Welcome to the Event Modeling workshop exercises. The goal of the exercises is to help you gain familiarity with both CQRS and event modeling We have 4 languages available for you to learn CQRS and apply event modeling with. We're also assuming that you have some basic familiarity with CQRS and event modeling. If you don't, no worries. Listen to the talks in the workshop and until you feel comfortable enough to begin some coding. Then, choose the language you're most familiar with or have always wanted to learn and get started.

Exercise Overview

For our exercise domain, we have chosen a simplified hotel reservation system. Reserving a room is a familiar process to most of us yet contains enough complexity to cover some of the most interesting aspects of CQRS and event modeling. Here is the first scenario we are going to cover.

At a hotel front desk, rooms are reserved from a pool of available rooms. In our simplified reservation system we can reserve a room as long as we have access to the current number of available rooms. We'll highlight other required features for a working reservation system later but for now, we'll restrict the requirements for reserving a room to the make a reservation request and the inventory of available rooms.

In order to get a successfully confirmed reservation, we need to send in the MakeReservation command, successfully generate the ReservationMade event and then save Reservation data in a read model for use in reports or a UI.

When you get to the code, you might notice that we've omitted some of the information you would normally expect to see in a reservation, namely dates. You can add that information if you like, but we've kept things simple as we are more interested in teaching you about CQRS then hotel management.

Our CQRS-based hotel reservation system has been implemented in C#, golang, kotlin and python. The Csharp and Kotlin repositories are listed below. As with all CQRS based systems, everything starts with a command. However, we are going to introduce some of the patterns we have used successfully for a number of Adaptech implementations. Also, most of the CQRS stacks in each language are optimized to work on Cloud services and take advantage of low-cost compute platforms like Azure functions or AWS lambdas. Details are in the readme section of each repository.

Location of repositories here and choose the repo for the language you want to work with:
C# https://github.com/timhamelin/ServerlessCQRSCSharp.git
Kotlin https://github.com/timhamelin/ServerlessCQRSKotlin.git

The repositories also contain folders for each exercise that requires coding and its solution so you can see how we suggest solving the exercises. Use the discussion forum to ask questions and give comments.

Exercise 1: Get familiar with your surroundings and choose an IDE

Your first exercise is to check out the type of code you're most interested in and load it into one of the IDE's. identify the command and events as well as the bus, command and event handlers and aggregate.

Clone the code you're most interested in working with down to a local directory. All repositories should be public for the course of the workshop, so pulling and cloning the code should be easy. Ensure that you have git installed on your workstation. To clone the C# repository, you would create a new working directory, change to it an execute the following command:
`git clone` https://github.com/timhamelin/ServerlessCQRSCSharp.git If you prefer Kotlin, you would clone the Kotlin repo using the following command:
`git clone` https://github.com/timhamelin/ServerlessCQRSKotlin.git

The Readme files for each repository will provide you with information on specific build instructions, IDE's and any specific setup information unique to the code base there are even Readme files for using a VM that is already set up for you with everything pre-installed.

IMPORTANT all of the repo's require that you have a locally installed version of EventStore DB from https://eventstore.com/ .   We'll give specific instructions on how to setup Event Store in the next exercise. But if you have it installed already, now would be a great time to ensure that it's running properly.

That's enough information for now, it's time to look at some code. The following diagram is a brief overview of the process flow you'll see implemented in each language.

The process flow you'll see in each language is
1. A command is sent to the API which is the entry point for our CQRS system
2. The command is published on the Bus. Rather than a separated command/event bus we tend to implement a single bus for both commands and events
3. The Bus determines the appropriate command handler for the command and calls it.
4. An Aggregate is created that holds the current state of the reservation. State is set by hydrating our Aggregate from an event store. The state of the Aggregate determines whether our Reservation command can be completed
5. If the reservation is valid, a ReservationMade event is created that indicates that a Reservation has been made.
6. The event is placed on our centralized bus via a **Send** function to a "denormalizer". The denormalizer saves the event as a read model. A read model is a denormalized table containing all information for future reporting on reservations.

Go through the code repository you've checked out and review our implementations of the bus, commands, events and aggregates. We'll provide more information to increase your understanding of the code later, but for now, just get familiar with the code and the IDE you've chosen to use.

Exercise 2: Install EventStoreDB and run the application

As mentioned in the previous exercise, all of the repo's require that you have a locally installed version of EventStoreDB running on your current workstation. We recommend EventStoreDb at Adaptech as it has all the necessary features required for a full Event Sourcing implementation.

Docker https://docs.docker.com/get-docker/ provides you with the easiest method of running the event store db. Each repo contains a file called `docker-compose.yml` in the repo's root directory. If you have docker installed, simply open up an administrative command line in your repo's root directory and execute the command :
`docker-compose up`

Instructions on the installation of EventStore can be found at the Event Store website at https://eventstore.com/ Instructions for different operating systems are available at https://eventstore.com/docs/getting-started/index.html

Once you have the event store up and running, it's time to get your code base running. All code bases are designed to provide one or more http access points to the API used to issue CQRS commands. Examine the Readme file for the instructions on starting your API. The IDE you use should also provide you with the capability for debugging and running the project of your choice.

Exercise 3: Send in a Make Reservation command

Now that your EventStore is installed and you are familiar with how to start your project, let's send in a command. Commands can be sent to a running api using curl or postman. In order for a reserve room command to be processed, it must contain some information. First, the reservation command itself, which consists of a Hotel Identifier, a Reservation identifier and a Room Type to be reserved (suite, queen, etc). When you start your project in an IDE, the IDE will usually print out the URL where commands are processed. For example, if you are running your API as an Azure function, the URL to make a reservation will be http://localhost:8081/makeReservation A POST command can be sent to this URL. A complete post command for curl will look like this:

curl --location --request POST 'http://localhost:7071/reservation/make' \
--header 'Content-Type: application/json' \
--data-raw '{
"id": "7feda51d-785f-4a3d-9d0c-09bc41764706",
"hotelId": "83d5c0d5-08e3-447b-935c-c2c9253b7674",
"roomtype": "Presidential"
}'

If you are unfamiliar with curl, then the same address can be loaded into postman or a similar app and POSTed to the url. If you use our VM postman is already installed with the api calls Send in a Make Reservation command using the above format. If you're successful, your command will return a 200 status code.

Exercise 4: Make Reservation again and review the event store

The command from Exercise 3 has reserved a room of type Presidential. There is only one presidential suite available in our hotel, so a second reservation should result in failure. Let's verify that by sending in the command from Exercise 3 again. What happens? If everything is set up correctly, trying to reserve the Presidential Suite a second time will result in a an error message. Verify this result and then we'll go through the pieces that are keeping the presidential room reserved.

When the room was reserved, two key pieces of information were saved. First the event ReservationMade was saved in the event store database. Second, a Reservation read model was created and stored in a SQLite database. The read model is used to determine the number of reserved rooms of any type.

You can easily determine if the event was saved successfully by looking at the contents of the Event Store DB. Log in to the event store db at http://localhost:2113 . Username will be admin and the password is changeit. Select the stream browser button in the top of the screen and you should be able to see your Reservation event

Note that the guid beside the word reservation matches the guid used to identify the reservation in our MakeReservation command.

Exercise 5: Canceling a Reservation (Commands and Events)

In this exercise, we're going to start the process of canceling a room reservation. Room reservations could be canceled for any number of reasons which might be of interest to the business domain. However, we're going to keep things simple, when a room reservation is removed, we simply modify the state of reserved rooms in our Reservation read model.

The first step in the development of any state change in a domain is the creation of a command and event. In this exercise, the commands and events used to cancel a reservation will be identical to those used to make a reservations. This isn't uncommon in CQRS and event driven programming. In fact, at Adaptech we encourage 'copy & paste' programming when we work in an Event sourced system. So, rather than reuse an existing command or an event, we rely more on the **type** of command to determine the actions a state change could perform.

Our event model for removing a reservation does re-use the Reservations read model, but new commands and events are required even though they will hold values identical to those used for reserving a room.

From this point forward, all exercises have a folder for starting the exercise as well as a folder

with the completed exercise implemented by the author of the repository.

Open `ex-5-start` and create the new commands and events required for this exercise. To see the finished solution completed by the creator of the repository, look in the folder `ex-5-complete`. When you are complete, you should have a command and an event suitable for removing a reservation for a room. Make sure you use the names from the above event model when naming your new command and event. That's another important point about Event Modeling, the names we use in an event model go directly into our code.

For a hint, look for `//virtual workshop ex-5` to see where to implement the Command and Event

Exercise 6: Command Handlers

Now that you have created a command for canceling a reservation, you can create a new command handler that will start the process of updating our reservation state. A command handler is called by our command/event bus and publishes the command. It has several tasks to complete including ensuring that the event from the published command is persisted onto the event store.

Just like commands and events, the creation of a command handler should be a "copy & paste" exercise. You can use the Make Reservation command handler as the starting point for the Cancel Reservation. You might also be getting the idea that at Adaptech, "copy and paste" programming is not seen as a bad practice. An event modeled system reuses the command to event to read model pattern, allowing developers to reuse their existing code with only minor modifications..

In this exercise, you will create a command handler that will process the CancelReservation command. The command handler will create an aggregate, but it will **NOT** be able to pass the command to the aggregate for processing. That happens in the next exercise. Still, all the other functionality of the command handler can be completed. So, in this exercise you will complete a command handler that:
   1. Creates a Reservation aggregate which will be used for event processing in the next exercise
   2. Publishes any created events onto the event store database

You might think this is a pretty sparse list for something as central to our application as a command handler. But this is common in CQRS and Event Driven programming. We want each piece of code to be repeatable and simple. We also want each Handler to perform only a small amount of orchestration thus making the functionality of a handler idempotent.

You can start the exercise by checking opening the folder `ex-6-start` from your chosen repository and creating your command handler there. Create a command handler that works with the CancelReservation command and does the two tasks listed above. Remember, this command handler will be very similar to the Reserve room command handler.

Look for `//virtual-workshop ex-6` for hints.

When you're done, you can check your work against the code in folder `ex-6-complete`, which is the implementation created by the repositories author.

Exercise 7: Aggregates/Completing the Command Handler

Now that we have events and commands available for canceling a reservation it is time to implement the business logic that will ensure that our commands are correct and to generate the corresponding ReservationCanceled event. In the previous exercise, you created a Command Handler for the ReservationCanceled command. The command handler needs to trigger a state change and receive an event that will be saved in the EventStore DB.

An Aggregate is responsible for performing all business logic associated with the command. After successfully performing the business logic, the aggregate creates an event describing the state change and returns the event to the Command Handler.

After completing changes to the aggregate for this exercise, you can then complete the command handler. In our case, the Aggregate will be responsible for creating the ReservationCanceled event. The command handler from the previous exercise is missing a key piece. There is no returned event to save. After completing this exercise, you will have the event to save. With these pieces of information you can complete the Command Handler in Exercise 6.

In this exercise, you do not need to build a new aggregate. Rather, the existing Reservation aggregate needs to be enhanced with a function to remove a room reservation. If successful, the Reservation aggregate will build and return a ReservationCanceled event.

You can start the exercise by checking out the folder `ex-7-start` from your chosen repository and modifying the aggregate and command handlers for the Remove Reservation command there.

When you're done, you can check your work against `ex-7-complete`, which is the implementation created by the repositories author.

Look for `//virtual-workshop ex-7` for hints.

Exercise 8: Denormalizers

We've implemented commands and events, performed some basic error checking and have ensured that the correct event is generated from our aggregate. Now it's time to ensure that the read models are updated correctly with the data from our event. The event contains the change that occurred in our system. When a ReservationMade event is received in the denormalizer we must decrement the count of rooms reserved by one. Note that this change occurs after the ReservationCanceled event has been saved to the Event Store DB. This order of processing is important. We always want to save an event before updating the read model. If the read model update fails, for any reason, we can replay all past events and rebuild our read models.

As with most of our coding exercises, we use a "copy & paste" approach to building denormalizers. The denormalizer for canceling a reservation will be slightly less complex than the denormalizer used to create a reservation.

Using the Make Reservation Denormalizer as a template, handle a cancel reservation and implement the functionality described above.

You can start the exercise by checking out the folder `ex-8-start` from your chosen repository and implement your denormalizer there.

When you're done, you can check your work against `ex-8-complete`, which is the implementation created by the repositories author.

For hints, look for `//virtual workshop ex-8`

Exercise 9: Putting it all together

At this point we've created all the pieces required to process our new Cancel Reservation command. There is a command to trigger a change in the state of the reservation and an event to record that change. In [Exercise 6: Command Handlers](#) and [Exercise 7: Aggregates/Completing the Command Handler](#) we created a command handler that will take our command, create an aggregate of the correct type to process the command and return our ReservationCanceled event. In [Exercise 8: Denormalizers](#), we modified the denormalizer for the reservation read model to handle the ReservationCanceled event and update our Reservations read model. Now it's time to tie all these pieces together so we can send the command in via the API.

Your first step should be to add a function that adds a endpoint allowing for a POST into the REST API for your code base. This API entry point is responsible for creating the CancelReservation command and placing it on the Bus for your repository.

In order for the Bus to process your command, the command and event handlers must be mapped to the command and event you created in [Exercise 5: Canceling a Reservation (Commands and Events)](#). Each project has a specific method for mapping a command to a command handler and an event to an event handler. You need to determine which method is used in your code base and register your command with the bus. You would bus.RegisterCommandHandler in Reservations.Api.StartUp.cs. Once your command are registered, you can send a POST command to the API and remove a reservation.

You can start the exercise by checking out the folder `ex-9-start` from your chosen repository and implement your API changes. Then register your command handlers. That should allow you to send a CancelRegistration command in via the API.

When you're done, you can check your work against `ex-9-complete`, which is the implementation created by the repositories author.

Look for `//virtual workshop ex-9`
for hints