Wetselm Reference

Glen Larsen

Version 1.0b, 21 Jan 2026

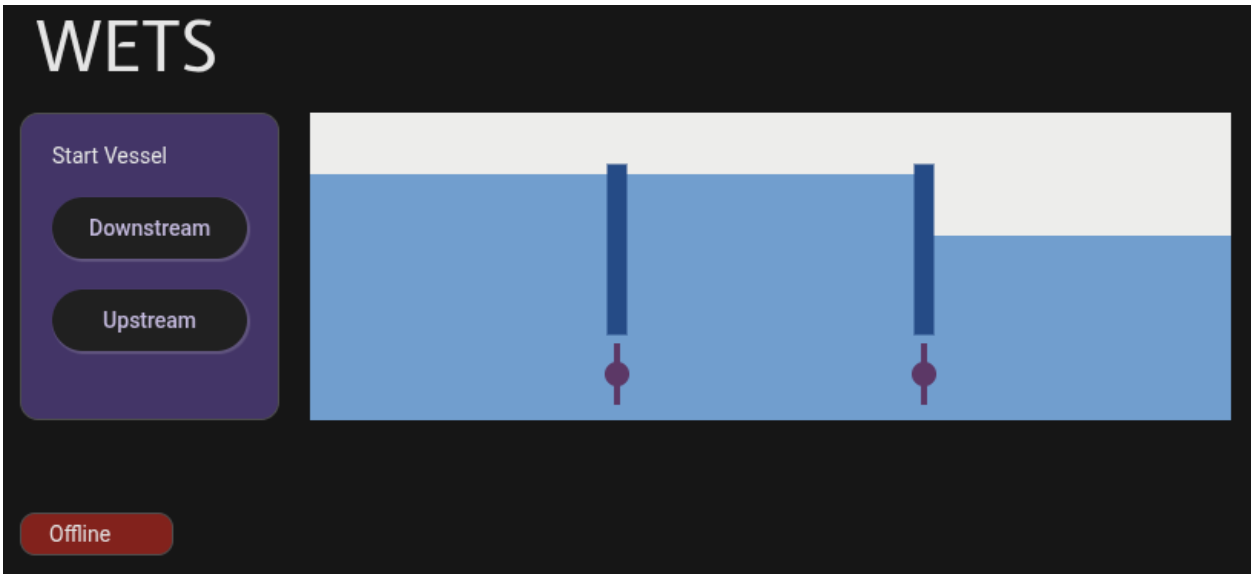
Table of Contents

Application Overview

This project documents and builds a graphical simulation tool for driving a `WETS` information model that has been instrumented to communicate its state to a NATS server. You are reading a literate programing document that contains both documentation and source code.

The document product is a PDF containing both documentation and code.

The generated code product — `wetselm` — is a web application in the elm programming language. Some details may be provided regarding the `elm` language but, since this is not meant as a tutorial, the focus is on the purpose and implementation of the application. You are encouraged to read the extensive Elm Guide for more information.



`wetselm` presents a 2-gate canal lock system that interacts with a `WETS` executable information module to display the state of various entities represented in that model: gates, valves, chambers, and vessels. Communication is accomplished via a NATS server.

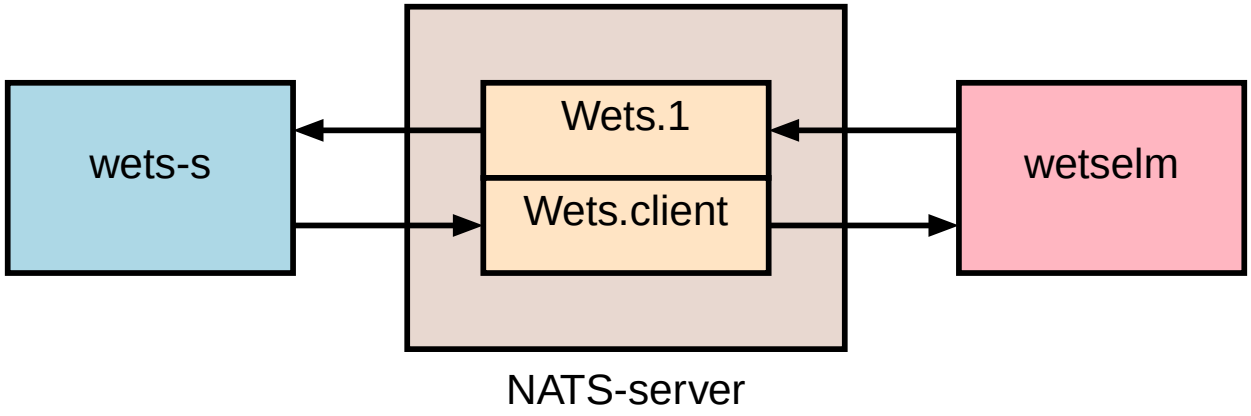


Figure 1. WETS messaging path using NATS

`wetselm` will send `Wets.1` messages to `wets-s` via the `nats-server` and `wets-s` will publish client messages on the `Wets-client` channel. All messaging is done in JSON.

This documented is structured so that every `elm` module has its own section,

Main
: The main `elm` entry point

Wets
: Contains helper functions for encoding and decoding JSON messages for NATS

Util
: Utility functions are refactored here to reduce noise in other modules

UI
: User interface details like various glyphs as well as theming and typography.

Literate programming

As a literate programming document the application source code is woven into this document and can be untangled into working source code as well as the document you are reading. [1] A `Makefile` is provided to build the code and document artifacts so the detailed mechanism is described there and not in this document.

Since the application is generated by this document, fixing bugs and adding features **must** be done by editing this document.

The `atangle` tool is used for extracting the tangled code. Within this document you will find code blocks headed with tags that `atangle` will recognize for organizing the output of the code.

In this document, final assembly of the code blocks is done in a section titled *Code Layout*.

[1] The words, tangle and weave, are established literate programming terms.

Main module

Elm programming dictates the use of a fair number of packages and instead of sprinkling the import statements throughout this section they are presented here so that function and type documentation is more readable.

The elm-animator package has been chosen for animation.

```
<<main-imports>>=
import Animator
import Time
```

Here we import all of the elm-nats packages needed for web socket communication. The `Random` package is here because it is used during init.

```
<<main-imports>>=
import Nats
import Nats.Config
import Nats.Effect
import Nats.Events
import Nats.PortsAPI
import Nats.Protocol
import Nats.Socket
import Nats.Sub
import Random
```

elm-ui is used for layout and UI design.

```
<<main-imports>>=
import Element as E
import Element.Background as Background
import Element.Border as Border
import Color
```

Using elm-orus-ui, a UI toolkit modeled after the Material Design System, provides reasonable theme coloring without too much hassle.

```
<<main-imports>>=
import OUI.Button as Button
import OUI.Material as Material
import OUI.Material.Color
import OUI.Material.Theme
```

```
import OUI.Text as Text
```

For animated graphics, structured vector graphics (SVG) are employed.

```
<<main-imports>>=
import Svg exposing (Svg)
import Svg.Attributes as SvgA
```

NOTE

Not all links for packages used in this application are annotated in this document. Please peruse the absolute source, the local `elm.json` file, for every package used. All elm packages can be found in the Elm package catalog.

Our application communicates with the `WETS` model using `NATS`, which is accomplished with web sockets. This is fairly well hidden from us but the first order of business is to indicate to `elm` that we will be expecting a port to be set up in the `index.html` file by declaring `Main` a port module. Note that only a single function is exposed by the `Main` module.

```
<<main-module>>=
port module Main exposing (main)
```

This is an `elm` browser document which requires us to declare various functions that will inform the architecture how it is going to deal with the state (`Model`) of our application.

view

The assembly of the graphical parts and their layout.

init

Initialization of the model

update

When messages arrive, they are handled here. For a NATS application this function has a wrapper to check to see if a message arrived on the port before calling into the proper update function.

subscriptions

Look here to see how we subscribe to NATS messages and animation timers.

Think of this in this way. The `update` function is, literally, where all the action takes place. Any message defined on the `Model` must be accounted for in `update`. Once updated, the

`view` function is called to read the DOM hierarchy and make sure any graphical updates are made.

`wetselm` will need the following packages as a minimum to accomplish this.

```
<<main-imports>>=
import Browser
import Wets
import UI
import Util
```

The main entry point is called `main` in this module and describes a jump table telling the elm run-time how the `Browser` is built.

```
<<main-functions>>=
main : Program { now : Int } Model Msg
main =
    Browser.document
        { view = view
        , init = init
        , update = wrappedUpdate
        , subscriptions = subscriptions
        }
```

A *wrapped* update is used so the NATS messages arriving can be inserted into the *update* functionality. Take a look at the wrappedUpdate function for more detail.

The subscriptions function is necessary for both message handling and animation.

Model

The `Model` type alias is used within the Elm architecture to declare the state of the application in record form.

```
<<main-types>>=
type alias Model =
    { nats : Nats.State String Msg
    , socket : Nats.Socket.Socket
    , serverInfo : Maybe Nats.Protocol.ServerInfo
    , actuatorStates : Animator.Timeline (Dict Id ActuatorState)
    , chamberStates : Animator.Timeline (Dict Id ChamberState)
```

```
    , vesselStates : Animator.Timeline (Dict Id TransitState)
    , activeVessels : VesselDict
    , vesselCount : Int
    , message : String
    }
```

For NATS communication we need the `nats`, `socket`, and `serverInfo` elements. The `actuator-`, `chamber-`, and `vessel- States` declare the timelines for animation. The `activeVessels` dictionary helps track the transit steps, `vesselCount` is instrumental in vessel naming, and `message` is used for displaying informational messages.

See the init function for how this `model` is initialized.

We use a lot of dictionaries so the import is directed to also expose the `Dict` type to improve code readability.

```
<<main-imports>>=
import Dict exposing (Dict)
```

init

This function is directed here from the main function and initializes all elements of the Model.

- The *flags* argument is populated in the `index.html` file as the current date and is used as a random seed for initializing NATS.
- Motor-driven animations (valves and gates) are initialized here to their population definitions (see the WETS documentation). Note that the naming here reflects the naming in the population as well.
- The center chamber is animated with the `chamberStates` element.
- The vessel states are initialized to an empty dictionary.
- Messages are initialized to an empty string.

The NATS socket is defined to address the web-socket address defined in the `server.config` file.

There is only one `Chamber` even though the use of a dictionary suggests otherwise. This was done for future cases with multiple chambers.

```
<<main-functions>>=
init : { now : Int } -> ( Model, Cmd Msg )
```

```
init flags =
    let
        nats : Nats.State String Msg
        nats = Nats.init (Random.initialSeed flags.now)
             (Time.millisToPosix flags.now)
    in
    ( { actuatorStates =
            Animator.init <|
                Dict.fromList
                    [ ( "Valve-M01", Closed )
                    , ( "Valve-M02", Closed )
                    , ( "Gate-M01", Closed )
                    , ( "Gate-M02", Closed )
                    ]
      , chamberStates =
            Animator.init <|
                Dict.fromList [ ( "Chamber-01", High ) ]
      , vesselStates = Animator.init Dict.empty
      , nats = nats
      , socket = Nats.Socket.new "0" "ws://localhost:8087"
      , serverInfo = Nothing
      , vesselCount = 0
      , activeVessels = Dict.empty
      , message = ""
      }
    , Cmd.none
    )
```

A design decision has been made with regards to vessels. Their state — where they are located during transit — and the vessel instance are managed separately. This is probably more efficient for animating but, mostly, this decision is about navigating the difficulty of a dynamic entity in the `Animator` construct.

ChamberState

This application has a single animated chamber and it has two states.

```
<<main-types>>=
type ChamberState
    = High
    | Low
```

This local function provides an association between a `ChamberState` and its depth in an SVG

glyph. You can thank the SVG coordinate system for any confusion about the value for `High` being lower than the value for `Low`.

```
<<main-functions>>=
chamberDepth : ChamberState -> Float
chamberDepth cstate =
    case cstate of
        High -> 40.0
        Low -> 80.0
```

ActuatorState

Motors (valves and gates) are either `Opened` or `Closed`.

```
<<main-types>>=
type ActuatorState
    = Opened
    | Closed
```

Id

An `Id` is an alias for a `String`.

```
<<main-types>>=
type alias Id =
    String
```

Vessels

TransitState

A `TransitState` defines location milestones in a vessels path through a transit lane. Each vessel will have a specific transit sequence that it will traverse either down- or up- stream.

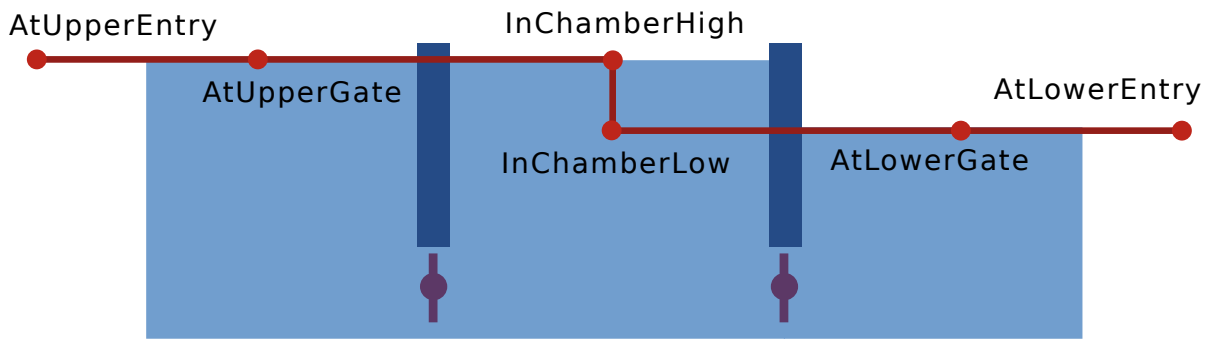


Figure 2. Transit milestones in a vessel's path

- The `*Entry` states are offstage (outside the SVG view) positions and not meant to be directly viewed by the user. This allows an initial animation from offstage to a waiting state.
- The `InChamber*` locations are employed for both vessel and chamber animations.

```
<<main-types>>=
type TransitState
    = AtUpperEntry
    | AtUpperGate
    | InChamberHigh
    | InChamberLow
    | AtLowerGate
    | AtLowerEntry
    | AtSea
```

toPosition

Provide an XY position for each TransitState. These are SVG positions within a SVG view.

See animVessel for how this is used.

```
<<main-functions>>=
toPosition : TransitState -> Util.Position
toPosition state =
    case state of
        AtUpperEntry ->
            { x = -70.0, y = chamberDepth High }
        AtUpperGate ->
            { x = 100.0, y = chamberDepth High }
        InChamberHigh ->
```

```
            { x = 300.0, y = chamberDepth High }
        InChamberLow ->
            { x = 300.0, y = chamberDepth Low }
        AtLowerGate ->
            { x = 500.0, y = chamberDepth Low }
        AtLowerEntry ->
            { x = 650.0, y = chamberDepth Low }
        AtSea ->
            { x = 0.0, y = 0.0 }
```

TransitSequence

This is a list of `TransitState` values and used to define a path along which a vessel is expected to travel.

```
<<main-types>>=
type alias TransitSequence =
    List TransitState
```

downStreamSequence

A `downStreamSequence` is the sequence a vessel travelling downstream is expected to take. Notice how its final move is from the `InChamberLow` state to the `AtLowerEntry` state. During animation, this will whisk the vessel image offstage right (i.e., it skips over the `AtLowerGate` milestone.)

```
<<main-literals>>=
downStreamSequence : TransitSequence
downStreamSequence =
    [ AtUpperEntry
    , AtUpperGate
    , InChamberHigh
    , InChamberLow
    , AtLowerEntry
    ]
```

upStreamSequence

For the `upStreamSequence`, note how the final move is from `InChamberHigh` to `AtUpperEntry` to whisk the vessel image offstage left.

```
<<main-literals>>=
upStreamSequence : TransitSequence
upStreamSequence =
    [ AtLowerEntry
    , AtLowerGate
    , InChamberLow
    , InChamberHigh
    , AtUpperEntry
    ]
```

Vessel

A `Vessel` has a Direction and a TransitSequence. This will be used later as a value for the vessel dictionary.

```
<<main-types>>=
type alias Vessel =
    { direction : Util.Direction
    , sequence : TransitSequence
    }
```

VesselDict

Declare a type we can use to store `Vessel` objects by name.

```
<<main-types>>=
type alias VesselDict =
    Dict Id Vessel
```

nextTransition

The path of a Vessel is determined by its TransitSequence. The next milestone in a sequence is always at the head of the list. A Maybe type is used here because the sequence could very well be empty, yielding `Nothing` from the function.

```
<<main-functions>>=
nextTransition : Id -> VesselDict -> Maybe TransitState
```

```
nextTransition license vdict =
    Dict.get license vdict
        |> Maybe.andThen (\v -> List.head v.sequence)
```

adjustSequence

When the next milestone in a sequence is reached, the front of the list is popped off and the Vessel's dictionary is updated.

```
<<main-functions>>=
adjustSequence : Id -> VesselDict -> VesselDict
adjustSequence license vdict =
    let
        pophead : Vessel -> Vessel
        pophead v =
            { v | sequence = List.drop 1 v.sequence }
    in
    Dict.update license (Maybe.map pophead) vdict
```

newVessel

Construct a Vessel going in a specific direction. This is meant to reside as a value in a Vessel's dictionary.

```
<<main-functions>>=
newVessel : Util.Direction -> Vessel
newVessel dir =
    { direction = dir
    , sequence =
        case dir of
            Util.Upstream ->
                upStreamSequence

            Util.Downstream ->
                downStreamSequence
    }
```

animator

The elm-animator package dictates that an `Animator` literal needs to be defined. These are better described in the package documentation but, briefly, this is for telling the animator

what timelines are active and how their states are accessed.

Each `Timeline` in the Model is represented here.

```
<<main-literals>>=
animator : Animator.Animator Model
animator =
    Animator.animator
        |> Animator.watching
            .actuatorStates
            (\newActuatorStates model ->
                { model | actuatorStates = newActuatorStates}
            )
        |> Animator.watching
            .chamberStates
            (\newChamberStates model ->
                { model | chamberStates = newChamberStates}
            )
        |> Animator.watching
            .vesselStates
            (\newVesselStates model ->
                { model | vesselStates = newVesselStates}
            )
```

subscriptions

There are two subscriptions to declare,

- Messages arriving on the port will trigger a message to update.
- Animation ticks on the timeline will trigger update as well.

```
<<main-functions>>=
subscriptions : Model -> Sub Msg
subscriptions model =
    Sub.batch
        [ Nats.subscriptions natsConfig model.nats
        , Animator.toSubscription AnimationRuntimeStep model animator
        ]
```

Ports

natsConfig

```
<<main-functions>>=
port natsSend : Nats.PortsAPI.Send String msg
port natsReceive : Nats.PortsAPI.Receive String msg

natsConfig : Nats.Config String String Msg
natsConfig =
    Nats.Config.string NatsMsg
        { send = natsSend
        , receive = natsReceive
        }
        |> Nats.Config.withDebug False
```

receiveProg

```
<<main-functions>>=
receiveProg : Nats.Protocol.Message String -> Msg
receiveProg natsMessage =
    ReceiveProg natsMessage.data
```

natsSubscriptions

```
<<main-functions>>=
natsSubscriptions : Model -> Nats.Sub String Msg
natsSubscriptions model =
    Nats.Sub.batch
        [ Nats.connect
            (Nats.Socket.connectOptions "Wets UI" "0.1"
                |> Nats.Socket.withUserPass "test" "test"
            )
            model.socket
            OnSocketEvent
        , Nats.subscribe "Wets.client" receiveProg
        ]
```

wrappedUpdate

```
<<main-functions>>=
wrappedUpdate : Msg -> Model -> ( Model, Cmd Msg )
wrappedUpdate msg model =
    let
        ( newModel, natsEffect, cmd ) =
            update msg model

        ( nats, natsCmd ) =
            Nats.applyEffectAndSub
                natsConfig
                natsEffect
                (natsSubscriptions model)
                newModel.nats
    in
    ( { newModel | nats = nats }
    , Cmd.batch [ cmd, natsCmd ]
    )
```

update

Msg

A core part of the Elm Architecture, the `Msg` type defines the actions that can change the state `model`. For each `Msg` there will be a corresponding update case. Literate programming allows us to itemize these messages next to their update functionality.

```
<<main-types>>=
type Msg
    <<main-messages>>
```

```
<<main-functions>>=
update : Msg -> Model -> ( Model, Nats.Effect String Msg, Cmd Msg )
update msg model =
    case msg of
        <<main-update-case>>
```

The return value includes a `Nats.Effect` which is a departure from the

> standard Elm architecture due to the wrapped update (which got us here.)

AnimationRuntimeStep

This is the animation frame tick which is used to update the Animator instance.

```
<<main-messages>>=
= AnimationRuntimeStep Time.Posix
```

```
<<main-update-case>>=
AnimationRuntimeStep tick ->
    ( Animator.update tick animator model
    , Nats.Effect.none
    , Cmd.none
    )
```

MoveVessel

```
<<main-messages>>=
| MoveVessel Id Int
```

Move a vessel from its current position to the next position in its sequence. The move will take the given amount of milliseconds. There is very little error recovery here because we expect the state transition to be correct from the WETS model.

Both `vesselState` and `activeVessels` dictionaries are updated to reflect the move to the new `TransitState`.

If the next state in the sequence is at either entry point it means that the vessel is completing its transit through the lock system. This will precipitate a `VesselFinished` message being delivered for this vessel after it completes its move.

```
<<main-update-case>>=
MoveVessel license millis ->
    case nextTransition license model.activeVessels of
        Nothing ->
            ( { model
                | message =
                    "handleVessel: No next transit state for " ++ license
              }
            , Nats.Effect.none
```

```
            , Cmd.none
            )

        Just nextState ->
            let
                setVesselState : TransitState -> Dict Id TransitState
                setVesselState newState =
                    Dict.insert license newState
                    <| Animator.current model.vesselStates
            in
            ( { model
                | vesselStates =
                    Animator.go (Animator.millis <| toFloat millis)
                        (setVesselState nextState)
                        model.vesselStates
                , activeVessels =
                    adjustSequence license model.activeVessels
              }
            , Nats.Effect.none
            , if nextState == AtLowerEntry || nextState == AtUpperEntry
              then
                  Delay.after millis <| VesselFinished license
              else
                  Cmd.none
            )
```

VesselFinished

```
<<main-messages>>=
| VesselFinished Id
```

```
<<main-update-case>>=
VesselFinished id ->
    ( { model
          | activeVessels = Dict.remove id model.activeVessels
          , vesselStates =
              Animator.go Animator.immediately
              (Dict.remove id <| Animator.current model.vesselStates)
              model.vesselStates
        , message = "Finished \"" ++ id ++ "\""
      }
    , Nats.Effect.none
    , Cmd.none
    )
```

VesselPassedGate

```
<<main-messages>>=
| VesselPassedGate Id
```

```
<<main-update-case>>=
VesselPassedGate id ->
    ( model
    , Nats.publish "Wets.1" <| Wets.vesselPassedGate id
    , Cmd.none
    )
```

FlowComplete

```
<<main-messages>>=
| FlowComplete Id
```

```
<<main-update-case>>=
FlowComplete id ->
    ( model
    , Nats.publish "Wets.1" <| Wets.flowEqualized id
    , Cmd.none
    )
```

ActuatorMoveDone

```
<<main-messages>>=
| ActuatorMoveDone Id
```

```
<<main-update-case>>=
ActuatorMoveDone id ->
    ( model
    , Nats.publish "Wets.1" <| Wets.actuatorMoveDone id
    , Cmd.none
    )
```

StartVessel

```
<<main-messages>>=
| StartVessel Util.Direction
```

Start a vessel on its TransitSequence.

1. A new Vessel is created.
2. A license is assigned to the vessel which will be used as key in the activeVessels dictionary.
3. The start state is pulled from its `transitSequence`.
4. The start state is inserted into the animator for this vessel.

Following that,

- The model is updated with an incremented vessel count.
- The vesselStates is updated to be at this start state immediately.
- The active vessels dictionary is updated by purging finished vessels and inserting the new vessel.
- Notice of the arriving vessel is sent to WETS.
- The MoveVessel command is sent to update after a brief delay.

```
<<main-update-case>>=
StartVessel direction ->
    let
        vessel : Vessel
        vessel = newVessel direction

        license : String
        license = Util.newVesselName model.vesselCount

        startState : TransitState
        startState =
            vessel.sequence |> List.head |> Maybe.withDefault AtSea

        startVessel : Dict Id TransitState
        startVessel =
            Animator.current model.vesselStates
                |> Dict.insert license startState
    in
    ( { model
        | vesselCount = model.vesselCount + 1
        , vesselStates =
            Animator.go Animator.immediately
```

```
                startVessel
                model.vesselStates
        , activeVessels =
            model.activeVessels
                |> Dict.insert license
                    { vessel | sequence = List.drop 1 vessel.sequence }
        , message =
            "Starting vessel \"" ++ license ++ "\""
      }
    , Nats.publish "Wets.1" <|
        Wets.newVessel license <|
            Util.toString direction
    , Delay.after 100 <| MoveVessel license 1000
    )
```

NatsMsg

When a `NatsMsg` arrives, the `NATS` model is updated which may result in a message delivery to our model.

```
<<main-messages>>=
| NatsMsg (Nats.Msg String Msg)
```

```
<<main-update-case>>=
NatsMsg natsMsg ->
    let
        ( nats, natsCmd ) =
            Nats.update natsConfig natsMsg model.nats
    in
    ( { model | nats = nats }
    , Nats.Effect.none
    , natsCmd
    )
```

OnSocketEvent

```
<<main-messages>>=
| OnSocketEvent Nats.Events.SocketEvent
```

Update the model's `serverInfo` structure on a socket event. There are a number of events here that are ignored.

```
<<main-update-case>>=
OnSocketEvent event ->
    ( { model
        | serverInfo =
            case event of
                Nats.Events.SocketOpen info ->
                    Just info
                _ ->
                    Nothing
      }
    , Nats.Effect.none
    , Cmd.none
    )
```

ReceiveProg

This message arrives when a `Wets.client` message is received via NATS, the handling of which is deferred to the wetsHandler function.

```
<<main-messages>>=
| ReceiveProg String
```

```
<<main-update-case>>=
ReceiveProg data ->
    let
        ( new_model, cmd ) = wetsHandler data model
    in
    ( new_model
    , Nats.Effect.none
    , cmd
    )
```

wetsHandler

Handle incoming `Wets.client` messages based on their messageType so that,

handleMotor
: handles `Motor` messages.

handleFlow
: for `Flow` messages.

handleVessel

for `Vessel` messages.

To accomplish this, `Json.Decode` is needed for the decode function but an appropriate decoder from the `Wets` module is used for the hard work.

```
<<main-imports>>=
import Json.Decode as Decode
```

```
<<main-functions>>=
wetsHandler : String -> Model -> ( Model, Cmd Msg )
wetsHandler data model =
    case Wets.messageType data of
        Just Wets.MotorMsg ->
            case Decode.decodeString Wets.motorDecoder data of
                Ok cmd -> handleMotor model cmd
                Err e ->
                    ( { model | message = Decode.errorToString e }
                    , Cmd.none
                    )

        Just Wets.FlowMsg ->
            case Decode.decodeString Wets.flowDecoder data of
                Ok cmd -> handleFlow model cmd
                Err e ->
                    ( { model | message = Decode.errorToString e }
                    , Cmd.none
                    )

        Just Wets.VesselMsg ->
            case Decode.decodeString Wets.vesselDecoder data of
                Ok cmd -> handleVessel model cmd
                Err e ->
                    ( { model | message = Decode.errorToString e }
                    , Cmd.none
                    )

        Nothing ->
            ( model, Cmd.none )
```

handleVessel

There is only a single vessel command — MOVE — so this results in a MoveVessel command. Vessels are instructed to MOVE when they are ready to pass through a gate so the returning command is set as a sequence to send the appropriate message when it completes.

```
<<main-functions>>=
handleVessel : Model -> Wets.VesselCommand -> ( Model, Cmd Msg )
handleVessel model cmd =
    ( model
    , Delay.sequence
        [ ( 0, MoveVessel cmd.license 1000 )
        , ( 1000, VesselPassedGate cmd.license )
        ]
    )
```

```
<<main-imports>>=
import Delay
```

handleFlow

This command is received a valve has been opened that will result in manipulating the level of water in a chamber. There may or may not be vessels in the chamber.

The name of the sensor is used to derive the target `Chamber-` and `Transit-` states. The target chamber state (`cstate`) is used in the chamber animation. The target transit state (`tstate`) is used to build a list of (possible) vessel moves that are sequenced after the model is updated.

The chamber move takes a fixed amount of time after which the response message is queued to run.

```
<<main-functions>>=
handleFlow : Model -> Wets.FlowCommand -> ( Model, Cmd Msg )
handleFlow model cmd =
    let
        setChamber : Id -> ChamberState -> Dict Id ChamberState
        setChamber id newState =
            Dict.insert id newState <|
                Animator.current model.chamberStates

        targetStates : String -> (ChamberState, TransitState)
        targetStates sname =
```

```
            if sname == "Sensor-F01"
            then ( Low,  InChamberHigh )
            else ( High, InChamberLow )

        (cstate, tstate) = targetStates cmd.name

        chamberMoves : List ( Int, Msg )
        chamberMoves =
            Animator.current model.vesselStates
                |> Dict.filter (\_ s -> s == tstate)
                |> Dict.foldl
                    (\vid _ -> List.append [ ( 0, MoveVessel vid 2000 ) ])
                    []
    in
    ( { model
        | chamberStates =
            Animator.go (Animator.millis 2000)
                (setChamber "Chamber-01" cstate)
                model.chamberStates
      }
    , Delay.sequence <|
        chamberMoves
            ++ [ ( 2000, FlowComplete cmd.name ) ]
    )
```

handleMotor

```
<<main-functions>>=
handleMotor : Model -> Wets.MotorCommand -> ( Model, Cmd Msg )
handleMotor model cmd =
    let
        opAsState : String -> ActuatorState
        opAsState op =
            if op == "RUN_IN" then
                Closed
            else
                Opened

        setActuator : Id -> ActuatorState -> Dict Id ActuatorState
        setActuator id newState =
            Dict.insert id newState
                 <| Animator.current model.actuatorStates

        actState : ActuatorState
        actState =
            opAsState cmd.operation
```

```
    in
    ( { model
        | actuatorStates =
            Animator.go (Animator.millis 1000)
                (setActuator cmd.name actState)
                model.actuatorStates
      }
    , Delay.after 1000 <| ActuatorMoveDone cmd.name
    )
```

View

When any graphical bit changes in the model, the `view` function is called to change the graphics presented to the user. This might appear to be an immense amount of work in a UI with animated graphics, but the work being done here updates the Domain Object Model (DOM) and then … magic happens to update the screen.

Because of all the layout work a view function can be quite large so it is broken out into a number of parts.

view
: The main entry point for view functionality.

locks
: The graphical depiction of a canal lock.

startPanel
: The left-side button controls.

infoPanel
: The ServerInfo panel that has morphed to a simple Offline/Online graphic.

view

```
<<main-functions>>=
view : Model -> Browser.Document Msg
view model =
    { title = "Sim Proto"
    , body =
        [ UI.layout <|
            E.column
                [ E.paddingEach
                    { top = 40, right = 0, bottom = 0, left = 80 }
                ]
```

```
                [ E.row
                    [ E.spacing 20
                    , E.paddingEach { top = 0, right = 0, bottom = 0, left = 30} ]
                    [ Text.displayMedium "WETS"
                        |> Material.text UI.theme
                    ]
                , E.row
                    [ E.paddingEach
                        { top = 20, right = 0, bottom = 60, left = 20 }
                    , E.spacing 20
                    ]
                    [ startPanel
                    , E.el
                        [ E.width <| E.px 600
                        , E.height <| E.px 200
                        ] <| locks model
                    ]
                , E.row
                    [ E.paddingEach
                        { top = 0, right = 0, bottom = 0, left = 20 }
                    , E.spacing 20
                    ]
                    [ infoPanel model
                    , Text.bodyMedium model.message
                        |> Material.text UI.theme
                    ]
                ]
        ]
    }
```

locks

The `locks` function composes an SVG view of a canal lock containing,

- two animated gates
- two animated valves
- an animated center chamber

SVG is z-layered. After defining a 600X200 pixel view the components are built up within that.

- A light-colored background *sky*.
- The chambers are filled rectangles to represent water, the center one is animated.
- The gate is a darker rectangle whose opacity is animated.
- The valve is a hub and vane combination in which the vane is animated.

- Finally the vessels are animated on top of all that.

```
<<main-functions>>=
locks : Model -> E.Element msg
locks model =
    Svg.svg
        [ SvgA.viewBox "0 0 600 200"
        , SvgA.width "600"
        , SvgA.height "200"
        ]
        (List.append
            [ Svg.rect
                [ SvgA.width "600"
                , SvgA.height "200"
                , SvgA.x "0"
                , SvgA.y "0"
                , SvgA.fill <| Color.toCssString Color.lightGray
                ]
                []
            , UI.chamber 0 40
            , animChamber "Chamber-01" model 200
            , UI.chamber 400 80
            , animGate "Gate-M02" model 194 34
            , UI.hub 200 170
            , animVane "Valve-M02" model 200 170
            , animGate "Gate-M01" model 394 34
            , UI.hub 400 170
            , animVane "Valve-M01" model 400 170
            ]
         <| allVessels model
        )
        |> E.html
```

startPanel

This forms the start controls that can introduce vessels into the system. Keypresses are directed to the StartVessel message.

```
<<main-functions>>=
startPanel : E.Element Msg
startPanel =
    let
        scheme : OUI.Material.Color.Scheme
```

```
        scheme =
            OUI.Material.Theme.colorscheme UI.theme
    in
    E.column
        [ Border.rounded 12
        , Border.color <|
            OUI.Material.Color.toElementColor scheme.outlineVariant
        , Border.width 1
        , Background.color <|
            OUI.Material.Color.toElementColor scheme.primaryContainer
        , E.padding 20
        , E.spacing 20
        , E.height E.fill
        ]
        [ "Start Vessel"
            |> Text.bodyMedium
            |> Material.text UI.theme
        , Button.new "Downstream"
            |> Button.onClick (StartVessel Util.Downstream)
            |> Material.button UI.theme []
        , Button.new "Upstream"
            |> Button.onClick (StartVessel Util.Upstream)
            |> Material.button UI.theme [ E.width E.fill ]
        ]
```

infoPanel

This shows the state of the NATS connection.

```
<<main-functions>>=
infoPanel : Model -> E.Element msg
infoPanel model =
    let
        isOnline : Bool
        isOnline =
            case model.serverInfo of
                Nothing -> False
                Just _ -> True

        panelAttr : List (E.Attribute msg)
        panelAttr =
            if isOnline then
                [ Background.color <| E.rgb255 0x1b 0x82 0x2f
                ]
            else
                [ Background.color <|
```

```
                    OUI.Material.Color.toElementColor scheme.errorContainer
                ]

        scheme : OUI.Material.Color.Scheme
        scheme = OUI.Material.Theme.colorscheme UI.theme
    in
    E.el
        ([ Border.rounded 10
         , Border.color <|
            OUI.Material.Color.toElementColor scheme.outlineVariant
         , Border.width 1
         , E.paddingEach { left = 18, top = 6, right = 10, bottom = 6 }
         , E.width <| E.px 100
         ] ++ panelAttr
        )
    <|
        Material.text UI.theme <|
            Text.bodyMedium <|
                if isOnline then
                    "Online"

                else
                    "Offline"
```

Animation

The following functions implement animated bits in the lock function.

Vane animation

The moving part of the valve depiction.

Chamber animation

The chamber resized its height to simulate filling or draining of a chamber.

Gate animation

The gate changes visually when opened or closed.

Vessels

All active vessels are checked for animation.

animVane

Only the vane of the valve is animated. This makes a quarter circle rotation to represent open (π/2) or closed (0).

```
<<main-functions>>=
```

```
valveValue : ActuatorState -> Float
valveValue astate =
    case astate of
        Opened -> pi / 2.0
        Closed -> 0.0
```

This animates a single vane for the actuator with the given `Id`.

```
<<main-functions>>=
animVane : Id -> Model -> Int -> Int -> Svg msg
animVane id model x y =
    UI.vane x y
    <| Animator.linear model.actuatorStates <|
        \actuatorStates ->
            Animator.at <|
                case Dict.get id actuatorStates of
                    Just astate -> valveValue astate
                    Nothing -> 0.0
```

animChamber

Animating a chamber is a matter of manipulating the Y coordinate in the appropriate direction.

- The `Id` of the chamber is passed in so we can find it in our chamber state dictionary
- The `Model` is needed since that holds the `chamberStates`.
- The fixed `xoffset` is passed in.
- The chamber depth is given by the depth associated with the state.

```
<<main-functions>>=
animChamber : Id -> Model -> Int -> Svg msg
animChamber id model xoffset =
    UI.chamber xoffset
    <| round
    <| Animator.linear model.chamberStates <|
         \chamberStates ->
             Animator.at <|
                 case Dict.get id chamberStates of
                     Just cs -> chamberDepth cs
                     Nothing -> 0.0
```

animGate

Animating a gate is tricky since the side view will look like a large slab and the vessel will be hidden behind it as it passes through. Here the opacity of the gate is animated so it transitions from fully opaque to almost transparent.

```
<<main-functions>>=
gateOpacity : ActuatorState -> Float
gateOpacity astate =
    case astate of
        Opened -> 0.25
        Closed -> 1.0
```

```
<<main-functions>>=
animGate : Id -> Model -> Int -> Int -> Svg msg
animGate id model x y =
    UI.gate x y
    <| Animator.linear model.actuatorStates <|
        \actuatorStates ->
            Animator.at <|
                case Dict.get id actuatorStates of
                    Just gstate -> gateOpacity gstate
                    Nothing -> 0.0
```

animVessel

Vessel animation is interesting because it moves in both X and Y. All the animations are linear and, for some reason, this `xy` animator has some easing to it. Since it may be that the vessel being animated is in a chamber with a changing water level, this easing is zeroed in `y` to force a linear motion.

```
<<main-functions>>=
animVessel : Id -> Model -> Util.Direction -> Svg msg
animVessel id model direction =
    UI.vessel id direction
    <| Animator.xy model.vesselStates <|
        \vStates ->
            let
                xypos : Util.Position
                xypos =
                    Dict.get id vStates
```

```
                        |> Maybe.withDefault AtSea
                        |> toPosition
            in
                { x = Animator.at xypos.x
                , y = Animator.at xypos.y
                    |> Animator.leaveSmoothly 0
                    |> Animator.arriveSmoothly 0
                }
```

allVessels

This is a view function for displaying all active vessels.

```
<<main-functions>>=
allVessels : Model -> List (Svg msg)
allVessels model =
    model.activeVessels
        |> Dict.toList
        |> List.map
           (\t -> animVessel (Tuple.first t)
                model
                (Tuple.second t).direction)
```

Code Layout

```
<<Main.elm>>=
<<main-module>>

<<main-imports>>

<<main-types>>

<<main-literals>>

<<main-functions>>
```

UI helper functions

This module contains miscellaneous functions graphical elements of the UI. It is broken into two parts,

1. Theming, guiding the overall look.
    - typescale
    - theme
    - layout
2. Lock Graphics, animated glyphs.
    - vane
    - hub
    - gate
    - chamber
    - vessel

```
<<ui-imports>>=
import Angle
import Color
import OUI.Material.Color as Color
import OUI.Material.Theme as Theme
import Util
```

Theming

typescale

About the only reason this is modified is to set the *hero* font for the UI. The chosen font needs some supporting code in the `index.html` file.

```
<<ui-functions>>=
typescale : Theme.Typescale
typescale =
    let
        base =
            Theme.defaultTypescale
```

```
        display =
            base.display

        displayMedium =
            display.medium
    in
    { base
        | display =
            { display
                | medium =
                    { displayMedium
                        | font = "Expletus sans"
                    }
            }
    }
```

theme

```
<<ui-exports>>=
theme
```

Besides the `Typescale` all we're doing here is selecting a base color scheme for the material look.

```
<<ui-functions>>=
theme : Theme.Theme ()
theme =
    Theme.defaultTheme
        |> Theme.withTypescale typescale
        |> Theme.withColorscheme Color.defaultDarkScheme
```

layout

```
<<ui-exports>>=
, layout
```

The layout sets up the overall color and options of the UI. The `focusStyle` is *disappeared* so as to not interfere with the material look.

```
<<ui-imports>>=
```

```
import Element as E
import Element.Background as Background
import Element.Font as Font
import Html exposing (Html)
import Html.Attributes
```

```
<<ui-functions>>=
layout : E.Element msg -> Html msg
layout =
    let
        scheme : Color.Scheme
        scheme =
            Theme.colorscheme theme
    in
    E.layoutWith
        { options =
            [ E.focusStyle
                { borderColor = Nothing
                , backgroundColor = Nothing
                , shadow = Nothing
                }
            ]
        }
        [ E.height E.fill
        , E.width E.fill
        , Background.color <| Color.toElementColor scheme.surface
        , Font.color <| Color.toElementColor scheme.onSurface
        , E.htmlAttribute <|
            Html.Attributes.style "-webkit-tap-highlight-color" "transparent"
        ]
```

Lock graphics

```
<<ui-imports>>=
import Svg exposing (Svg)
import Svg.Attributes as SvgA
```

vane

```
<<ui-exports>>=
, vane
```

This glyph represents the *paddles* of a valve that can be rotated in any direction around the center. It is a `polyline` that uses a transform to do the heavy lifting.

```
<<ui-functions>>=
vane : Int -> Int -> Float -> Svg msg
vane x y radians =
    let
        xForm : String
        xForm =
            [ "translate ("
            , String.fromInt x
            , " "
            , String.fromInt y
            , ") rotate ("
            , String.fromInt
                <| round <| Angle.inDegrees <| Angle.radians radians
            , ")"
            ] |> String.concat
    in
    Svg.g
        [ SvgA.transform xForm
        ]
        [ Svg.polyline
            [ SvgA.points "0 20 0 -20"
            , SvgA.strokeWidth "4"
            , SvgA.stroke <| Color.toCssString Color.darkPurple
            ]
            []
        ]
```

hub

```
<<ui-exports>>=
, hub
```

This is the center of the *valve* and isn't meant to animate.

```
<<ui-functions>>=
hub : Int -> Int -> Svg msg
hub x y =
    Svg.circle
```

```
        [ SvgA.cx <| String.fromInt x
        , SvgA.cy <| String.fromInt y
        , SvgA.r "8"
        , SvgA.fill <| Color.toCssString Color.darkPurple
        ]
        []
```

chamber

```
<<ui-exports>>=
, chamber
```

A *chamber* is an enclosed body of water represented by a rectangle. To achieve animation the *depth* parameter is used to vary the height.

```
<<ui-functions>>=
chamber : Int -> Int -> Svg msg
chamber xoffset depth =
    Svg.rect
        [ SvgA.width "200"
        , SvgA.height <| String.fromInt <| 200 - depth
        , SvgA.x <| String.fromInt xoffset
        , SvgA.y <| String.fromInt depth
        , SvgA.fill <| Color.toCssString Color.lightBlue
        ]
        []
```

gate

```
<<ui-exports>>=
, gate
```

A gate is represented by a rectangle and allows the opacity to be passed in as a parameter so it can be animated externally.

```
<<ui-functions>>=
gate : Int -> Int -> Float -> Svg msg
gate x y opacity =
```

```
    Svg.rect
        [ SvgA.width "12"
        , SvgA.height "110"
        , SvgA.x <| String.fromInt x
        , SvgA.y <| String.fromInt y
        , SvgA.fill <| Color.toCssString Color.darkBlue
        , SvgA.stroke <| Color.toCssString Color.darkBlue
        , SvgA.fillOpacity <| String.fromFloat opacity
        ]
        []
```

vessel

```
<<ui-exports>>=
, vessel
```

A vessel is depicted as a simple triangle pointing either east or west to describe its up- or down- stream direction. It was done this way out of worry that something more elaborate might be too heavy for animation but it represents the look nicely. It is not fully opaque so that passing vessels appear to pass through each other.

The vessel license is annotated below the vessel. There is only a vague effort to center the text.

```
<<ui-functions>>=
vessel : String -> Util.Direction -> Util.Position -> Svg msg
vessel name direction pos =
    let
        xForm : String
        xForm =
            [ "translate ("
            , String.fromInt <| round <| pos.x - 12
            , " "
            , String.fromInt <| round <| pos.y - 12
            , ")"
            ] |> String.concat

        points : String
        points =
            case direction of
                Util.Downstream ->
                    "2,2 22,12 2,22"

                Util.Upstream ->
```

```
                  "22,2 2,12 22,22"
    in
    Svg.g
        [ SvgA.transform xForm
        ]
        [ Svg.polygon
            [ SvgA.points points
            , SvgA.fill <| Color.toCssString Color.darkGreen
            , SvgA.fillOpacity ".8"
            ]
            []
        , Svg.text_
            [ SvgA.y "45"
            , SvgA.x "-24"
            , SvgA.fill <| Color.toCssString Color.yellow
            , SvgA.fontSize ".6em"
            ]
            [ Svg.text name
            ]
        ]
```

Code Layout

```
<<UI.elm>>=
module UI exposing
  (
  <<ui-exports>>
  )
<<ui-imports>>
<<ui-functions>>
```

Wets messaging helpers

This module contains functions for encoding and decoding JSON messages directed to and from the `wets-s` service.

There are three incoming `Wets.client` message types that need to be addressed.

Motor
: Motor messages directing the movement of a valve or gate.

FlowSensor
: Flow Sensor message to monitor flow.

Vessel
: Message directing a vessel to move.

The outgoing `Wets.1` messages, requiring encoding, are,

VesselArrived
: Vessel arrived, ready for transit.

MotorCompleted
: The motor command has completed, the extent has been reached for a valve or gate.

VesselPassedGate
: Vessel has moved through and is clear of the gate.

FlowEqualized
: The flow has equalized on either side of a gate, signalling that a gate can be opened.

```
<<wets-module>>=
module Wets exposing (
    <<wets-exports>>
    )
```

As expected, the Elm JSON packages are used for the decoding.

```
<<wets-imports>>=
import Json.Encode as Encode
import Json.Decode as Decode
```

Decoding incoming messages

WetsMessage

This type declaration is useful so that an `elm update` case can handle a specific type of message.

```
<<wets-exports>>=
WetsMessage (..)
```

```
<<wets-types>>=
type WetsMessage
    = MotorMsg
    | FlowMsg
    | VesselMsg
```

messageType

Given a string, assumed to be an incoming message, return the message type.

```
<<wets-exports>>=
, messageType
```

To get the message type, the `object` field of the message is decoded, then it is translated into a WetsMessage type.

Yes, you could do more error tracking here, perhaps letting the Result propagate up to the client?

```
<<wets-functions>>=
messageType : String -> Maybe WetsMessage
messageType s =
    case Decode.decodeString (Decode.field "object" Decode.string) s of
        Ok cmd ->
            case cmd of
                "Motor" ->
                    Just MotorMsg
                "FlowSensor" ->
```

```
                    Just FlowMsg
                "Vessel" ->
                    Just VesselMsg
                _ ->
                    Nothing
        Err _ ->
            Nothing
```

MotorCommand

The strategy for all incoming messages is to decode the string as given and presume the client has previously determined that it is, in fact, the *type* of message they are expecting. That is, the client first uses wetsMessageType before requesting a decoder to break it apart.

```
<<wets-types>>=
type alias MotorCommand =
    { object : String
    , operation : String
    , name : String
    }
```

motorDecoder

```
<<wets-exports>>=
, motorDecoder, MotorCommand
```

```
<<wets-functions>>=
motorDecoder : Decode.Decoder MotorCommand
motorDecoder =
    Decode.map3 MotorCommand
        (Decode.field "object" Decode.string)
        (Decode.field "operation" Decode.string)
        (Decode.field "name" Decode.string)
```

FlowCommand

```
<<wets-functions>>=
```

```
type alias FlowCommand =
    { object : String
    , operation : String
    , name : String
    }
```

flowDecoder

```
<<wets-exports>>=
, flowDecoder, FlowCommand
```

```
<<wets-functions>>=
flowDecoder : Decode.Decoder FlowCommand
flowDecoder =
    Decode.map3 FlowCommand
        (Decode.field "object" Decode.string)
        (Decode.field "operation" Decode.string)
        (Decode.field "name" Decode.string)
```

VesselCommand

```
<<wets-types>>=
type alias VesselCommand =
    { object : String
    , operation : String
    , license : String
    , gate : String
    }
```

vesselDecoder

```
<<wets-exports>>=
, vesselDecoder, VesselCommand
```

```
<<wets-functions>>=
vesselDecoder : Decode.Decoder VesselCommand
```

```
vesselDecoder =
    Decode.map4 VesselCommand
        (Decode.field "object" Decode.string)
        (Decode.field "operation" Decode.string)
        (Decode.field "license" Decode.string)
        (Decode.field "gate" Decode.string)
```

Encoding outgoing messages

newVessel

This message encodes for the `VesselArrived` `Wets.1` message.

Create a message for a newly arriving vessel with a name (`license`) and a particular direction. This message has the important distinction of starting the process of movement through the WETS system. [1]

```
<<wets-exports>>=
, newVessel
```

```
<<wets-functions>>=
newVessel : String -> String -> String
newVessel license direction =
    Encode.object
        [ ("object", Encode.string "VesselArrived")
        , ("license", Encode.string license)
        , ("direction", Encode.string direction)
        , ("wets_name", Encode.string "Wets_1")
        ]
        |> Encode.encode 0
```

actuatorMoveDone

Encodes for the `MotorCompleted` `Wets.1` message.

```
<<wets-exports>>=
, actuatorMoveDone
```

```
<<wets-functions>>=
actuatorMoveDone : String -> String
actuatorMoveDone name =
    Encode.object
        [ ("object", Encode.string "MotorCompleted")
        , ("name", Encode.string name)
        ]
        |> Encode.encode 0
```

vesselPassedGate

Encodes for the `VesselPassedGate` `Wets.1` message.

```
<<wets-exports>>=
, vesselPassedGate
```

```
<<wets-functions>>=
vesselPassedGate : String -> String
vesselPassedGate license =
    Encode.object
        [ ("object", Encode.string "VesselPassedGate")
        , ("license", Encode.string license)
        ]
        |> Encode.encode 0
```

flowEqualized

Encodes for the `FlowEqualized` `Wets.1` message on a named entity.

```
<<wets-exports>>=
, flowEqualized
```

```
<<wets-functions>>=
flowEqualized : String -> String
flowEqualized name =
    Encode.object
        [ ("object", Encode.string "FlowEqualized")
```

```
        , ("name", Encode.string name)
        ]
        |> Encode.encode 0
```

Code Layout

```
<<Wets.elm>>=
<<wets-module>>

<<wets-imports>>

<<wets-types>>

<<wets-functions>>
```

[1] It is at this point that the WETS population is chosen. In our case this is `Wets_1`, which is the graphical description of our client expects. It is expected that other model populations may be used in a more elaborate manner.

Util, the utilities module

The `Util` module holds various types and functions that are used throughout the application, typically refactored here after review to reduce weight in modules that require clarity.

```
<<util-module>>=
module Util exposing (
    <<util-exports>>
    )
```

Direction

```
<<util-exports>>=
Direction(..)
```

```
<<util-types>>=
type Direction
    = Upstream
    | Downstream
```

```
<<util-exports>>=
, toString
```

toString (from Direction)

```
<<util-functions>>=
toString : Direction -> String
toString d =
    case d of
        Upstream ->
            "up"

        Downstream ->
            "down"
```

Position

This XY record structure was being used in several places so it was refactored here.

```
<<util-exports>>=
, Position
```

```
<<util-types>>=
type alias Position =
    { x : Float
    , y : Float
    }
```

newVesselName

Make up a vessel name from Lorem Ipsum text. Not much to see here really, I just wanted something that would provide unique names. An index is tacked to the end just to make sure. Some care is taken to remove punctuation and, since its a boat title, capitalize the text.

```
<<util-exports>>=
, newVesselName
```

```
<<util-imports>>=
import Lorem
import String.Extra
```

```
<<util-functions>>=
newVesselName : Int -> String
newVesselName index =
    let
        nmax : Int
        nmax = 50

        n : Int
        n = modBy nmax index

        wordpair : List String
```

```
        wordpair =
            Lorem.words (nmax + 1)
                |> List.drop n
                |> List.take 2
    in
        List.singleton (String.fromInt index)
        |> List.append wordpair
        |> List.map (String.filter Char.isAlphaNum)
        |> String.join " "
        |> String.Extra.toTitleCase
```

Code Layout

```
<<Util.elm>>=
<<util-module>>

<<util-imports>>

<<util-types>>

<<util-functions>>
```

Index