Un-RESTful Services

•••

Building gRPC Services with .NET Core

Overview

- We assume microservices are RESTful
  - Should we?
- Realities of REST
- Documenting RESTful endpoints
- Introduction to gRPC
- gRPC Demo
- Q&A

Why do we think we need REST?

- Client language agnostic
- Easy to test
- Easy to consume
- Works over HTTP
  - “Router friendly”
- HTTP already comes with “verbs”
  - Resources are “nouns”

The reality of REST

- Not the only client language agnostic protocol out there
- Testing isn’t always as easy as people think
- Consumption isn’t as easy as it looks
- Forcing our mental model into verb-over-resource REST design
    - REST model bleeds into application internal model
    - We start “thinking RESTfully” - pollutes design process
    - Don’t separate the API from the real code
    - Impedance mismatch between what we want to do and how we express it
- JSON is loosely typed
    - Need add-ons like JSON schema to strongly type message payloads
- Approaching cargo cult levels
    - Not applying critical thinking to our problems anymore

But we can document RESTful endpoints...

```
paths:
  /users/{userId}:
    get:
      summary: Returns a user by ID.
      parameters:
        - in: path
          name: userId
          required: true
          type: integer
          minimum: 1
          description: Parameter description in Markdown.
      responses:
        200:
          description: OK
```

Code generation and Swagger

- Swagger-to-Code
- Code-to-Swagger
- Swagger UI
- Swagger-to-HTML
  - Because swagger is **not** human readable
- Complex dance, sharing twice-removed generated artifacts to ensure client and server API compliance

Introducing gRPC

- RPC API described using popular protocol buffer IDL
  - Concise, human-readable format
  - Single source of truth (and code gen)
  - Opinion: Easier to separate API facade/contract from implementation details
  - Strongly typed
- Low latency
  - Compact, binary payloads
- High performance
  - Single or bi-directional *streaming*.
- Works over HTTP/2
  - Router friendly*
  - Doesn’t interfere with regular HTTP connections

gRPC (via Protocol Buffers) is Strongly Typed

- Messages and Primitives
  - Messages can contain other messages as fields - nested structure
- Arrays of Messages and Primitives
- Primitives
  - Double
  - Float
  - [u]int32, [u]int64
  - Fixed32, fixed64
  - Bool
  - String
  - Bytes

Sample gRPC Service Definition

```
syntax = "proto3";

package PartialFoods.Services;

import "partialfoods.proto";

service OrderManagement {
    rpc GetOrder(GetOrderRequest) returns (GetOrderResponse);
    rpc OrderExists(GetOrderRequest) returns (OrderExistsResponse);
}
```

Sample gRPC Service Definition - Cont’d

```
message GetOrderRequest {
      string OrderID = 1;
}

message GetOrderResponse {
      string OrderID = 1;
      uint64 CreatedOn = 2; // UTC milliseconds of time terminal created transaction
      string UserID = 3; // User ID of the order owner
      uint32 TaxRate = 4; // Percentage rate of tax, whole numbers because reasons
      ShippingInfo ShippingInfo = 5; // Information on where order is to be shipped
      repeated LineItem LineItems = 6; // Individual line items on a transaction
      OrderStatus Status = 7;
}
```

More strong types - enums

```
enum OrderStatus {
      UNKNOWN = 0;
      OPEN = 1;
      CANCELED = 2;
}
```

Defining a Streaming Service

```
service TestAPI {
      rpc agent_heartbeat(Agent) returns (Ack);
      rpc update_library_entries(stream LibraryEntryUpdate) returns (LibraryUpdateResponse);
      rpc stage_file(stream LargeFileComponent) returns (LargeFileAck);
      rpc download_file(StagedFile) returns (stream LargeFileComponent);
}
```

Implementing a gRPC Service

```
<PackageReference Include="Grpc" Version="1.11.0-pre2" />
<PackageReference Include="Grpc.Tools" Version="1.11.0-pre2" />
<PackageReference Include="Grpc.Reflection" Version="1.11.0-pre2"/>
```

Generating Code from IDL

- Add **Grpc.Tools** to project dependencies
- Invoke **protoc** with the **protoc-gen-grpc** plugin
- This produces C# code for:
  - Message definitions
  - Enums
  - Interfaces and abstract base classes for services
  - Strongly-typed gRPC client class

Generating Code from IDL

```
PROJDIR=`pwd`
cd ~/.nuget/packages/grpc.tools/1.6.1/tools/linux_x64

protoc -I $PROJDIR/../../proto --csharp_out $PROJDIR/RPC --grpc_out
$PROJDIR/RPC $PROJDIR/../../proto/inventory.proto
--plugin=protoc-gen-grpc=grpc_csharp_plugin

cd -
```

Implementing a gRPC Service

```
public class OrderManagementImpl : OrderManagement.OrderManagementBase {

...

   public override Task<SomeResponse> SomeMethod() { ... }

}
```

Implementing a gRPC Service

```
public override Task<OrderExistsResponse> OrderExists(
  GetOrderRequest request, grpc::ServerCallContext context)
{
    bool exists = repository.OrderExists(request.OrderID);
    var resp = new OrderExistsResponse
    {
       Exists = exists,
       OrderID = request.OrderID
    };
    return Task.FromResult(resp);
}
```

Implementing a gRPC Service - Starting the Server

```
var port = int.Parse(Configuration["service:port"]);
var refImpl = new ReflectionServiceImpl(
    ServerReflection.Descriptor, OrderManagement.Descriptor);
Server server = new Server
{
    Services = { OrderManagement.BindService(new  OrderManagementImpl(repo)),
                 ServerReflection.BindService(refImpl) },
    Ports = { new ServerPort("localhost", port, ServerCredentials.Insecure)
}
};
server.Start();
mre.WaitOne();
```

But what about ... ?

- Curl or Postman?
  - Use **grpcurl**
- Human-friendly output?
  - Use **jq**
- Clients that cannot consume gRPC?
  - Use the generic gRPC gateway as a reverse proxy
- Swagger and docs?
  - Plenty of tooling to generate docs from protobuf IDLs

gRPC Services DEMO

Q&A