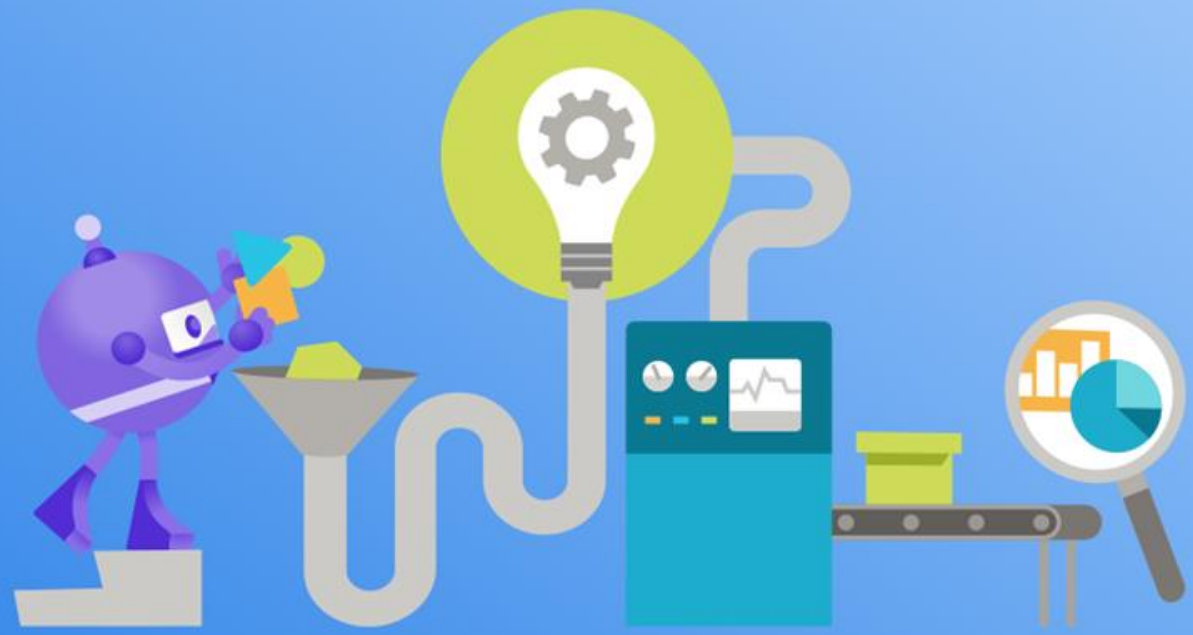


.NET Conf

Focus on F#

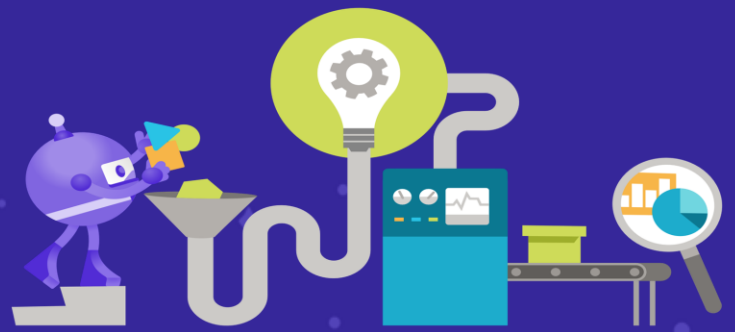
July 29, 2021

focus.dotnetconf.net

DEMETRIX

Into the Land of Biotech, with F# as an Ally

Olya Samusik | @w0lya
Software Engineer at Demetrix, Inc

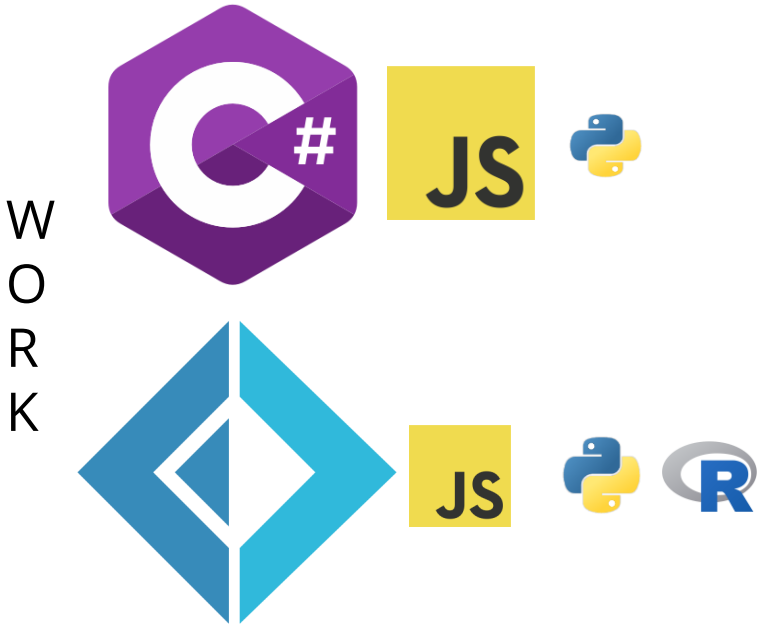


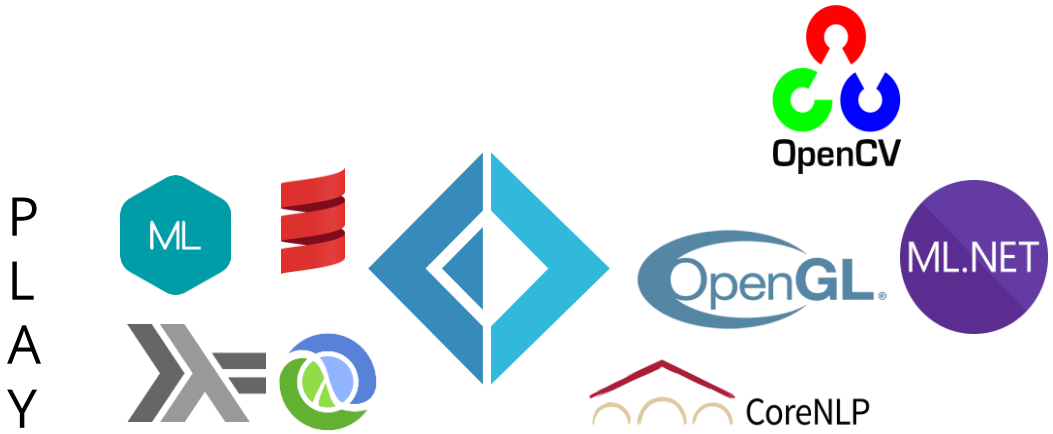
Thanks!

- Demetrix
  - Software Engineers
  - Data Scientists
  - IT
  - Scientists
- Dotnet Conf, Focus on F# organizers
- F# Community
  - Don Syme - Language + Type provider support
  - Alfonso Garcia-Caro - Fable and solid advice
  - Dustin Moris Gorski - Giraffe
  - Maxime Mangel - Elmish
  - Postgres Npgsql maintainers - improvements in type safety
  - SAFE stack maintainers
  - Elliot Menschik, Paul Underwood - AWS Biotech support
  - Krzysztof-Cieslak - Ionide
  - Matthias Brandewinder - Recruiting
  - Chris Macklin - Genotype Specification Language collaboration
  - Cody Johnson - Feliz.Plotly
  - a lot of other people...

About me

- Software engineer
  - 9 y of (mostly) C# coding by day at work, ~5y of occasional F# coding by night
  - 2 y of full stack F# at work! And trying to wrap my head around biotech
  - computer graphics / vision | ML, data science |…
- Contact info
  - w0lya on Twitter, GitHub, …
  - olya@demetrix.com



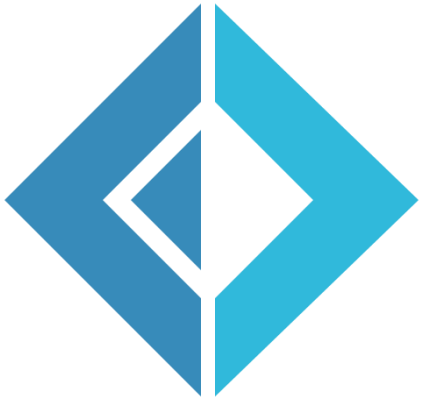


Plan

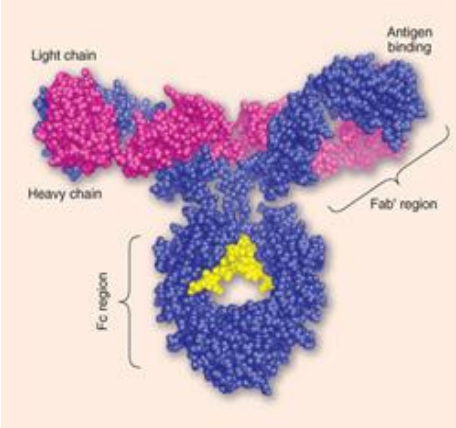
- Quick peek at biotechnology
- Demetrix, what do we do
- Challenges we face, usual / unusual
- Our technical landscape
- Software we build
- Why F#, how it helps us
- Problems and solutions
- Lessons learned, opportunities for improvement
- Q&A

DEMETRIX

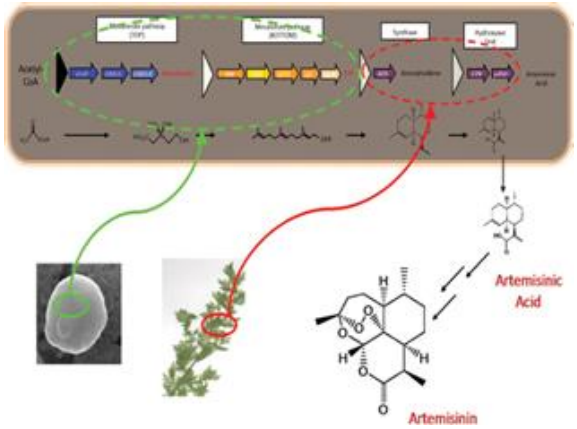
+



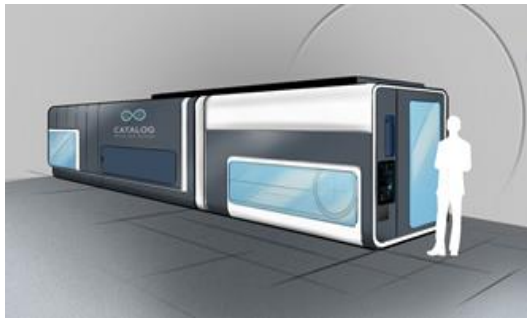
Biotech - the future is now!



Therapeutic antibodies (cancer , infectious disease)



Malaria medicine manufactured with yeast
Keasling, Amyris



Prototype disk drive writing data into DNA
Catalog DNA

IMPOSSIBLE™

Alternative foods.
Heme molecule

BOLT THREADS

Biosynthecized construction materials, e.g. cement

Modern Meadow

Biosynthesized sustainable materials, e.g. leather

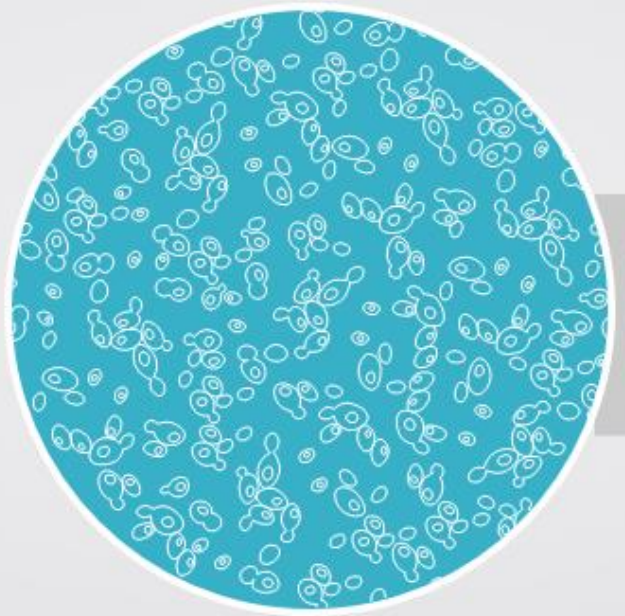
ZBiotics - hangover prevention (bacteria that eats acetaldehyde)

I Grew Real Spider Silk Using Yeast

Justin Atkin, DIY enthusiast (The Thought Emporium)

What does Demetrix do?

Demetrix takes DNA sequences from plants and puts them into yeast – yeast now makes what the plant does



Letter | Published: 27 February 2019

Complete biosynthesis of cannabinoids and their unnatural analogues in yeast

Xiaozhou Luo, Michael A. Reiter, Leo d'Espaux, Jeff Wong, Charles M. Denby, Anna Lechner, Yunfeng Zhang, Adrian T. Grzybowski, Simon Harth, Weiyin Lin, Hyunsu Lee, Changhua Yu, John Shin, Kai Deng, Veronica T. Benites, George Wang, Edward E. K. Baidoo, Yan Chen, Ishaan Dev, Christopher J. Petzold & Jay D. Keasling

Nature **567**, 123–126(2019) | Cite this article

Panoramic view

Teams

Strain Engineering
Design and build yeast strains, aiming to optimize production

DNAOps
Make the process of Strain Engineering efficient and consistent

USP (UpStream Processing)
Engineer robust and scalable fermentation processes that allow for optimal strain performance

DSP (DownStream Processing)
Develop and support processes for extraction, separation, purification and preservation of the products

HTPE (High-Throughput Process Engineering)
Characterize, optimize and execute scaled down processes, perform strain screening

HTS (High Throughput Screening)
Auto-testing large numbers of compounds for a specific biological target

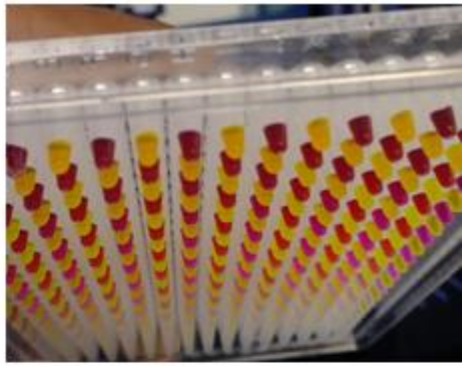
Analytics
Develop assays and process samples to get qualitative (which ones) and quantitative data (concentration) about analytes

Two avenues necessary for success

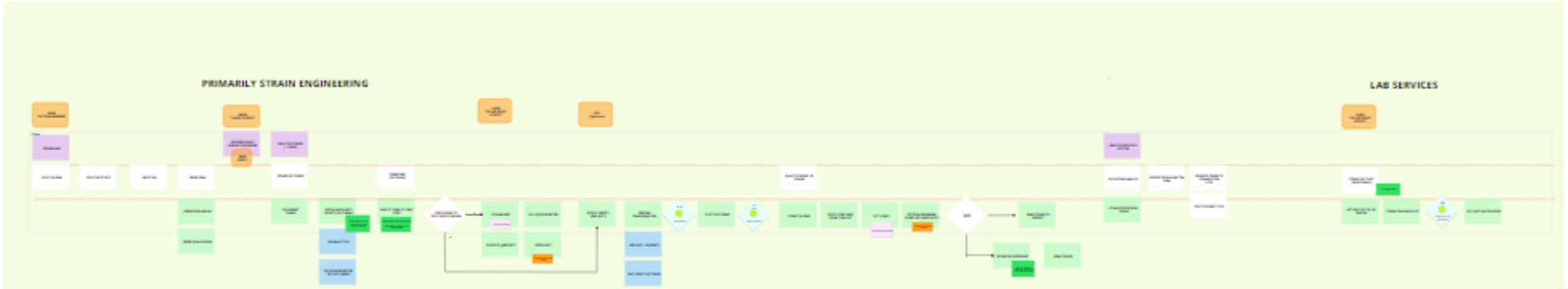
R&D/Strain Engineering

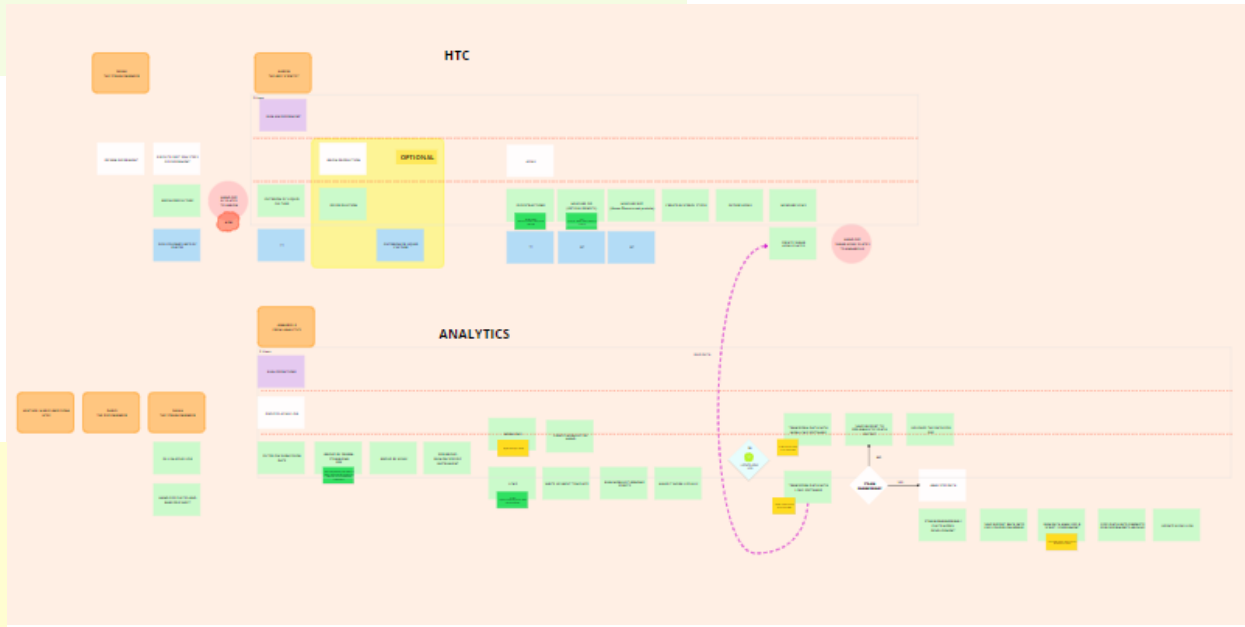
Manufacturing/Process Engineering

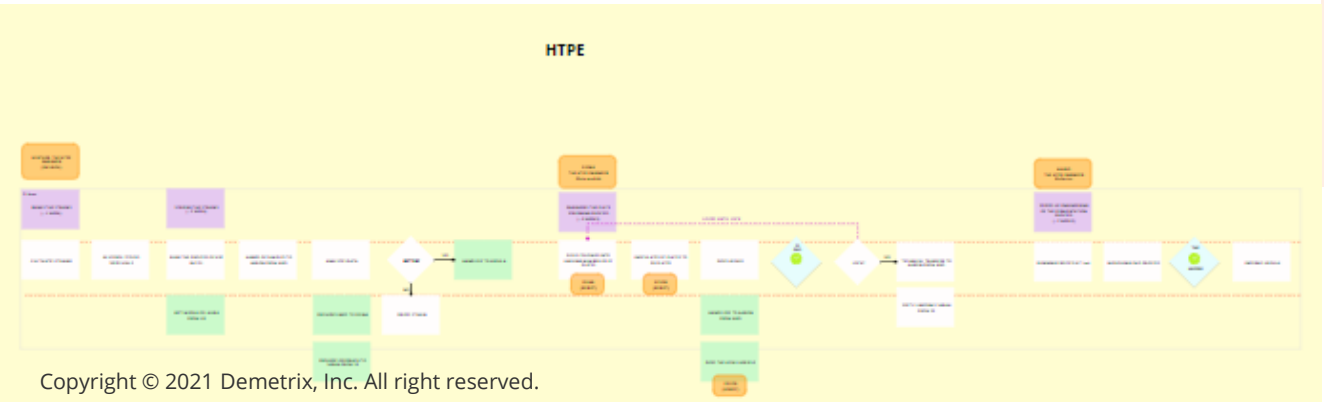
Let’s zoom in a bit… (partial) real world view



Let’s zoom in a bit… (partial) logical view. Complex!







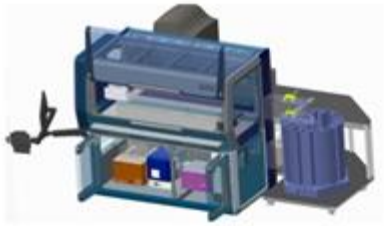
Some of the challenges we face

- Modelling multiple complex domains
- Gathering, understanding, prioritizing requirements
- Enabling maximum user engagement & involvement in building software (for their own good!)
- Addressing rapidly evolving requirements
- Gathering, structuring, making use of a lot of heterogeneous data
- Ensuring secure access for humans and robots
- Building nice UI to help users solve unconventional problems
- Being efficient regardless of being spread thin across different areas

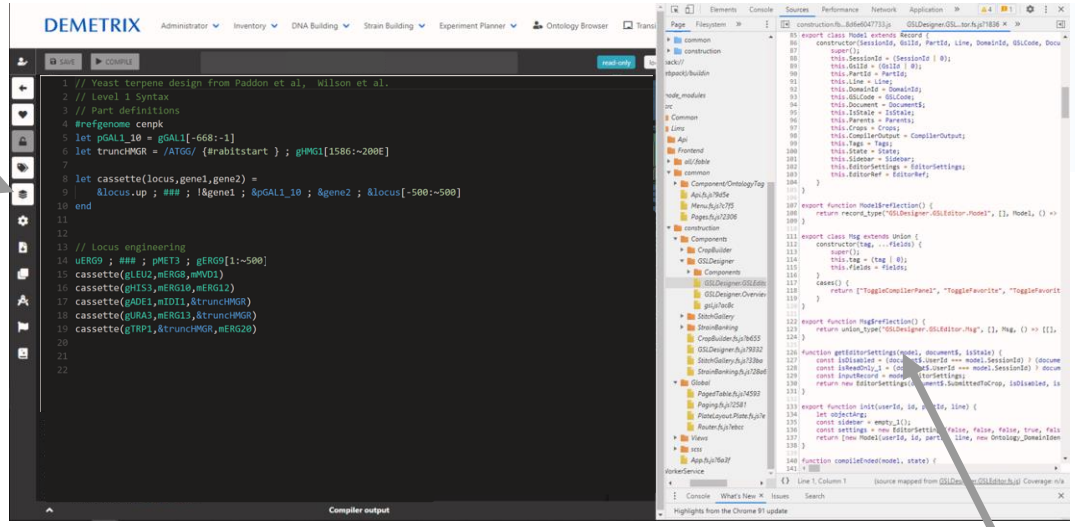
- Web server
- Hosting
- Prototyping
- Testing
- Iterations
- CI / CD
- Maintenance
- Error handling
- Logging
- Batch, recurring tasks
- Reliable and performant data storage
- Reusing functionality, infrastructure across applications

Tech landscape

User interfaces
Data visualization
Lab / Robot data services



Client



TABLE

aws
Auth

ASP.NET Core

GIRAFFE

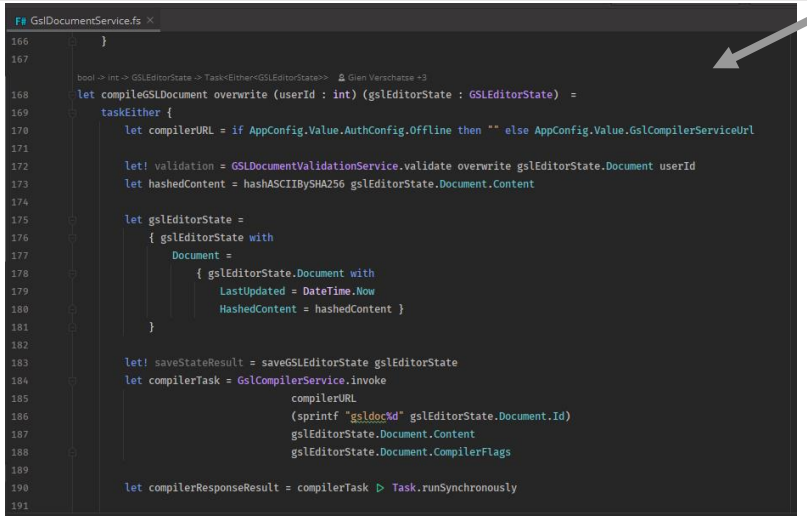
Full stack F#

Shared
code and
data definitions

aws
Hosting

Hangfire
(batch tasks)

Server



PostgreSQL

Grafana

FAKE (F# make)

PAKET

User services
– mail / docs

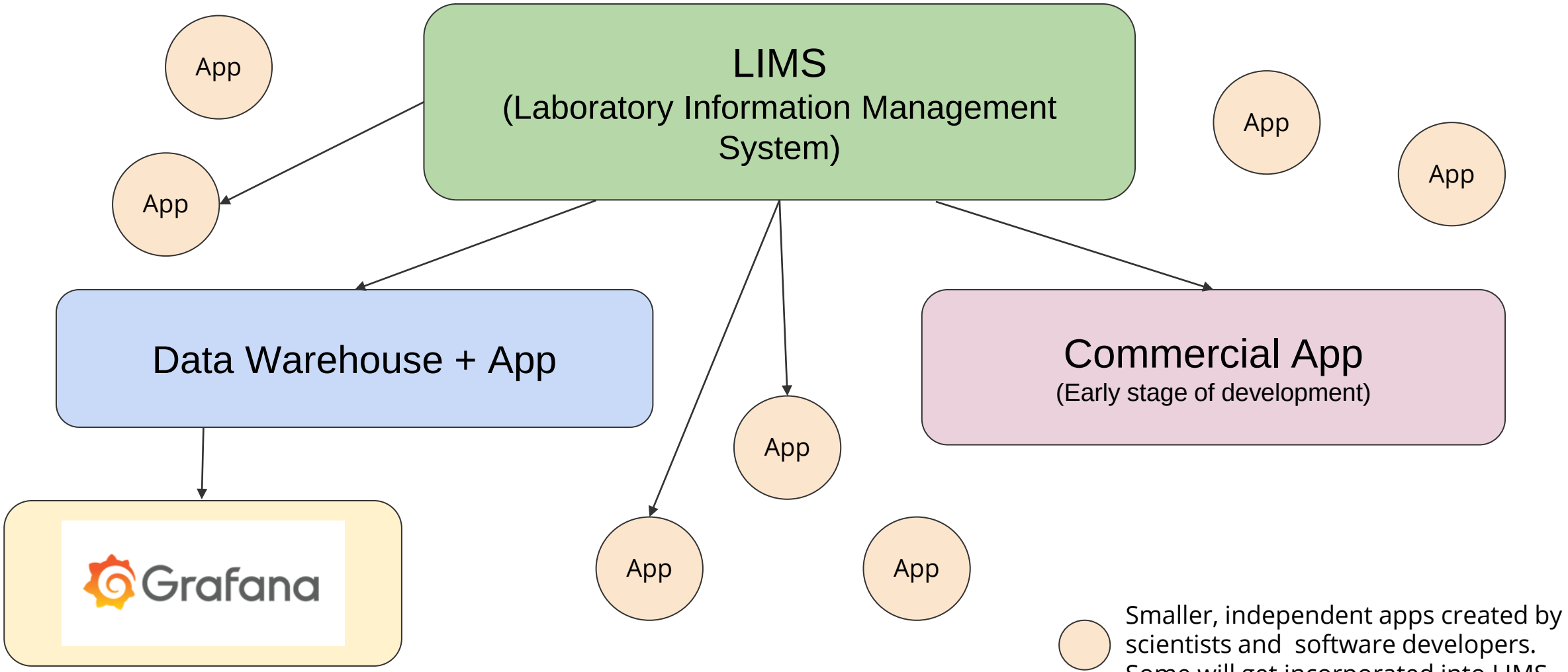
Prometheus

Metrics

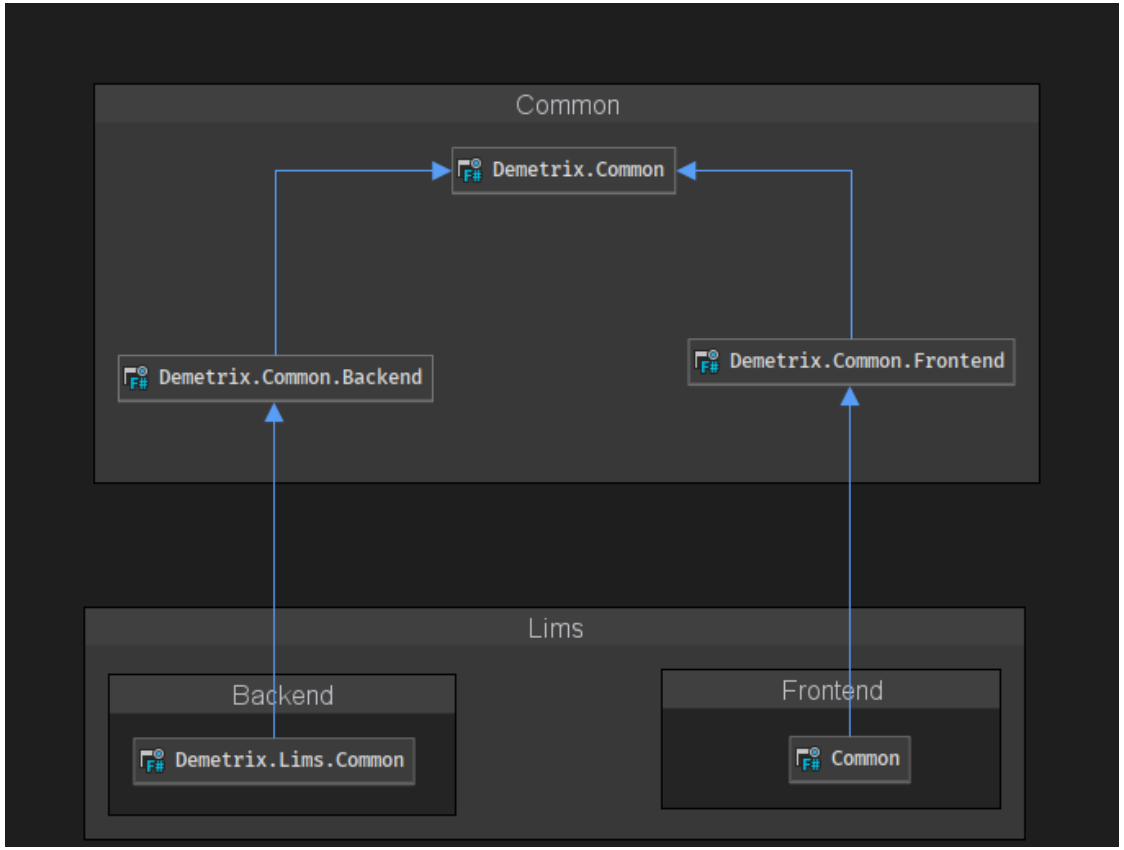
logstash
elasticsearch
kibana

Logging

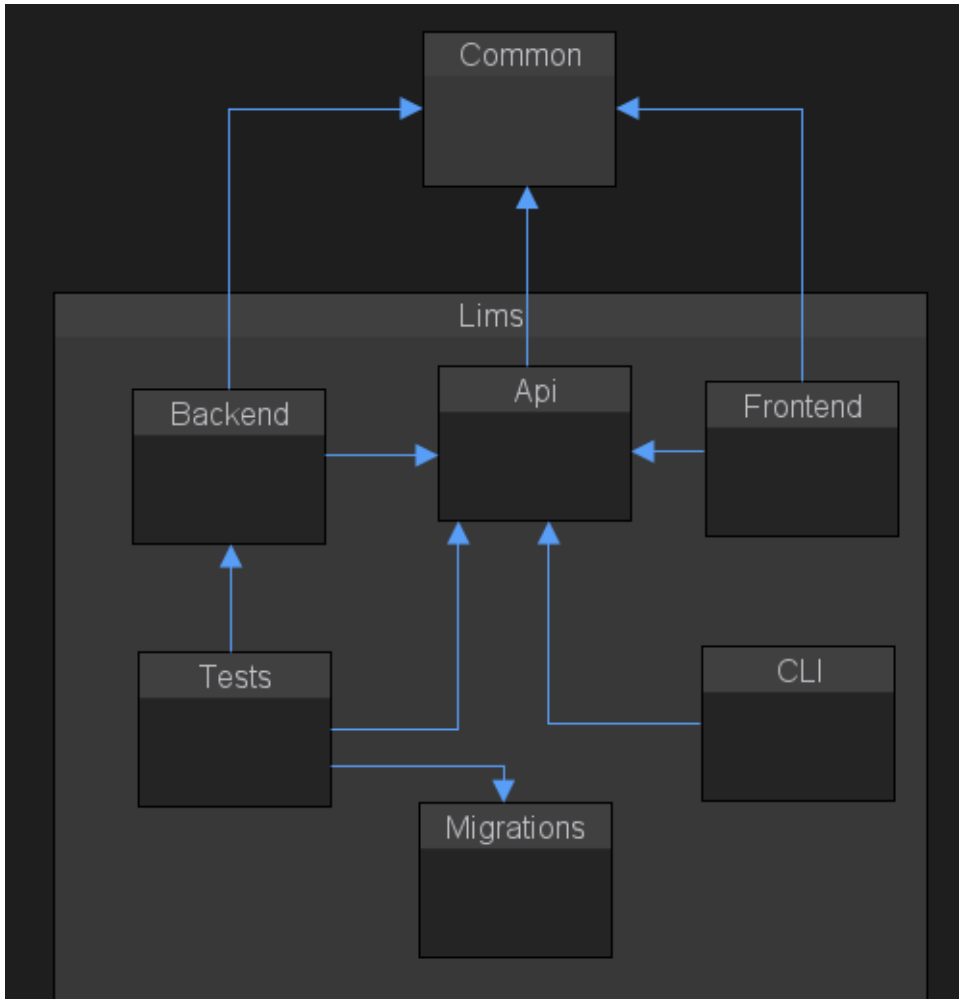
Demetrix software world



A bit about code structure



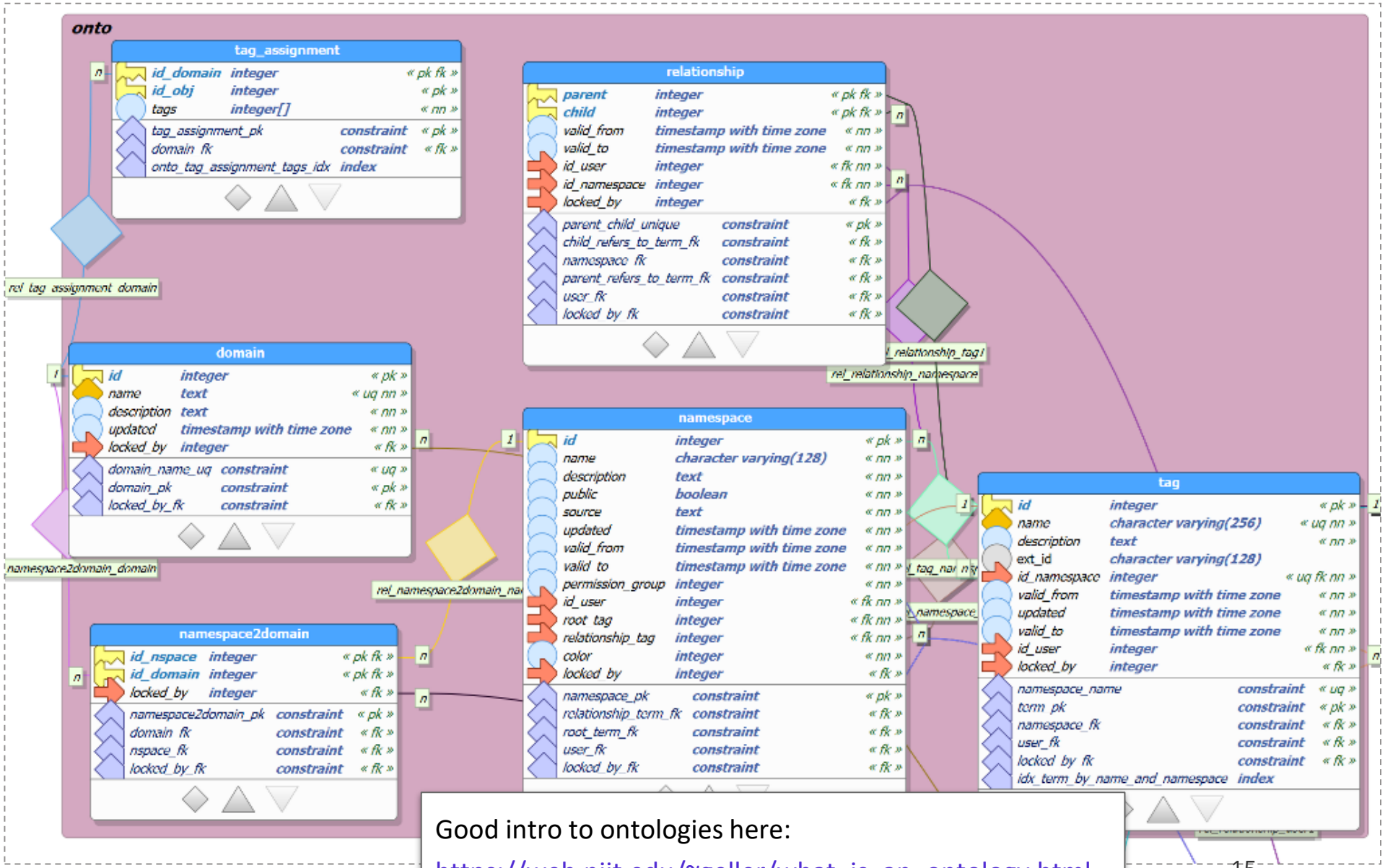
What common code sharing looks like for any of our apps



A look at an individual app and its code parts

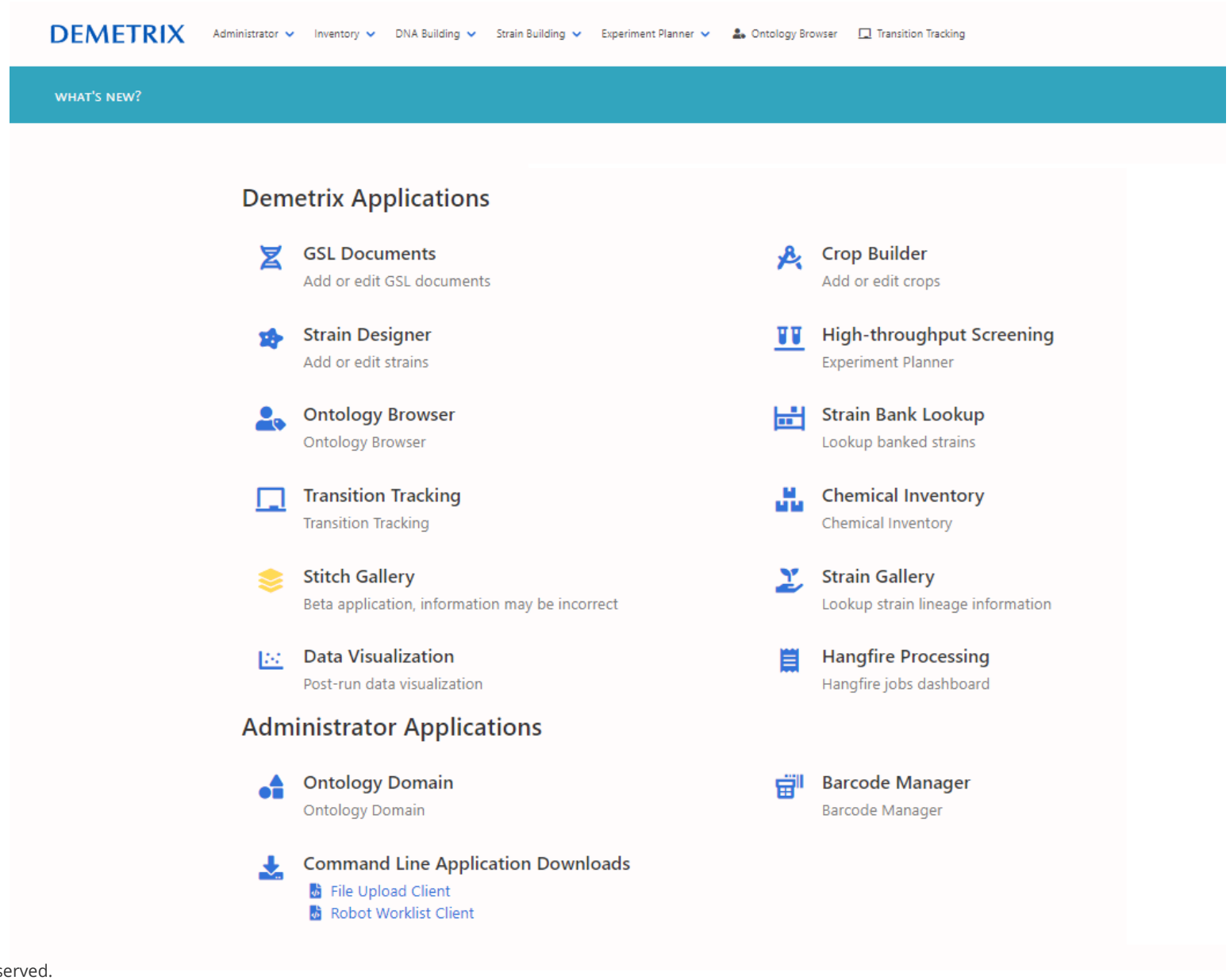
Ontologies to represent domain metadata

- vessel_type root tag
  - 250mL Bottle
  - 500uL Assay Plate
  - 50mL Conical Tube
  - Assay media
  - Bottle 500mL
  - PC media
  - PR media
  - Vial
- chemical_vendor
  - Demetrix Media Prep
  - Elga LabWater
  - Fisher Chemical
  - Sigma-Aldrich
  - VWR
- demetrix_lab
- equipment
  - LidValet
  - Microserve
  - freezer
- liquid_transferer
  - Echo
  - Stellaluna
  - Vantage
    - Babe
      - 8channel
        - channel1
        - channel2
        - channel3

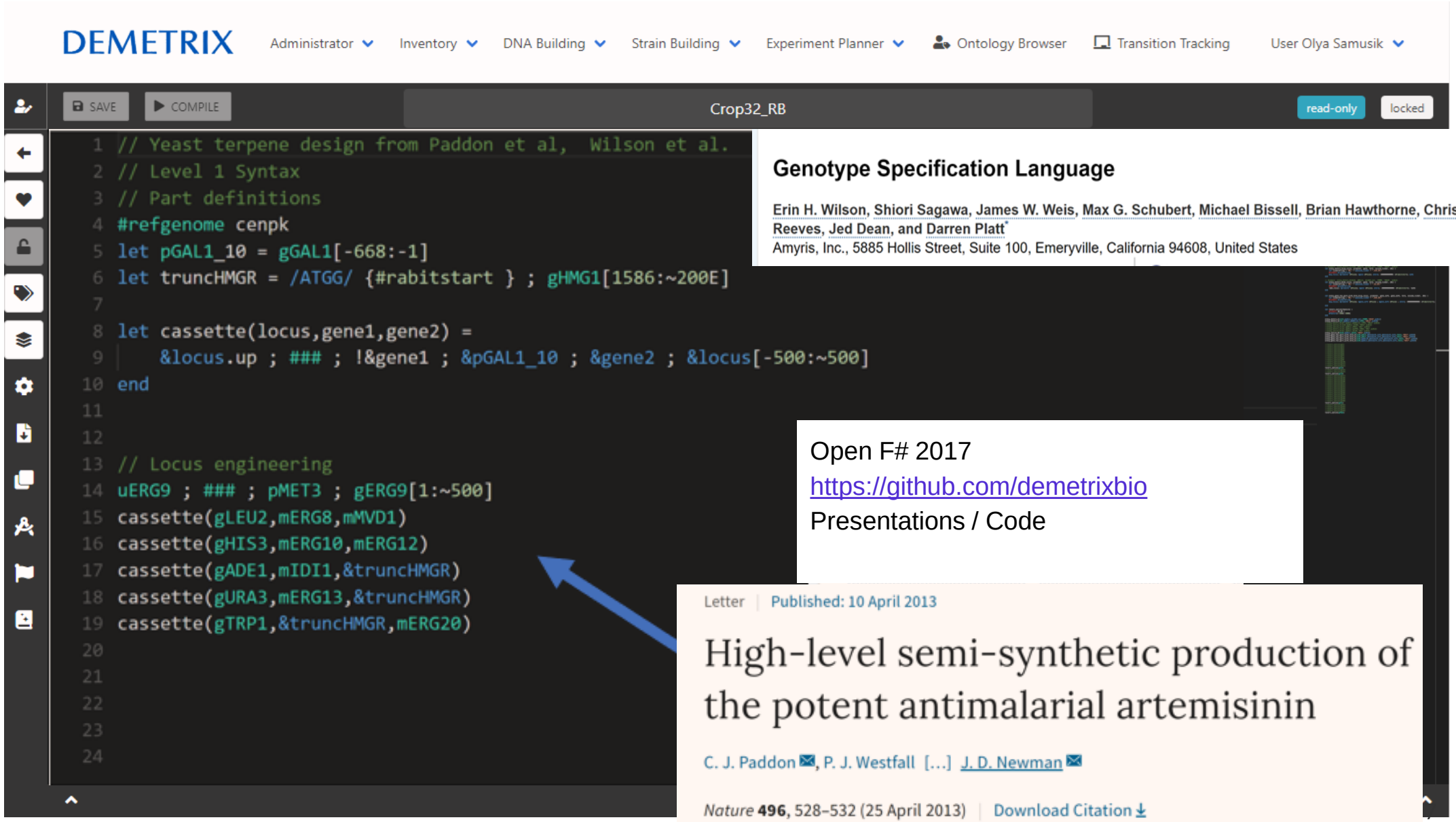


Good intro to ontologies here:
https://web.njit.edu/~geller/what_is_an_ontology.html

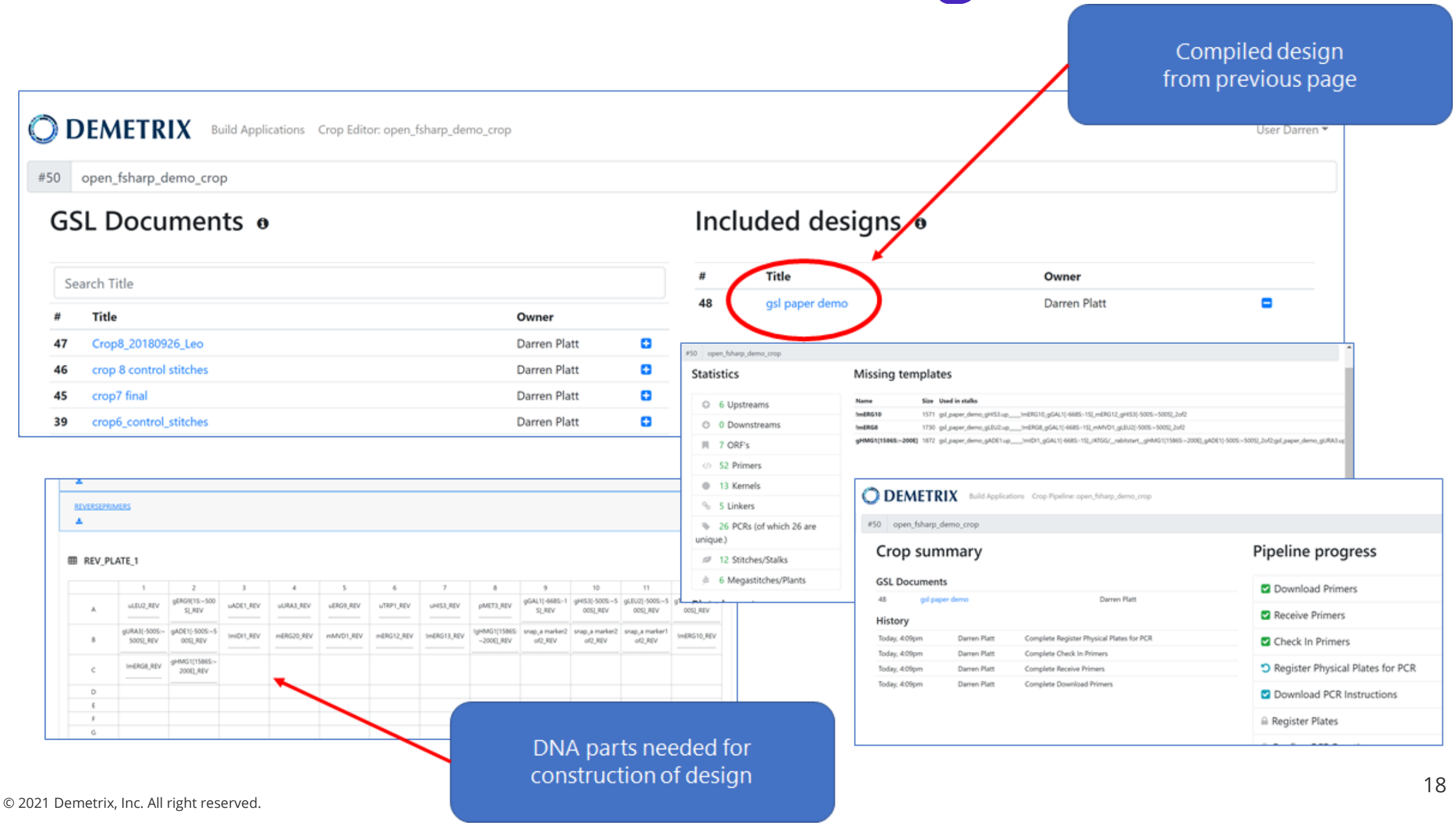
LIMS - a federated set of Fable applications



DNA Designer application

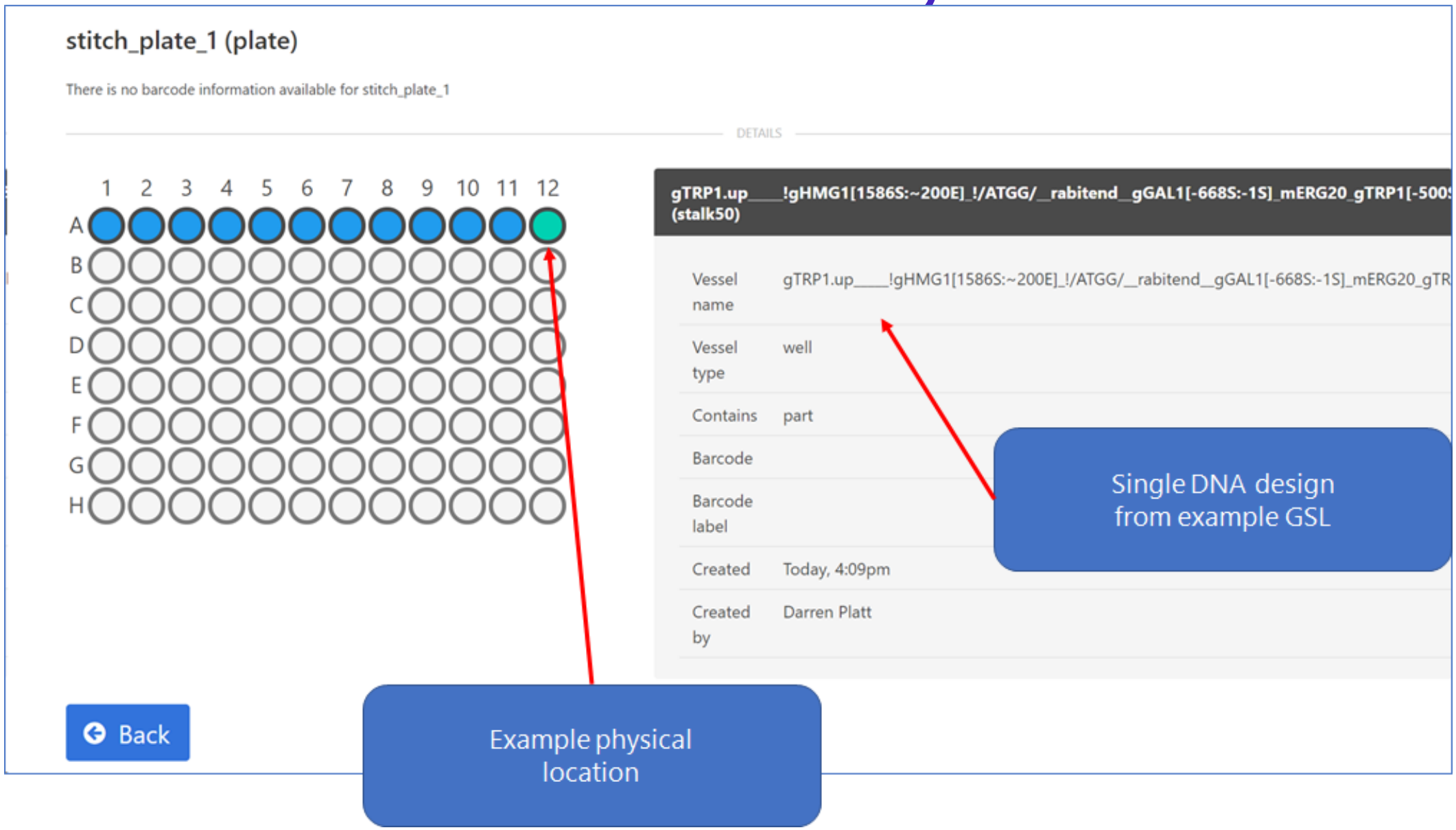


DNA Building

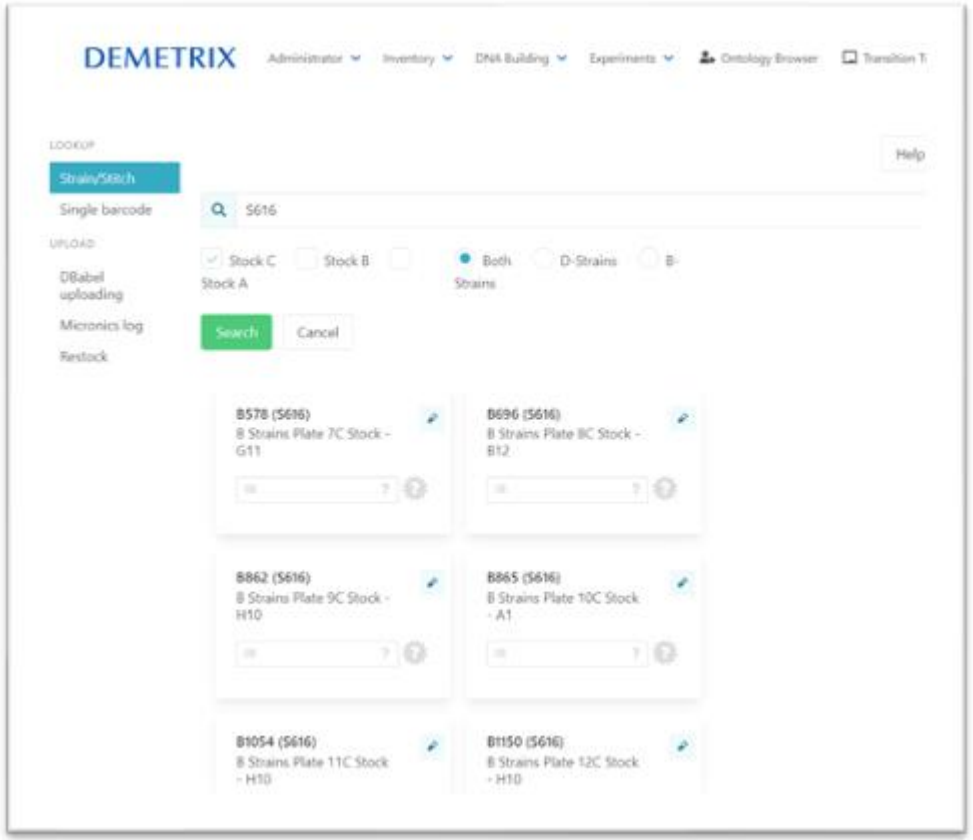


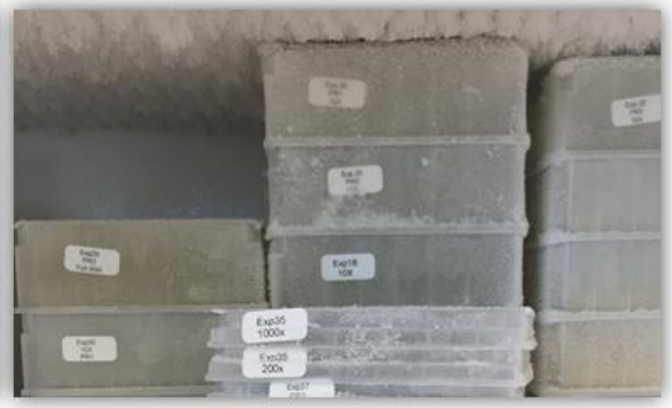
Constructing and then QC-ing DNA

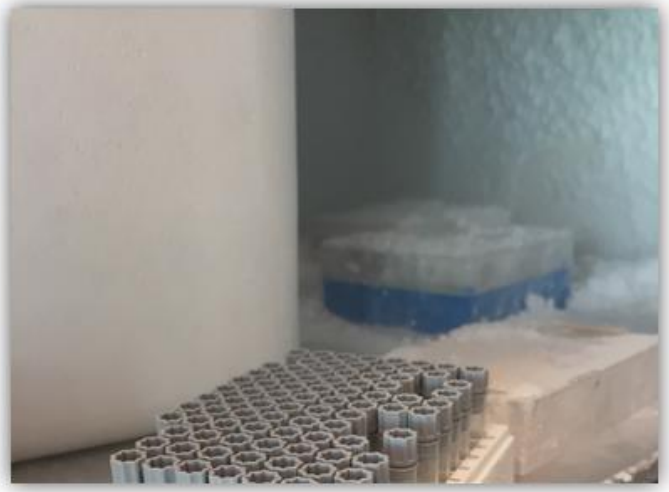
Inventory



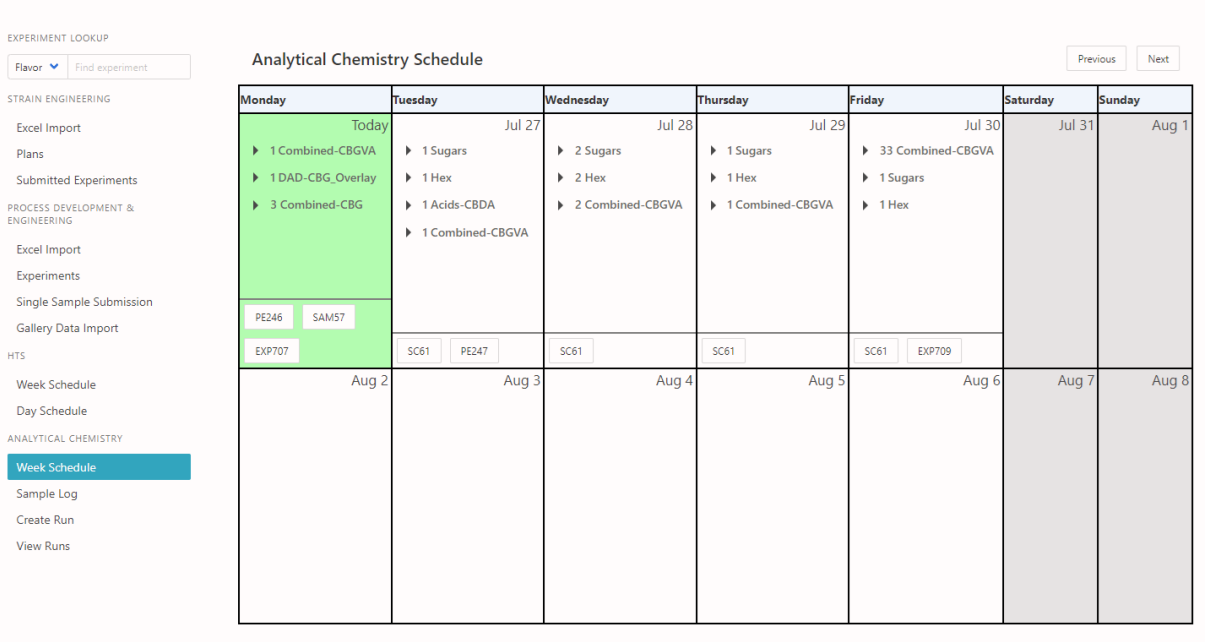
Virtual model of Physical World

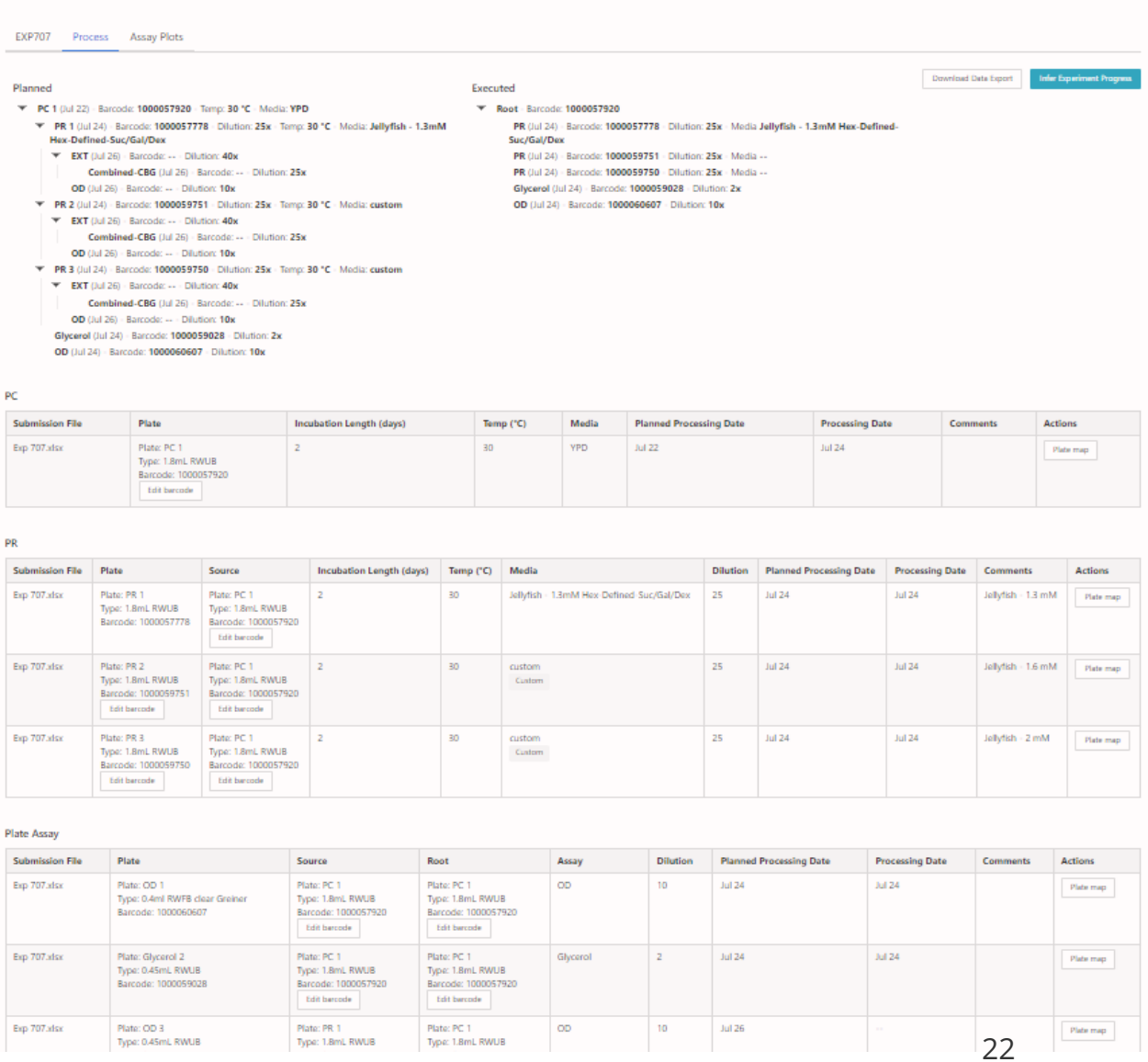




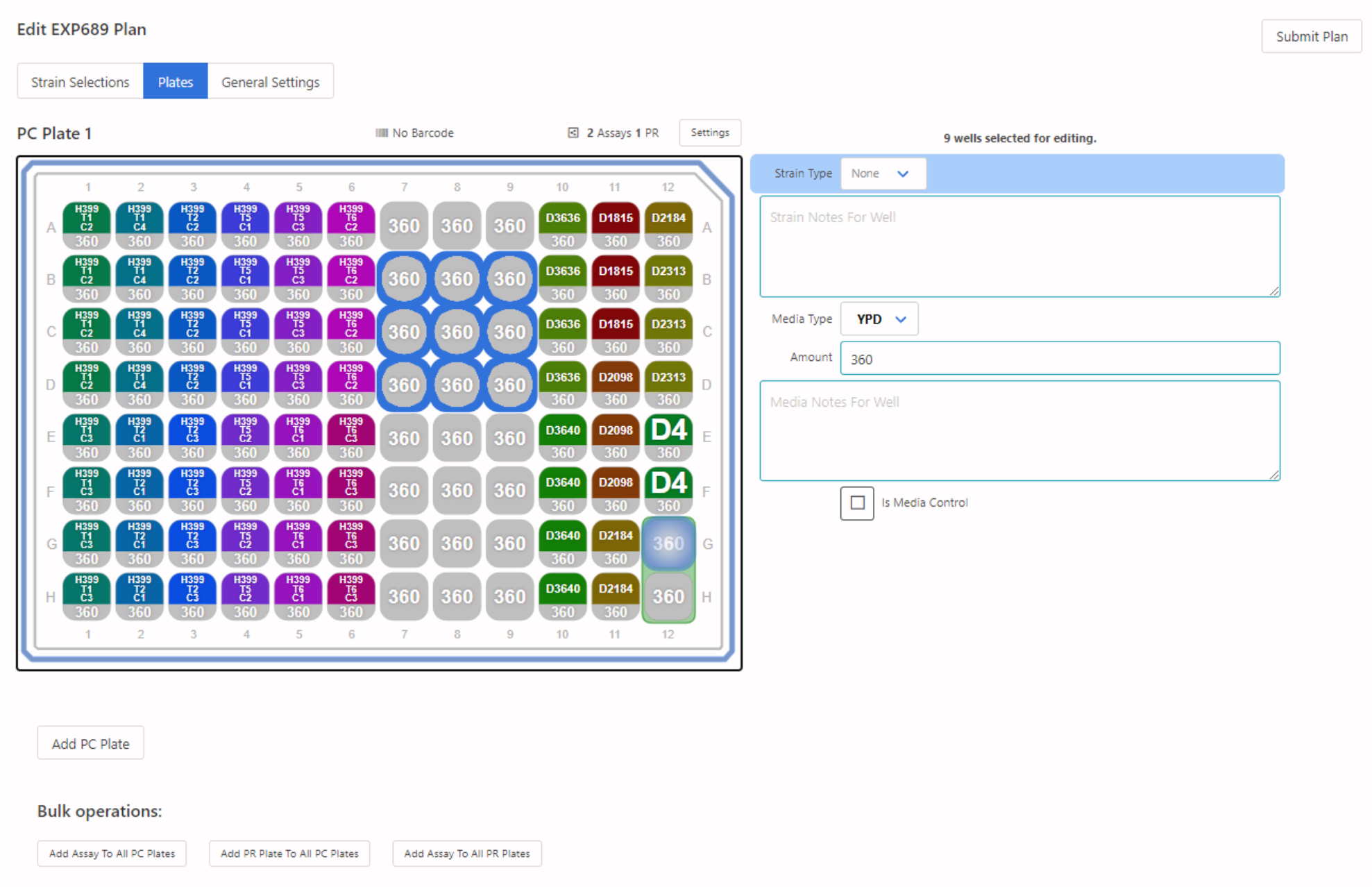


Experiment planning, human-robot coordination

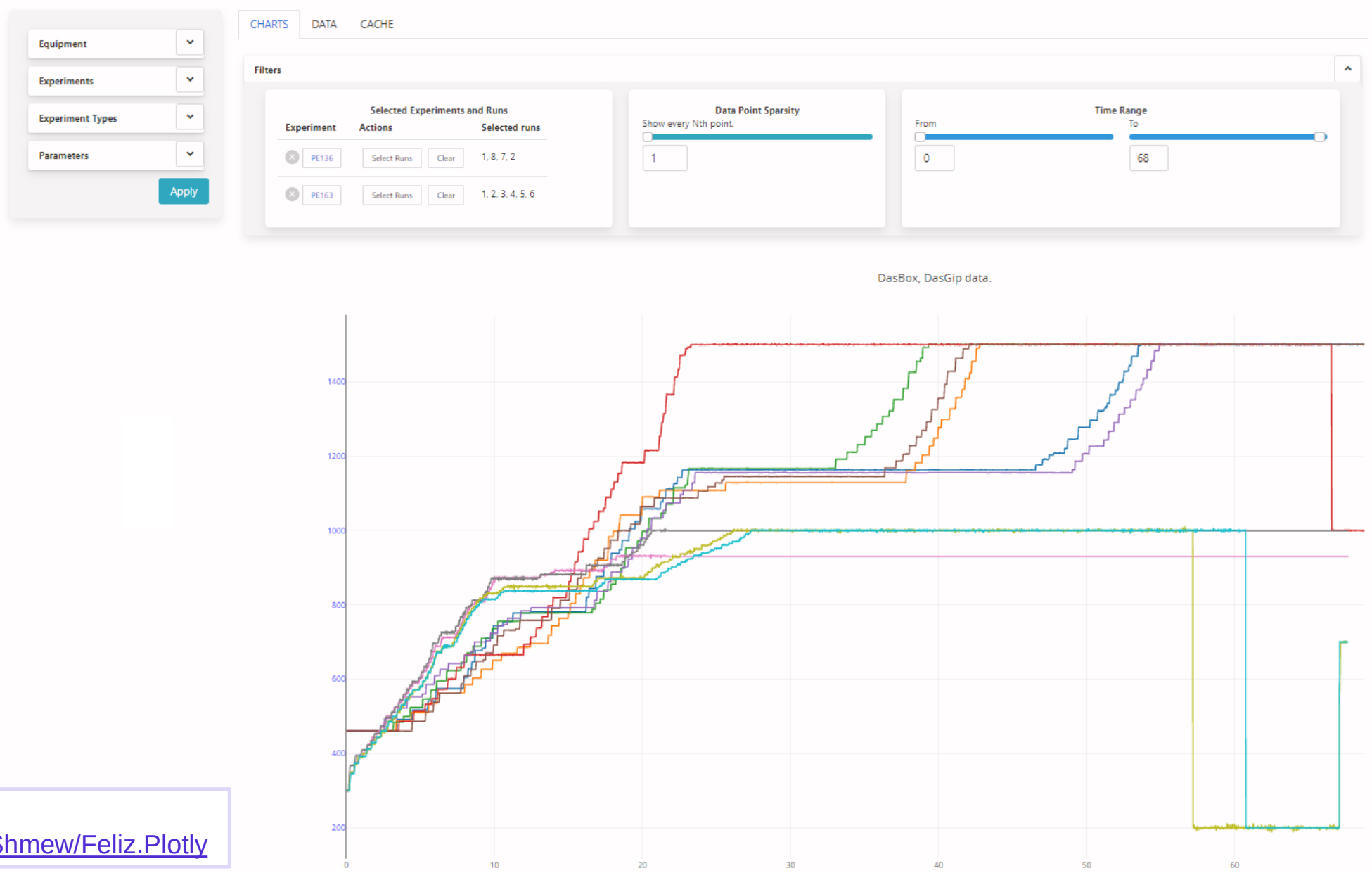




Complex user interactions - experiment designer

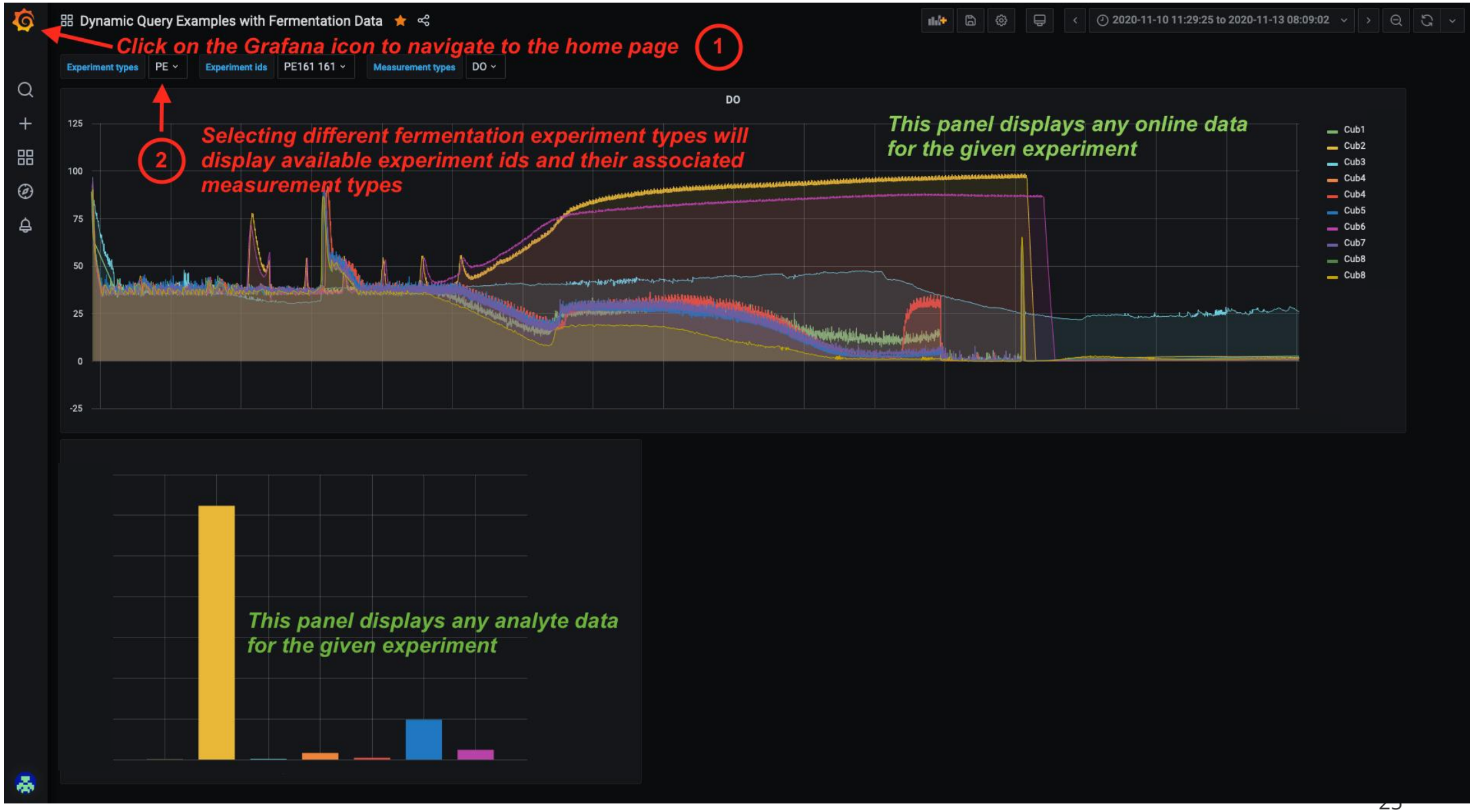


Experimental Data Visualization



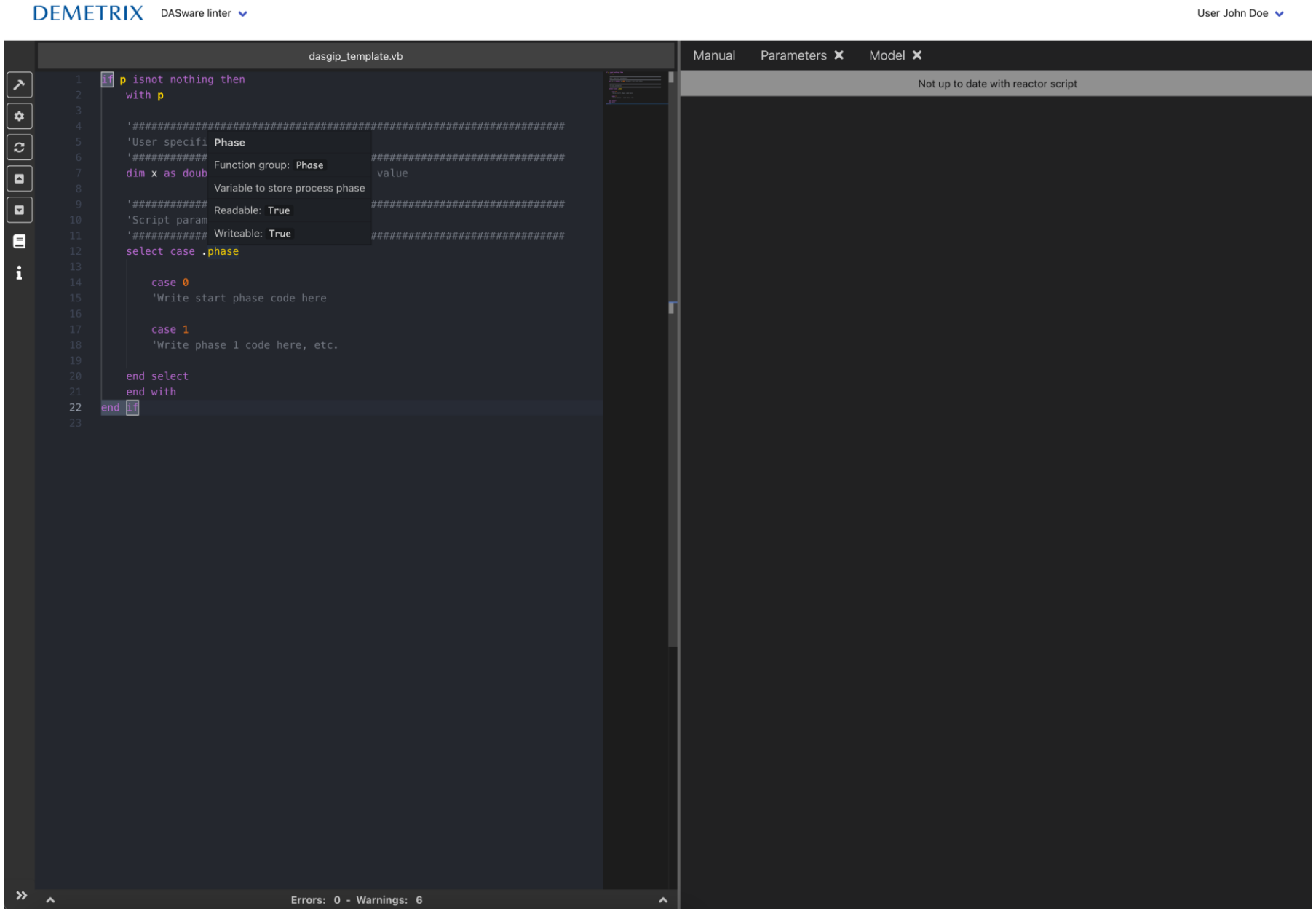
Feliz.Plotly
https://github.com/Shmew/Feliz.Plotly

Data Warehouse & Grafana



DASware Linter – makes reactor setup easier

- Syntax & sanity checking
- Hover provider
- Linter with superpowers: uses set points parsed from the script to predict titers using a deep learning model trained on previous fermentation runs



So... why F#?

What makes building software enjoyable?

- Easy domain modelling and data manipulations
- NOT spending most of my time on error hunting
- Focusing on solving business problems, not on boilerplate and ceremonies
- Convenient async and parallel programming
- Extensive, mature tooling and libraries for
  - talking to databases
  - dealing with various kinds of data
  - developing web applications
  - easier coding (e.g. IDE itself)
- Supportive tech community (especially if smaller company)

So... why F#?

Easy domain modelling and data manipulation

- Records (AND types)
- Discriminated Unions (OR-types)
- Single-case Unions
- Pattern matching
- Collection types (Sequence / List / Map / Array ...)
- Handy functions for collection processing
- Pipe operator! |>

```
[<Struct>]
type TagIdentifier = TagIdentifier of id : int
with
    int
    member this.Value with get() = let (TagIdentifier value) = this in value
module TagIdentifier =
    TagIdentifier -> int
    let get (TagIdentifier id) = id
```

3

```
module StageTopology =
    type Single =
        { VesselId : VesselId
          StageTypeTagId : TagIdentifier }

    type SingleWithChildren =
        { VesselId : VesselId
          StageTypeTagId : TagIdentifier
          Children : VesselId list }
```

1

```
let vesselTypes =
    if m.SelectedVesselTypes.IsEmpty then
        None
    else
        m.SelectedVesselTypes : Map<TagIdentifier,Tag>
        |> Seq.map (fun s -> s.Key) : seq<TagIdentifier>
        |> Array.ofSeq : TagIdentifier []
        |> fun s -> VesselType.Any(transitionRole, s |> Array.map TagIdentifier.get)
        |> Some
```

4

```
type Msg =
    | BarcodeInputMsg of SearchInput.Msg<Vessel>
    | VesselRetrieved of Vessel
    | RelatedExperimentsRetrieved of Experiment list
    | ShowError of exn
```

2

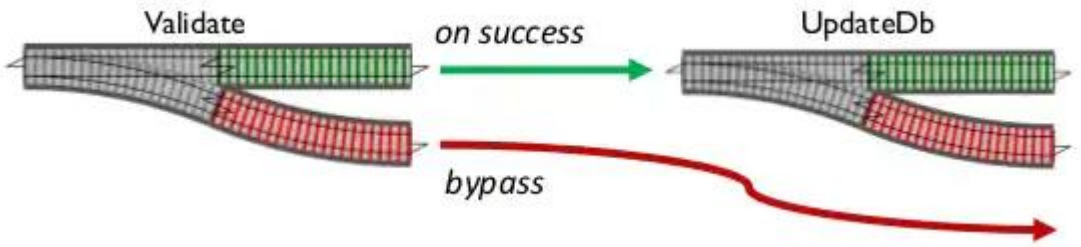
```
let executedTransferMap =
    plannedToExecutedAssayMap.Executed : Map<TransferPathKey,TransferPath>
    |> Seq.map (fun s -> s.Value) : seq<TransferPath>
    |> Seq.groupBy (fun s -> s.SourceBarcode) : seq<string * seq<TransferPath>>
    |> Seq.map (fun (key, rows) ->
        let byPath = rows |> Seq.map (fun s -> s.BarcodePath, s) |> Map.ofSeq
        key, byPath) : seq<string * Map<string [],TransferPath>>
    |> Map.ofSeq
```

5

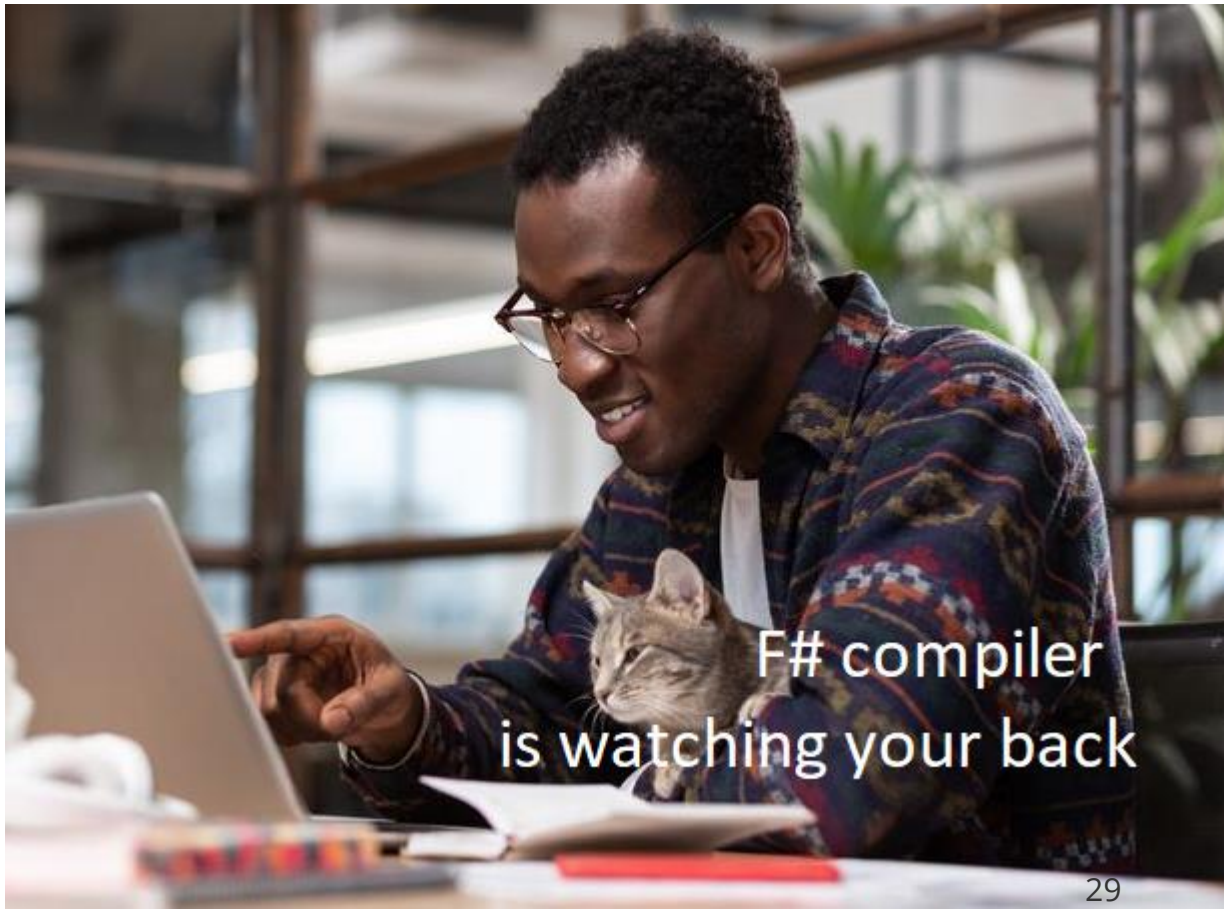
So… why F#?

NOT spending most of your time debugging and error hunting

- Smart compiler is your friend!
  - Static typing and type inference
- Immutability by default
- Option types



- Units of measure
- Railway Oriented Programming (ROP)



So... why F#?

Focusing on solving business problems, not on boilerplate and ceremonies

- ROP-style control flow
- Declarative nature of the language
- Terseness of the language

```
type FailureMessage =
    | Unknown of string
    | Database of string
    | Parse of string
    | Validation of string
    | Conflict of string
    | NotFound of string
    | ExceptionFailure of exn
    string
    member this.Message =
        match this with
        | Unknown s
        | Database s
        | Parse s
        | Validation s
        | Conflict s
        | NotFound s → s
        | ExceptionFailure e → e.Message

    FailureMessage -> string
    static member unwrap (failure : FailureMessage) =
        match failure with
        | FailureMessage.Database msg → msg
        | FailureMessage.Parse msg → msg
        | FailureMessage.Validation msg → msg
        | FailureMessage.Conflict msg → msg
        | FailureMessage.Unknown msg → msg
        | FailureMessage.NotFound msg → msg
        | FailureMessage.ExceptionFailure exn → exn.ToString()

type Success<'TSuccess> =
    { Data : 'TSuccess
      Warnings : string list }

type Either<'a> = Result<Success<'a>, FailureMessage>
```

```
ExperimentPK -> (unit -> 'a) -> Task<Either<Experiment>>
let private updateExperiment experimentId update =
    taskEither {
        do! ExperimentStorage.assertExperimentExists experimentId false : System.Threading.Tasks.Task<unit option>
            ▷ TaskEither.isSome (NotFound $"expId={experimentId.Value}")
        use scope = Db.createTransactionScope()
        do! update () ▷ TaskEither.ofTask
        let! experiment =
            ExperimentStorage.getExperimentById experimentId true : System.Threading.Tasks.Task<Experiment option>
            ▷ TaskEither.isSome (Validation $"expId={experimentId.Value}")
        scope.Complete()
        return experiment
    }
```

https://github.com/demetrixbio/Plough.ControlFlow

So... why F#?

Convenient async and parallel programming primitives

```
int -> SaveGSLDocument -> Task<Either<GSLSrcDocument>>
let saveGSLDocument userId cmd =
    taskEither {
        let (document, overwrite) =
            match cmd with
            | Strict doc → (doc, false)
            | Overwrite doc → (doc, true)

        let! validation = GSLDocumentValidationService.validate overwrite document userId

        let hashedContent = hashASCIIBySHA256 document.Content
        let doc =
            { document with
                LastUpdated = DateTime.Now
                HashedContent = hashedContent }

        return! GslDocumentCommandStorage.saveGSLDocument doc
    }
```

```
open System
open System.IO

let printTotalFileBytes path =
    async {
        let! bytes = File.ReadAllBytesAsync(path) |> Async.AwaitTask
        let fileName = Path.GetFileName(path)
        printfn $"File {fileName} has %d{bytes.Length} bytes"
    }

[<EntryPoint>]
let main argv =
    argv
    |> Seq.map printTotalFileBytes
    |> Async.Parallel
    |> Async.Ignore
    |> Async.RunSynchronously

    0
```

- Async.StartChild
- Async.Parallel
- Async.StartAsTask

...

[Async programming in F# (Microsoft Docs)]

[Writing Concurrent Programs Using F# Mailbox Processors]

https://github.com/fsprojects/FSharpx.Async

So... why F#?

Extensive, mature tooling and libraries

Data

- FSharp.Data (CSV, XML)
- Fsharp.Data.Npgsql – real time resolution of SQL / type issues (ours)
- Dapper (+ custom SQL query building)
- Dbup + custom wrapper for migrations

Web

- Giraffe / Asp.Net Core
- Fable
- Elmish
- Fulma
- SAFE stack
- Thoth
- Feliz.Plotly

Infra

- Paket
- FAKE
- Hangfire – routine batch computing
- Nunit - testing
- Argu - dealing with CLI arguments

Misc

- Jupyter Notebooks + IFsharp

IDEs

- VS Code + Ionide
- JetBrains Rider

Supportive community

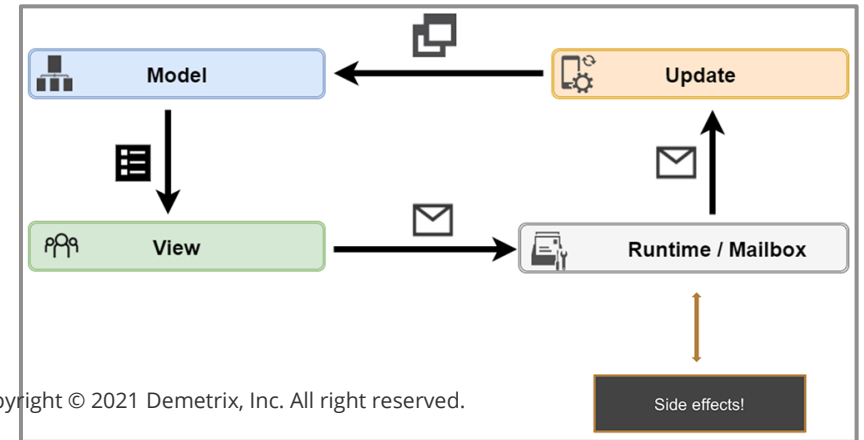
- F# Twitter helps to solve both tech and employment problems
- F# Slack - very welcoming, helps F# beginners to learn and strive
- Friendly OSS project maintainers
- F# mentorship program!

Elmish 'Model-View-Update' example

```
type Model =
    { Orientation : Orientation
      FileName : string
      Content : string
      ValidationStatus : Validation.State
      ValidationMessage : string option }

type ExternalMsg =
    | SaveFileContent of string * string
    | NoOp

type Msg =
    | SaveContent of name : string * content : string
    | ErrorReadingFile
```

```
Orientation -> Model
let init orientation =
    {
        Orientation = orientation
        FileName = ""
        Content = ""
        ValidationStatus = Validation.NotTriggered
        ValidationMessage = None
    }
```

```
Msg -> Model -> Model * Cmd<'a> * ExternalMsg
let update msg model =
    match msg with
    | SaveContent (name, content) →
        { model with FileName = name
                     Content = content
                     ValidationStatus = Validation.NotTriggered
                     ValidationMessage = None }, Cmd.none, SaveFileContent(name, content)

    | ErrorReadingFile →
        model, Cmd.none, NoOp
```




```
Model -> (Msg -> unit) -> ReactElement
let view model dispatch =
    Text.div [] [
        Fulma.File.file [ if model.FileName <> "" then yield Fulma.File.HasName
                          if model.Orientation = Vertical then yield Fulma.File.IsBoxed
                          yield Fulma.File.Size IsSmall
                          if model.ValidationStatus = Validation.Failed then yield Fulma.File.Color IsDanger ] [
            Fulma.File.label [ ] [
                yield Fulma.File.input [ GenericOption.Modifiers [ Modifier.BackgroundColor IsBlack; Modifier.TextColor IsDark ]
                                         GenericOption.Props [ OnInput (fun evt →
                                            let file = (evt.target :?> HTMLInputElement).files.[0]
                                            let name = file.name
                                            let reader = FileReader.Create()
                                            reader.onload ← fun _ →
                                                let content = reader.result :?> string
                                                (name, content) ▷ SaveContent ▷ dispatch

                                            reader.onerror ← (fun _ → dispatch ErrorReadingFile)

                                            reader.readAsText file) ] ]
                yield Fulma.File.cta [ ]
                    [ Fulma.File.icon [ ] [ Icon.icon [ ] [ Fa.i [ Fa.Solid.Upload  ] [ ] ] ]
                      Fulma.File.label [ ] [ str "Choose a file ..." ] ]
                if model.FileName <> "" then yield Fulma.File.name [ ] [ str model.FileName ]
            ]
        ]
```

Web server & API example

```fsharp
type Api(client : ApiClient) =
    OntologyNamespaceIdentifier -> TaskEither<NamespaceAndTags>
    member x.GetNamespaceById (OntologyNamespaceIdentifier namespaceId) : TaskEither<NamespaceAndTags> =
        client.Get <| sprintf "/api/onto/namespace/%i" namespaceId

    CreateNamespace -> TaskEither<NamespaceAndTags>
    member x.CreateNamespace (cmd : CreateNamespace) : TaskEither<NamespaceAndTags> =
        client.Post("/api/onto/namespace/new", cmd)
```

Thin API Proxy layer

https://github.com/demetrixbio/Plough.WebApi

```fsharp
unit -> HttpFunc -> HttpContext -> HttpFuncResult
let api () = subRoute "/onto" <| choose [
    GET >=> requiresScope AccessScope.OntologyRead >=> choose [
        route "/namespace/all" >=> makeJSONHandlerAsync<NamespacesAndMasterTags> getNamespacesAndMasterTags
        routef "/namespace/%i" (makeJSONHandlerWithArgAsync<int, NamespaceAndTags> getNamespaceAndTags)
        routef "/namespace/name/%s" (makeJSONHandlerWithArgAsync<string, NamespaceAndTags> getNamespaceAndTagsByName)
        route "/domain" >=> makeJSONHandlerAsync<Domain list> getDomains
        route "/domain/all" >=> makeJSONHandlerAsync<OntologyDomain list> getOntologyDomains
        routef "/domain/name/%s" (makeJSONHandlerWithArgAsync<string, OntologyDomain> getOntologyDomainByName)
        routef "/domain/%s/namespaces" (makeJSONHandlerWithArgAsync<string, DomainForTagAssigment> getDomainNamespacesByName)
        routef "/tagassignment/domainnameandobj/%s/%i" (makeJSONHandlerWithArgAsync<string * int, TagAssignment> getAssignedTagsForDomainNameAndObj)
        routef "/tagassignment/domainandobj/%i/%i" (makeJSONHandlerWithArgAsync<int * int, TagAssignment> getAssignedTagsForDomainAndObj)
        routef "/tagassignment/domainname/%s" (makeJSONHandlerWithArgAsync<string, TagAssignment list> getAssignedTagsForDomainName)
    ]
    POST >=> requiresScope AccessScope.OntologyWrite >=> choose [
        route "/namespace/new" >=> makeJSONHandlerWithObjAsync<CreateNamespace, NamespaceAndTags> createNamespace
        route "/namespace/edit" >=> makeJSONHandlerWithObjAsync<EditNamespace, Namespace> editNamespace
        route "/namespace/delete" >=> makeJSONHandlerWithObjAsync<DeleteNamespace, OntologyNamespaceIdentifier> deleteNamespace
        route "/namespace/retire" >=> makeJSONHandlerWithObjAsync<RetireNamespace, Namespace> retireNamespace
        route "/tag/new" >=> makeJSONHandlerWithObjAsync<CreateTag, Tag> createTag
        route "/tag/edit" >=> makeJSONHandlerWithObjAsync<EditTag, Tag> editTag
```

Dispatching endpoints + authorization + JSON wrapping

Interacting with our Database

Fsharp.Data.Npgsql

```
let getPointStructuresByNames (psNames : string []) =
    task { // like C# await
        use! connection = Db.openConnectionAsync()
        use getPointStructureIds = Db.CreateCommand<"
            SELECT
                p.id, p.name, p.component_types, p.component_units, p.id_user, p.created, p.updated,
                u.alias_name AS u_alias_name
            FROM measure.point_structure p
            INNER JOIN admin.user u ON p.id_user = u.id
            WHERE name = ANY(@names)", SingleRow = false>(connection)
        let! result = getPointStructureIds.TaskAsyncExecute(names = psNames)

        return result : List<FSharp.Data.Npgsql.NpgsqlConnection<...>.Commands.CreateCommand<...>.Record>
               |> Seq.map (fun s ->
                   let componentTypes = s.component_types
                   let componentUnits = s.component_units
                   let compDescriptions =
                           Array.zip componentTypes componentUnits : (int * int) []
                           |> Array.map (fun c ->
                                       let (cType, cUnits) = c
                                       { Type = cType |> TagIdentifier
                                         Units = cUnits |> TagIdentifier })
                   { Id = s.id
                     Name = s.name
                     Components = compDescriptions
                     Owner = { Id = s.id_user; GivenName = s.u_alias_name }
                     Created = s.created
                     Updated =  s.updated }) : seq<PointStructure>
               |> Seq.toList
    }
```

Dapper + QueryBuilder

```
let getChemicalPageByQuery (query : ChemicalPredicate Filter) pageIndex pageSize =
    task {
        let state : QueryInterpreter.State =
            { Template = chemicalPageSqlTemplate
              InnerJoins = Dictionary()
              LeftJoins = Dictionary()
              Parameters = Dictionary() }

        let mapSimple state = function
            | ShortName operation → mapInvariantString state operation "c.short_name"
            | InChI operation → mapInvariantString state operation"c.inchi_key"
            | IUPAC operation → mapInvariantString state operation "c.iupac_name"
            | Formula operation → mapInvariantString state operation "c.formula"

        let pageParam = [ "offset", pageIndex * pageSize |> box
                          "limit", box pageSize ] |> dict

        let select =
            QueryInterpreter.mapToSql query state mapSimple : SqlBuilder
            |> SqlBuilder.parametrized pageParam : SqlBuilder
            |> SqlBuilder.build

        let chemicalCountSql = sprintf "SELECT COUNT(1) FROM (%s) AS t;" select.Sql
        let chemicalQuerySql = sprintf "%s ORDER BY c.id DESC OFFSET @offset LIMIT @limit;" select.Sql

        use! connection = Db.openConnectionAsync()
        let! count = connection.QuerySingleAsync<int64>(chemicalCountSql, select.Parameters)
        let! chemicals = connection.QueryAsync<ChemicalProjection>(chemicalQuerySql, select.Parameters)

        return { Items = chemicals : IEnumerable<ChemicalProjection>
                         |> Seq.map (fun s →
                             { Chemical.Id = s.Id
                               InChI = s.InChI |> Option.ofObj
                               IUPAC = s.IUPAC |> Option.ofObj
                               Formula = s.Formula |> Option.ofObj
                               Description = s.Description |> Option.ofObj
                               State = s.State |> Option.ofObj |> Option.map Storage.stringToMatterState |> Option.get
                               ShortName = s.ShortName
                               SynonymGroup = s.SynonymGroup }) : seq<Chemical>
                         |> List.ofSeq
                 PageSize = pageSize
                 PageIndex = pageIndex
                 TotalItemCount = Some count }
    }
```

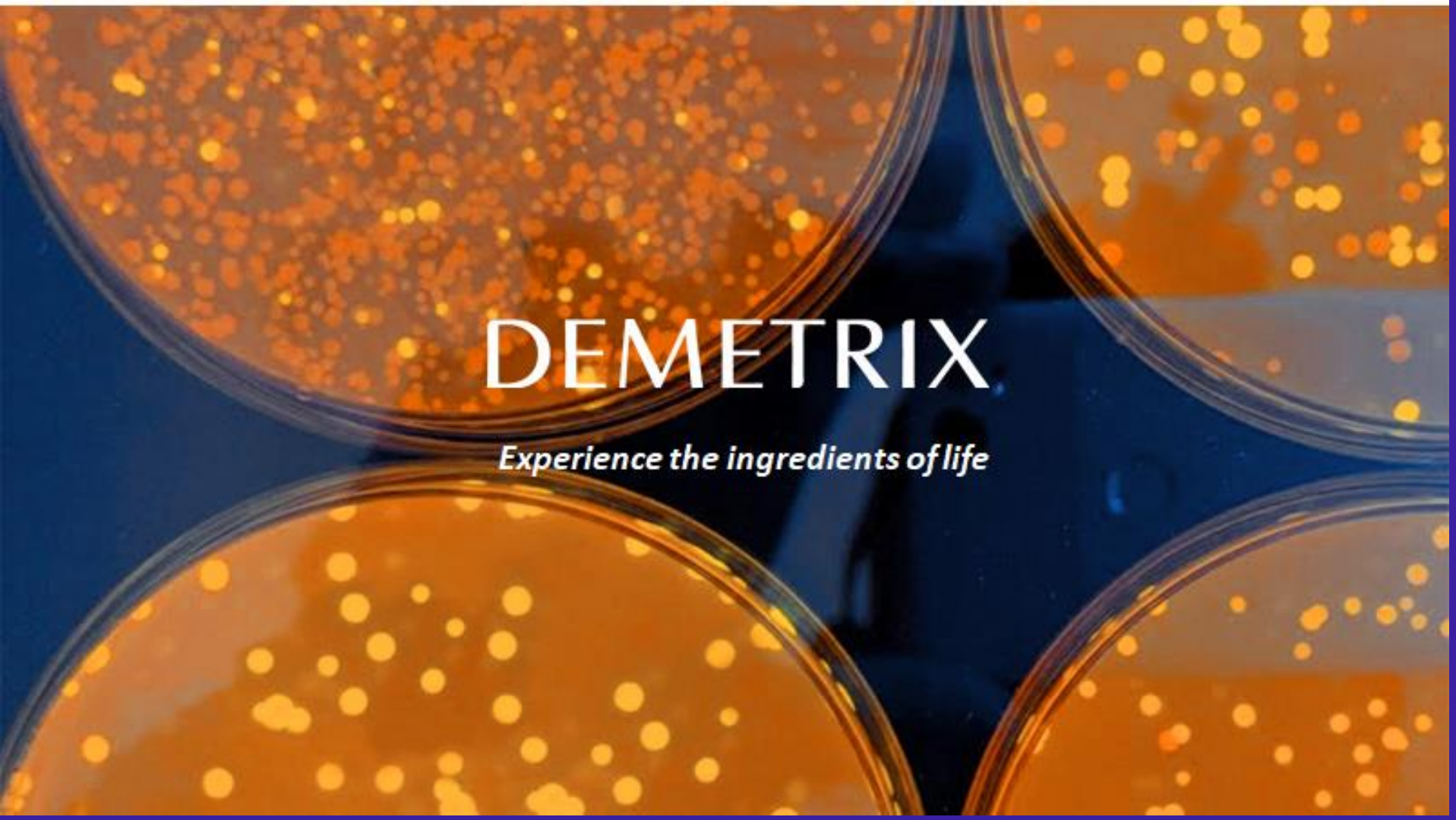
https://github.com/demetrixbio/FSharp.Data.Npgsql
https://github.com/demetrixbio/Plough.DynamicQuery

Room for improvement?

- Native task {} support with state machines
  - (fsharp/fslang-suggestions/581) (https://github.com/dotnet/fsharp/pull/6811)
  - We rely a lot on computational expressions
  - Async / Task heterogeneity
  - Excited about work that's going on to improve this whole area
- Tooling performance something we watch (and push) – shout out to all the IDE maintainers
- Native JS + Elmish model friction around state
- Scripting seems harder than it should be still (e.g. dependency loading) but that's improving
- Jupyter F# experience still not as good as Python but improving
- For novices: challenging to set things up and get started, documentation inconsistencies

Conclusions

- Biotech is a tough proving ground for a programming environment
- F# has been a great fit for developing software to manage our lab, both backend and frontend
- F# helps to handle complex, changing requirements gracefully, reduces code refactoring-PTSD
- Result is compact, type-safe, reliable code, quickly delivered that is maintainable in face of rapidly evolving requirements
- Learning curve is a bit steep for novices but manageable -- interns and scientists keep making valuable contributions to Demetrix's software



Thanks for joining!

Ask your questions live on Twitter #dotNETConf

Darren's minimal shopping list for a programming language

• Strongly typed	- reduces errors
• Semantic whitespace	- reduces errors
• Functional first	- reduces errors
• Option types	- reduces errors
• Units of measure	- reduces errors
• Type inference	- reduces RSI, makes #1 palatable
• Static typing	- fast
• Good parallel primitives	- lazy fast
• Rich built in data types	- reduces development time
• Type providers	- reduces development time
• Open source	- reduces dependency
• Decent library support	- don't want to roll my own db driver
• Language interop (*)	- increases library support
• Mature tooling	- make CPU work for developer
• Supportive community	- life is too short to hang out with jerks

Past Demetrix presentations in the F# world

Darren Platt
President and Head of Computing

2020

F# for Biotech — Stranger Things edition

Darren Platt

Biotech is a broad and demanding field for software development. It combines a messy domain, rapidly changing requirements and brutal timelines for software development that stress the software development life cycle. In the middle of this controlled software explosion, we have developed a platform using F# starting from a compiler that targets life itself, through to data management systems overseeing millions of liquid transfers a year. The end product of the lab engineering is a living organism (yeast based) that can manufacture the active ingredients in the Cannabis plant via fermentation. This talk will give a user-focused view into the types of problems we have to solve, and why F# and Fable have been good companions on this journey. Also why you don't want to be the only BASIC programming in a lab full of roaming monsters.

2018

A YEAR IN PARADISE. FULL STACK F# FOR SYNTHETIC BIOLOGY

THU 27TH - 12:20 PM (TALK)

We are now about one year into implementing a largely F# environment for designing, building and analyzing engineered microorganisms. This is cloud deployed full stack F# (Fable, React, Elmish, Giraffe, Postgres) with algorithms for designing DNA, and web based systems for controlling robots and managing a lab. I will talk about what has worked well and where there are rough spots. I will touch on some areas of biology but it will be mostly about implementing a system for managing a complex manufacturing environment that is widely applicable to many industries.

2017

GSL: A COMPILER TOOL-CHAIN WHOSE TARGET ARCHITECTURE IS LIFE ITSELF

At Amyris, we are taking advantage of rapid advances in automation, biology and computing to engineer microorganisms into platforms that efficiently manufacture molecules for a wide range of industrial, consumer and pharmaceutical applications. We are currently working on hundreds of molecules and as the rate of engineering has increased, biologists are facing many of the same issues faced by early software pioneers. We have been introducing many software practices to the biology community at Amyris to increase efficiency and accuracy of genetic designs.

Kate Curran
Associate Director, Automated Research Operations

2019

LEARNING F# ON THE JOB AS A PROGRAMMING NOVICE

THURSDAY, SEPTEMBER 26 @ 16:00

Demetrix is a biotech start-up built on Fsharp infrastructure. We have some great Fsharp developers along with a lot of scientists and engineers that are relatively new to programming. I'm in the latter camp. I'll give some impressions and lessons learned from those of us learning F# as our first programming language.

https://github.com/demetrixbio/presentations