GRPC RUST

Cargo.toml

```
[package]
name = "test_grpc"
version = "0.1.0"
authors = [""]
edition = "2018"

# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html

[dependencies]
prost = "0.6.1"
tonic = {version="0.2.0",features = ["tls"]}
tokio = {version="0.2.18",features = ["stream", "macros"]}
futures = "0.3"

[build-dependencies]
tonic-build = "0.2.0"

[[bin]]
name = "grpcserver"
path = "src/server.rs"

[[bin]]
name = "grpcclient"
path = "src/client.rs"
```

proto/say.proto

```
 // version of protocol buffer used
syntax = "proto3";

// package name for the buffer will be used later
package hello;

// service which can be executed
service Say {
// function which can be called
rpc Send (SayRequest) returns (SayResponse);
}

// argument
message SayRequest {
// data type and position of data
```

```
string name = 1;
}

// return value
message SayResponse {
// data type and position of data
string message = 1;
}
```

build.rs

```
fn main() -> Result<(), Box<dyn std::error::Error>> {
    let config = tonic_build::configure().out_dir("src");
    config.compile(&["proto/say.proto"],&["proto/"])?;
    Ok(())
}
```

src/server.rs

```
use hello::say_server::{Say, SayServer};
use hello::{SayRequest, SayResponse};
use tonic::{transport::Server, Request, Response, Status};
mod hello;

// defining a struct for our service
#[derive(Default)]
pub struct MySay {}

// implementing rpc for service defined in .proto
#[tonic::async_trait]
impl Say for MySay {
    // our rpc impelemented as function
    async fn send(&self, request: Request<SayRequest>) ->
Result<Response<SayResponse>, Status> {
        // returning a response as SayResponse message as defined in .proto
        Ok(Response::new(SayResponse {
            // reading data from request which is awrapper around our
SayRequest message defined in .proto
            message: format!("hello {}", request.get_ref().name),
        }))
    }
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // defining address for our service
    let addr = "[::1]:50051".parse().unwrap();
```

```
    // creating a service
    let say = MySay::default();
    println!("Server listening on {}", addr);
    // adding our service to our server.
    Server::builder()
        .add_service(SayServer::new(say))
        .serve(addr)
        .await?;
    Ok(())
}
```

src/client.rs

```
use hello::say_client::SayClient;
use hello::SayRequest;

mod hello;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // creating a channel ie connection to server
    let channel =
tonic::transport::Channel::from_static("http://[::1]:50051")
        .connect()
        .await?;
    // creating gRPC client from channel
    let mut client = SayClient::new(channel);
    // creating a new Request
    let request = tonic::Request::new(SayRequest {
        name: String::from("hello world"),
    });
    // sending request and waiting for response
    let response = client.send(request).await?.into_inner();
    println!("RESPONSE={:?}", response);
    Ok(())
}
```

src/hello.rs

```
/// argument
#[derive(Clone, PartialEq, ::prost::Message)]
pub struct SayRequest {
    /// data type and position of data
    #[prost(string, tag = "1")]
    pub name: std::string::String,
}
/// return value
```

```
#[derive(Clone, PartialEq, ::prost::Message)]
pub struct SayResponse {
    /// data type and position of data
    #[prost(string, tag = "1")]
    pub message: std::string::String,
}
#[doc = r" Generated client implementations."]
pub mod say_client {
    #![allow(unused_variables, dead_code, missing_docs)]
    use tonic::codegen::*;
    #[doc = " service which can be executed"]
    pub struct SayClient<T> {
        inner: tonic::client::Grpc<T>,
    }
    impl SayClient<tonic::transport::Channel> {
        #[doc = r" Attempt to create a new client by connecting to a given
endpoint."]
        pub async fn connect<D>(dst: D) -> Result<Self,
tonic::transport::Error>
        where
            D: std::convert::TryInto<tonic::transport::Endpoint>,
            D::Error: Into<StdError>,
        {
            let conn =
tonic::transport::Endpoint::new(dst)?.connect().await?;
            Ok(Self::new(conn))
        }
    }
    impl<T> SayClient<T>
    where
        T: tonic::client::GrpcService<tonic::body::BoxBody>,
        T::ResponseBody: Body + HttpBody + Send + 'static,
        T::Error: Into<StdError>,
        <T::ResponseBody as HttpBody>::Error: Into<StdError> + Send,
    {
        pub fn new(inner: T) -> Self {
            let inner = tonic::client::Grpc::new(inner);
            Self { inner }
        }
        pub fn with_interceptor(inner: T, interceptor: impl
Into<tonic::Interceptor>) -> Self {
            let inner = tonic::client::Grpc::with_interceptor(inner,
interceptor);
            Self { inner }
        }
        #[doc = " function which can be called"]
        pub async fn send(
            &mut self,
            request: impl tonic::IntoRequest<super::SayRequest>,
        ) -> Result<tonic::Response<super::SayResponse>, tonic::Status> {
            self.inner.ready().await.map_err(|e| {
                tonic::Status::new(
                    tonic::Code::Unknown,
                    format!("Service was not ready: {}", e.into()),
```

```
                )
            })?;
            let codec = tonic::codec::ProstCodec::default();
            let path =
http::uri::PathAndQuery::from_static("/hello.Say/Send");
            self.inner.unary(request.into_request(), path, codec).await
        }
    }
    impl<T: Clone> Clone for SayClient<T> {
        fn clone(&self) -> Self {
            Self {
                inner: self.inner.clone(),
            }
        }
    }
    impl<T> std::fmt::Debug for SayClient<T> {
        fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result
{
            write!(f, "SayClient {{ ... }}")
        }
    }
}
#[doc = r" Generated server implementations."]
pub mod say_server {
    #![allow(unused_variables, dead_code, missing_docs)]
    use tonic::codegen::*;
    #[doc = "Generated trait containing gRPC methods that should be
implemented for use with SayServer."]
    #[async_trait]
    pub trait Say: Send + Sync + 'static {
        #[doc = " function which can be called"]
        async fn send(
            &self,
            request: tonic::Request<super::SayRequest>,
        ) -> Result<tonic::Response<super::SayResponse>, tonic::Status>;
    }
    #[doc = " service which can be executed"]
    #[derive(Debug)]
    #[doc(hidden)]
    pub struct SayServer<T: Say> {
        inner: _Inner<T>,
    }
    struct _Inner<T>(Arc<T>, Option<tonic::Interceptor>);
    impl<T: Say> SayServer<T> {
        pub fn new(inner: T) -> Self {
            let inner = Arc::new(inner);
            let inner = _Inner(inner, None);
            Self { inner }
        }
        pub fn with_interceptor(inner: T, interceptor: impl
Into<tonic::Interceptor>) -> Self {
            let inner = Arc::new(inner);
            let inner = _Inner(inner, Some(interceptor.into()));
            Self { inner }
```

```
        }
    }
    impl<T, B> Service<http::Request<B>> for SayServer<T>
    where
        T: Say,
        B: HttpBody + Send + Sync + 'static,
        B::Error: Into<StdError> + Send + 'static,
    {
        type Response = http::Response<tonic::body::BoxBody>;
        type Error = Never;
        type Future = BoxFuture<Self::Response, Self::Error>;
        fn poll_ready(&mut self, _cx: &mut Context<'_>) -> Poll<Result<(),
Self::Error>> {
            Poll::Ready(Ok(()))
        }
        fn call(&mut self, req: http::Request<B>) -> Self::Future {
            let inner = self.inner.clone();
            match req.uri().path() {
                "/hello.Say/Send" => {
                    #[allow(non_camel_case_types)]
                    struct SendSvc<T: Say>(pub Arc<T>);
                    impl<T: Say>
tonic::server::UnaryService<super::SayRequest> for SendSvc<T> {
                        type Response = super::SayResponse;
                        type Future =
BoxFuture<tonic::Response<Self::Response>, tonic::Status>;
                        fn call(
                            &mut self,
                            request: tonic::Request<super::SayRequest>,
                        ) -> Self::Future {
                            let inner = self.0.clone();
                            let fut = async move {
inner.send(request).await };
                            Box::pin(fut)
                        }
                    }
                    let inner = self.inner.clone();
                    let fut = async move {
                        let interceptor = inner.1.clone();
                        let inner = inner.0;
                        let method = SendSvc(inner);
                        let codec = tonic::codec::ProstCodec::default();
                        let mut grpc = if let Some(interceptor) =
interceptor {
                            tonic::server::Grpc::with_interceptor(codec,
interceptor)
                        } else {
                            tonic::server::Grpc::new(codec)
                        };
                        let res = grpc.unary(method, req).await;
                        Ok(res)
                    };
                    Box::pin(fut)
                }
```

```
                _ => Box::pin(async move {
                    Ok(http::Response::builder()
                        .status(200)
                        .header("grpc-status", "12")
                        .body(tonic::body::BoxBody::empty())
                        .unwrap())
                }),
            }
        }
    }
    impl<T: Say> Clone for SayServer<T> {
        fn clone(&self) -> Self {
            let inner = self.inner.clone();
            Self { inner }
        }
    }
    impl<T: Say> Clone for _Inner<T> {
        fn clone(&self) -> Self {
            Self(self.0.clone(), self.1.clone())
        }
    }
    impl<T: std::fmt::Debug> std::fmt::Debug for _Inner<T> {
        fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result
{
            write!(f, "{:?}", self.0)
        }
    }
    impl<T: Say> tonic::transport::NamedService for SayServer<T> {
        const NAME: &'static str = "hello.Say";
    }
}
```