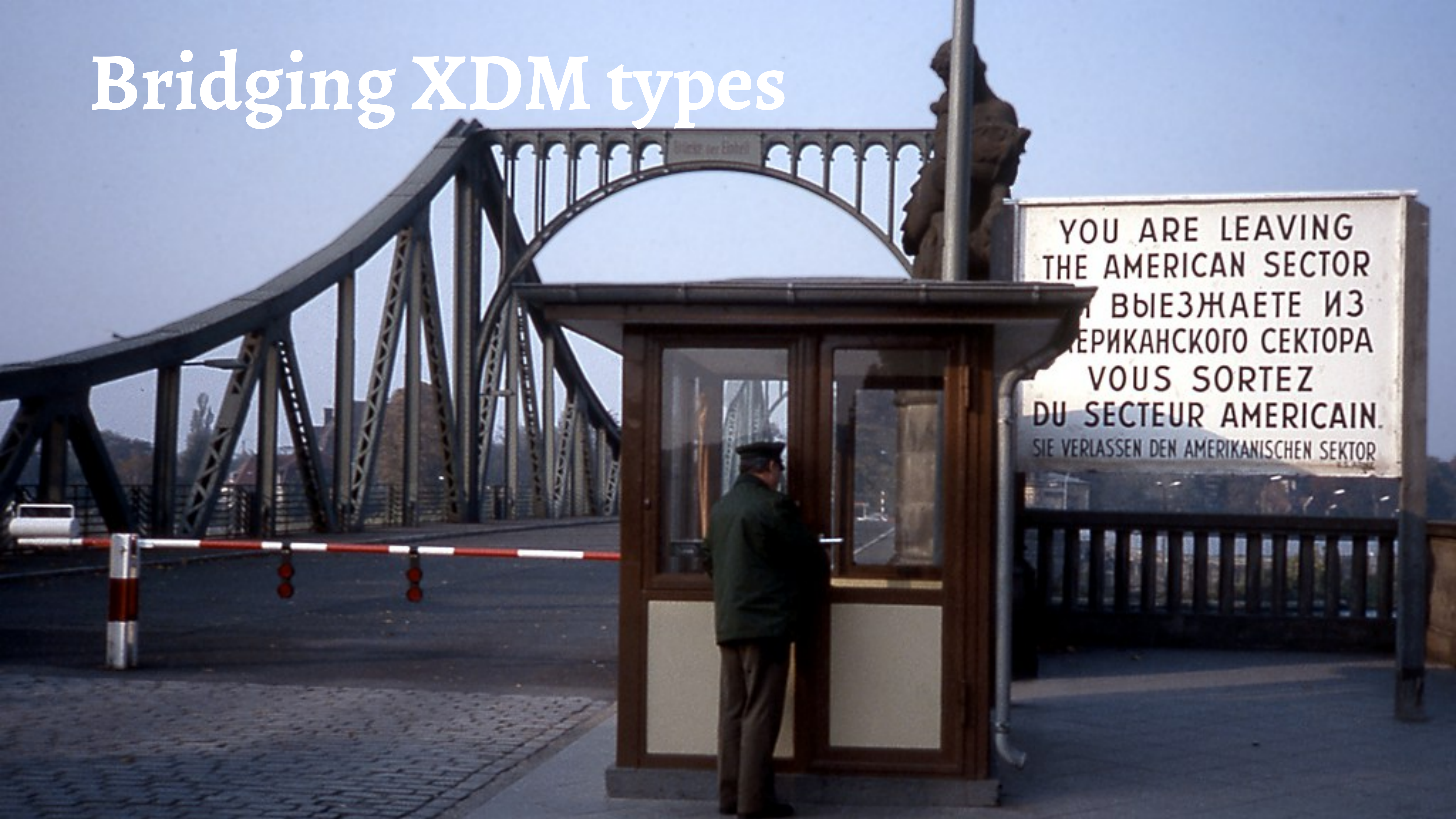
Bridging XDM types in multiple native type systems

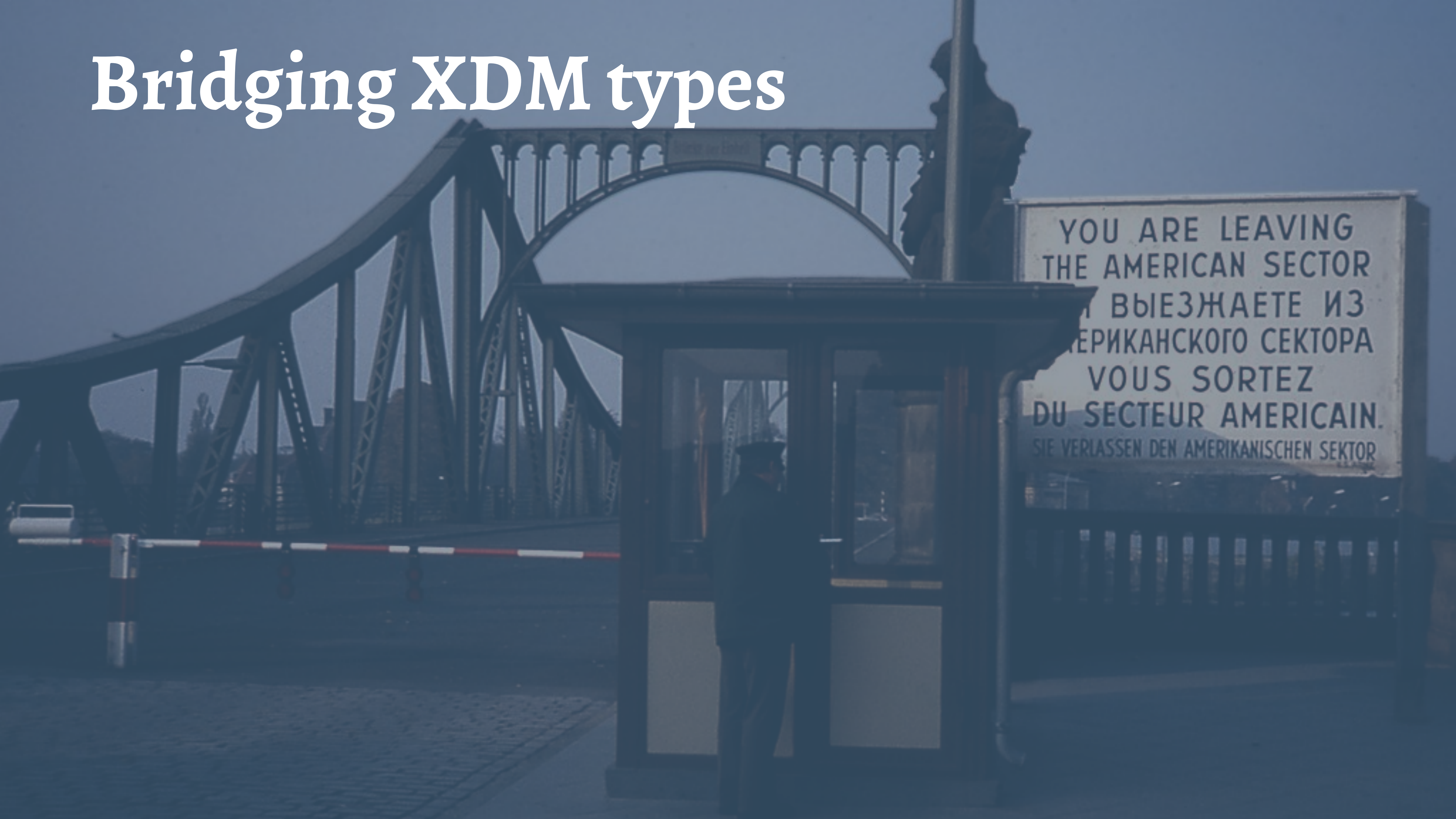
O'Neil Delpratt, oneil@saxonica.com
Matt Patterson, matt@saxonica.com

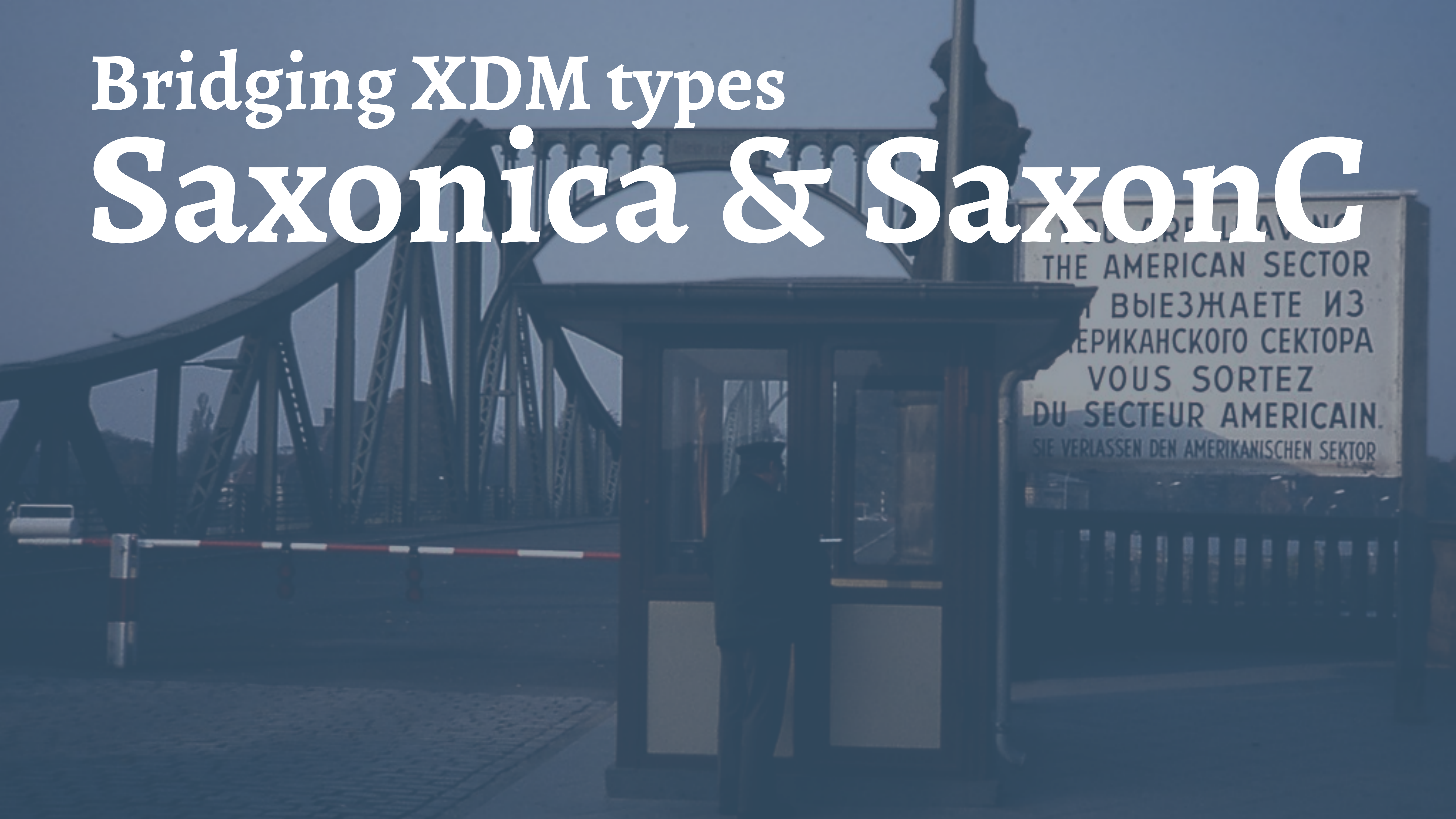
· *XMLPrague 2024 Conference*

Bridging XDM types



Bridging XDM types



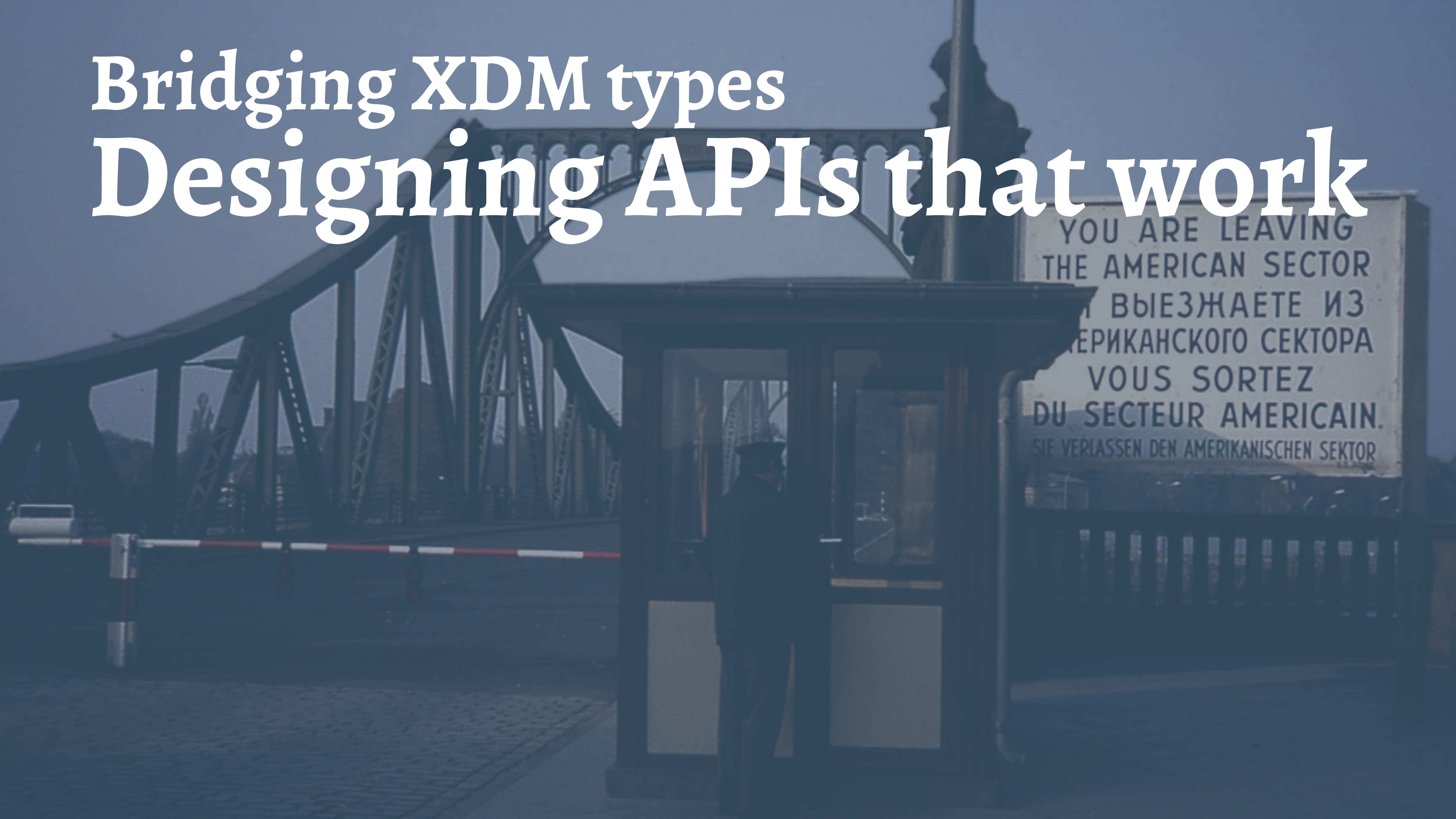


Bridging XDM types

Saxonica & SaxonC

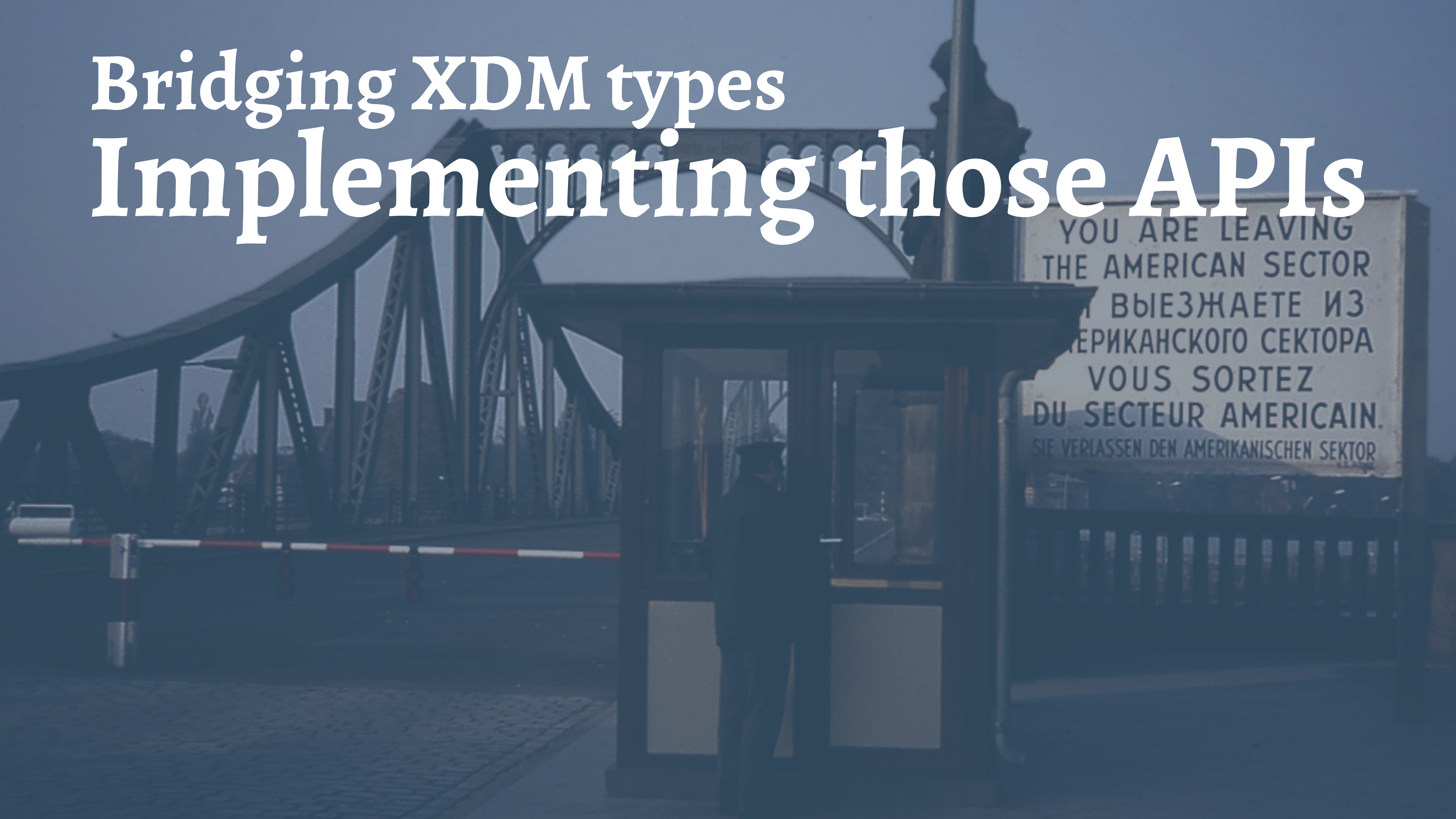
Bridging XDM types

Designing APIs that work



Bridging XDM types

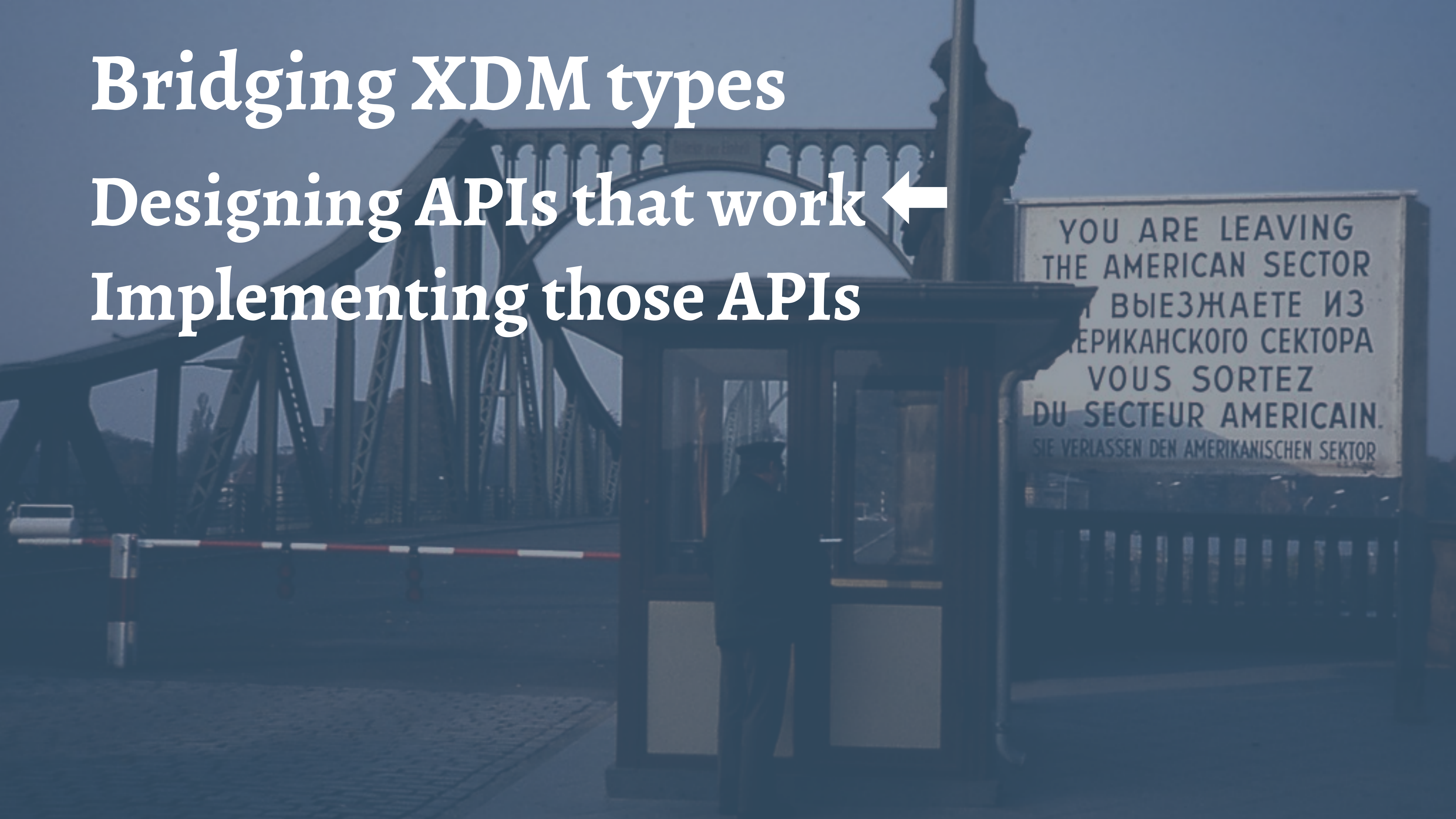
Implementing those APIs



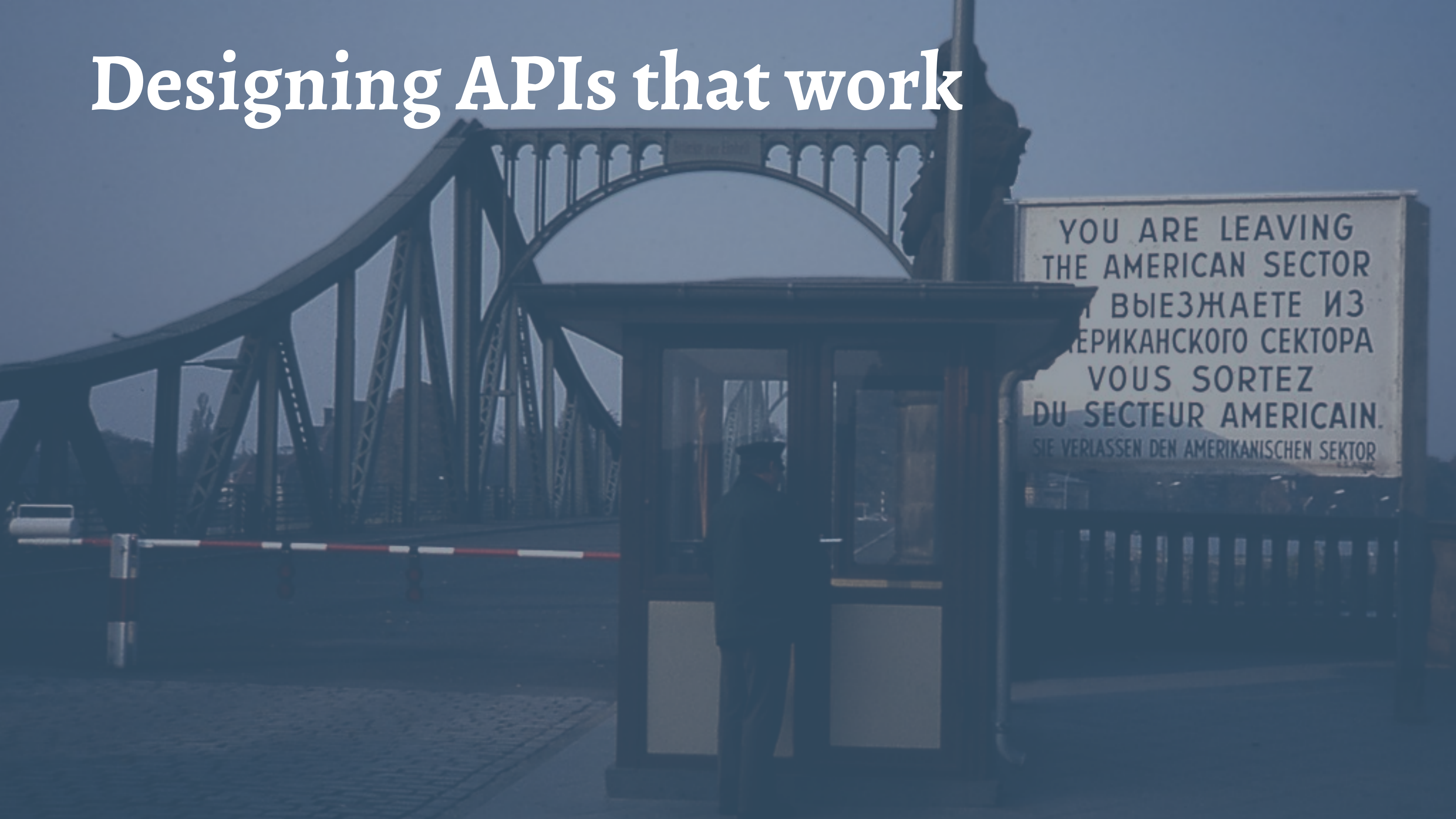
Bridging XDM types

Designing APIs that work ←

Implementing those APIs



Designing APIs that work



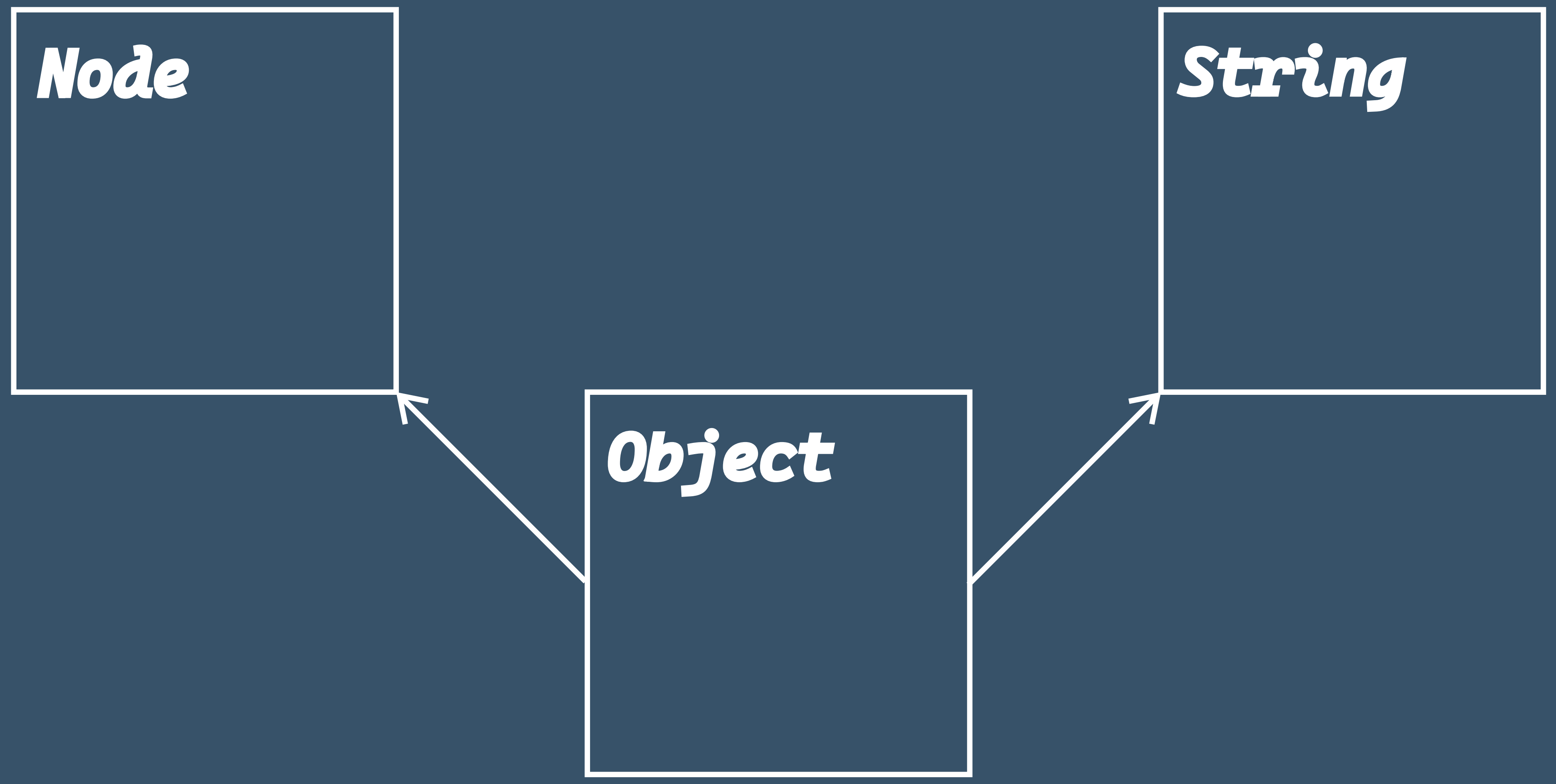
Exploring the type system gap

Exploring the type system gap

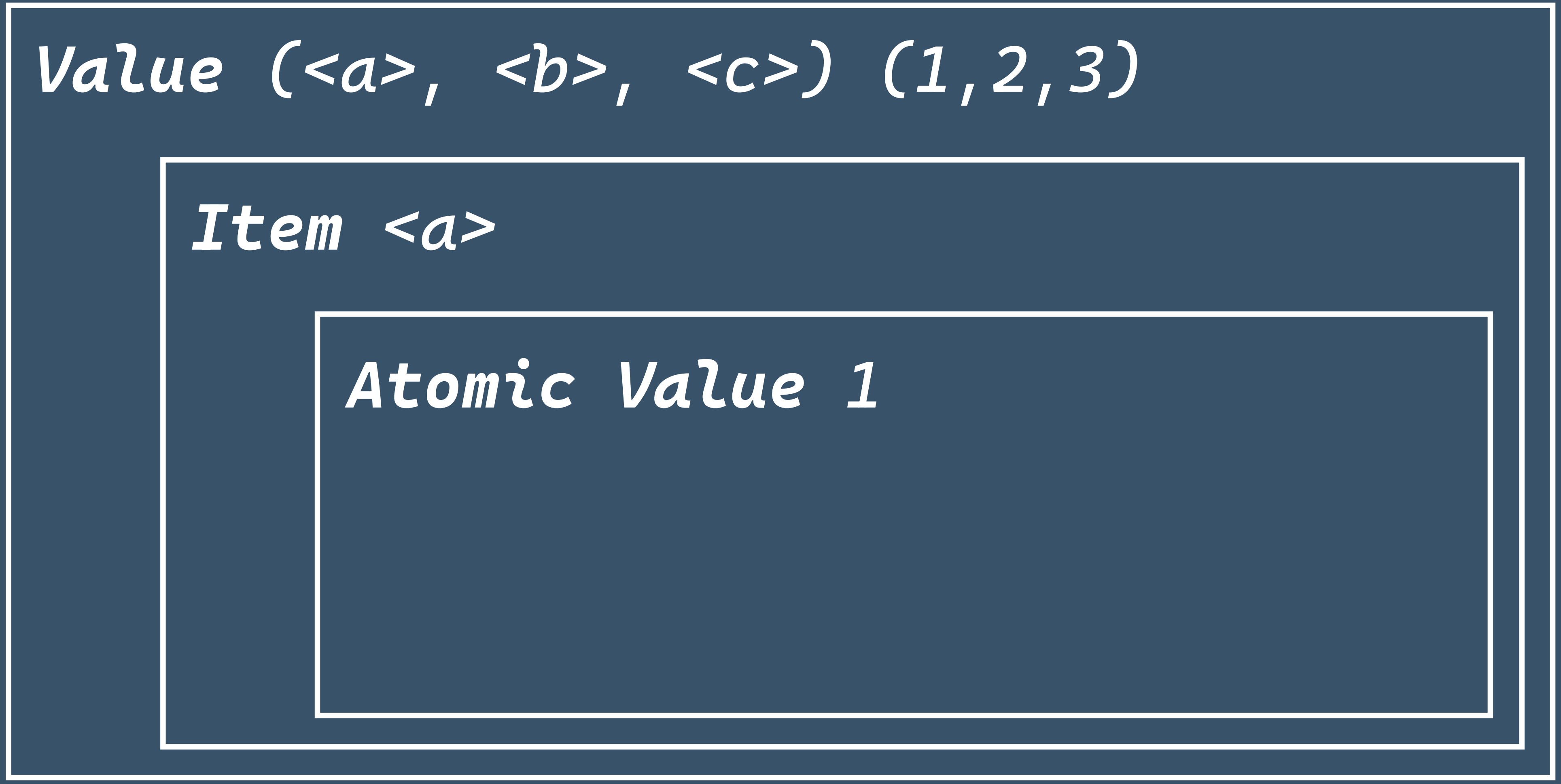
— What's ‘Primitive’?

— What's Inheritance

Exploring the type system gap



Exploring the type system gap



Value (<a>, <b>, <c>) (1,2,3)

Item <a>

Atomic Value 1

The sequence-likeness monster

The sequence-likeness monster

- Values (Sequences) are the ‘simplest’ type
- All Atomic Values are also 1-item sequences.

How long is a (piece of) String?

How long is a (piece of) String?

The following statements are all true in XDM:

- The length of the XDM Atomic Value representing the string "Hello World" is 11.
- The length of the XDM Atomic Value representing the string "Hello World" is 1.
- The length of the XDM Atomic Value representing a string containing 2^10 characters is 1.

Functionally speaking

In XPath, `fn:string-length()` and `fn:count()` make the distinction between the sequence-likeness of a string and the stringiness of a string clear

Similarly, `map:get()` is very different to the `[]` operator.

Numbers

Numbers (XDM)

```
xs:float
xs:double
xs:decimal
   └─xs:integer
       ├─xs:nonPositiveInteger
       │   └─xs:negativeInteger
       │ xs:long
       │   └─xs:int
       │       └─xs:short
       │           └─xs:byte
       └─xs:nonNegativeInteger
           ├─xs:unsignedLong
           │   └─xs:unsignedInt
           │       └─xs:unsignedShort
           │           └─xs:unsignedByte
           └─xs:positiveInteger
```

Numbers (Javascript)

`Number`

Maps

Maps

Consider a simple XDM Map:

```
let $simples := map { "a" : "obviously" }
```

We can get the value of "a" with map:get()

```
map:get($simples, "a")
```

Maps

Of course, we can also call the map to get the value, because it's also an XDM `FunctionItem`:

```
$simples("a")
```

And let's not forget about also being a Value:

```
$simples[1]("a") = "obviously"
```

Maps

Compare this with a Python Dictionary:

```
>>> simples = {"a": "obviously"}
>>> simples[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 1
>>> simples["a"]
'obviously'
```

Maps

```
map(xs:date, map(xs:byte, xs:string))
```

What do you need to be able to do in order to provide idiomatic local language API access to XDM Maps?

```
map[datetime.date.today()][42]
```

```
map.get(xdm.date.today()).get(42)
```

```
map.get(xdm.date.today()).get(xdm.byte(42))
```

Whose idiom is it anyway?

Whose idiom is it anyway?

Competing conventions

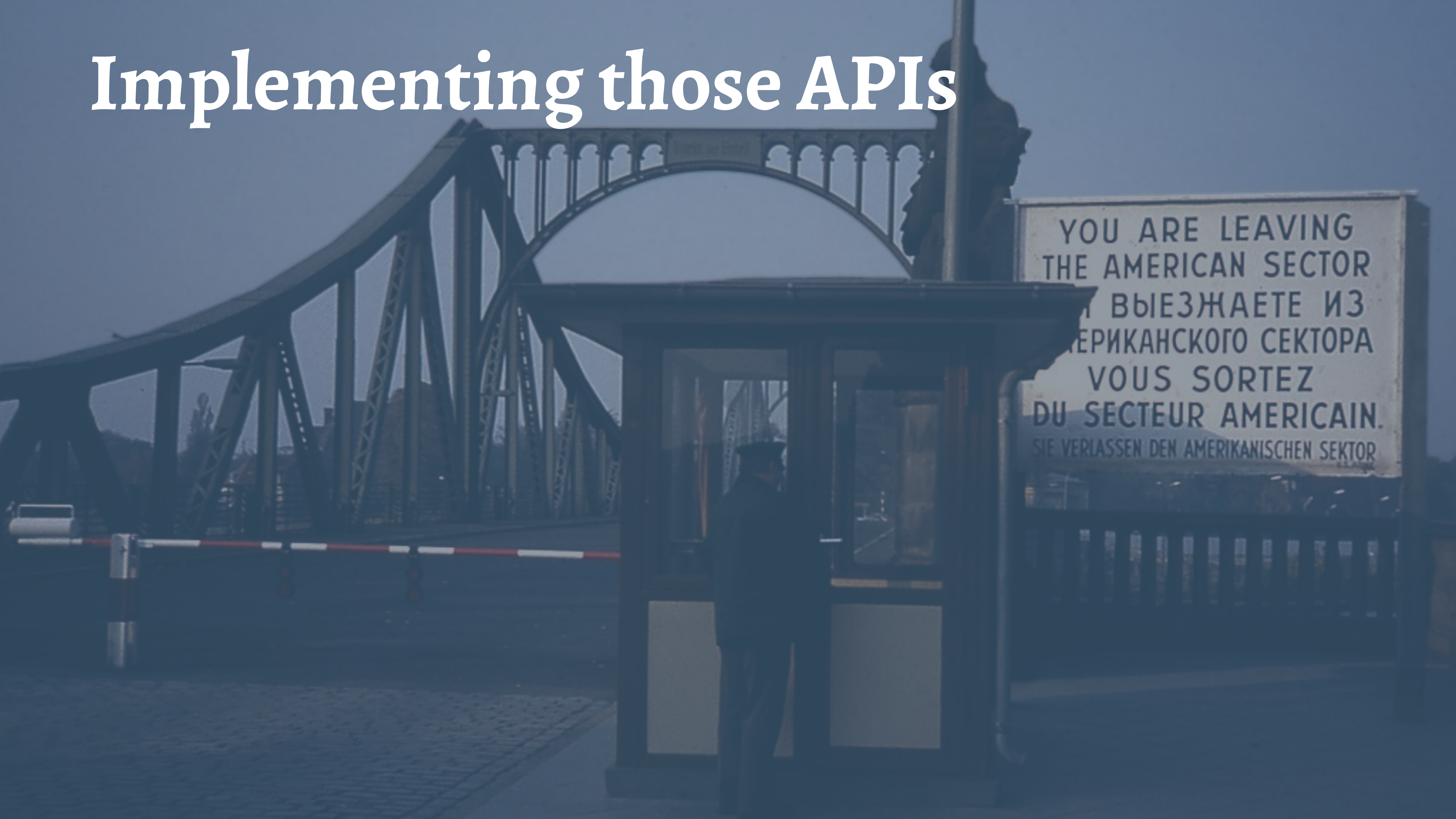
- What do you choose when `[]` would make sense for indexing into a sequence, looking up a key in a map, and slicing a string?

Whose idiom is it anyway?

Contextual ignorance

- Maybe it's okay to ignore some XDM aspects that make no sense in the context of a different language and type system.
- Atomic Values make no sense as sequences outside of XPath, so maybe we can just make them be non-sequence-like in our API.

Implementing those APIs



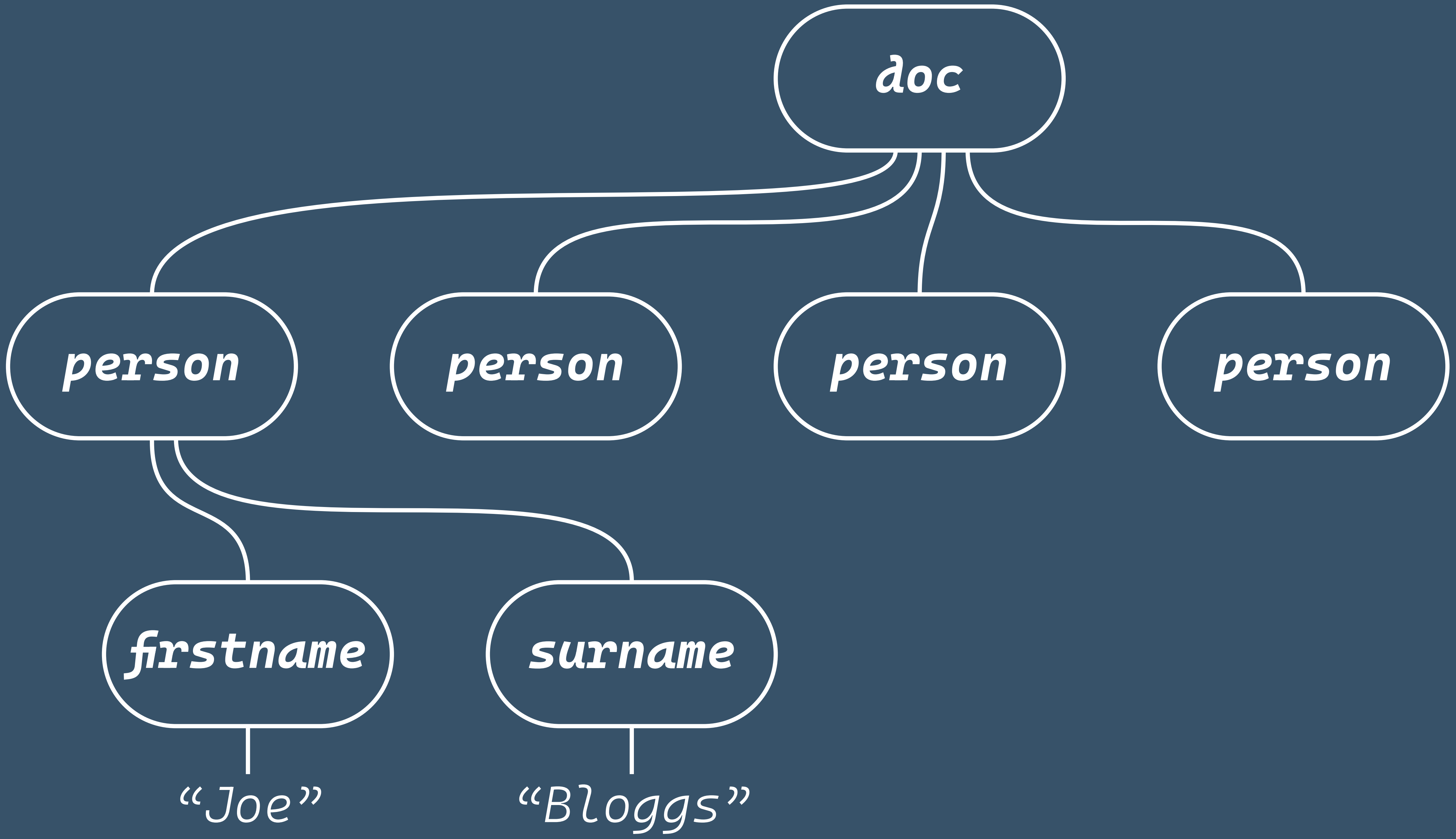
Traverse an XML Document

```
<doc>
  <person id='x1'>
    <firstname>joe</firstname>
    <surname>bloggs</surname>
    <telephone>+4400000</telephone>
    ...
  </person>
</doc>
```

XPath

```
//person
/doc/person[2]/firstname
/expr/../..
```

XML Tree structure



XDM Node Navigation via API

```cpp
//C++ Code
int childCountA = node->getChildCount();
XdmNode **childrenA = node->getChildren();
XdmNode *child = childrenA[0];
XdmNode **children = child->axisNodes(EnumXdmAxis::CHILD);
int childCount = child->axisNodeCount();
for (int i = 0; i < childCount; i++) {
    const char *childStr = children[i]->toString();
    cout << "child node:" << (childStr) << endl;
    operator delete((char *)childStr);
}
for (int i = 0; i < childCount; i++) {
    delete children[i];
}
delete[] childrenA;
delete node;
```

Code stuff

```
(: XPath :)
package[@role='secondary']

// Java
for (XdmNode pack : testInput.select(
    child("package").where(
        attributeEq("role","secondary"))).asListOfNodes()
{...}

# Python
packs = (pack for pack in testInput.children if (
        pack.name == 'package' and
                pack.get_attribute_value('role') == 'secondary')
)
```

Implementing APIs in Multi-tier systems?

SaxonC:

- built on Saxon-J (Java)
- GraalVM: Cross-compiled to native

  GraalVM: JVM implementation that provides the ability to compile Java down to native code ahead-of-time

Core languages: Java <- -> C/C++
Extensions in Python (using cython), PHP

Problems with Multi-tier systems

Multi-tier programming languages
Garbage Collection (GC) issues with created XDM objects ->
* Java GC,
* C++ unmanaged code
* PHP/Python GC

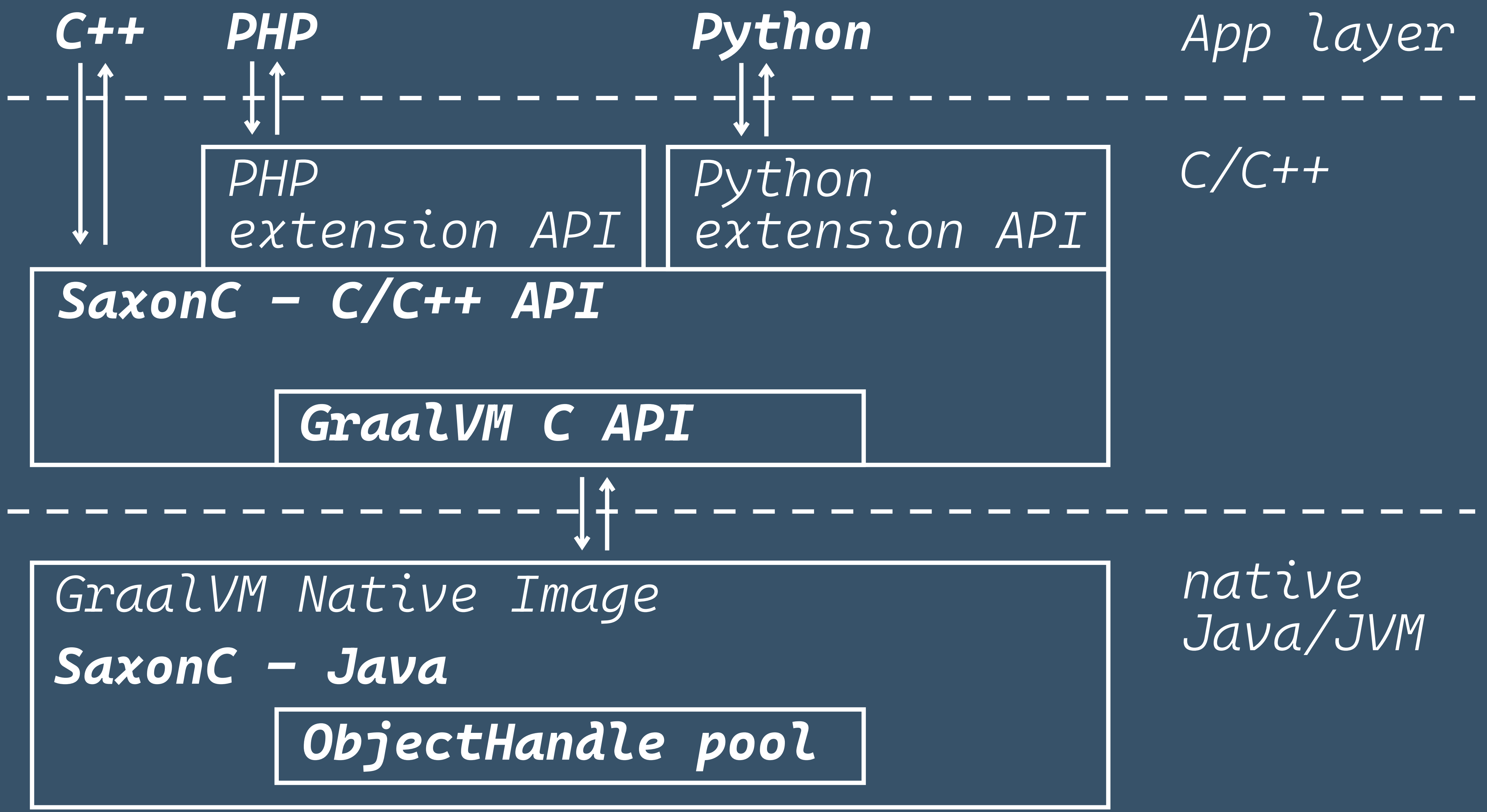
Solution for the Java GC problem

keeping objects alive when still in use in C++

GraavlVM's API:

- ObjectHandle
- ObjectHandle pool

Solutions in SaxonC



PHP/Python GC problems

Examples where problems can occur

```
node_ = sp.parse_xml(xml_file_name=xmlFile)
output = executable.apply_templates_returning_value(xdm_value=node_)
executable.set_parameter("param1", node_)

# What happens to node_ at this point?
```

Another Examples where problems can occur

```
saxonproc = PySaxonProcessor()
valuei = saxonproc.make_array([saxonproc.make_integer_value(i) for i in [8,9,10]])
executable.set_parameter("param2", values) # Undefined behaviour
....
```

How do we solve this problem?

Our own Memory management in C++
* XDM Object reference counting
* Caching of child nodes for XDM parent node/
* Tracking accessed XDM Items in XdmValue and XDM child nodes

Examples

```
node_ = sp.parse_xml(xml_file_name=xmlFile) # refCount +1
output = executable.apply_templates_returning_value(xdm_value=node_) # node_ refCount??,  output refCount +1

valuei = saxonproc.make_array([saxonproc.make_integer_value(i) for i in [8,9,10]]) # int refCount +1
executable.set_parameter("param1", node_) # node_ refCount +1,
executable.set_parameter("param2", values) # value refCount +1

# What is the refCount of node_ at this point?
```

Conclusion

Conclusion

XDM's view of the world is different

XDM wrappers need to be a first-class part of any API

Conclusion

Mixing managed GraalVM, unmanaged C++, and managed host language code is complex.

Conclusion

We still have a lot of room to improve, and we hope that this survey of some of the higher-level challenges and lower-level engineering will be useful to other implementers and users, as well as ourselves.

Thank you & Questions

Glienicker Brücke in the Cold War photo: David Stanley[1]

Glienicker Brücke now photo: Konstantin's Europe and more[2]

[1] https://www.flickr.com/photos/davidstanleytravel/21587205403. Some rights reserved, CC-BY-2.0.

[2] https://www.flickr.com/photos/konstantinseurope/34278642053. Some rights reserved, CC-NC-BY-ND-2.0