
  - renderHelpers
  - renderMembers
  - renderGroups
  - renderMessages

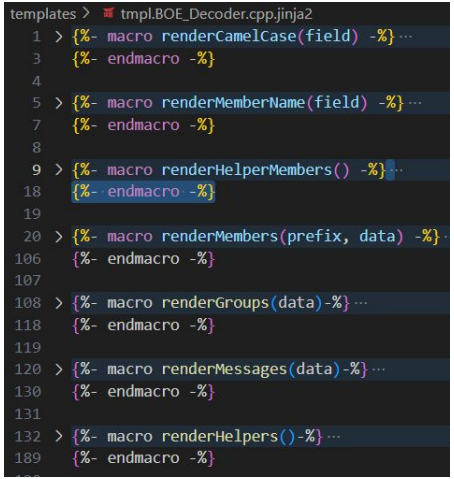
Repetitive and Intuitive structure across all the macros

- to handle different cases such as data types, constraints, and groups.





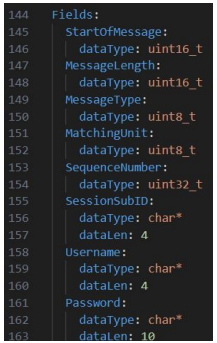




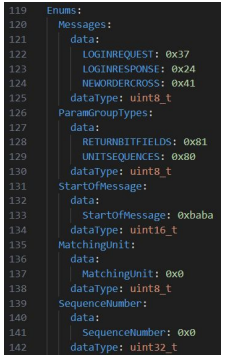
boe.yaml

```
Messages: ···

Groups: ···

Optionals: ···

Enums: ···

Fields: ···
```

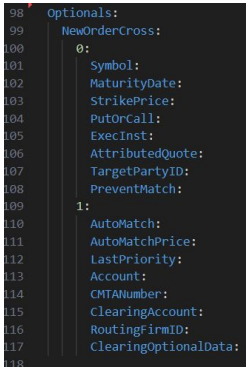
```
RepeatingGroupsOfNewOrderCross: ···

NumberOfUnits: ···

UnitSequences: ···

NumberOfReturnBitfields:
  Fields:
    ReturnBitfield:

ReturnBitfields:
  Fields:
    ParamGroupLength:
    ParamGroupType: ParamGroupTypesEnum::RETURNBITFIELDS
    MessageType:
    NumberOfReturnBitfields:
      ParamGroups:
        NumberOfReturnBitfields:
```



```
Fields:
  StartOfMessage:
    dataType: uint16_t
  MessageLength:
    dataType: uint16_t
  MessageType:
    dataType: uint8_t
  MatchingUnit:
    dataType: uint8_t
  SequenceNumber:
    dataType: uint32_t
  SessionSubID:
    dataType: char*
    dataLen: 4
  Username:
    dataType: char*
    dataLen: 4
  Password:
    dataType: char*
    dataLen: 10
```



```
Enums:
  Messages:
    data:
      LOGINREQUEST: 0x37
      LOGINRESPONSE: 0x24
      NEWORDERCROSS: 0x41
    dataType: uint8_t
  ParamGroupTypes:
    data:
      RETURNBITFIELDS: 0x81
      UNITSEQUENCES: 0x80
    dataType: uint8_t
  StartOfMessage:
    data:
      StartOfMessage: 0xbaba
    dataType: uint16_t
  MatchingUnit:
    data:
      MatchingUnit: 0x0
    dataType: uint8_t
  SequenceNumber:
    data:
      SequenceNumber: 0x0
    dataType: uint32_t
```



```
Optionals:
  NewOrderCross:
    0:
      Symbol:
      MaturityDate:
      StrikePrice:
      PutOrCall:
      ExecInst:
      AttributedQuote:
      TargetPartyID:
      PreventMatch:
    1:
      AutoMatch:
      AutoMatchPrice:
      LastPriority:
      Account:
      CMTANumber:
      ClearingAccount:
      RoutingFirmID:
      ClearingOptionalData:
```

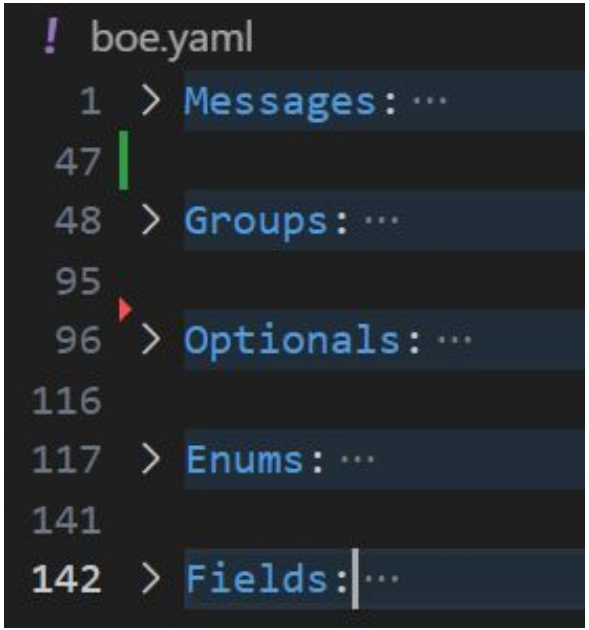
```
Messages:
  LoginRequest:
    Fields:
      StartOfMessage: StartOfMessageEnum::STARTOFMESSAGE
      MessageLength:
      MessageType: MessagesEnum::LOGINREQUEST
      MatchingUnit:  MatchingUnitEnum::MATCHINGUNIT
      SequenceNumber: SequenceNumberEnum::SEQUENCENUMBER
      SessionSubID:
      Username:
      Password:
      NumberOfParamGroups:
        ParamGroups:
          UnitSequences:
          ReturnBitfields:

NewOrderCross: ···
```

Modular Design

- Intuitive to understand, Meaningful to interpret, Easy to edit
  - Similar structure at all levels.
- Detangling specification from presence
  - Same group or Field can be present in multiple messages and groups.
  - Keeping Group and Field specifications away from presence at a common place, allows us to quickly edit configuration, with minimal changes.
- Yet allowing constraints on Fields per Message, and per Group
  - Constant fields
  - Restricted fields
  - Optional fields

Where and How to specify atomic fields?

Where and How to specify common enumerators?



Highlights schema

Where to specify a message or group?

- message specification in Messages section, and group specification in Groups section
- Both message and group specification have the same schema.

How do we specify a Message and Group?

- List all fields present in Fields section, along with CONSTRAINTS.
  - Fields with Enum are considered as **constant fields**.
  - Field with List are considered as **restricted fields**.
  - Fields with Bitmap are considered as **optional fields**.
  - Fields with Param Groups section is considered as **group count fields**.
  - The ParamGroups section lists all the groups which could be present.
    - One group indicates special case **group**.
    - More than one group indicates **param groups**.
    - Group specification having Group indicates **nested group**.
- metaData has information
  - whether it has optional fields
  - which bitmap the optional fields are associated with
  - which message the group with optional field is part of
- Constraints are specified at message-level or group-level
  - A field can different constraints based on where they are present.
- According to BOE repeating groups can only have optional fields, and they have one-to-one mapping to message.

Common schema to write the presence of Param Groups, Groups, Repeating Groups.

Custom names for group-count fields and many other fields.

```
NewOrderCross:
  Fields:
    StartOfMessage: StartOfMessageEnum::STARTOFMESSAGE
    MessageLength:
    MessageType: MessagesEnum::NEWORDERCROSS
    MatchingUnit: MatchingUnitEnum::MATCHINGUNIT
    SequenceNumber:
    CrossID:
    CrossType: [{Unknown: 0}, {AIM: 1}, {QCC: 2}, {SAM: 3}, {P
    CrossPrioritization: [{Buy: 1}, {Sell: 2}, {Unknown: 0}]
```

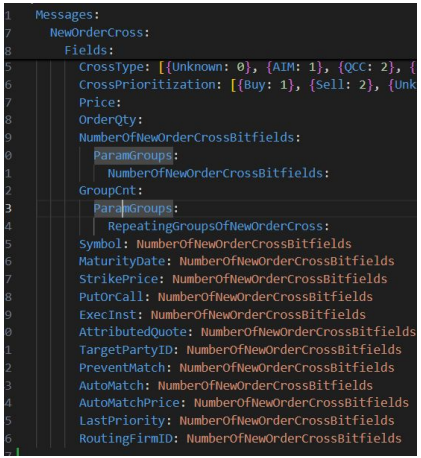
```
Groups:
  NumberOfNewOrderCrossBitfields: ···
  RepeatingGroupsOfNewOrderCross: ···
  NumberOfUnits: ···
  UnitSequences: ···
  NumberOfReturnBitfields:
    Fields:
      ReturnBitfield:
  ReturnBitfields:
    Fields:
      ParamGroupLength:
      ParamGroupType: ParamGroupType
      MessageType:
      NumberOfReturnBitfields:
        ParamGroups:
          NumberOfReturnBitfields:
```

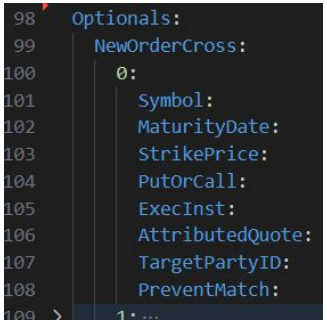
Highlights schema

Bitfields and Optional Fields

- How is field specified as optional field?
  - By the value passed, for example `NumberOfNewOrderCrossBitfields`
  - Same interface for message or group
- How to specify repeating group with optional fields?
  - By setting `metaData`, specifying of which message, and bitfield it depends on.
  - According to BOE repeating groups are associated with a unique message only, and repeating groups can only have optional fields.
- How to specify which bitfield and bit will represent which optional field?
  - Optionals section specifies mapping b/w Optional fields and their associated BitfieldIdx, BitIdx
- How to specify which optional fields should be passed at the end-of-message?
  - Simply append them at the end-of message configuration
  - This solves two issues
    - Not passing optional fields part of repeating group
    - Those which are disabled by the BOE
  - Merits of solution
    - Easy to edit configuration or enable new optional fields.
    - Keeps jinja2 simple, which makes it easy to manage and maintain.
- How are we handling presence of group with optional fields, and constraints on not to pass the information at the end?
  - Explained above

Divide the problem into two pieces and attack them individually to keep the Jinja2 template simple.





Highlights YAML config file

Constant fields can be specified per message.

Allowing flexibility across messages.

Code generator

- Optimizes then as compile time constants using constexpr.
- automatically removes the setter for constant fields
- runtime validation for constant fields according to spec

boe.yaml

```yaml
Messages:
  LoginRequest:
    Fields:
      StartOfMessage: StartOfMessageEnum::STARTOFMESSAGE
      MessageLength:
      MessageType: MessagesEnum::LOGINREQUEST
      MatchingUnit:  MatchingUnitEnum::MATCHINGUNIT
      SequenceNumber: SequenceNumberEnum::SEQUENCENUMBER
      SessionSubID:
      Username:
      Password:
      NumberOfParamGroups:
        ParamGroups:
          UnitSequences:
          ReturnBitfields:

  NewOrderCross: ···
```

Highlights

```
74  UnitSequences:
75    Fields:
76      ParamGroupLength: # might need to handle the name
77      ParamGroupType: ParamGroupTypesEnum::UNITSEQUENCE
78      NoUnspecifiedUnitReplay:  [ {True : 0x01},
79                                  {False: 0x00},
80                                  {Unknown : 0x02} ]
```

Enumerators to handle constant and constrained fields.

What are different types of enumerators?

- Global and within class, optional enumerators to represent different use cases.
- Common enumerators which are shared across different messages and groups are placed global scope.

Support for Constrained Fields

- Message and Group class set Constrained Fields with unknown value when the message object is initialized without any arguments, If the unknown value is not modified the encoder is equipped detect this and mark it as error.
- Encoder and decoder ensures that payload is not corrupted, and populates status object accordingly.
- Unknown is used to ensure that payload is corrupted.
- Message object is by default initialized with Unknown for constrained types.

Code generator takes the responsibility to identifying ParamGroup or Group based on spec.

- abstracts the setter for Group Size, and automating it inside the setter for adding group element.

boe.yaml

```
116
117   Enums:
118     Messages:
119       data:
120         LOGINREQUEST: 0x37
121         LOGINRESPONSE: 0x24
122         NEWORDERCROSS: 0x41
123       dataType: uint8_t
124     ParamGroupTypes:
125       data:
126         RETURNBITFIELDS: 0x81
127         UNITSEQUENCES: 0x80
128       dataType: uint8_t
129     StartOfMessage:
130       data:
131         StartOfMessage: 0xbaba
132       dataType: uint16_t
```

BOE Protocol

Introduction

- Unlike FIX, BOE has **No delimiters**, **No Field Names**, **Only Field Data**
- Login Request Payload
  - BABA3D0037000000000030303031544553545445535449 4E47000000030F008001020 14ABB0100020000000008008125030041050B00812C06004107004000
- New Order Cross Payload
  - BABAB00041006400000004E5A315637424A5F41636365707442757900000031312 04E0000000000006400000002413003003164000000514C37535A37435F6167656E6 37900000000000004343444546470000000000000003228000000514C394B3855565F636F6E747261310000000000464F41424344270200005758595A323C000000514C39543559445F636F6E747261320000000000464F414243447B0000005758595A303051306B41000043444546
- Complex structure with fields, optional fields, nested repeating groups, param groups
- Multiple protocol versions, each with distinct differences.

Each message has **10 byte header**

Layout Identifier

Field	Offset	Length	Data Type	Description
StartOfMessage	0	2	Binary	Must be `0xBA 0xBA`.
MessageLength	2	2	Binary	Number of bytes for the message, including this field but not including the two bytes for the *StartOfMessage* field.
MessageType	4	1	Binary	Message type.
MatchingUnit	5	1	Binary	The matching unit which created this message. Matching units in BOE correspond to matching units on Multicast PITCH. **For session level traffic, the unit is set to 0.** **For messages from Member to Cboe, the unit must be 0.**
SequenceNumber	6	4	Binary	The sequence number for this message. Messages from Cboe to Member are sequenced distinctly per matching unit. Messages from Member to Cboe are sequenced across all matching units with a single sequence stream. Member can optionally send a 0 sequence number on all messages from Member to Cboe. Cboe highly recommends that Members send sequence numbers on all inbound messages.

Login Request

Field	Offset	Length	Data Type
StartOfMessage	0	2	Binary
MessageLength	2	2	Binary
MessageType	4	1	Binary
MatchingUnit	5	1	Binary
SequenceNumber	6	4	Binary
SessionSubID	10	4	Alphanumeric
Username	14	4	Alphanumeric
Password	18	10	Alphanumeric
NumberOfParam Groups	28	1	Binary
ParamGroup$_1$			
…			
ParamGroup$_n$			

Field Name	Hexadecimal	Notes
StartOfMessage	BA BA	Start of message bytes.
MessageLength	3D 00	61 bytes
MessageType	37	Login Request
MatchingUnit	00	Always 0 for inbound messages
SequenceNumber	00 00 00 00	Always 0 for session level messages
SessionSubID	30 30 30 31	0001
Username	54 45 53 54	TEST
Password	54 45 53 54 49 4E 47 00 00 00	TESTING
NumberOfParam Groups	03	3 parameter groups
ParamGroupLength	0F 00	15 bytes for this parameter group
ParamGroupType	80	0x80 = Unit Sequences
NoUnspecified UnitReplay	01	True (replay only specified units)
NumberOfUnits	02	Two unit/sequence pairs to follow;
UnitNumber$_1$	01	Unit 1
UnitSequence$_1$	4A BB 01 00	Last received sequence of 113,482
UnitNumber$_2$	02	Unit 2
UnitSequence$_2$	00 00 00 00	Last received sequence of 0
ParamGroupLength	08 00	8 bytes for this parameter group
ParamGroupType	81	0x81 = Return Bitfields
MessageType	25	0x25 = Order Acknowledgment
NumberOfReturn Bitfields	03	3 bitfields to follow
ReturnBitfield$_1$	00	No bitfields from byte 1
ReturnBitfield$_2$	41	*Symbol, Capacity*
ReturnBitfield$_3$	05	*Account, ClearingAccount*
ParamGroupLength	0B 00	11 bytes for this parameter group
ParamGroupType	81	0x81 = Return Bitfields
MessageType	2C	0x2C = Order Execution
NumberOfReturn Bitfields	06	6 bitfields to follow
ReturnBitfield$_1$	00	No bitfields from byte 1
ReturnBitfield$_2$	41	*Symbol, Capacity*
ReturnBitfield$_3$	07	*Account, ClearingFirm, ClearingAccount*
ReturnBitfield$_4$	00	No bitfields from byte 4
ReturnBitfield$_5$	40	*BaseLiquidityIndicator*

Terminology

Message is a set of fields.

Atomic field are fields which cannot be broken down further.

- StartOfMessage, GrpCnt, ReturnBitfield are atomic.
- Groups, ParamGroups, Bitfields are not atomic.

Groups

G(N, {A_1, A_2, …A_m}, K) = {{A_1, A_2, …A_m}, {A_1, A_2, …A_m} …N times}

NumberOfUnits	02	Two unit/sequence pairs to follow;
UnitNumber$_1$	01	Unit 1
UnitSequence$_1$	4A BB 01 00	Last received sequence of 113,482
UnitNumber$_2$	02	Unit 2
UnitSequence$_2$	00 00 00 00	Last received sequence of 0

A set of atomic fields, repeated k times.

Nested Groups

Can groups have groups?

$G_2(N_2, \{X, G_1(N_1, \{A, B\}), Z\})$

$G_2(N_2, \{X, G_1, Z\})$

NumberOfParamGroups	03	3 parameter groups
ParamGroupLength	0F 00	15 bytes for this parameter group
ParamGroupType	80	0x80 = Unit Sequences
NoUnspecifiedUnitReplay	01	True (replay only specified units)
NumberOfUnits	02	Two unit/sequence pairs to follow;
$UnitNumber_1$	01	Unit 1
$UnitSequence_1$	4A BB 01 00	Last received sequence of 113,482
$UnitNumber_2$	02	Unit 2
$UnitSequence_2$	00 00 00 00	Last received sequence of 0
ParamGroupLength	08 00	8 bytes for this parameter group
ParamGroupType	81	0x81 = Return Bitfields
MessageType	25	0x25 = Order Acknowledgment
NumberOfReturnBitfields	03	3 bitfields to follow
$ReturnBitfield_1$	00	No bitfields from byte 1
$ReturnBitfield_2$	41	*Symbol, Capacity*
$ReturnBitfield_3$	05	*Account, ClearingAccount*
ParamGroupLength	0B 00	11 bytes for this parameter group
ParamGroupType	81	0x81 = Return Bitfields
MessageType	2C	0x2C = Order Execution
NumberOfReturnBitfields	06	6 bitfields to follow
$ReturnBitfield_1$	00	No bitfields from byte 1
$ReturnBitfield_2$	41	*Symbol, Capacity*
$ReturnBitfield_3$	07	*Account, ClearingFirm, ClearingAccount*
$ReturnBitfield_4$	00	No bitfields from byte 4
$ReturnBitfield_5$	40	*BaseLiquidityIndicator*
$ReturnBitfield_6$	00	No bitfields from byte 6

Param Groups

Group whose repeating units can vary.

G(N, {F_1, F_2}, {A_1, A_2, A_3})

G(2, {F_1, F_2}, {A_1, A_2, A_3})

can be

{{F_1, F_2}, {F_1, F_2}} Or

{{F_1, F_2}, {A_1, A_2, A_3}} Or

{{A_1, A_2, A_3}, {A_1, A_2, A_3}}

NumberOfParamGroups	03	3 parameter groups
ParamGroupLength	0F 00	15 bytes for this parameter group
ParamGroupType	80	0x80 = Unit Sequences
NoUnspecifiedUnitReplay	01	True (replay only specified units)
NumberOfUnits	02	Two unit/sequence pairs to follow;
UnitNumber₁	01	Unit 1
UnitSequence₁	4A BB 01 00	Last received sequence of 113,482
UnitNumber ₂	02	Unit 2
UnitSequence₂	00 00 00 00	Last received sequence of 0
ParamGroupLength	08 00	8 bytes for this parameter group
ParamGroupType	81	0x81 = Return Bitfields
MessageType	25	0x25 = Order Acknowledgment
NumberOfReturnBitfields	03	3 bitfields to follow
ReturnBitfield₁	00	No bitfields from byte 1
ReturnBitfield₂	41	*Symbol, Capacity*
ReturnBitfield₃	05	*Account, ClearingAccount*

Optional Fields & Bitfields

ClOrdID	10	20	Text	Corresponds to *ClOrdID* (11) in Cboe FIX. ID chosen by the client. Characters in the ASCII range 33-126 are allowed, except for comma, semicolon, pipe, the 'at' symbol (@) and double quotes. If the *ClOrdID* matches a live order, the order will be rejected as duplicate. Note: Cboe only enforces uniqueness of *ClOrdID* values among currently live orders, which includes long-lived, persisting GTC/GTD orders. However, we strongly recommend that you keep your *ClOrdID* values unique.
Side	30	1	Alphanumeric	Corresponds to *Side* (54) in Cboe FIX. 1 = Buy 2 = Sell
OrderQty	31	4	Binary	Corresponds to *OrderQty* (38) in Cboe FIX. Order quantity. System limit is 999,999 contracts.
NumberOf NewOrder Bitfields	35	1	Binary	Bitfield identifying which bitfields are set. Field values must be appended to the end of the message.
NewOrderBitfield^{1}	36	1	Binary	Bitfield identifying fields to follow.
….				
NewOrderBitfield^{n}		1	Binary	Last bitfield.
Optional fields. . .				

Byte	Bit	Field	
1	1	*ClearingFirm*	●
	2	*ClearingAccount*	●
	4	*Price*	●
	8	*ExecInst*	●
	16	*OrdType*	●
	32	*TimeInForce*	●
	64	*MinQty*	●
	128	*MaxFloor*	●
2	1	*Symbol*	**R**
	2	*SymbolSfx*	
	4	*Currency*	
	8	*IdSource*	
	16	*SecurityId*	
	32	*SecurityExchange*	
	64	*Capacity*	**R**
	128	*RoutingInst*	●

Optional Fields & Bitfields

Bitfields control which optional fields are expected.

Optional fields part of groups are not supplied at end-of-message.

Others will be appended at the end.

NumberOfNewOrder Bitfields	04	Four bitfields to follow
NewOrderBitfield1	04	*Price*
NewOrderBitfield2	C1	*Symbol, Capacity, RoutingInst*
NewOrderBitfield3	01	*Account*
NewOrderBitfield4	17	*MaturityDate, StrikePrice, PutOrCall, OpenClose*
Price	70 17 00 00 00 00 00 00	*0.60*
Symbol	4D 53 46 54 00 00 00 00	MSFT
Capacity	43	C = Customer
RoutingInst	52 00 00 00	R = Routable
Account	44 45 46 47 00 00 00 00 00 00 00 00 00 00 00 00	DEFG
MaturityDate	EF DB 32 01	2011-03-19
StrikePrice	98 AB 02 00 00 00 00 00	17.50
PutOrCall	31	1 = Call
OpenClose	4F	O = Open

Optional Fields & Bitfields

NumberOfReturn Bitfields	02	Two bitfields to follow
ReturnBitfield1	00	No fields from byte 1
ReturnBitfield2	41	*Symbol, Capacity*
GroupCnt	03 00	Two repeating groups to follow
ClOrdID	4E 5A 31 56 37 47 4E 5F 61 67 65 6E 63 79 00 00 00 00 00 00	NZ1V7GN_agency
OrderID	02 C0 91 A2 94 AB 78 04	2G4GYK000002 (base 36)
Capacity	43	C = Customer
ClOrdID	4E 5A 31 56 37 4B 46 5F 63 6F 6E 74 72 61 31 00 00 00 00 00	NZ1V7KF_contra1
OrderID	03 C0 91 A2 94 AB 78 04	2G4GYK000003 (base 36)
Capacity	46	F = Firm
ClOrderID	4E 5A 31 56 37 4E 48 5F 63 6F 6E 74 72 61 32 00 00 00 00 00	NZ1V7NH_contra2
OrderID	04 C0 91 A2 94 AB 78 04	2G4GYK000004 (base 36)
Capacity	46	F = Firm
Symbol	30 30 51 30 6B 41 00 00	00Q0kA

NumberOfReturn Bitfields	47	1	Binary
ReturnBitfield[1]	48	1	Binary
…			
ReturnBitfield[n]		1	Binary
GroupCnt		2	Binary
Repeating Groups of…			
ClOrdId		20	Text
OrderId		8	Binary
Side (Optional)		1	Alphanumeric
AllocQty (Optional)		4	Binary
Capacity (Optional)		1	Alpha
OpenClose (Optional)		1	Alphanumeric
GiveUpFirmID (Optional)		4	Alpha
Account (Optional)		16	Text
CMTANumber (Optional)		4	Binary
ClearingAccount (Optional)		4	Text
Optional fields. . .			