Application Programming Interfaces (APIs)

Software Architecture

March 28, 2022

Brae Webb

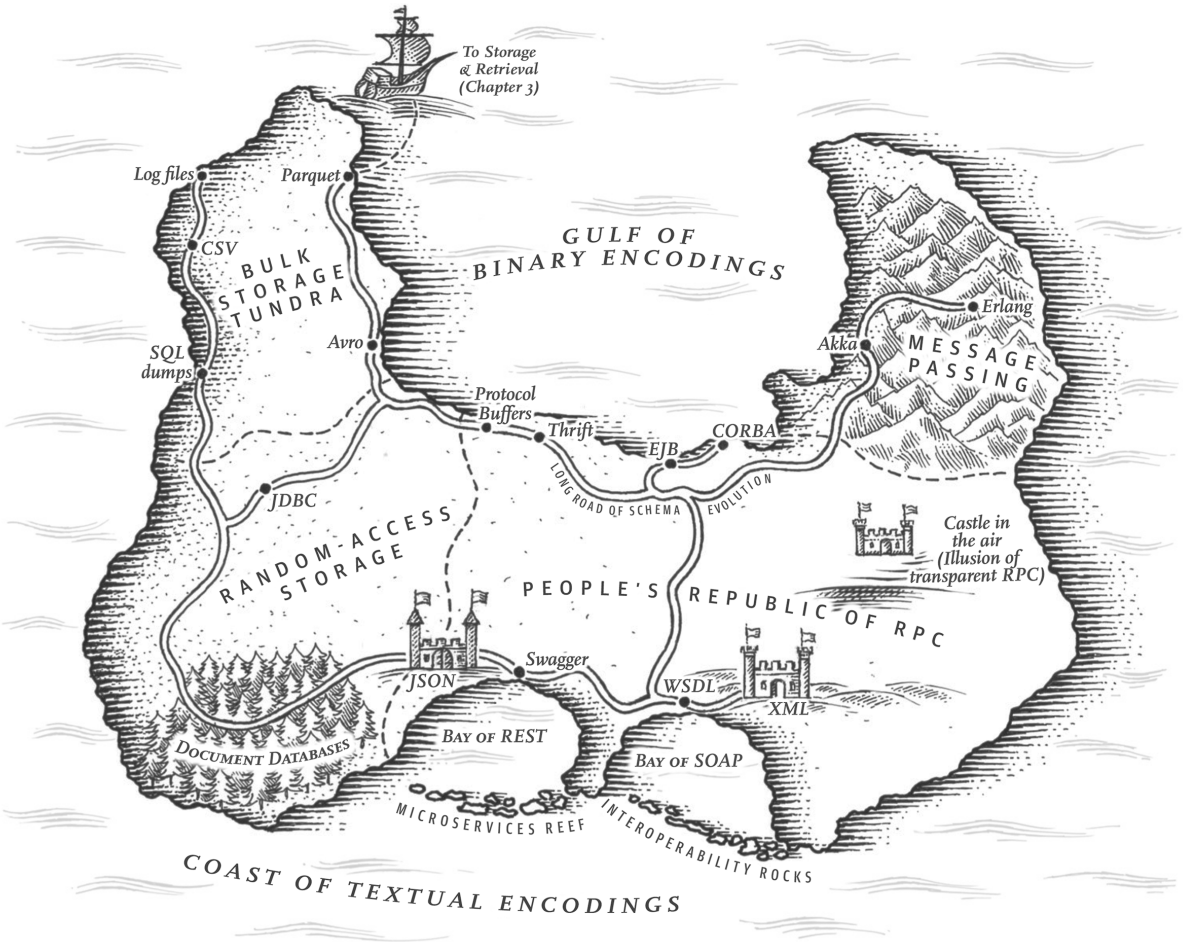

Figure 1: A map of communication techniques from Designing Data-Intensive Applications [1].

1 This Week

This week our goal is to:

- explore the various techniques developers use to communicate between distributed systems;
- deploy a static website on an S3 bucket; and
- configure the static website to communicate with an EC2 backend.

2 Communication

This week we have started to explore the world of distributed systems. The world relies (heavily) on distributed systems, there is no machine in the world powerful enough to process the requests Google receives every second[1]. Even systems of a much smaller scale can still need many hundreds or thousands of machines working seamlessly together. This inter-machine teamwork requires machines to know how to talk to each other. For the practical this week we will look at APIs as a mechanism for communication.

3 Data Formats

Communication requires an exchange of information. On computer systems information is stored at runtime in memory as primitive data which your programming language can interpret; bytes, integers, strings, etc. In object-oriented languages, primitive data is wrapped up into useful packages: objects. If we want this information to escape the confines of our programming language runtime, we need to package it up in a language-independent format. For a format to be language-independent we just need many languages to implement an encoding and decoding mechanism for the format. We have a number of language-independent formats available but a few defacto standards.

3.1 XML

Extensible Markup Language (XML) is one of the most widely used language-independent formats. The use cases of XML are extensive, it's the foundation for many popular utilities[2], such as SVG file formats, SAML authentication, RSS feeds, and ePub books.

```
   » cat csse6400.xml
<root>
  <item>
    <key>Course Code</key>
    <value>CSSE6400</value>
  </item>
  <item>
    <key>Course Title</key>
    <value>Software Architecture</value>
  </item>
</root>
```

XML is designed as a markup language, similar to HTML, it is not designed as a data exchange format. Developers have come to point out that the verbosity and complexity of XML, compared to alternatives such as JSON, are deal breakers. While XML can be used as a data exchange format it is not designed for it, and as a result APIs built around XML as a data format are becoming less common.

3.2 JSON

JavaScript Object Notation (JSON) is quickly replacing XML as the data format used in APIs. As you will note, it is more succinct and communicates the important points to a human reader better. The popularity

[1] Current estimates are that Google requires over a million servers

[2] https://en.wikipedia.org/wiki/List_of_XML_markup_languages

of JSON is largely due to its compatibility with JavaScript which has taken over web development. JSON is the map-esque data type in JavaScript. Detractors of JSON claim that its main disadvantage compared to XML is that it lacks a schema. However, schemas are possible in JSON[3], they are optional, just as in XML, but are used much less than in XML.

```
» cat csse6400.json
{
    "Course Code": "CSSE6400",
    "Course Title": "Software Architecture"
}
```

3.3 MessagePack

It should not be a surprise that the JSON and XML formats are not resource efficient. Nowadays, we are less concerned with squeezing data into a tiny amount of data on the hard drive as our hard drives are massive. However, we are often concerned with how much data is being transmitted via network communication.

In the example JSON snippet above, we use 78 bytes to encode the message. MessagePack[4] is a standard for encoding and decoding JSON. When encoding our original JSON snippet with MessagePack we shrink to just 57 bytes. At our scale, a negligible difference, but at the scale of terrabytes or petabytes, a significant consideration.

```
» cat csse6400.msgpack
82 ab 43 6f 75 72 73 65 20 43 6f 64 65 a8 43 53 53 45 36 34 30 30 ac 43 6f 75 72 73
   65 20 54 69 74 6c 65 b5 53 6f 66 74 77 61 72 65 20 41 72 63 68 69 74 65 63 74 75
   72 65
```

Info

For those interested, 0x82 specifies a map type (0x80) with two fields (0x02). Followed by a string type (0xa0) of size eleven (0x0b). The rest is left as an exercise: https://github.com/msgpack/msgpack/blob/master/spec.md.

3.4 Protobuf

Protocol Buffers (protobuf) is another type of binary encoding. However, unlike MessagePack, the format was designed from scratch, allowing a more compact and better designed format. Protobufs require all data to be defined by a schema. For example:

```
» cat csse6400.proto
```

[3] https://json-schema.org/
[4] https://msgpack.org/

```
message Course {
    required string code = 1;
    required string name = 2;
}
```

Protobufs differ from XML, JSON, and MessagePack via their method of integration. In the previous examples, your language would have a library to encode and decode the data format into and out of your language's type system. With protobuf, an external tool, *protoc*, takes the schema and generates a model of the schema in your target language. This gives every language a native method to interact with the data format, it often means that developers do not need to be aware of the underlying encoding.

```
   » cat csse6400.java

Course softarch = Course.newBuilder()
    .setCode("CSSE6400")
    .setName("Software Architecture")
    .build();
output = new FileOutputStream(args[0]);
softarch.writeTo(output);
```

4 Application Programming Interfaces

4.1 XML-RPC

4.2 SOAP

4.3 REST

4.4 JSON-RPC

4.5 GraphQL

4.6 gRPC

5 Deploying a Todo Part Two

In the practical last week we deployed a very simple todo application. Recall that we deployed a database and an EC2 instance which acted as a docker container. The EC2 instance was responsible for:

1. allowing users to download the client-side HTML and Javascript frontend; and
2. providing a REST API the client talked to in order to access persistent data.

This means that each time the client opens the website they will need to download the frontend and make all subsequent requests to the same instance. This does not scale well. In this practical, we will make one simple change that helps increase our capacity.

The todo application was built using a simple layered architecture.

1. The client-side (or presentation layer) is developed in Elm[5] which compiles into static HTML and JavaScript.
2. The presentation layer communicates with a Laravel[6] persistence layer via a REST API.
3. The persistence layer in turn communicates with a MySQL[7] database layer.

We have already deployed the database layer separately from the presentation and persistence layer. Now, we want to deploy the service which serves the presentation layer separately from the persistence layer. Since our presentation layer is static, we can take advantage of the AWS S3 service. AWS S3 is scalable, which means that we do not have to worry about manually scaling our frontend serving (and can focus on the challenge of backend scaling).

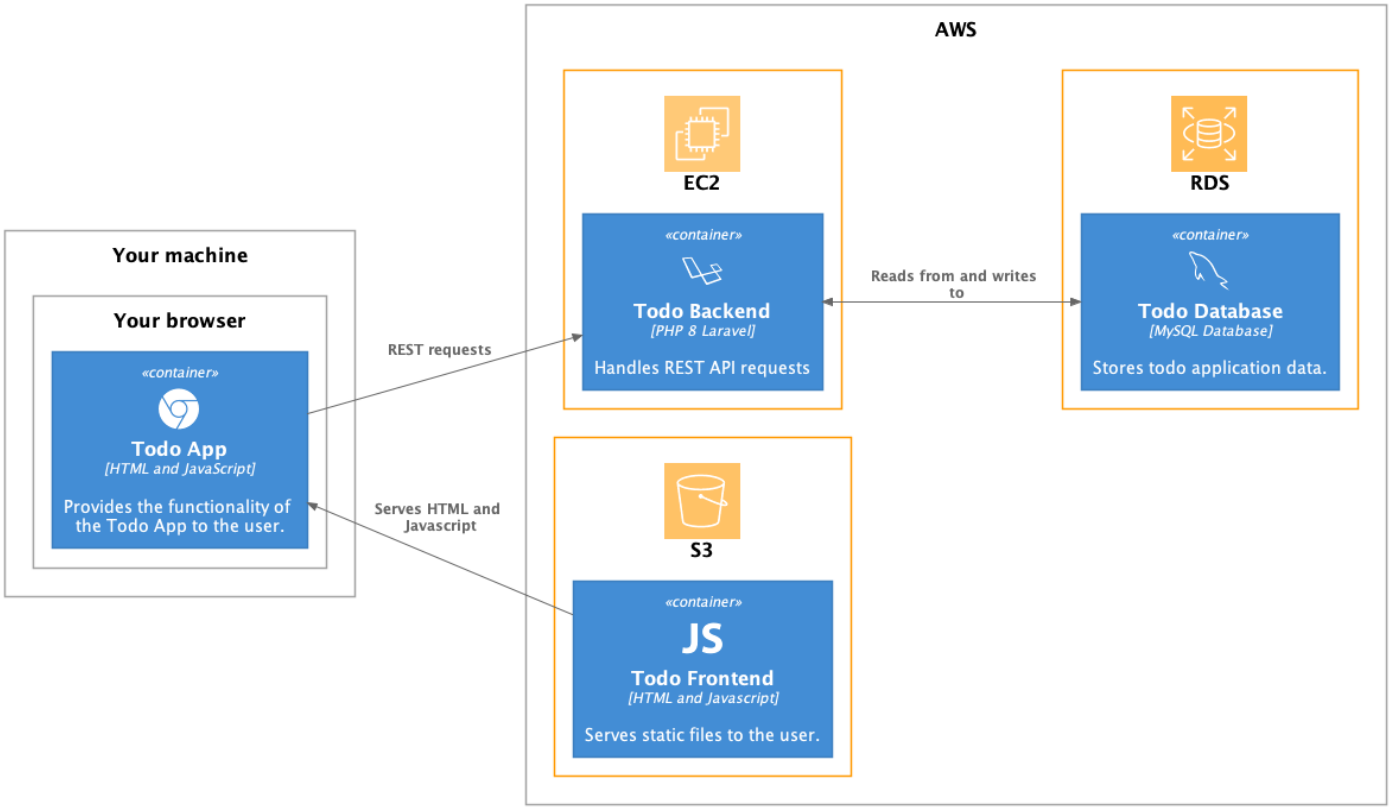

Figure 2: Desired deployment diagram of the todo application.

References

[1] M. Kleppmann, *Designing Data-Intensive Applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media, Inc., March 2017.

[5] https://elm-lang.org/
[6] https://laravel.com/
[7] https://www.mysql.com/