



What's Old is New Again!

Adventures in Metaprogramming with Scalameta

Chicago Area Scala Enthusiasts
June 25, 2020

<https://github.com/eric-fredericks/floppy-demo>



Eric J. Fredericks

*Senior Principal Engineer
Rally Health (Chicago)*

@k-lergic (slack) ~ eric.fredericks@gmail.com



We put health in the hands of the
individual

Rally Health is hiring!



Who is Eric?

Hometown: Guilderland, New York (Albany)

Software Engineer at Rally Health (Chicago)

20+ years experience in Software Engineering in many different business contexts



Interests and Goals:

Scala is now my “native language”

(it used to be C, then C++, then, Java...)

Get more functional!

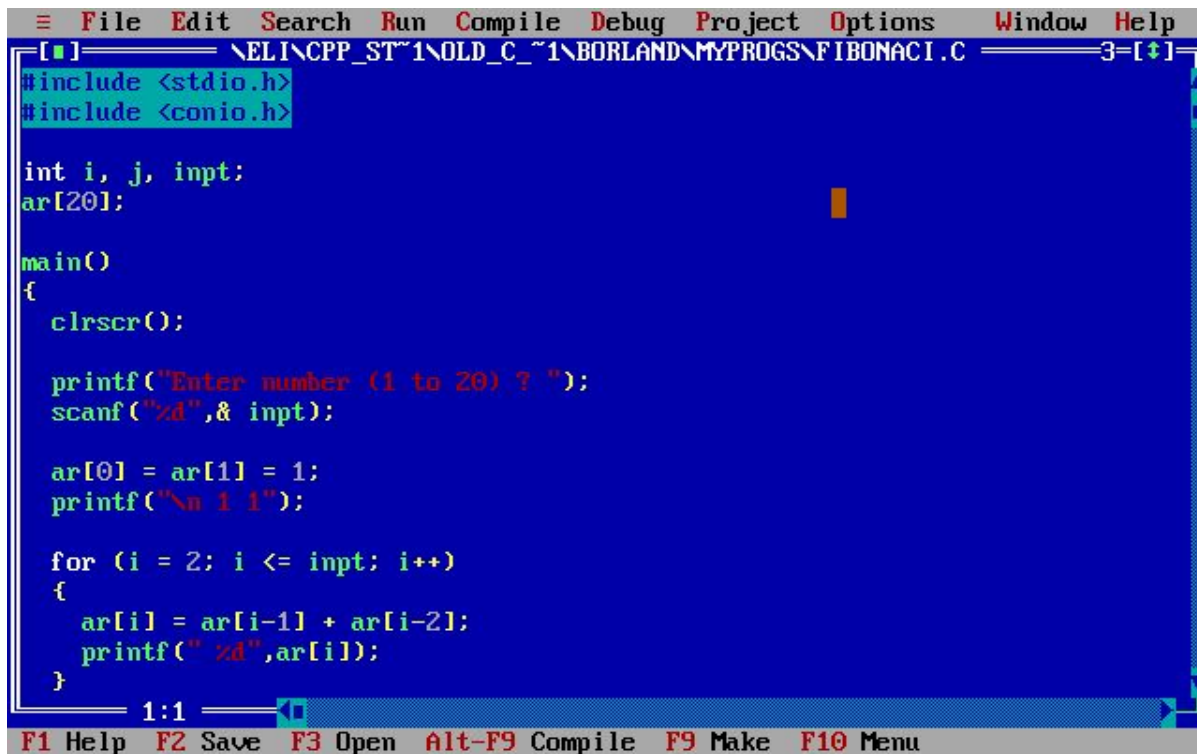
Enable teams to do things faster

*Also: I’m known at Rally for not caring for
Rally’s favorite ~~superfood~~....*



Who is Eric?

My first IDE (1993)

A screenshot of the Borland C++ IDE from 1993. The window title is "\ELI\CPP_ST~1\OLD_C_~1\BORLAND\MYPROGS\FIBONACI.C". The menu bar includes File, Edit, Search, Run, Compile, Debug, Project, Options, Window, and Help. The code editor has a blue background with text in various colors: preprocessor directives and keywords in green, identifiers and literals in red, and operators and punctuation in cyan. The code is a Fibonacci sequence calculator. The status bar at the bottom shows "1:1" and function key shortcuts: F1 Help, F2 Save, F3 Open, Alt-F9 Compile, F9 Make, and F10 Menu.

```
File Edit Search Run Compile Debug Project Options Window Help
\ELI\CPP_ST~1\OLD_C_~1\BORLAND\MYPROGS\FIBONACI.C 3=1
#include <stdio.h>
#include <conio.h>

int i, j, inpt;
ar[20];

main()
{
    clrscr();

    printf("Enter number (1 to 20) ? ");
    scanf("%d",& inpt);

    ar[0] = ar[1] = 1;
    printf("\n 1 1");

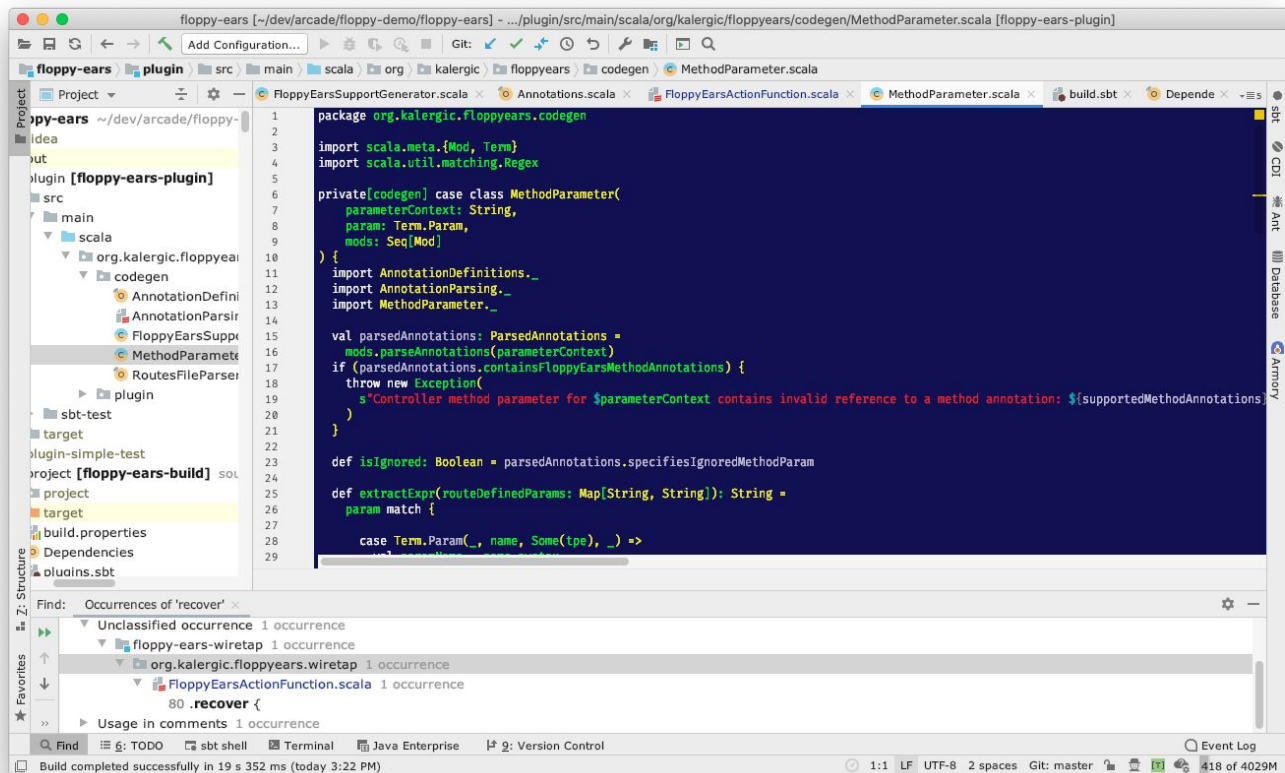
    for (i = 2; i <= inpt; i++)
    {
        ar[i] = ar[i-1] + ar[i-2];
        printf(" %d",ar[i]);
    }
}
```

1:1

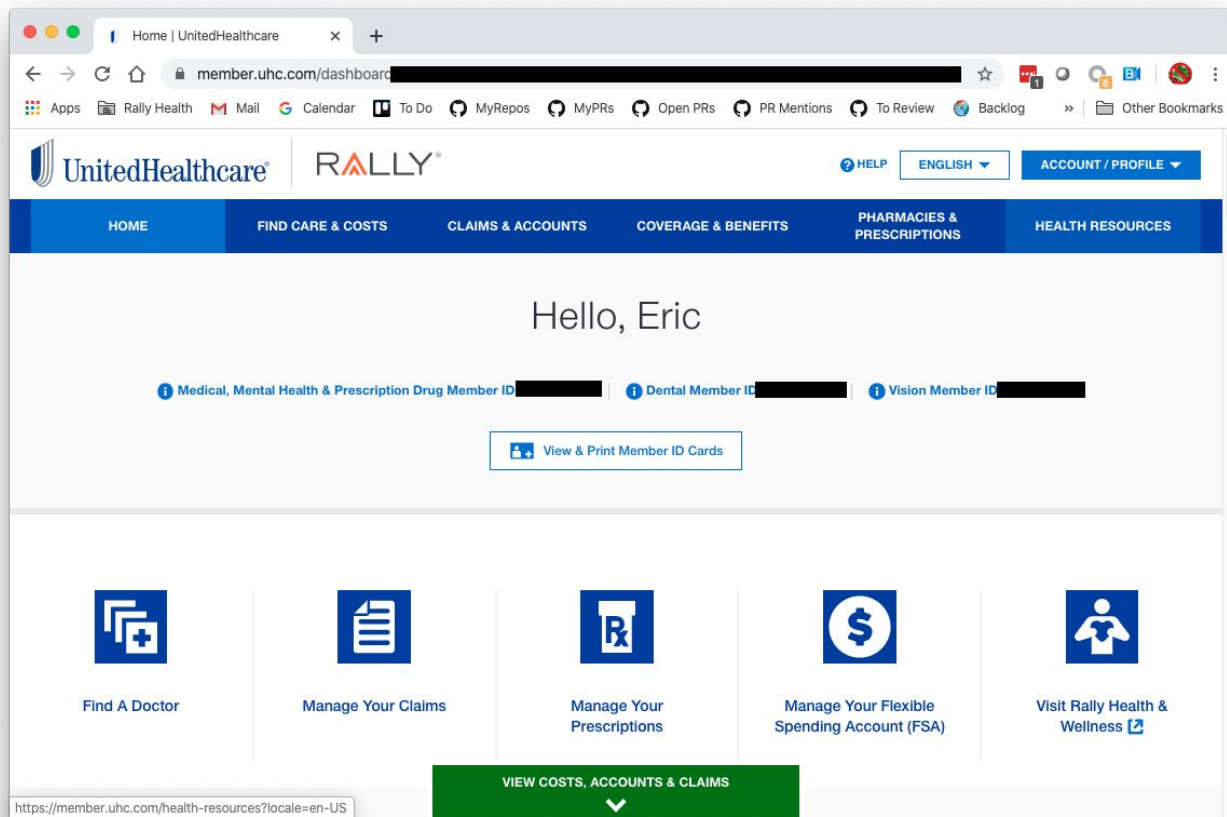
F1 Help F2 Save F3 Open Alt-F9 Compile F9 Make F10 Menu

Who is Eric?

My current IDE:



Rally Product: Digital Health Plan (look familiar?)



Data Science and Data Collection

Questions we ask:

- How do consumers interact with our products?
- What product features do consumers use?
- How do consumers use our product features?
- ... *and probably others*

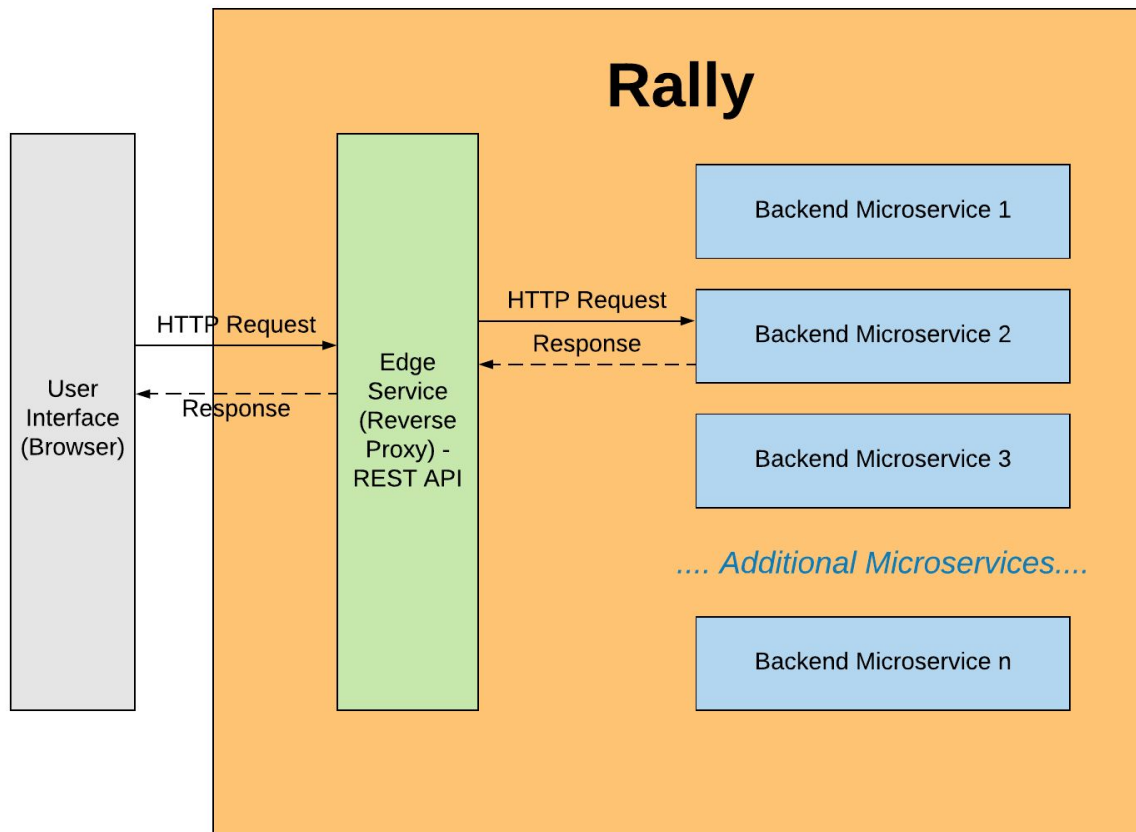
This translates (in a lot of cases) to:

- Where do consumers click in the UI / SPA?
- Which service endpoints does the UI call on our consumers' behalf?
- What data flows through those endpoints?

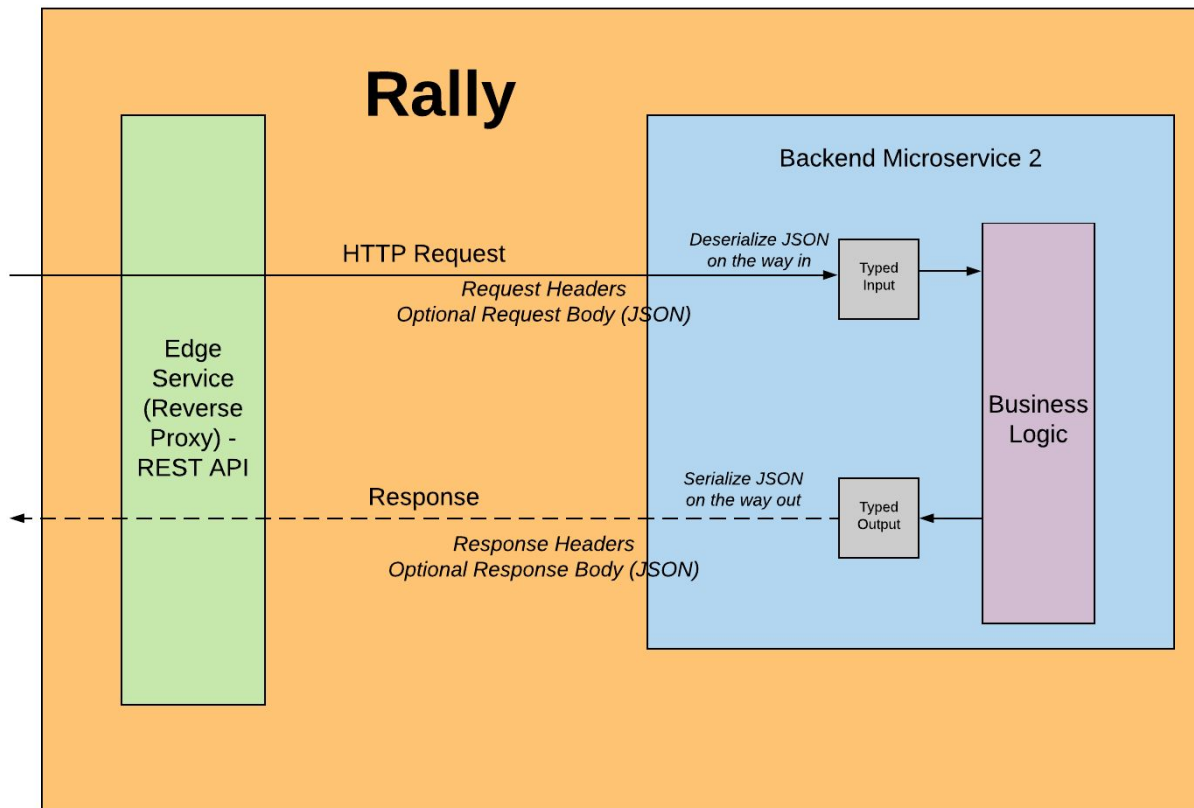
Original Data Science Reporting Requirements

- **UI-related** events such as
 - page load, section load
 - click
 - view
-
- Events for **backend endpoint** calls, including:
 - request parameters
 - request and response bodies
- Reporting of user **session** information as a session is established or refreshed
- Tagging of all reported events with **user** and **session** information
- **Schema-free data reporting.**
 - Data team: “We’ll make sense of your data!”

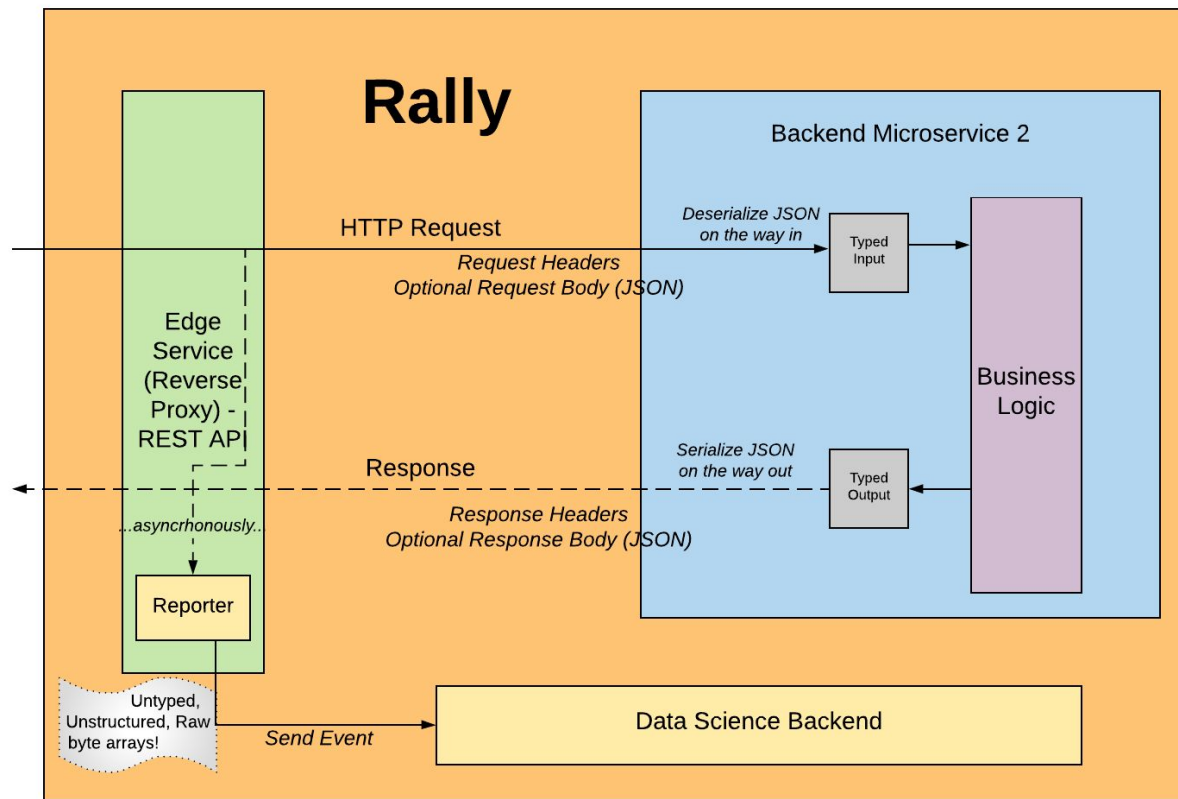
Request Processing



Zooming in a little...



Reporting Data to Data Backend (Original Implementation)



Edge Service Controller Instantiation

```
trait MyController {  
  def getSomething(id: UUID): EssentialAction  
}
```

```
class MyControllerImpl(cc: ControllerComponents, /*...*/ )  
  extends AbstractController(cc) with MyController {
```

```
  private[this] val myActionContext: ActionContext // ... (can derive a “source” object from this)  
  private[this] val myAction: ActionBuilder[Request, AnyContent] =  
    // Execute the action and then asynchronously report the data  
    Action andThen ReporterActionFunction(myActionContext, /* ... */)
```

```
  def getSomething(uuid: UUID): EssentialAction = myAction.async {  
    proxyRequest(s"/path/to/backend/resource/${uuid.toString}")  
  }  
}
```

Schema-free data reporting with a Play Action Function

```
class ReporterActionFunction[R[_] <: Request[_]](actionContext: ActionContext, /* ... */)
  extends ActionFunction[R, R] {

  override def invokeBlock[A](request: R[A], block: R[A] => Future[Result]): Future[Result] = {
    // Execute the action...
    val resultF = block(request)
    // Wiretap as a side-effect!
    resultF.onComplete {
      case tryResult => tryResult.foreach { result =>
        if (result.header.status >= 200 && result.header.status < 300) {
          tap(request, result)
        } else {
          logger.debug("Reporting skipped because non-successful status")
        }
      }
    }
    resultF
  }
}
```

Benefits of this design

Separation of concerns!

- Data reporting is done asynchronously
 - We used a different execution context for data reporting
- Data reporting is orthogonal to business logic
 - Developers can add endpoints and not think (too much) about data reporting
 - Follow the pattern and you get data reporting *for free!*

... And then the plot thickened...

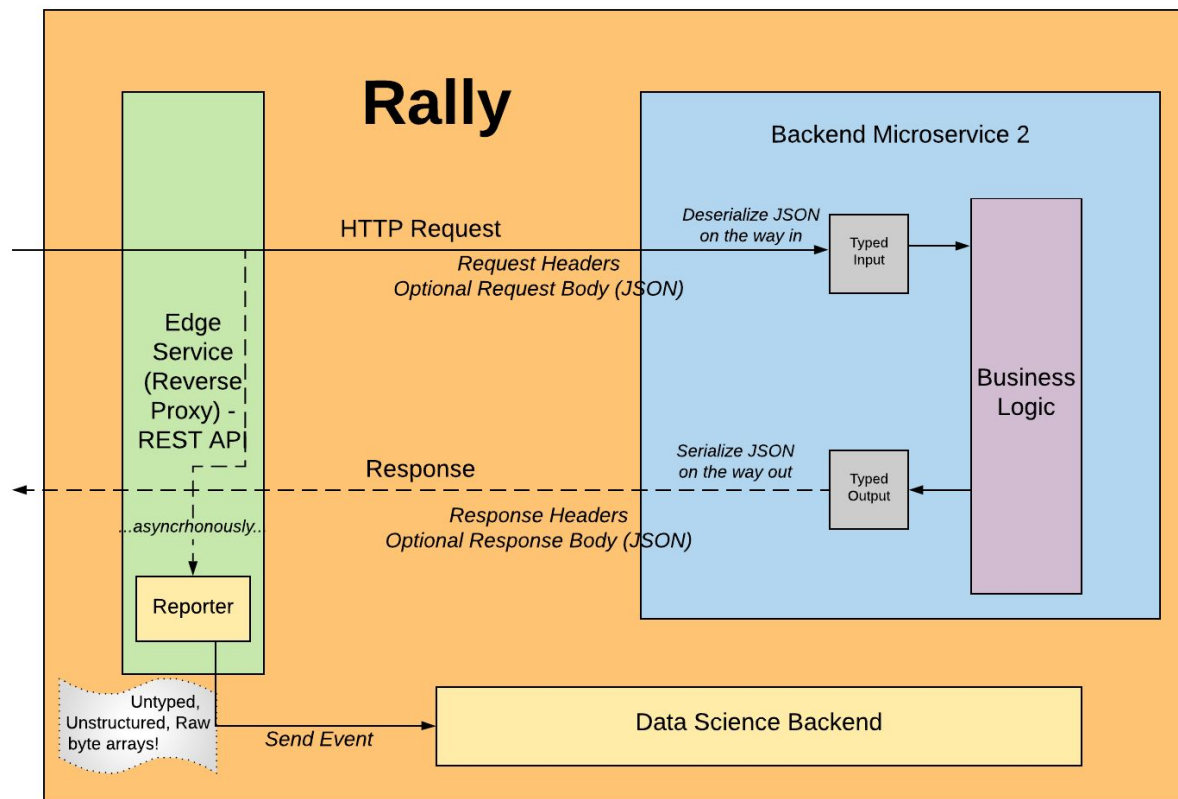
NextGen Data Science Reporting Requirements

Exact same kinds of events as before...

But:

- You must **register a schema** for each event type (app startup)
- Schemas are **semantically versioned**
 - *X.Y, where X is major version and Y is minor version*
- Any new schema you register for a type must be **backward compatible** within the same major version
- Data you send for an event type will be **validated** against the schema you registered for that type and rejected if it is invalid
- **We'll help you:** **Macros for schema generation** are provided by the data team
 - *All you have to do is call our macros for your model objects and we'll build the schemas for you...*

Reporting Data to Data Backend (Original Implementation)



What to do?

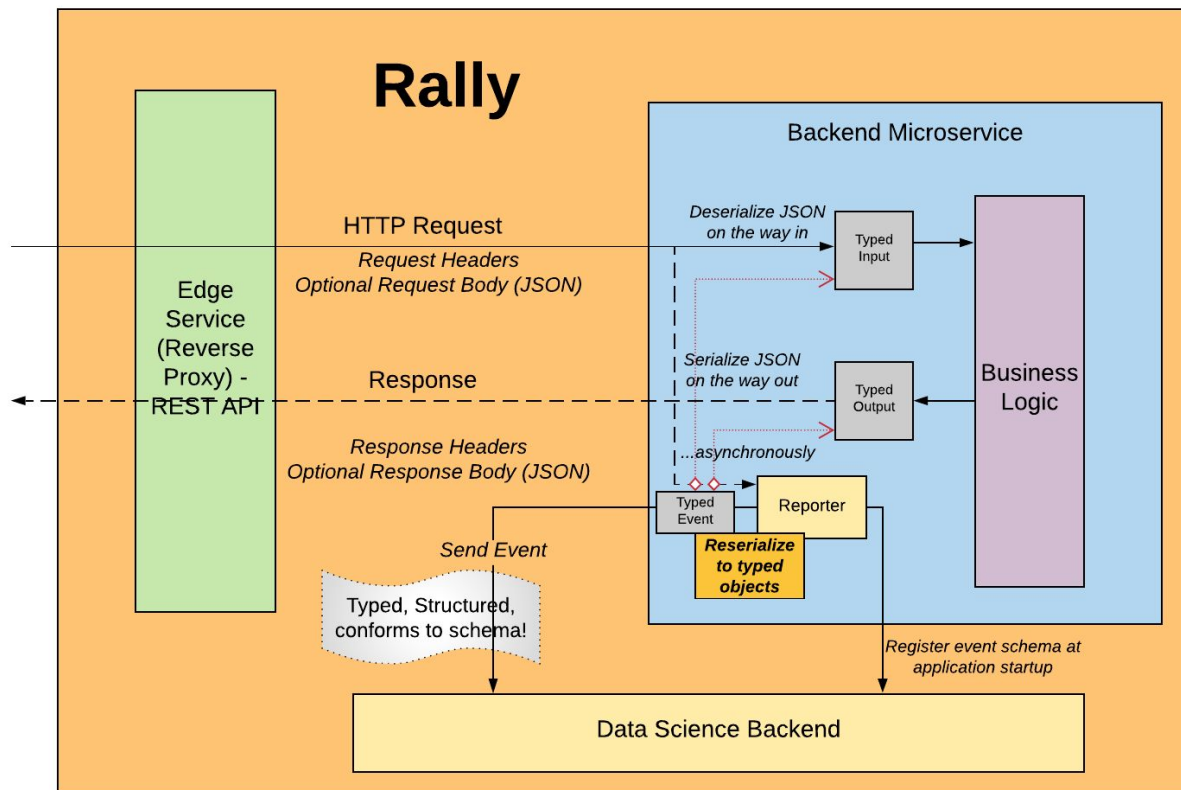
Schemas need to be generated from case classes, **driven by their type!**

Our **edge services** do data reporting, but have **no knowledge of the types** of the payloads!

Options:

- Add another service that (somehow) takes care of all this... ?
- Factor out the data models into a library (or libraries)
and make the edge services “type-aware”
- Move data reporting to backend services? ← **WINNER**

NextGen Data Reporting



What does an Event look like?

For **each endpoint**, there would be something like:

```
case class Event(  
  userId: Option[String],  
  sessionId: Option[String],  
  /* probably headers */  
  /* request parameters (and we want param types!) */  
  request: Array[Byte] /* --> */ request: ??? // The request body type  
  response: Array[Byte] /* --> */ response: ??? // The response body type  
)
```

Ruh roh.

I need the **types** to deserialize request and response bodies.

... I don't have them on the edge service.

And **Play** uses **play-json**. Which requires implicitly declared **Formats**, **Reads**, **Writes** ...

```
val event = Event(  
  userId = appContext.extractUserId(request),  
  sessionId = appContext.extractSessionId(request),  
  /* maybe headers and request params with types */  
  // Deserialize with Play JSON:  
  request =  
    Json.fromJson[MyRequestBody](request.body.asInstanceOf[AnyContentAsJson].get),  
  response =  
    Json.fromJson[MyResponseBody](Json.parse(responseBytes.toArray))  
)  
client.sendEvent(  
  sourceFor(actionContext),  
  // Serialize event (here I use json4s, but at Rally I used play-json again (for a bunch of reasons)  
  Extraction.decompose(event).asInstanceOf[JObject]  
)
```

Taking Stock

We're trying to move the data reporting functionality to backend services.

How can we best preserve the separation of concerns we have already achieved?

- Data reporting functionality decoupled from business logic
- Data reporting execution context decoupled from Play (user-serving) execution context

Hand coding all this stuff:

... for all ~180 instrumented endpoints in our product ...

... is going to be ugly and time consuming ... (my teams are busy as it is!)

Can we make it declarative and minimize the amount of developer overhead?

Taking Stock

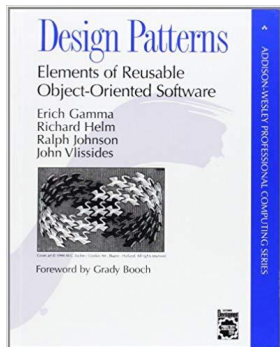
Idea: What if we could *generate* all the type-aware code we need?

Can we keep our integration points as declarative as possible?

....Let's abstract out what we can and

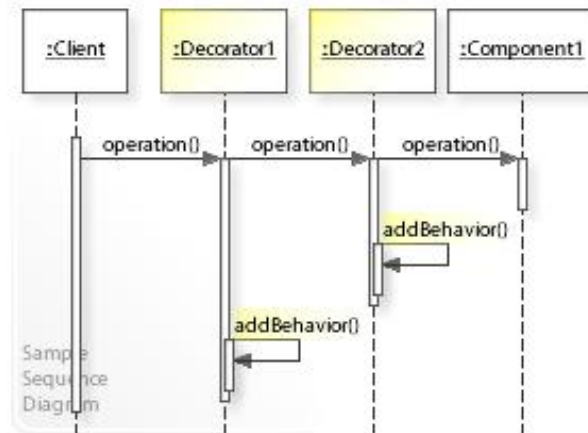
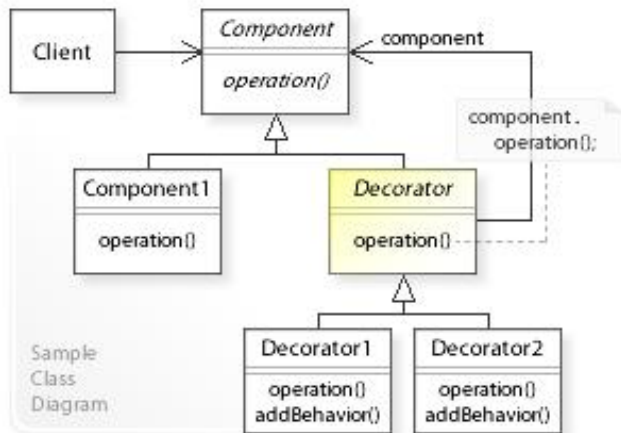
....Let's generate the “type-aware” code with **SCALAMETA!**

Going back in time... circa 2000...



Go4 Decorator Pattern

OO Pattern, you have all the types you need!



From: Wikipedia, Vanderjoe / CC BY-SA (<https://creativecommons.org/licenses/by-sa/4.0>)

Going back in time... circa 2000...

The system I worked on was implemented in Java and used CORBA.

We **defined system-level interfaces** using CORBA Interface Definition Language.

The typing appealed to me...

We used **code generation** to define *interceptors* (another name for *decorators*).

- One thing they did was instrumentation / stats collection / error tracking
- Another thing they did was logging

OO Pattern - you get “**type awareness**” for free!

*... for both inputs **and** outputs!*

... Incidentally: When I left that world, I was like, why are you all using *Strings* everywhere? (XML, JSON)

Controllers and Services

```
trait MyController {  
  // Play routes file maps a URL to this, and you have to return an action that has no knowledge of type  
  def getSomething(id: UUID): EssentialAction  
}  
  
trait MyService {  
  def getSomething(id: UUID): Future[Something]  
}  
  
import play.api.libs.json.{JsError, JsSuccess, Json}  
import play.api.mvc._  
  
class MyControllerImpl(cc: ControllerComponents, service: MyService, /*...*/ )  
  extends AbstractController(cc) with MyController {  
  
  private[this] val GetSomethingAction: ActionBuilder[Request, AnyContent] = ???  
  
  def getSomething(uuid: UUID): EssentialAction = GetSomethingAction.async {  
    service.getSomething(uuid).map {  
      case Some(thing) => Ok(Json.toJson(thing)) // I have to serialize the result and return a status here  
      case None        => NotFound  
    }  
  }  
}
```

EssentialAction

Where'd the types go? There are no type variables...

.... oh that's right, I have to deal with serializing my *output* (and deserializaing inputs, maybe my request body...)

```
/**
 * An `EssentialAction` underlies every `Action`. Given a `RequestHeader`, an
 * `EssentialAction` consumes the request body (an `ByteString`) and returns
 * a `Result`.
 *
 * An `EssentialAction` is a `Handler`, which means it is one of the objects
 * that Play uses to handle requests.
 */
trait EssentialAction extends (RequestHeader => Accumulator[ByteString, Result]) with Handler { self =>

  /** @return itself, for better support in the routes file. */
  def apply(): EssentialAction = this

  def asJava: play.mvc.EssentialAction = new play.mvc.EssentialAction() {
    def apply(rh: play.mvc.Http.RequestHeader) = self(rh.asScala).map(_.asJava)(Execution.trampoline).asJava
    override def apply(rh: RequestHeader)      = self(rh)
  }
}
```

Generating code...

Could we **drive code generation** from the service interface?

... we have all the **input parameters**, and their **types**, return **types**,
and the **event name** information we need there ...

```
trait MyService {  
  
  @Wiretap(/* params for code generator */)   
  def getSomething(id: UUID, someParam: String): Future[Something]  
  
  @Wiretap(/* params for code generator */)   
  def postSomeItem(item: SomeItem): Future[SomeItem]  
}
```

Controllers and Services

```
import play.api.libs.json.{JsError, JsSuccess, Json}
import play.api.mvc._
```

```
class MyControllerImpl(cc: ControllerComponents, service: MyService, /*...*/ )
  extends AbstractController(cc) with MyController {
```

```
  private[this] val GetSomethingAction: ActionBuilder[Request, AnyContent] = ???
```

```
  def getSomething(uuid: UUID): EssentialAction = GetSomethingAction.async {
```

```
    // BUT: Teams may not be using this idiom! (and I want to support multiple teams)
```

```
    service.getSomething(uuid).map {
      case Some(thing) => Ok(Json.toJson(thing))
      case None        => NotFound
    }
```

```
  }
```

```
}
```

Generating code...

Could we **drive code generation** from the controller trait?

All Play apps have to define a routes file pointing to controller methods...

... and we have all the **input parameters**, and their **types**,

but we need a little help for the **request bodies** and their **types**
and the **response bodies** and their **types**...

```
trait MyController {  
  
  @Wiretap(/* params for code generator */)   
  @WiretapResponse(classOf[Something])  
  def getSomething(id: UUID): EssentialAction  
  
  @Wiretap(/* params for code generator */)   
  @WiretapRequest(classOf[SomeItem])  
  @WiretapResponse(classOf[SomeItem])  
  def postSomeItem(item: SomeItem): EssentialAction  
}
```

Floppy Ears!

*Because when I listened in on conversations when I was a kid,
my Mom called me “Floppy Ears.”*

Sidebar: If you work at Rally, there is a different name for the “original” in-house version of this library and its companion plugin. It includes lots of our shared libraries and tooling. Ping me on **Rally Slack** (@k-lergic) for details!

The “Floppy Ears” example/demo code is a “port” of the code to Play 2.8 and does not have any of the “internal” Rally stuff in it.

Demo Code: <https://github.com/eric-fredericks/floppy-demo>

Floppy Ears!

Wiretap Library:

- A set of annotations to drive the code generator
 - Define what methods need to be instrumented
- Facilities for providing both application-level and endpoint (action) level context
- An action function:
 - Asynchronously invokes the type-aware function to create the event
 - Asynchronously sends the event to the data science backend
- A bunch of helper code to deal with parameter extraction (especially url path params)
 - *Because Play didn't expose url path parameter extraction utility code*

Floppy Ears!

sbt plugin (code generator):

- A routes file parser/validator
 - Uses the Play routes file parser! (At least they exposed this)
 - Requires your controller trait to use the same parameter names as those used in the routes file
- Code to parse our custom annotations
- Code to deal with controller method parameter conversion
- Code generator driver

Code generation result

- a **case class** for the generated event object
with all the properly typed values involved
- a **companion object** for the case class
*with a schema and a customized “type-aware” function that takes a request, and the response bytes, and produces a **JSON representation of the event** object*
*** This is how the **wiretap** library’s action function uses the generated code*

This “type-awareness” involves essentially hard-coding JSON deserialization of request and response bodies and re-packaging them into the event object before re-serialization

- a **helper method** that developers can use in controllers to get an `ActionContext` holding the type-specific information, including a reference to the type-aware function

Floppy Ears: Code generation

Scalameta usage

- My code generator reads each source file and parses the source with Scalameta
- Each source file's syntax tree is traversed
- Pattern matching (often with quasiquotes) is used extensively
- Information from the source is interpreted and translated into source text that is written to a data structure collecting strings holding the “output” source
- I didn't construct new syntax trees or transform syntax trees with Scalameta
 - Perhaps this is an area worth exploring?
 - I was on a deadline and didn't have time to reason deeply about it
 - It seemed more appropriate to traverse, detect what I was looking for, and build up a sequence of strings representing a new source file (as a side-effect)

Action Context

```
case class ActionContext[R[_] <: Request[_]](  
  actionDef: ActionDef,  
  eventSchema: Schema,  
  createEvent: R[_] => // The request  
    (R[_] => Option[String]) => // The user id extraction function  
    (R[_] => Option[String]) => // The session id extraction function  
    ByteString => // The byte array containing the result produced by the action  
    JObject // The resulting event object, as a Json object  
)
```

**** The entire design hinges on this `createEvent` function!**

Play Action Function

```
class FloppyEarsActionFunction[R[_] <: Request[_]](actionContext: ActionContext, /* ... */)
  extends ActionFunction[R, R] {

  override def invokeBlock[A](request: R[A], block: R[A] => Future[Result]): Future[Result] = {
    // Execute the action...
    val resultF = block(request)
    // Wiretap as a side-effect!
    resultF.onComplete {
      case tryResult => tryResult.foreach { result =>
        if (result.header.status >= 200 && result.header.status < 300) {
          tap(request, result)
        } else {
          logger.debug("Reporting skipped because non-successful status")
        }
      }
    }
    resultF
  }
}
```

Play Action Function

```
private[this] def tap[A](request: R[A], tryResult: Try[Result]): Unit = tryResult.map { result =>
  result.body.consumeData.onComplete {
    case Success(bytes) =>
      val event = createEvent(request)(extractUserId)(extractSessionId)(bytes)
      client.sendEvent(source, event).onComplete {
        case Success(_) =>
          logger.debug(s"Completed sending event for source=$source")
        case Failure(e) => logger.error("Error reporting data", e)
      }(tapEC)
    case Failure(e) =>
      logger.error("Reporting skipped, materialization failed!", e)
  }(tapEC)
}
```

Recipe Box Controller example

```
trait RecipeController {  
  import RecipeController._  
  
  @Wiretap(actionName = "GetRecipe", majorVersion = 1)  
  @WiretapResponse(classOf[Recipe])  
  def getRecipe(id: Long): EssentialAction  
  
  @Wiretap(actionName = "SaveRecipe", majorVersion = 1)  
  @WiretapRequest(classOf[Recipe])  
  @WiretapResponse(classOf[Recipe])  
  @WiretapResponseTransform(classOf[RecipeInfo], toRecipeInfo)  
  def saveRecipe(): EssentialAction  
  
  @Wiretap(actionName = "RecipeSearch", majorVersion = 1)  
  @WiretapResponse(classOf[Seq[RecipeInfo]])  
  def search(q: Seq[String]): EssentialAction  
}
```

Example generated code: Save recipe endpoint

```
// Event object for which schema is derived for data reporting
```

```
case class SaveRecipeV1Event(  
  _userId: Option[String],  
  _sessionId: Option[String],  
  _request: Recipe,  
  _response: RecipeInfo  
)
```

```
// Helper method to get context in the controller
```

```
def saveRecipeV1Context[R[_] <: Request[_]]: ActionContext[R] = {  
  import SaveRecipeV1Event._  
  ActionContext[R](  
    SaveRecipeV1Action,  
    SaveRecipeV1EventSchema,  
    createSaveRecipeV1Event[R] // Your friend, the create event function!  
  )  
}
```


Example generated code: Save recipe endpoint

```
// Companion object
object SaveRecipeV1Event {
  implicit val SaveRecipeV1EventSchema: Schema = AvroSchema[SaveRecipeV1Event]
  val SaveRecipeV1Action = ActionDef("SaveRecipe", 1)
  def createSaveRecipeV1Event[R[_] <: Request[_]]( // It's your friend again!
    request: R[_](extractUserId: R[_] => Option[String])(extractSessionId: R[_] => Option[String])(
      bytes: ByteString): JObject = {
    val _userId = extractUserId(request) // From app-level context
    val _sessionId = extractSessionId(request) // From app-level context
    val requestBodyAsJson: AnyContentAsJson = request.body.asInstanceOf[AnyContentAsJson]
    val _requestBody: Recipe = Json.fromJson[Recipe](requestBodyAsJson.json).get
    val responseJsValue = Json.parse(bytes.toArray)
    val responseBody: Recipe = Json.fromJson[Recipe](responseJsValue).get
    val _response: RecipeInfo = toRecipeInfo(responseBody) // here's that transformation function!
    val event = SaveRecipeV1Event(_userId = _userId, _sessionId = _sessionId, _request = _request,
      _response = _response)
    Extraction.decompose(event).asInstanceOf[JObject] // Json4s (because, play-json, ew!)
  }
}
```

Recipe Box Controller example

```
class RecipeControllerImpl( /* ... */ )(implicit floppyContext: FloppyEarsContext[Request], /* ... */)  
  extends AbstractController(components) with RecipeController {  
  
    import FloppyEarsActionFunction._  
    import RecipeControllerFloppyEarsSupport._ // Generated!  
    /* ... */  
  
    private[this] val SaveRecipeV1: ActionContext[Request] = saveRecipeV1Context[Request]  
  
    private[this] val saveRecipeAction: ActionBuilder[Request, AnyContent] =  
      Action.withFloppyEars(SaveRecipeV1) // Pimp my library: implicit class in FloppyEarsActionFunction  
  
    override def saveRecipe(): EssentialAction = saveRecipeAction.async(parse.json) { request =>  
      // as before...  
    }  
  }  
}
```

Taking Stock

Did we accomplish our goals?

1. We need to provide and adhere to **semantically versioned schemas**.

We can now support schemas driven by types.

Compliance is externally enforced.

= We can now move our data reporting integration points to our backend services.

2. We want to preserve **separation of concerns**.

- Data reporting functionality decoupled from business logic
- Data reporting execution context decoupled from Play (user-facing) execution context

= Our design can still realize these requirements.

3. We want to keep **developer overhead at a minimum**.

Add a library dependency, and an sbt plugin

Sprinkle in some annotations and controller action wiring!

= Pretty minimal, if you ask me!

Future Work

- Exploring/exploiting syntax tree construction/transformation with Scalameta
 - Is it worth doing something different than generating source as a side-effect?
- Generating controller unit tests
 - Or even just stubs? Something to get you most of the way would be nice...
- Seeing how this tool can support other web service frameworks?
 - http4s, akka4s, ...
- Avoiding play-json (Rally's bane for Play upgrades)
- Scalameta/scalafix for automatically updating source code to declare schemas



Thank You

<https://github.com/eric-fredericks/floppy-demo>