Sending real-time updates from the state machine via SignalR

Script start

Let's complete our purchase experience by updating our state machine so it can send real time notifications back to the frontend when the purchase transaction is complete.

In Trading repo

1. Consume the updated NuGet package:
   <PackageReference Include="Play.Common" Version="1.0.5" />

2. Add SignalR directory

3. Add MessageHub.cs:

```
namespace Play.Trading.Service.SignalR
{
   [Authorize]
   public class MessageHub : Hub
   {
      public async Task SendStatusAsync(PurchaseState status)
      {
           If (Clients != null){
              await Clients.User(Context.UserIdentifier)
                    .SendAsync("ReceivePurchaseStatus", status);
           }
      }
   }
}
```

4. Add UserIdProvider.cs:

```
namespace Play.Trading.Service.SignalR
{
   public class UserIdProvider : IUserIdProvider
   {
      public virtual string GetUserId(HubConnectionContext connection)
      {
         return connection.User.FindFirstValue(JwtRegisteredClaimNames.Sub);
      }
   }
}
```

5. Update PurchaseStateMachine:

```
public class PurchaseStateMachine : MassTransitStateMachine<PurchaseState>
{
  private readonly MessageHub hub;

  …

  public PurchaseStateMachine(MessageHub hub)
  {
    …
    ConfigureFaulted();
    this.hub = hub;
  }

  …

  private void ConfigureInitialState()
  {
    Initially(
      When(PurchaseRequested)
        …
        .Catch<Exception>(ex => ex
            .Then(context =>
            {
              …
            })
            .TransitionTo(Faulted)
            .ThenAsync(async context => await hub.SendStatusAsync(context.Instance)))
    );
  }

  private void ConfigureAccepted()
  {
    During(Accepted,
      …
      When(GrantItemsFaulted)
        .Then(context =>
        {
          …
        })
        .TransitionTo(Faulted)
        .ThenAsync(async context => await hub.SendStatusAsync(context.Instance))
```

```
        );
    }

    private void ConfigureItemsGranted()
    {
        During(ItemsGranted,
            …
            When(GilDebited)
                …
                .TransitionTo(Completed)
                .ThenAsync(async context => await hub.SendStatusAsync(context.Instance)),
            When(DebitGilFaulted)
                …
                .TransitionTo(Faulted)
                .ThenAsync(async context => await hub.SendStatusAsync(context.Instance))
            );
    }
    …
}
```

6. Update Startup:

```
public class Startup
{
    …

    public void ConfigureServices(IServiceCollection services)
    {
        …
        services.AddSwaggerGen(c =>
        {
            …
        });

        services.AddSingleton<IUserIdProvider, UserIdProvider>()
                .AddSingleton<MessageHub>()
                .AddSignalR();
}

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            …
```

```
        app.UseCors(builder =>
        {
            builder.WithOrigins(Configuration[AllowedOriginSetting])
                .AllowAnyHeader()
                .AllowAnyMethod()
                .AllowCredentials();
          //Credentials must be allowed in order for cookie-based sticky sessions to work correctly.
        });
    }

    …

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
        endpoints.MapHub<MessageHub>("/messagehub");
    });
  }
  …
}
```

7. Start Trading service (USE F5)

In Frontend portal

8. Show the SignalR pieces of PurchaseForm.js

9. Try a Purchase

10. Verify quantities and gil

11. Try a purchase for more than user can purchase

12. Verify quantities and gil

And, beyond this, I invite you to try out a few more error scenarios across client and server, like crashes or connectivity issues, to see how well the different microservices and the frontend can handle the unexpected issues.

So, at this point all the Play Economy system backend features required to support the frontend, and our fictional game client, are complete and working end to end, all thanks to the collaboration of the four .NET based microservices that are powering the whole experience.

If you followed along up to this point, congratulations! I really hope you have enjoyed the learning journey so far.

In the next module, we will learn how to get our microservices containerized to get them ready to be deployed outside of the developer box.