Generate All The Things

Brian Ketelsen
@bketelsen

Thank you to the organizers of this conference for putting together such a unique and interesting program. I am grateful for the opportunity to share some of my passion with you here, and honored to share the stage with such a diverse and interesting group of speakers.

In this talk I’m going to show you how you can level up your skills and productivity by using Code Generation

Who Am I - @bketelsen

- 10 Years as CIO/CTO, 25 years programming
- Co-Author of Go In Action (Manning Press)
- GopherCon / Gopher Academy
- BrianKetelsen.com + learn.brianketelsen.com
- @gotimefm Go Time Podcast

So — Who I am and why you care about what I have to say:
10 Years as CIO/CTO, 25 years programming
Co-Author of Go In Action (Manning Press)
Co-Organizer of GopherCon
Founder of Gopher Academy
BrianKetelsen.com + learn.brianketelsen.com
@gotimefm Go Time Podcast.
I've been using Go in production systems since 2010.

It’s Not Just A Meme - It’s a Lifestyle

Last year I re-discovered code generation. I’ve done some code generation in the past, but it never really caught on in the way it did for me last year. This presentation is the story of that rediscovery.

Why Code Generation

Over the course of this talk, I’m going to give a brief overview of the history of code generation with a little conceptual background. Then I’m going to describe my journey to enlightenment.
You’ve probably already used code generation. Let’s look at some popular examples:

rails new mywebapp

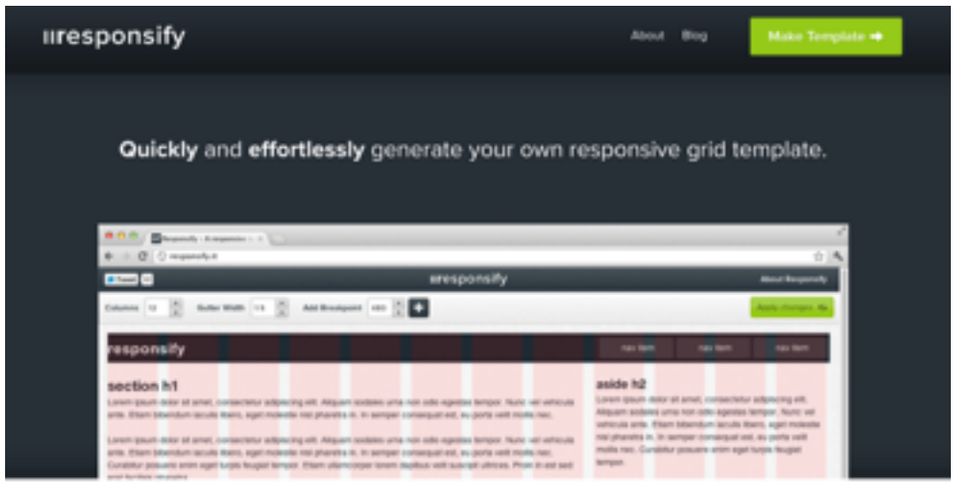
```
    create  README.md
    create  Rakefile
    create  config.ru
    create  .gitignore
    create  Gemfile
    create  app
    ...
    create  tmp/cache
    ...
       run  bundle install
```

How many people have created a web application or API using Ruby on Rails?

Rails wasn't the first framework to do code generation, but they sure did it right. Not only did rails generate the boilerplate of the web application for you, but rails went the extra mile and added generators for each of the different components you might need while you build your web application.

yo webapp

Next up on the example list is Yeoman - anybody who’s done a web application has seen yeoman. Yeoman opened the door to generating the boilerplate for any type of application with a format that is easy to implement. You can ask questions during the process and change the output based on the responses. Yeoman inspired a generation of tools like Ember CLI, React CLI and Angular CLI.



Here's another example of a code generator. This one is an online tool that lets you answer a few questions then it outputs HTML and CSS specific to your needs. There are hundreds of websites similar to this - answer a few questions, download the resulting boilerplate. Outputs can be anything from HTML and CSS to SQL schemas. I call this Code Generation As A Service.

History Of Code Generation

Before we talk about my life changing realization, I want to step back and talk a little bit about where code generation came from. Let’s go back in time.

Automatic Programming

The first incarnation of Code Generation was called “automatic programming” by David Parnas
Parnas helped shape Object Oriented Programming by pioneering the concept of “Information Hiding” — what many today call “Encapsulation” - the idea that the details of an implementation are hidden behind a public interface, allowing the details to change without impacting the consumers of the Object.

“automatic programming has always been a euphemism for programming in a higher-level language than was then available to the programmer.”

–David Parnas

"automatic programming has always been a euphemism for programming in a higher-level language than was then available to the programmer.”
That broad definition of automatic programming could be applied to many of the concepts that have evolved in computer programming - macros, C++ Templates, Domain Specific Languages…

Automatic Programming Through The Years

- 1940's - Punch Tape
- 1952 - Autocode - the first compiler
- 1950's - ALGOL, FLOW-O-MATIC, Speedcoding
- 1960's - BASIC, COBOL, BCPL

A drastically simplified history of automatic programming shows that each evolution brought us, the programmers farther and farther away from machine code and closer to human readable form.
1940 - punch tape simplified programing, allowing humans to input machine code quicker
1952 - autocode, the first compiler was created. Barely an abstraction above machine code, but far better none the less
1950's - ALGOL, FLOW-O-MATIC, and Speedcoding are introduced
1960's - the real language revolution begins, with each new language created to solve a particular business need

Note that the BCPL language invented in 1967 is the precursor to modern C

Automatic Programming

The lowest level of machine instructions are a little too low level for most of us humans. Every layer above the silicon is another abstraction, another level of Automatic Programming.

hmm… Abstractions…

Isn’t that why we’re here?

Layers of Abstraction*

*Grossly Simplified

Let’s take a look at the abstractions we take for granted as we write software today. We’ll start at the lowest level and work our way up to the software you write. I’ve taken some liberties, left out some smaller levels of abstractions, but the layers that I’m going to show you represent fundamental decreases in the amount of work you need to do to accomplish a task.

Layer 0
Machine Code

The bottom of the line is machine code - the language a computer’s processor speaks. Nobody wants to write this directly. It’s not a language at this level, it’s not even words. It’s just a pattern of bits.

Layer 1 Assembly Language

```
i = j + k;                1   ILOAD j    // i = j + k     0x15 0x02
if (i == 3)               2   ILOAD k                    0x15 0x03
    k = 0;                3   IADD                       0x60
else                      4   ISTORE i                   0x36 0x01
    j = j - 1;            5   ILOAD i    // if (i < 3)    0x15 0x01
                          6   BIPUSH 3                    0x10 0x03
                          7   IF_ICMPEQ L1               0x9F 0x00 0x0D
                          8   ILOAD j    // j = j - 1     0x15 0x02
                          9   BIPUSH 1                    0x10 0x01
                         10   ISUB                       0x64
                         11   ISTORE j                   0x36 0x02
                         12   GOTO L2                    0xA7 0x00 0x07
                         13 L1:          BIPUSH 0       // k = 0 0x10 0x00
                         14   ISTORE k                   0x36 0x03
                         15 L2:
                (a)                (b)                        (c)
```

Assembly language is the next layer up …. foreign to many of us, but still a giant layer of abstraction over machine code. It's not the most comfortable or forgiving way to write code.

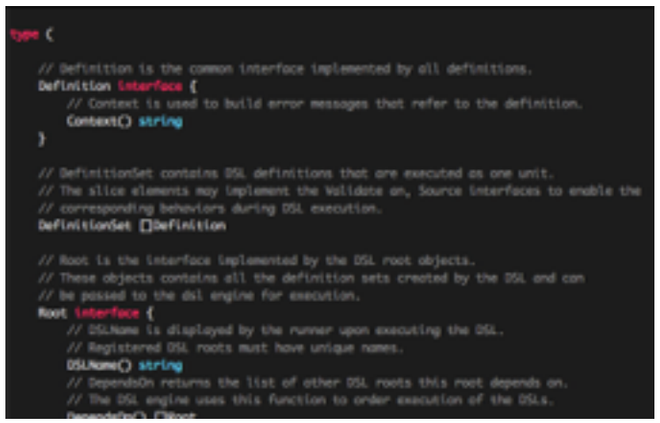
It’s a little bit of a stretch in this list of layers, but your operating system provides a multitude of abstractions for you, from file storage to network IO. I think it belongs in the list of abstraction layers.

Layer 3
Intermediate
Representation

- Go's Assembly Language
- Java ByteCode
- LLVM IR
- GNU RTL
- CIL
- Pascal p-code
- JRuby/Rubinius Bytecode

Many of the languages we use today have a layer of indirection between the source code and the compiled binary. Go generates its own flavor of assembly language which is later translated to processor specific code. Java and .NET have byte code, the LLVM generates IR, even the GNU compilers have RTL. Each of these is a layer of abstraction with the intent of making platform portability easier.

Layer 4
Source Code



The layer we as developers are most familiar with is Source Code. Each programming language provides a unique syntax, often with the intent of shaping the way you think about solving problems. We live in this layer, we breathe in this layer, we operate in this layer.
Source code allows us to program in higher level constructs, like loops, which are just syntactic sugar for branching and jumping at the assembly language level.
We also get the benefit of structures, records, or objects which allow us to store data larger than a single machine word or register can hold.

Layer 5 Standard Libraries

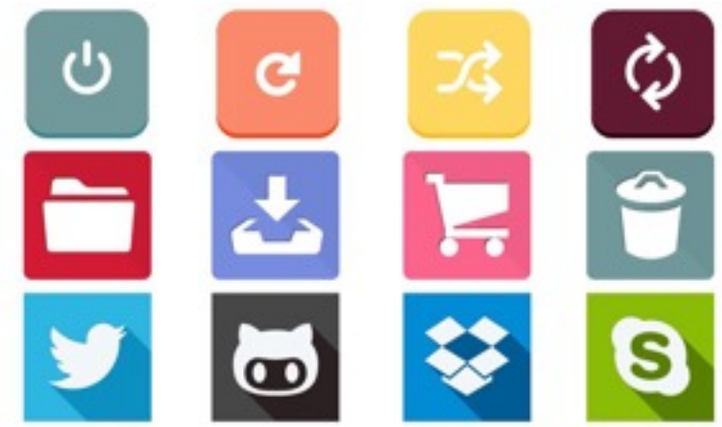
```go
// These routines end in 'f' and take a format string.

// Fprintf formats according to a format specifier and writes to w.
// It returns the number of bytes written and any write error encountered.
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error) {
    p := newPrinter()
    p.doPrintf(format, a)
    n, err = w.Write(p.buf)
    p.free()
    return
}

// Printf formats according to a format specifier and writes to standard output.
// It returns the number of bytes written and any write error encountered.
func Printf(format string, a ...interface{}) (n int, err error) {
    return Fprintf(os.Stdout, format, a...)
}

// Sprintf formats according to a format specifier and returns the resulting string.
func Sprintf(format string, a ...interface{}) string {
    p := newPrinter()
    p.doPrintf(format, a)
    s := string(p.buf)
    p.free()
    return s
}
```

Go, Java, C, .Net…. nearly every modern programming language has a standard library. This is code that has been written for you, on your behalf, to do things so common that nearly everyone will need it. Another layer of abstraction.

Layer 6
External
Libraries

Layer 6 - External Libraries : Whether you call them Gems, eggs, packages, modules, or my personal favorite — “Adopted Problems”, chances are good that your projects will depend on code that someone else wrote. These abstractions are most interesting because someone else abstracted them for you.

Layer 7
Your
Application



The last layer in our stack of abstractions is your application. You solved a problem by piecing together lower level abstractions and mixing them in the right order. The final product - your application - rests upon the abstractions below it to perform a useful task, solve a business problem, or more generally — provide value to someone.

My Journey

All of that classification and history brings us here - to my journey. My motivations in code generation may be familiar to you. Working as the CTO of several startups over the past few years I noticed a few things.

My Silly Generalizations

< 20% of your code solves the business problem

Take these generalizations with a grain of salt, I made up all the numbers based on the intuition of 25 years of experience.
Probably the most important reason I started exploring code generation with such a passion was my realization after doing several startups consecutively that the real core of your business logic is less than 20% of your application.

My Silly Generalizations

> 80% of your code is the boilerplate and glue

That means that more than 80% of the code you write is just bubblegum and bailing wire gluing the code with value together. Is there *ANY* business value in writing another data access layer? In hooking API controllers to an HTTP request router?

My Silly Generalizations

Nobody solves a problem the same way

Nobody solves a problem the same way This one hurts a lot - Put 5 developers in a room and ask them to solve the same problem and you'll get five wildly different solutions. 3 of them might even work. But all of them will be drastically different from the others. There are an infinite number of ways to build a widget. For your your team and your company, this means that projects, or even different parts of the same project may have radically different styles, different assumptions, different dependencies. Without the worlds largest architectural and code review board, it's impossible to escape this problem.

My Silly Generalizations

Copy Pasta is the enemy of agility

Sometimes you can’t abstract a code pattern into a reusable object, library, or package. The variable factors are just —> this much <— too far apart to make reusable code. You just can’t DRY it up effectively. So you copy the source code from solution A into solution B - then use your higher brain functions to change the pattern to match the new problem and you win, right? If shipping code is your only concern, then you’re a winner. But someday (HOPEFULLY) you’re going to have to maintain that code. Changes in solution A have to be applied to solution B too, but there’s no shared code, only Copy Pasta. This is why nobody wants to do code maintenance.

Time To Level Up

I think it’s time to level up. And code generation is the path I’ve chosen. Follow along with me now as I share my evolution from frustration to code-generating master.

Level One

I’ve categorized my code generation evolution into 5 levels, like belts in Karate. Level 1 is generating boilerplate. I am sure I’ve created my own projects that did this, but they weren't memorable or terribly useful. More commonly, this level comes from an IDE or similar tool when you create a new project from a Template.
You may have done this:
File -> New -> Windows Forms Application
Or, if you’re as old as I am, maybe you did this:
File -> New -> Enterprise Java Bean
This level of code generation is very useful - it provides a starting point for your project, it defines and enforces de-facto standards for how to work with a framework or technology.

Level Two

Level 2 is “Generating Code with Logic” — depending on how you define logic, most of the popular code generators like “rails” and “yeoman” fall here. When you generate a new controller in Rails, it populates one or more templates with some variables and places the code in the right place in your directory structure to cause the application to behave differently.

This level is also very important. I’d make the argument that Level 2 launched a million developers - the ease of rails and yeoman generation meant that developers could concentrate on solving actual business problems rather than reinventing the same infrastructure over and over. Level 2 also brings another important benefit - it enables the “Convention over Configuration” style of programming that eases the cognitive burden of writing and maintaining software. It allows people unfamiliar with your business to quickly grasp the intention of a piece of code because of it’s standardized name and location.

Level Three

Level 3 is Generating Complex Code - For me, this is where things get interesting. I'm going to dive in here and show you how I got hooked on code generation. When this journey started, the problem I was trying to solve was one that many of you have faced, or will face. I had to create a metric tonne of code in a short amount of time so that we could create business value for our investors, sell amazing services to our customers, … and keep the lights on. It's a timeless story, two or three developers, miles of code to write, and 90 days of burn rate left in the bank. I needed to create public facing APIs for multiple products. They had to be coded, tested, documented and deployed. I'm going to tell you a secret that may surprise you: the documentation was the real pain point for me. There's nothing more tedious in my book than documenting a REST API.

Time for a New Idea

Given a well-defined request and response:

- Create an API
- With Tests
- With Input Validations
- With Documentation
- Then deploy and monitor it

The problem at hand wasn't complex, but it was fairly tedious. The request and response types for each of the products in the API had dozens or sometimes hundreds of fields. Our task was to create the API, test it, document it for our customers - the consumers of the API — then deploy it and keep it running.

Generate Code from Struct Tags

```
type Blog struct {
    Name   string `json:"name"`
    URL    string `json:"url"`
}
```

My first attempt at solving this problem with code generation was a lot of fun. Go lets you assign arbitrary metadata to the fields of your structs (structs are similar to objects if you're coming from an OO language background) I took inspiration from several struct validation packages that are available - useful when you receive data in an HTTP POST and want to ensure it's valid. In the struct tag example here, the JSON struct tags tell Go's encoding package what to call the "Name" and URL fields when serializing to and from JSON

When I learned how easy it was to read the data out of struct tags, I quickly jumped into inventing my own tags.

Struct Tag Abuse

…
Name string `json:"name" description:"The name of your blog" type:”string" min:"5" required:"true"`
…

The benefit of this approach is that the domain logic for this type is embedded right within the type itself.

The obvious drawback is that the struct tags soon took on a life of their own.
The tags became a giant key/value store of domain knowledge that was buried too deep in the code to be useful.

It mostly worked, but I didn’t love it. It wasn’t practical, and it wasn’t elegant.

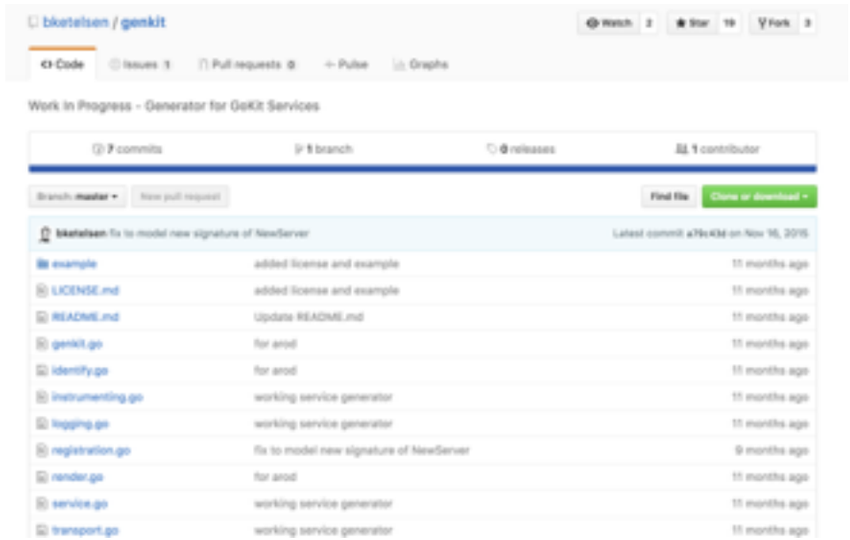
Generate From Code Annotations

```
// @REST {PUT,POST,GET}
type Blog struct {
    Name   string `json:"name"`
    URL    string `json:"url"`
}
```

http://www.onebigfluke.com/2014/12/generic-programming-go-generate.html

The next evolution of my code generation used code annotations much like you would frequently see in Java. I was inspired by a few Swagger packages that I found that generated swagger documentation by reading annotations in the comments of your structs.
Inspired by a blog post from Bret Slatkin, I wrote an application that parsed the code into an abstract syntax tree — or AST — and looked for my specific tags. When it found the tags, it stored the type below the comment in a list of candidate types for code generation. Later, I iterated over those candidates and applied a series of templates to them. If you've never tried parsing source code before, I highly recommend reading this post. It's not as hard or intimidating as I thought it would be.

GenKit



The result of this experimentation was GenKit - a code generation tool that reads annotations in the comments of your code and generates a micro service using go-kit, a popular framework in Go.

GenKit

```
//go:generate genkit $GOFILE
…
// @service
type Blog struct {
    Name   string `json:"name"`
    URL    string `json:"url"`
}
```

genkit uses Go’s ‘generate’ command, which inspects source files for the go:generate comment then invokes the named code generator, passing in the contents of the source code as context to the generator.

GenKit

```
func render(suffix string, w io.Writer, packageName string, types []GeneratedType) error {

    switch suffix {
    case "service":
        return serviceTemplate.Execute(w, generateTemplateData{packageName, types})
    case "instrumenting":
        return instrumentingTemplate.Execute(w, generateTemplateData{packageName,
types})
…
}
```

GenKit did two things:
It read the AST of the source code it was given, looking for types annotated with the @service comment
It applied five different templates to that type, generating code for the declaration, instrumentation, logging, registration, and network transport of the micro service.
Many who use go-kit complain that while the toolkit is amazing, there's an onerous amount of boilerplate required. This application solved that problem, but it did little to add additional value in terms of documentation. When my next stop looked like combining struct tags for the documentation with annotations for the service boilerplate, I knew I was barking up the wrong tree.

Enter goa

Enter GOA : I discovered goa on November 16th, 2015. goa is a code generator and library combination that generates your API based on a design that you create using goa's Domain Specific Language.

And for the record - it crossed a thousand stars a month or two ago.

Enter goa

Brian Ketelsen
@bketelsen

So I spent a few hours converting an internal API to goa (goa.design) and I think I want to marry the guy who wrote it.

Within 24 hours, I had converted a small API to goa and I was immediately sold on the concepts.

Fortunately, Raphael Simon, the author of goa was already spoken for, so no marriages were destroyed by my carelessly lustful tweeting.

What was it about goa that earned my adoration?

Things I <3 About goa

- Generates that 80% boilerplate code I don't want to write
- Generates that documentation I don't want to write
- Generates SO MUCH MORE

I came to goa for the code and documentation generation. goa creates all that boilerplate and glue code that I wanted to avoid. It also generates JSON schema and swagger output, so you have all the documentation you need to give to consumers of your API. It generates a command line application to interact with your API, a javascript library, and a Go package to interact with your API. With a little bit of bash glue I was able to generate a PDF of the API documentation suitable for distribution to my customers.

Inner Beauty

- Generates code that is idiomatic Go, looks handwritten
- Uses net/context better than I've ever seen it used

I said that I came to goa for the code generation and documentation. The reason I stayed was because of goa's inner beauty.

The code that is generated by goa is very carefully crafted, idiomatic Go code. With very few exceptions it looks like code you or I could have written.

Even more interestingly, goa generates a type for each request and response that embeds Go's Context package. I had never seen the Context package used this way and immediately I knew I had been underutilizing this powerful package.

Inner Beauty

```
// CreateAccountContext provides the account create action
context.
type CreateAccountContext struct {
    context.Context
    *goa.ResponseData
    *goa.RequestData
    Payload *CreateAccountPayload
}
```

Each action in a goa API gets a custom generated context that embeds Go's Context. If you're not familiar with Go's Context package, it's a way to store request specific metadata that survives across function and application boundaries, so it is common to send a context as the first parameter of any function that uses the network.

These strongly typed contexts were the icing on the cake. My API controllers now have strongly typed requests and responses, which reduces my ability to shoot myself in the foot a hundred fold.

goa's Design Language

Let's look at goa's design language to see why it's so powerful.

goa's Design Language

```
// User is the media type used to render user resources.
var User = MediaType("vnd.application/goa.users", func() {
    Description("A user of the API")
    Attributes(func() {
        Attribute("id", UUID, "Unique identifier")
        Attribute("href", String, "User API href")
        Attribute("email", String, "User email", func() {
            Format("email")
        })
    })
….
```

goa requires you to describe your API before you write a single line of code. This has the added benefit of forcing ME to actually think about my API before I write a single line of code.
The DSL lets you describe request and response types, the actions of your controllers, and more importantly it lets you document the entire API all the way down to required fields and field-level validations.

goa's Design Language

```
var _ = Resource("account", func() {
    DefaultMedia(Account)
    BasePath("/accounts")
    Action("list", func() {
        Routing(
            GET(""),
        )
        Description("Retrieve all accounts.")
        Response(OK, CollectionOf(Account))
        Response(NotFound)
    })
```

Each endpoint in your API is thoroughly documented, specifying the endpoint's URL, the actions available for the resource, the format and shape of any data that could be PUT or POSTed to that endpoint, and finally the format and shape of any possible response data or errors.

goa's Design Language

```
Attribute("varietal", func() {
    MinLength(4)
    Example("Merlot")
})
Attribute("vintage", Integer, func() {
    Minimum(1900)
    Maximum(2020)
    Example(2012)
})
Attribute("color", func() {
    Enum("red", "white", "rose", "yellow", "sparkling")
})
```

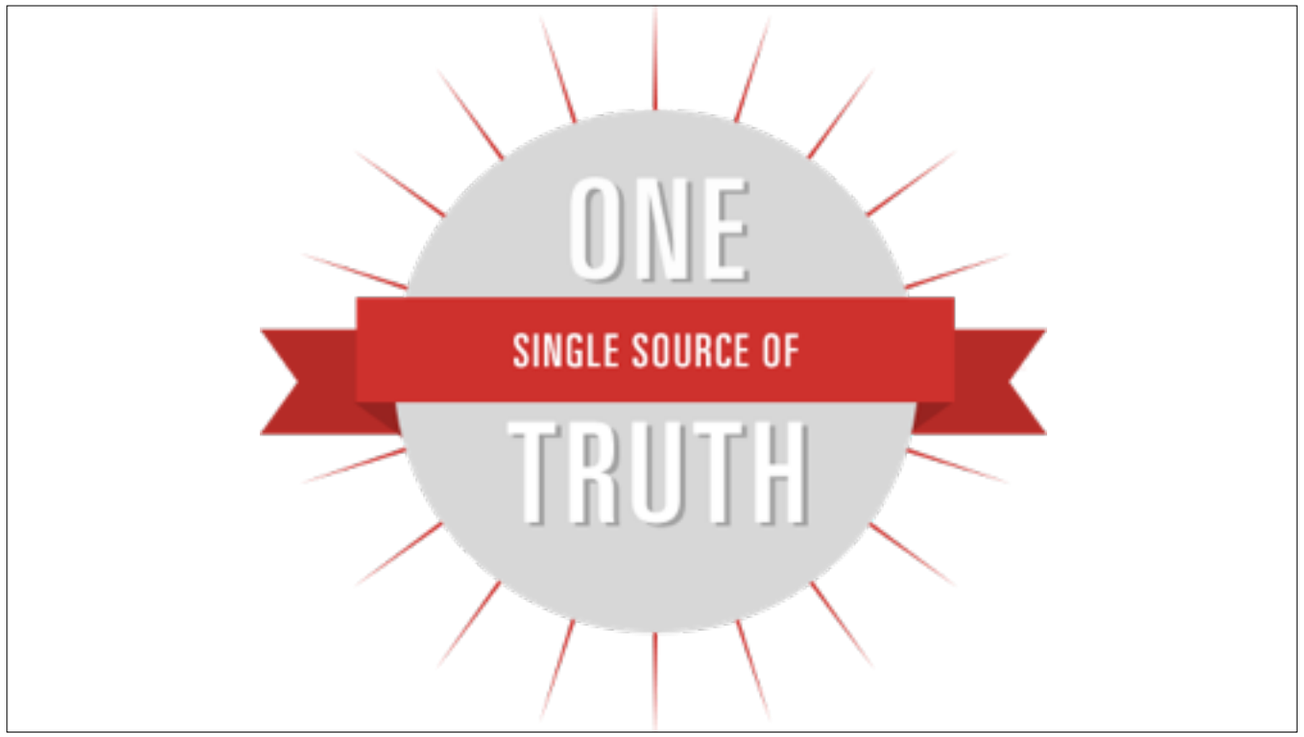
The DSL allows you to describe every field in precise detail, with minimum and maximum values, required fields, example values, even enumerations listing the acceptable values for a field.

When completed, the design is a thorough and easy-to-reference blueprint for your API.

Invoking goa

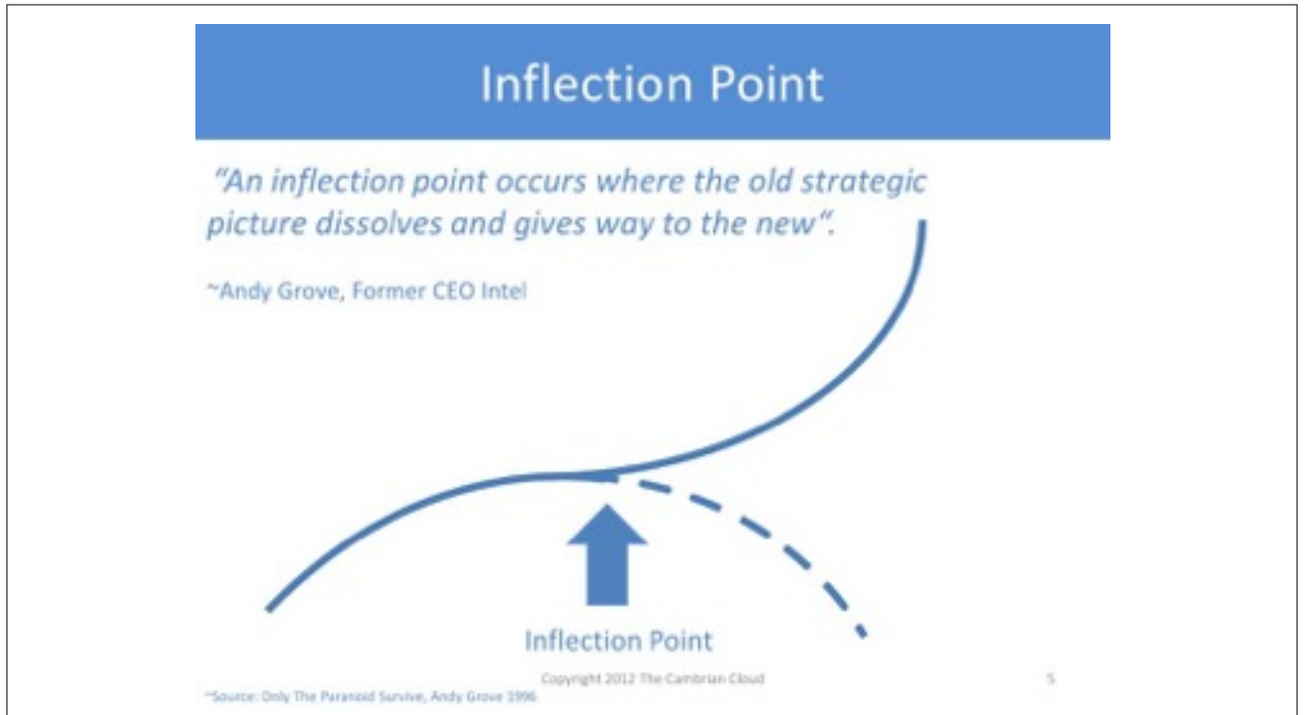
```
$> goagen -d your/project/design bootstrap
```

To generate source code from your design, you use the “goagen” command, passing in the path to your project’s Design DSL



My design DSL described my whole application. In that design I had described what inputs were required, what shape they must arrive in, what responses could be returned, and even where to send a request to get a desired response. I had documented the API as a whole, and each individual field of every request and response. I even documented the security requirements and how to get an API token.

This design was more than the blueprint for my API. It was the single source of truth for my application.



Andy Grove gave this definition of “Inflection Point” that represents the realization I had just a few days after using goa for the first time.

An Inflection Point occurs where the old strategic picture dissolves and gives way to the new

After spending a few days learning goa’s DSL I reached my inflection point.

Level Four

We’ve just spent a lot of time talking about my path to Level 3 of code generation. But there are still two more levels to go.

Level 4 is Generating Peripheral Artifacts - My first Pull Request to goa allowed me to extend the goa DSL by placing arbitrary metadata anywhere in the design. I used this metadata to describe how I wanted to store the data in the database.

The First 9* Iterations of Gorma

Use metadata from the goa DSL to generate a data access layer automatically from the types that are constructed by goa

*I actually stopped counting at some point. There were a lot of iterations.

Gorma represents my foray into Level 4 of my code generation evolution. Building on the idea that the API design was my single source of truth, I wanted to generate all the code required for my API to persist to and read from the database.

I didn’t really know what I was getting into, but I was very excited.

Gorma Metadata

```
// User is the media type used to render user resources.
var User = MediaType("vnd.application/goa.users",
func() {
    Description("A user of the API")
    Attributes(func() {
    Metadata("hasMany", "Bottles")
…
```

The metadata addition to goa allowed me to annotate goa's request and response types and generate data access code.

It mostly worked. But the metadata got ugly quickly when trying to shape the output of the generated code.

Moment of Clarity

After literally dozens of failed attempts, Raphael Simon - the creator of goa - finally convinced me I was doing it all wrong. Instead of shoe-horning metadata into goa's design language, I should create my OWN DSL for Gorma. That brought my second big Pull Request to goa:

Instead of goa's DSL parsing being coupled to the goa specific implementation of an API generator, we made it a generic DSL parsing engine. We decoupled the DSL parsing from the API generator which allowed the DSL engine to be used for *ANY* DSL.

Before and After

~~Metadata("belongsTo", "Account")~~

```
Model("Bottle", func() {
   BuildsFrom(func() {
      Payload("bottle","new")                    })
      RendersTo(Bottle)
       BelongsTo(Account)
})
```

The new DSL Engine package in goa allowed me to create a DSL specific to data storage definitions. I spent some time thinking through what that would look like and settled on something that has flavors of Active Record, but looks & feels like the rest of goa's DSL.

The Gorma DSL describes the relationships between the models that are used to store the API's data. It also describes the relationship between goa's types and the database models.
In this example, the Bottle database model uses the BuildsFrom() and RendersTo() functions to describe the goa API request and response types that map to this data model.

Because the goa design describes all of the API's types so specifically, Gorma knows everything it needs to know to generate 100% of the code for a complex CRUD application.

Generated goa Controller

```
// List Accounts
func (c *AccountController) List(ctx
*app.ListAccountContext) error {
  // TODO
  return ctx.OK(accounts)
}
```

goa generates your API all the way up to your controllers. You’re expected to get the data you need from somewhere and return it.

goa Controller with Gorma

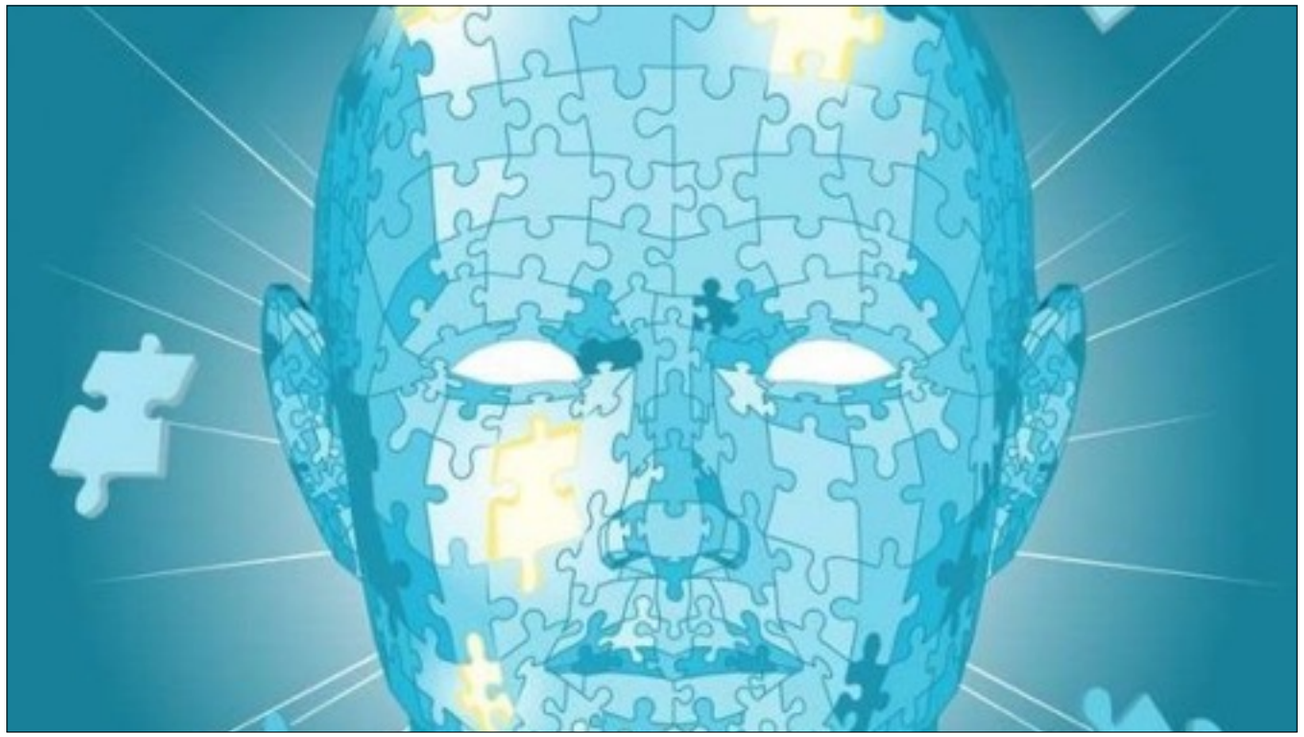
```
// List Accounts
func (c *AccountController) List(ctx
*app.ListAccountContext) error {
  accounts := adb.ListAccount(ctx.Context)
  return ctx.OK(accounts)
}
```

Gorma makes that process a one-liner in many cases.

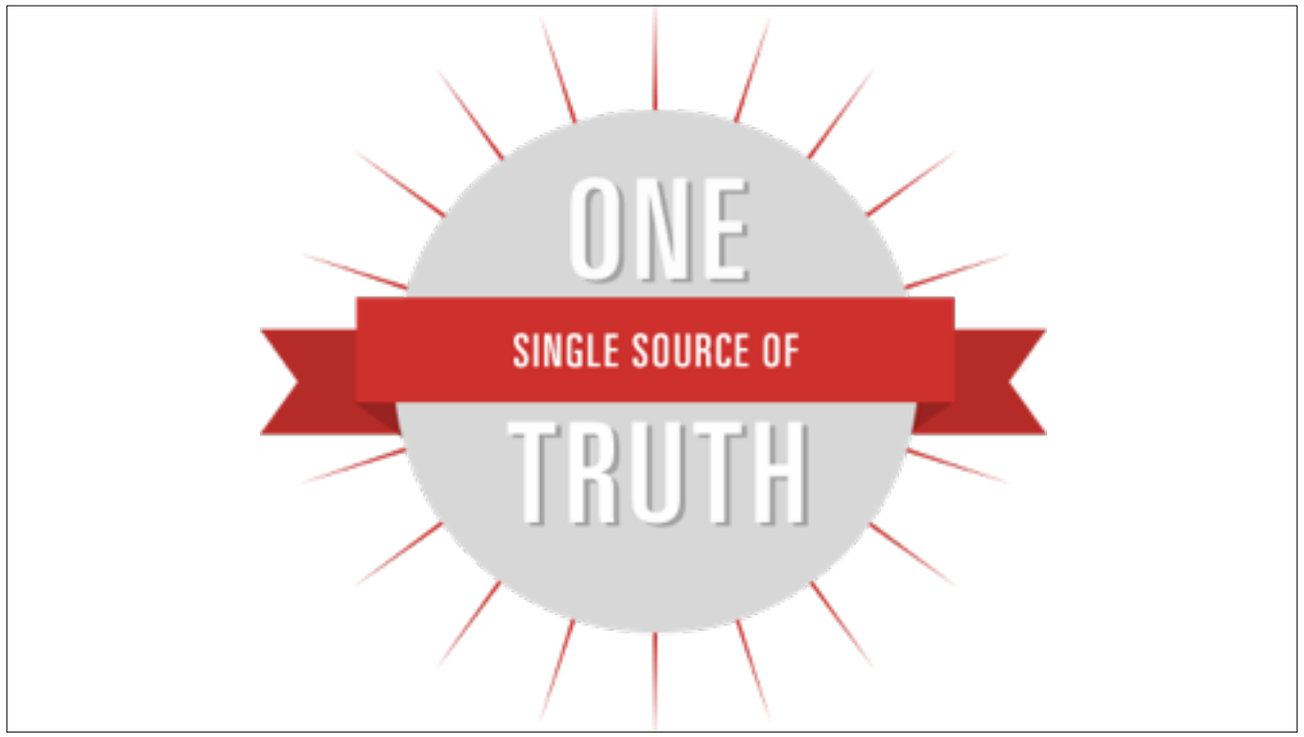
Invoking Gorma

```
$> goagen --design=your/project/design
gen --pkg-path=github.com/goadesign/
gorma
```

When we added the ability to create arbitrary DSL's and invoke them from goa, we opened up the possibility to generate anything.
You invoke a goa plugin using goagen's GEN subcommand, passing in the path of your plugin.



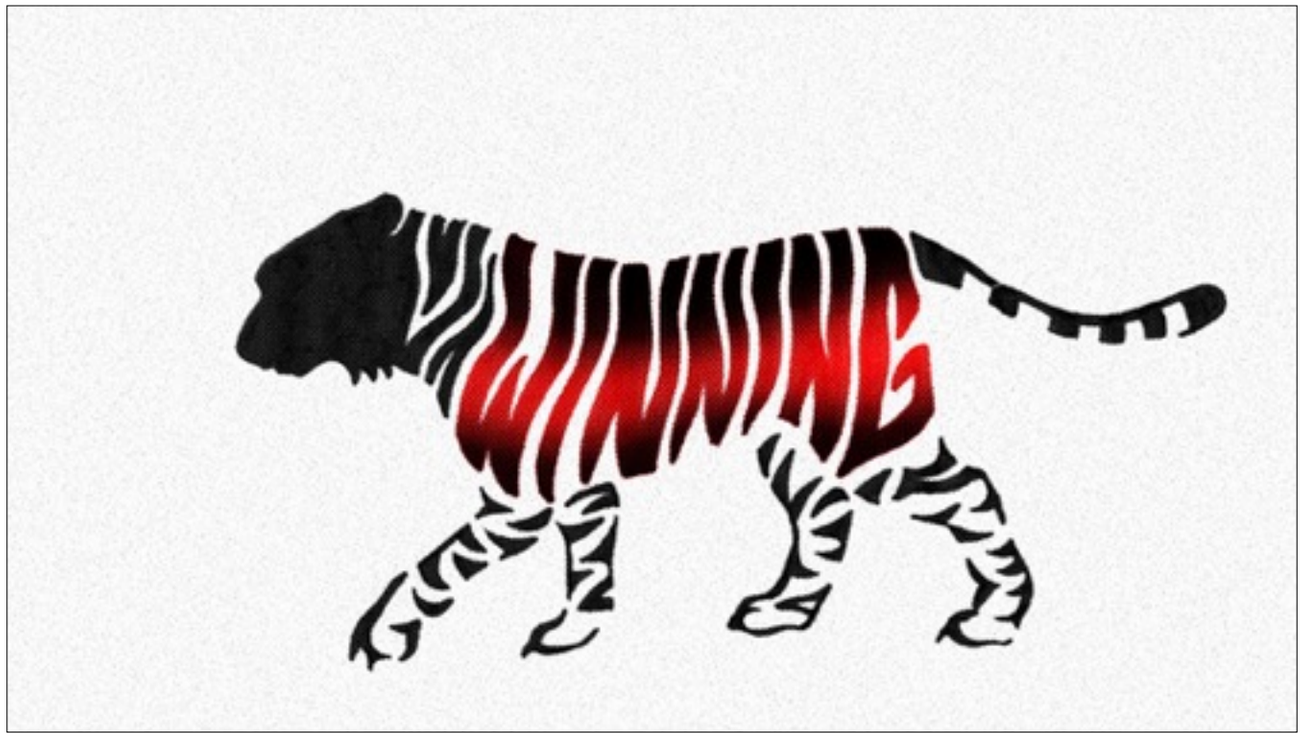
Let’s step back a moment and recap what just happened.
Using goa’s DSL I created a single source of truth for my application. A single place to look for documentation, expectations, requirements, and validation rules. I then generated an application that is compliant with that Single Source of Truth.
Later, by using goa’s DSL Engine to create my own DSL, I extended that Single Source of Truth all the way down to the database layer. Now I have a canonical place to look for information about the database schema, the queries that are used to access that database and the functions that are used to transform data to output through the API.



This Single Source of Truth concept is looking better and better.

At this point, I was starting to feel like I had leveled up in my code generation evolution. It was only then that I stopped focusing on the time-to-market benefits of code generation and looked at the solution with my CIO hat firmly planted on my head.

What if the Product team helped create the API Design? A few short hours of training later, the head of Product Development was committing API Designs into our git repository.



I stand here before you today to tell you that the only thing better than not writing tedious API documentation by hand is having the product team write most of your API design for you.
Our API design became 90% of our Business Requirements.
This immediately ended arguments over the intentions of requirements documents. The DSL is clear, concise and explicit in its intentions.
Even more powerful, as those requirements and designs evolved, we simply regenerated the API at each iteration. No code was thrown away, no bitter resentment over changing that stupid field from an integer to a string in 47 places throughout the codebase.

Level Five

It all crystallized when we reached Level 5. Level 5 in my made-up code-generation matrix is “Sharing Designs Among Projects and Teams”
If we offer two products that share a subset of request or response fields, why can’t we share that DSL between the products? Well it turns out we can. The DSL is really just Go code, which compiles down to a package. Packages can be imported anywhere. Now the Customer Model in Product A is exactly the same as the Customer Model in Product B. We know with 100% confidence that it’s the same, because it came from the same Single Source of Truth.

Level 5

This is where the single source of truth really shines

Once we realized we were able to share designs between products, we decided to create a central repository of all designs. Now we had a single home for our Single Source of Truth. All the domain objects throughout our product line and various APIs lived in a single repository. This repository became a catalog of widgets that can be plugged together to create new and interesting combinations to be used in new products.

Raise your hand if you've ever heard someone say "Let's take all that data we collected from the Woozle product, run it through the GEOIP database and add it to the Wubbles product" That new product took a third of the time because we already had DSL designs for Woozles and Wubbles.

In order to become a true master of Level 5, one must look beyond the application. And really, this isn't difficult once you see the benefits of generation from a Single Source of Truth. What's most likely to happen is your entire development team will rush off to build DSLs for everything they can think of.
True mastery of Level 5 comes when you are generating all of the artifacts required to build, test, deploy and even monitor your application.
True mastery comes when your Designs Become your Business Requirements.

Level Up

With a Single Source of Truth, there is no limit to what artifacts you can generate.
Docker files, Kubernetes and Mesos manifests.
even Travis or Jenkins configurations

Conclusion

The benefits of having a Single Source of Truth for your projects weren’t obvious to me from the start. I came to code generation because I’m a lazy programmer and I didn’t want to write a lot of tedious documentation by hand. What I found was unexpected.

Rapid Iterations

Code generation enabled rapid iterations. Making a change to a request or response type in my API became a single line edit in my design, rather than 30 minutes of search and replace. I found myself unafraid to radically change the types in my APIs because the cost of each change was negligible.

Documentation is Always Accurate

The Generated Documentation is Always Accurate
I can’t stress enough how important this point is. No development artifact suffers bit rot faster than your documentation. If the documentation is merely a generated artifact of your deployment process then it is always accurate and up to date. Pause with me for a moment and let that sink in.

Now raise your hands if your applications’ documentation is more than one revision behind the code in production. Those of you who didn’t raise your hands — LIARS.

Requirements Become Code

Requirements Become Code

The DSL that describes your project is code. By extension then, all the tools we have available to validate code are also available to validate requirements. If the requirements don’t compile, there’s something wrong with your design.

Design First

Generating code from a design means that you have to actually design before you code. Informal requirements won't work. When I was forced to take the time to design my API first, I drastically reduced the number of iterations required before achieving the desired business outcome. Thinking through the entire design takes a little more time up front, but saves time in the long run. Code generation taught me the value of the long game as a developer.

Generate All The Things

After embracing the Single Source of Truth mantra, I realized that there were very few things that couldn't be described with a DSL and generated. If I can describe the requirements of an API with a DSL, why can't I describe the requirements to deploy that API? The relationships between applications can be described. The deployment descriptors can be described.

Nearly everything can be generated.

Big Change Becomes Easier

Big changes become easier: This one is less apparent up front. What happens when you need to change databases, or when you move from Docker Swarm to Kubernetes? What happens when you need to support that one customer who can't use REST and has to use a TCP text protocol?

The solution is to generate new artifacts from your existing design DSL. Certainly there's a time cost in creating new templates and logic for each new generation target, but the hard work of thoroughly describing your application is already done. Changes that once felt ominous or radical are now within easy reach.

Layer 8
Design
DSL

```
package design

import (
    . "github.com/goadesign/goa/design"
    . "github.com/goadesign/goa/design/apidsl"
)

// Account is the account resource media type.
var Account = MediaType("application/vnd.account+json", func() {
    Description("A tenant account")
    Attributes(func() {
        Attribute("id", Integer, "ID of account")
        Attribute("href", String, "API href of account")
        Attribute("name", String, "Name of account")
        Attribute("created_at", DateTime, "Date of creation")
        Attribute("created_by", String, "Email of account owner", func() {
            Format("email")
        })

        Required("id", "href", "name", "created_at", "created_by")
    })

    View("default", func() {
        Attribute("id")
        Attribute("href")
        Attribute("name")
        Attribute("created_at")
        Attribute("created_by")
    })
```

Given all the benefits of code generation and having a single source of truth for your application, I submit to you that there should be another layer added to our stack of Abstractions.

Layer 8 - The Design DSL — Enabling the Single Source of Truth

In the last 40 minutes I've given you a brief history of code generation and an overview of the layers of abstraction that we take for granted as programmers. I walked you through my journey from Level One code generator to Level Five master. My call to action is simple:

Generate All The Things

Add a new layer to your Abstraction model that describes your application, and generate everything from it.

Automatic Programming

After all, every evolution in Automatic Programming came because someone was crazy enough to try it.

Thank You

https://brianketelsen.com http://learn.brianketelsen.com

Image Credits

- www.rubyonrails.org
- yeoman.io
- responsify.it
- abstractions.io
- magazineusa.com
- github.com
- apple.com
- microsoft.com
- iconion.com
- theodessyonline.com
- productguy.in
- lookatmedam.com.au
- manjr.com
- knowledgeplusaction.com