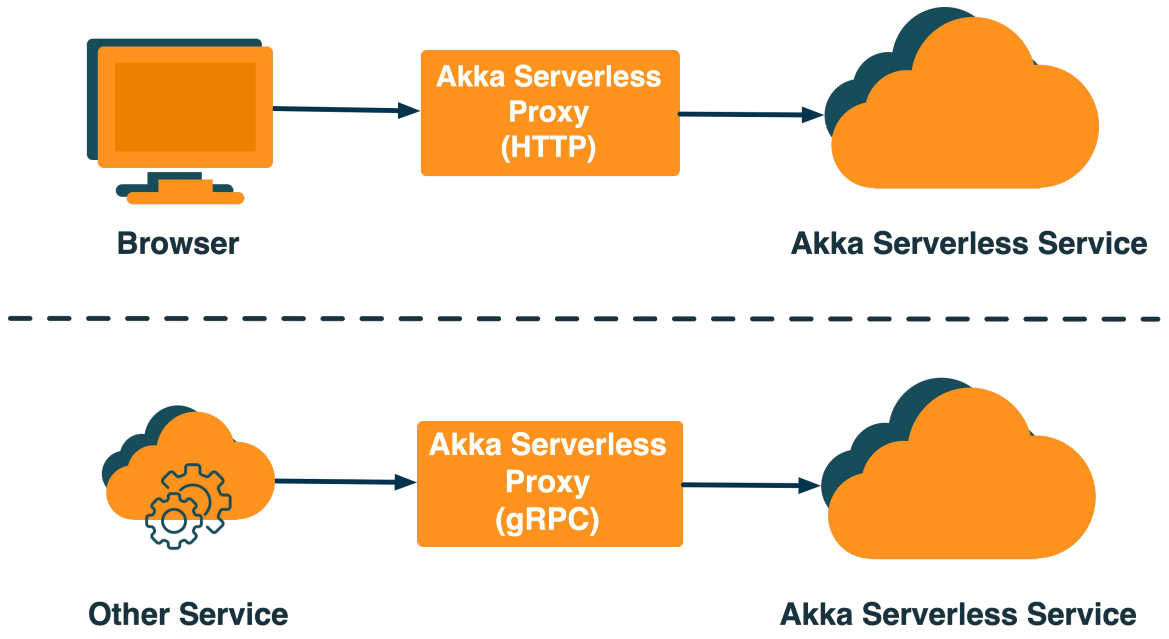
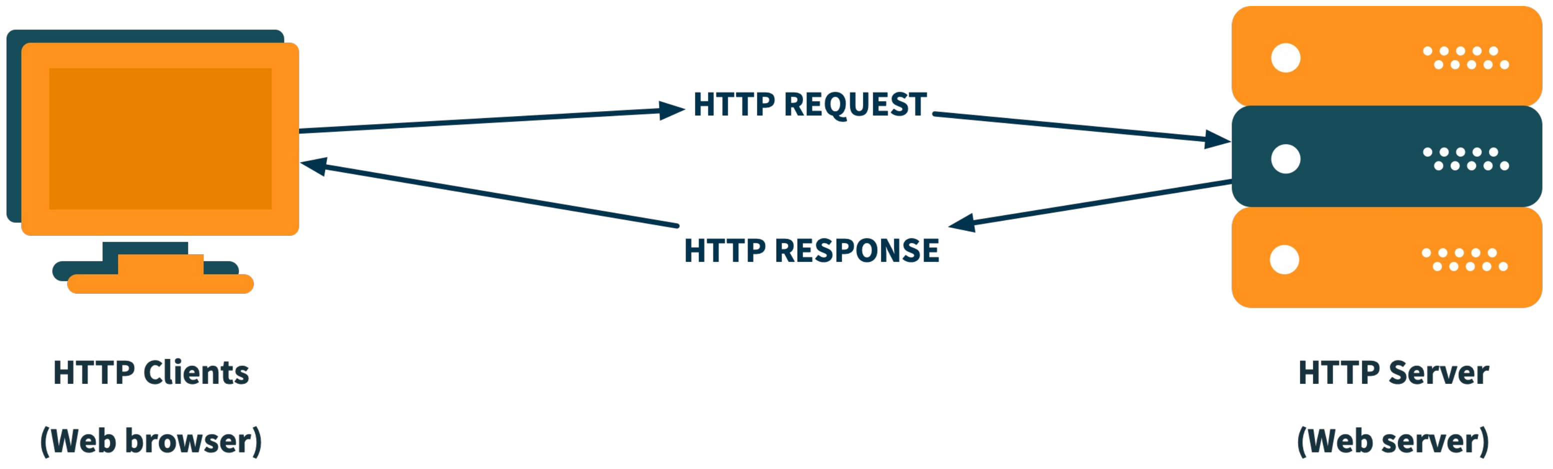
gRPC Client

Communicating



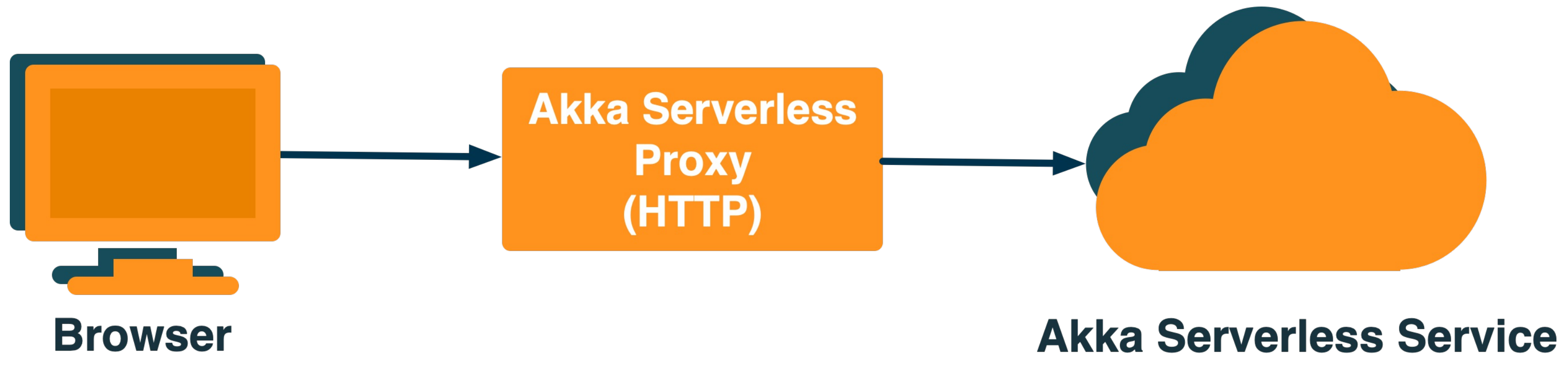
- Now that we have an Akka Serverless Service it is important to understand how to interact with it.
- Akka Serverless provides two methods of communicating:
  - HTTP
  - gRPC
- Typically, we might use HTTP from a browser, and gRPC from other services.

Notes on HTTP



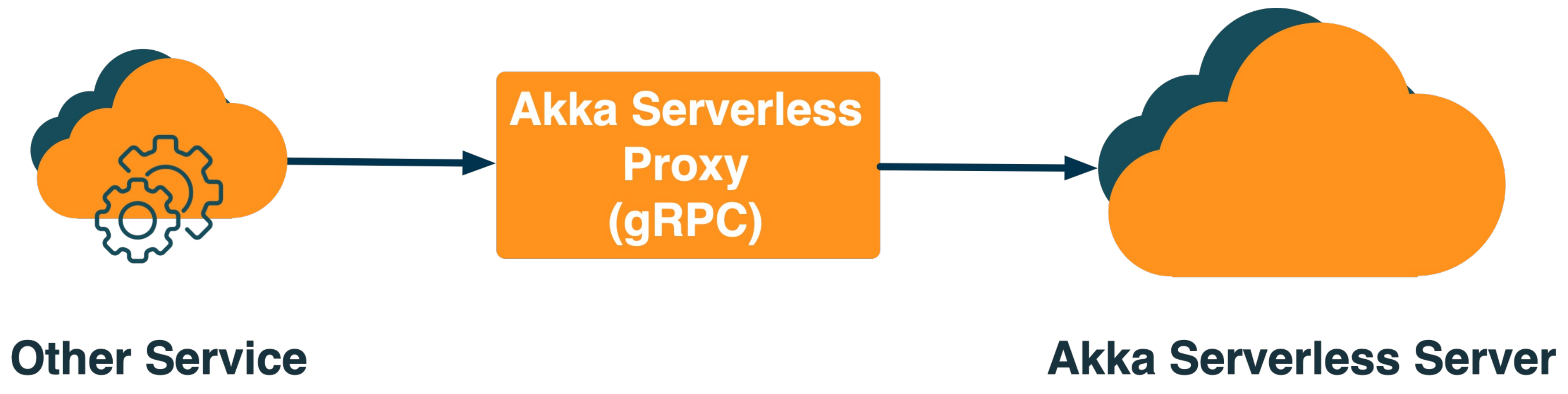
- HTTP (Hyper Text Transfer Protocol) is the basic protocol used by web browsers.
- The most widely supported version of HTTP is 1.1.
- gRPC is based on HTTP/2 which includes powerful new features.
  - Example: Bidirectional streams of data.
- Unfortunately, raw HTTP/2 is not available in all web browsers.

gRPC from the Browser



- The Akka Serverless Proxy includes a gRPC proxy to enable HTTP 1.1 support.
- It enables your frontend JavaScript code to make simple HTTP requests to your backend service.
- We will discuss this in more detail later in the course.

gRPC from Node.js



- For JavaScript on the server-side, node.js is the main runtime.
- In this environment, the constraints of browser compatibility are removed.
- We can use a JavaScript gRPC client natively to interact with our service.

Creating a gRPC Client

Dependencies

```
const grpc = require('@grpc/grpc-js');
const protoLoader = require('@grpc/proto-loader');
```

- In this course we will use grpc-js, a pure JavaScript implementation of *gRPC*.
- In order to create a *Client* we need two libraries:
  - '@grpc/grpc-js' allows us to construct our gRPC *Client*.
  - '@grpc/proto-loader' handles the loading and parsing of our .proto file.

Credentials

```
const credentials = grpc.credentials.createInsecure();
```

```
const credentials = grpc.credentials.createSsl();
```

- To connect to our gRPC *Service* we will need *Credentials*.
- When running locally we typically use *Insecure* credentials.
- When running on Akka Serverless we typically use *Secure* credentials (eg. Ssl).

Loading the .proto File

```
const packageDefinition = protoLoader.loadSync('helloworld.proto');
```

- To communicate with our *Service* we will need access to the *Schema*.
- loadSync from '@grpc/proto-loader' allows us to load our .proto file.
- It is a blocking operation that takes the path to the .proto file.
- It returns a packageDefinition, which is a JavaScript representation of our .proto definitions.

Creating the Descriptor

```
const pkg = grpc.loadPackageDefinition(packageDefinition);
const descriptor = pkg.com.example.helloworld;
```

- Next, we load the package from the packageDefinition.
  - loadPackageDefinition will load the messages and *Service* definition.
  - Note: package is a reserved word so we abbreviate with pkg.
- To get the descriptor we select the namespace of our package
  - In this case com.example.helloworld.

Creating the Client

```
const client = new descriptor.ServiceName(serverUri, credentials);
```

- We now have everything we need to create our *Client*.
- To create the *Client*, we create a new instance of the *Service*.
- This will require:
  - A URI where the *Service* is hosted.
  - Credentials.

Creating a Request

- .proto

```
message SayHello {
  string user_id = 1;
  string name = 2;
}
```

- .js

```
const request = new descriptor.SayHello.constructor({
  userId: '123', name: 'Wade'
});
```

- Now that we have a client, we need to prepare a message to send.
- Our descriptor also provides us with the messages from our .proto file.
  - The SayHello message takes two string fields, a user_id and a name.
  - Note: We can use JavaScript-friendly camelCase to refer to the

Using the Client

```
function handleResponse(error, response) { ... }

client.sayHello(request, handleResponse);
```

- To call our sayHello rpc method, we provide a request and callback.
- The callback will be executed when the service responds.
- It will receive either an error or a response which can be handled as appropriate.
- The request and response types will correspond to the protobuf messages specified by the service in the .proto file.

Dynamic vs Precompilation

- The @grpc/grpc-js library allows us to dynamically load the .proto file and make it accessible to JavaScript
- There is no precompilation step to generate JavaScript code.
    - This may be different from other *gRPC* libraries.
- Many other languages have similar *gRPC* client libraries.
    - They often require a compilation/code generation step to make use of the .proto definitions.