Designing C++ Code Generator Guardrails:

A Collaboration Among Outreach and Development Teams and Users

CB BAILEY
& SHERRY SONTAG

Cppcon
The C++ Conference

2024
September 15 - 20

Designing C++ Code Generator Guardrails

A collaboration between outreach and development teams & users

CppCon 2024
September 16, 2024

Sherry Sontag, Technical Expert
CB Bailey, Software Engineer

TechAtBloomberg.com

Bloomberg Engineering

About Us

Sherry Sontag

The journalist in our software shop she leads community-wide outreach projects to build support for improvements to our codebase

CB Bailey

Designs and maintains widely used infrastructure tools and processes underpinning thousands of applications and shared C++ libraries

Prologue

In response to a recent Request for Comment (RFC), a colleague complained that we had violated “the principle of least astonishment.”

We were offering a modern set of guardrails for Bloomberg’s 20-year-old Code Generator.

Others responded before we could.

They said they weren’t the least bit surprised.

Bloomberg

Engineering

Prologue

To be fair, the RFC was dense.

Still, these changes had been in motion and evolving for two years.

Teams across our Engineering department had collaborated, shared their use cases, accepted our changes, rewritten their code, and updated community cookie cutters and documentation.

Bloomberg
Engineering

Prologue

The RFC was a final alert, published to kick up any last issues before these recommendations were formally declared as policy.

Bloomberg

Engineering

Prologue

Of course, we actually *did* violate the principle of least astonishment.

Without planning, we had engineered a significant initiative that optimized thousands of libraries;

Ensured the best features were used in a feature rich tool; and

Made it far easier to continue modernizing the Code Generator.

We did it without risk!

And we did it with the consensus and participation of dozens of teams.

Our partnership

TechAtBloomberg.com

Bloomberg

Engineering

Our partnership just sort of happened

We work on different teams; different projects and on different continents.

We can’t remember actually deciding to work together.

There was never a formal assignment or project.

Bloomberg

Engineering

Our partnership

We are, however, both part of Bloomberg’s Technology Infrastructure (TI) team where shared the concern that that our huge codebase is outgrowing critical container executables.

Bloomberg has ~50,000 libraries, and the company has been around for 40 years.

In addition to infrastructure teams whose job it is to look at library architecture, there are dozens of engineers who have joined the effort as volunteers.

Bloomberg

Engineering

Our partnership across the Engineering department

The freedom to pursue an idea for the greater good is as much a part of Bloomberg's culture as is fierce independence of individual teams.

But, we would need their buy-in!

Bloomberg

Engineering

A growing service architecture

TechAtBloomberg.com

Bloomberg

Engineering

At the outset, shrinkage and growth

For nearly 20 years, there has been an effort to shrink executables to a growing network of schema-driven services.

For each subtraction, there was an addition of a client library with supporting serialization code.

The trade-off is an obvious win, but the tooling to support this shift was so attractive that it sparked an entirely new area of growth.

Bloomberg

Engineering

Enter the C++ Code Generator

Built to encourage the shift out of the containers, the Generator can create both frameworks for services and complete client libraries from the same XSD schema files.

It works so well that developers begin using it to create libraries of value semantic types. These become really popular because they are shareable across teams and even across languages.

They are so useful that Bloomberg has developed a huge vocabulary of these types, held in thousands of libraries, many outside of the service network.

Bloomberg

Engineering

The Generator can do it all

For services, impressive flexibility provides the framework for networking, threading, logging, and metrics reporting capabilities – allowing authors to focus on their business logic.

Bloomberg

Engineering

The Generator can do it all, but not always as well

For much simpler client libraries, all of that functionality is worse than irrelevant; it can be counterproductive.

Conflicting choices can hinder interoperability.

Too many options leads to bloat.

A hugely successful tool that is generating thousands of client libraries means lots and lots of bloat.

Bloomberg
Engineering

Two lines of attack converge

TechAtBloomberg.com

Bloomberg

Engineering

CB began silently saving massive amounts of space

Working under the hood to improving the Code Generator, her changes are invisible to users, risk free, and happen automatically.

Bloomberg

Engineering

Sherry made much more noise

Team by team, she reached out to ask about the very different choices being made to create similar code.

At first, most teams are protective of their processes, certain they need every option they use.

Sherry turns to CB as a fact checker and mediator.

Their conversations take off from there.

Bloomberg

Engineering

Seeing eye-to-eye

On Sherry's outreach team, this was expected.

CB's manager asked if he needed to intercede.

Thankfully CB declined.

Two lines of attack

Together, we began working out what matters and why, and enlisted other teams to comment, to complain, to negotiate.

We looked at…

- How these libraries will be built
- What they can be named
- What they can contain
- What options will be forever out of reach

Bloomberg

Engineering

Mapping it out with CMake

All of this gets a huge boost from a community shift to CMake.

For generated libraries, visibility is simple. There are essentially only two source files required – an XSD and a CmakeLists.txt.

CMake also gives CB an easy way to wrap the Generator with specialised and powerful APIs.

Counting our victories… one option at a time

TechAtBloomberg.com

Bloomberg

Engineering

Codec Choice

By default, the Generator builds-in support for all four major codecs, though often only BER, the simplest one, is needed.

Library owners are reluctant to remove existing support because a client somewhere could suffer a runtime failure. There is much less risk for new libraries, which can create the rules before they have clients.

Sherry begins pushing teams to choose a single codec, at least for new libraries.

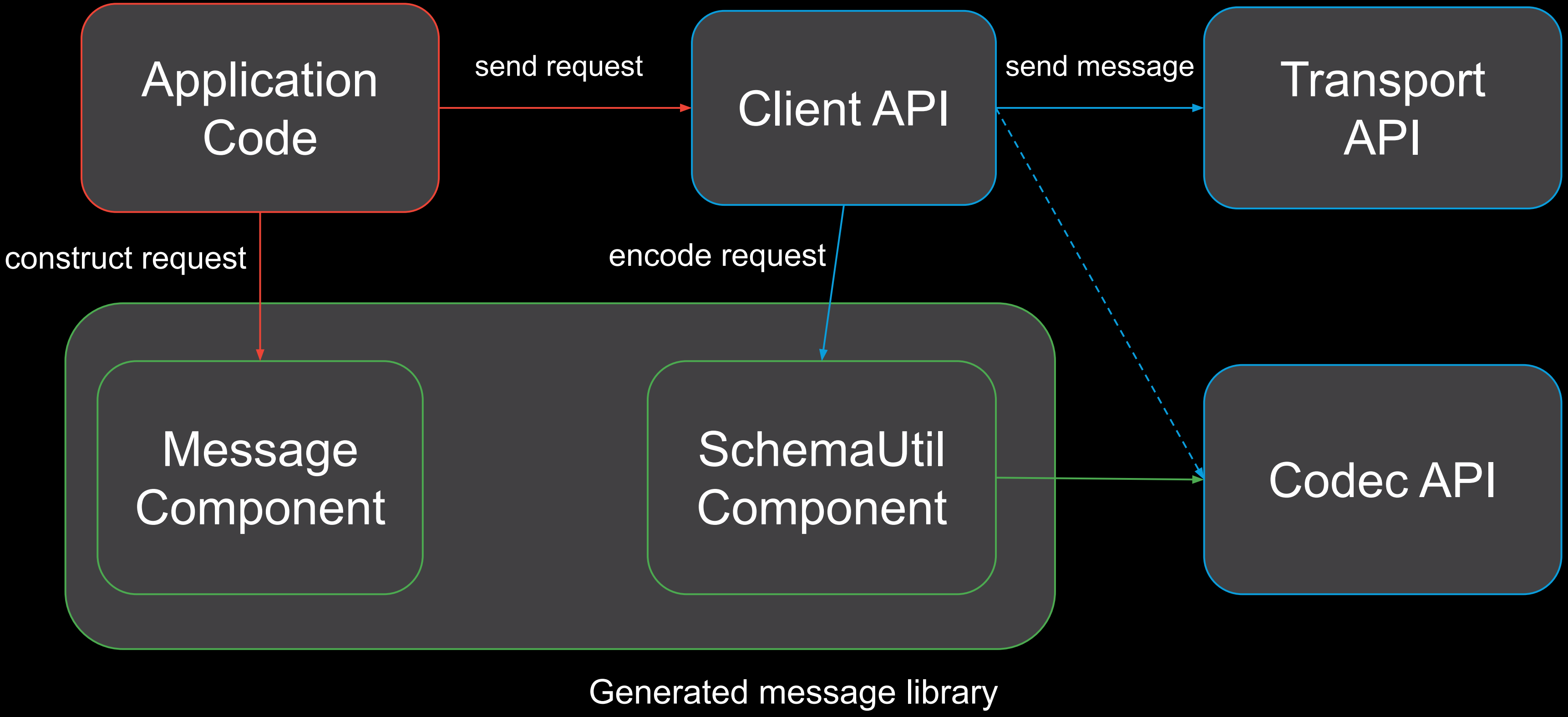
CB comes up with a solution for existing code

She asks one very important question:

What if unused generated codec support could just disappear?

She answers it with “Split CodecUtil.”

Client Message Library Use



Removing unused “SchemaUtil” code

Original client API (run time encoding choice):

```
class Client {
  public:
    template <typename t_RESPONSE, typename t_REQUEST>
    int sendRequest(t_RESPONSE       *response,
                    const t_REQUEST&  request,
                    EncodingType      encoding);
};
```

Removing unused “SchemaUtil” code (2)

Newer client API (compile time encoding choice):

```
class Client {
  public:
    template <typename t_RESPONSE, typename t_REQUEST,
              typename t_CODEC>
    int sendRequest(t_RESPONSE       *response,
                    const t_REQUEST&  request,
                    t_CODEC           codec);

};
```

Removing unused “SchemaUtil” code (3)

What does a “Codec” class look like?

```
class BerCodec {
  public:
    template <typename t_MESSAGE>
    int encode(std::streambuf   *out,
               const t_MESSAGE&  message);

    template <typename t_MESSAGE>
    int decode(t_MESSAGE      *message,
               std::streambuf *in);
};
```

Removing unused “SchemaUtil” code (4)

Originally-generated “SchemaUtil” component:

```
struct SchemaUtil {
    int encode(std::streambuf *out,
               const Request&  request,
               EncodingType    encoding);

    int decode(Response       *response,
               std::streambuf *in,
               EncodingType    encoding);
};
```

TechAtBloomberg.com

Bloomberg Engineering

Removing unused “SchemaUtil” code (5)

Newly-generated “SchemaUtil” component:

```
struct SchemaUtil {
    template <EncodingType t_ENCODING>
    struct Encoder : std::false_type {
    };

    template <EncodingType t_ENCODING>
    struct Decoder : std::false_type {
    };
};
```

Removing unused “SchemaUtil” code (6)

Newly-generated “SchemaUtil” component:

```
template <>
struct SchemaUtil::Encoder<EncodingType::e_BER>
: std::true_type {

    int encode(std::streambuf *out,
               const Request&  request);

};
```

It is a pure infrastructure win!

Removing unused code left no danger of breakage.

No negotiation was needed, since developers could continue to build with all codecs if they chose to…

The immediate reduction in task size is equivalent to several months growth at the current rate.

Defining the contents of a generated client library

Our next moves were neither silent nor subtle. It was going to take outreach, noise, and lots of negotiation.

We set out to change how these libraries were named and on what they could depend.

We mandated that libraries which contain generated code could only contain generated code, and that they must be fully regenerated every time they're updated.

Servicemsgs and Types

Teams were most accepting when we reasserted an older rule that mandated two kinds of client libraries named with suffixes that announced their purpose.

- ***“Servicemessage”* libraries** hold generated top-level request and response types used to communicate with services. These libraries never depend on one another.

- ***“Types”* libraries** contain value semantic types. Used within and outside of the service network, these generated types have evolved into a massive vocabulary that is shared across teams.

Servicemsgs and Types

The rule was practical, not technical.

Years ago, when our task linklines were hand-maintained, each new client library required a new -l rule.

At the start, that was easy.

With one client library for each service, these libraries were peers and blind to one another. There was little hierarchy to work out. They all just needed to be added to the linkline in a batch above their few core dependencies.

Servicemsgs and Types

When the first teams began using simple types from other services, we didn't realize this was the start of something much bigger.

To avoid a few hierarchy issues that were impacting the placement of hundreds of -ls, we asked that shareable libraries be named differently.

Servicemsgs and Types

As it turns out, that separation is more valuable than we anticipated, even now when linklines are auto-generated.

Types libraries are an important commodity, and declaring that design decision makes for cleaner code.

It also turns out that using one servicemessage library to provide types for a second leaves one set of request and response types orphaned and bloats applications that can only ever use one service.

TechAtBloomberg.com

Bloomberg

Engineering

Servicemsgs and Types

We asked CB to take advantage of the split and to create new client CMake APIs to wrap the Code Generator.

We made them mandatory, as they prevent most unnecessary options from ever being implemented for client code.

These APIs also enable CB's team to iterate, ensuring code is always built using their latest changes.

Stripping Debug Information

CB’s team has long been reclaiming critical space by stripping debug information from libraries containing well-understood generated code.

By design, this approach also strips debug information from any handwritten code.

Stripping Debug Information

Some people were annoyed that the new APIs always left no room for their own business logic.

Others had been adding in code they wanted in their client libraries.

CB taught the Code Generator new tricks – for instance, adding service identification and version information in a compile-time-friendly component.

Negotiations in full swing

Developers began pointing out generated library design issues from their own teams, often legacy design decisions that they found befuddling.

They allowed us to convince them that other Code Generator options they routinely added were never *actually* needed.

Bloomberg

Engineering

Ending this chapter

TechAtBloomberg.com

Bloomberg

Engineering

Turning our recommendations into policy

We presented our work as a related series of proposals in an RFC.

There was a bit of argument.

People mostly appreciated the changes.

Bloomberg

Engineering

The project taught us even more about outreach

- Never just say no. Look at the code. People have good ideas.
- Reasonable use cases need a solution, even when infrastructure needs to be tweaked.
- If you have a better idea, explain it.
- People will do the right thing when they know why!

The impact on the code base has been huge.

Our unplanned initiative helped optimize thousands of libraries.

We saved tons of critical space.

The wins came with NO RISK!

They were accomplished with overwhelming community consensus.

Bloomberg

Engineering

Thank you!

https://TechAtBloomberg.com/cplusplus

https://www.bloomberg.com/careers

Bloomberg Engineering

TechAtBloomberg.com