Chaining Functions Together

Mark Heath
MICROSOFT MVP

@mark_heath www.markheath.net

Chaining Functions

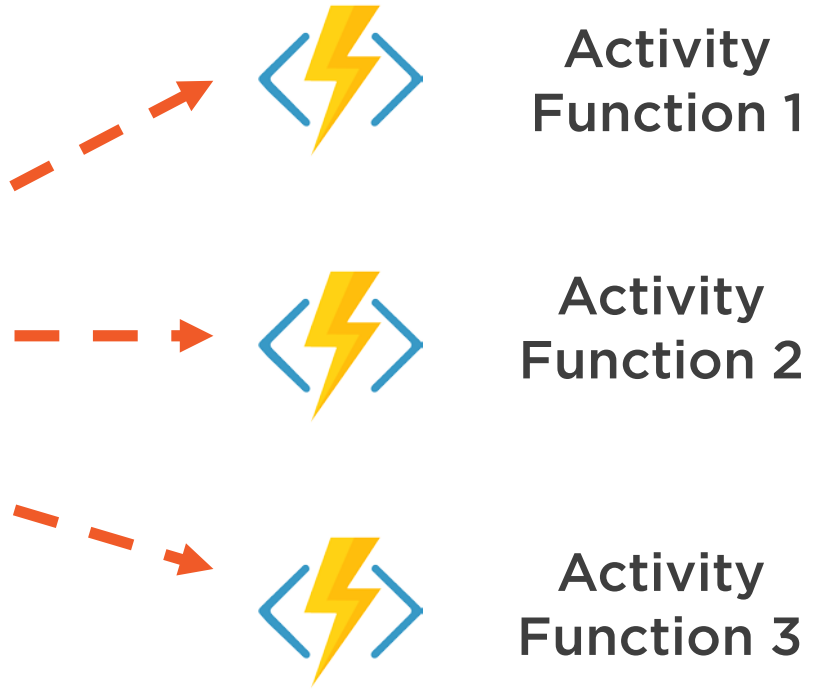
Function 1 Function 2 Function 3

No single place to see the whole workflow

Chaining with Durable Functions

Orchestrator Function

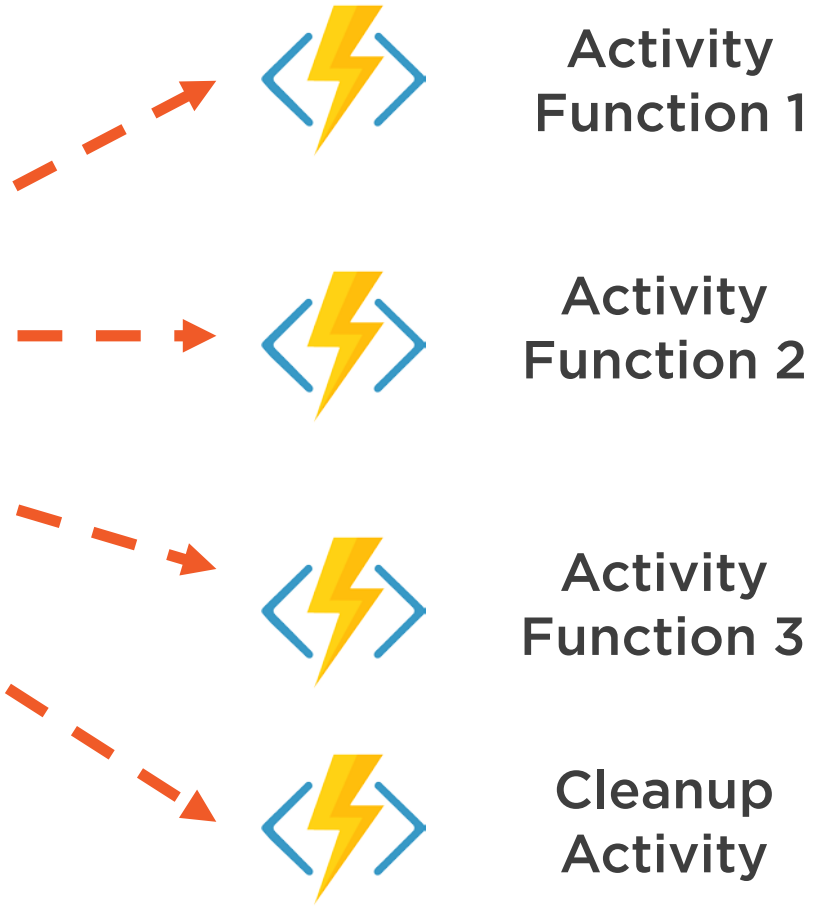
```
// call the first activity
await CallActivityAsync("Activity1");
// call the second activity
await CallActivityAsync("Activity2");
// call the third activity
await CallActivityAsync("Activity3");
```



Chaining with Durable Functions

Orchestrator Function

```
try {
    // call the first activity
    await CallActivityAsync("Activity1");
    // call the second activity
    await CallActivityAsync("Activity2");
    // call the third activity
    await CallActivityAsync("Activity3");
} catch (Exception e) {
    await CallActivityAsync("Cleanup");
}
```



Coming Up

Create a function chain with Durable Functions

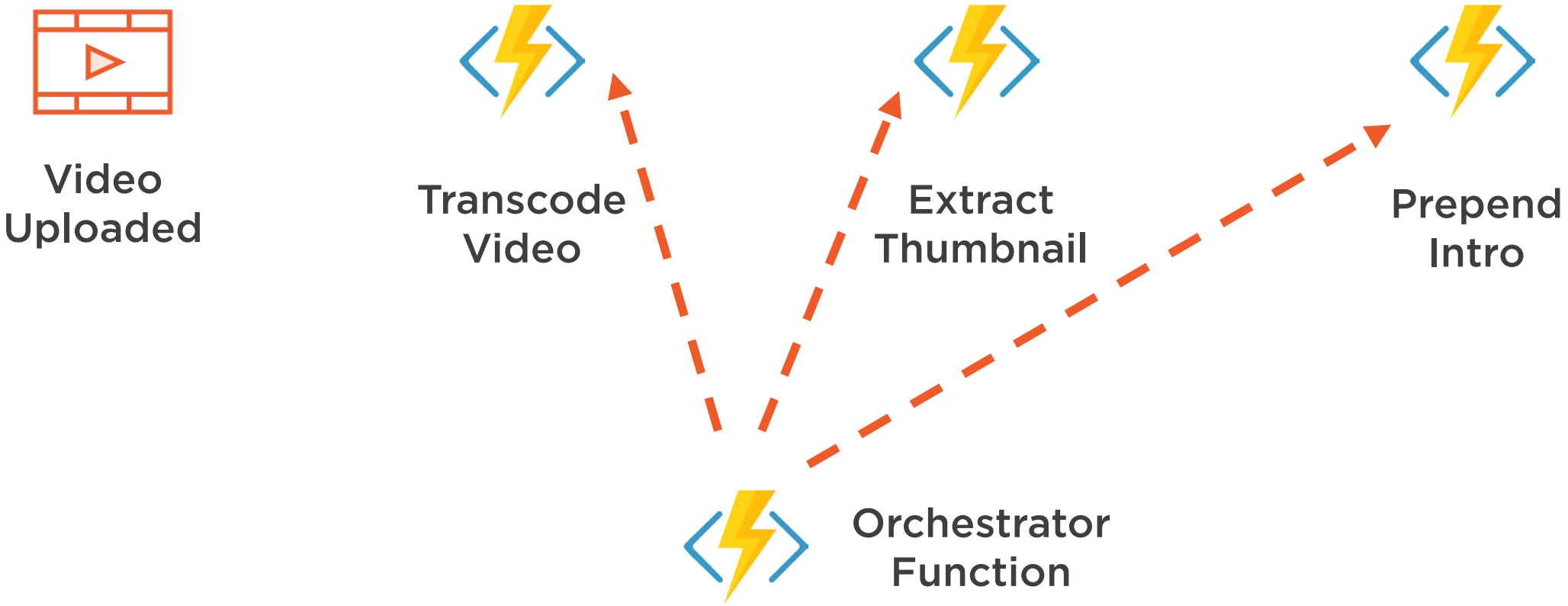
- Create an orchestrator function
- Create activity functions
- Start a new orchestration with the OrchestrationClient binding
- Test locally

Demo Scenario

Video Publishing Workflow

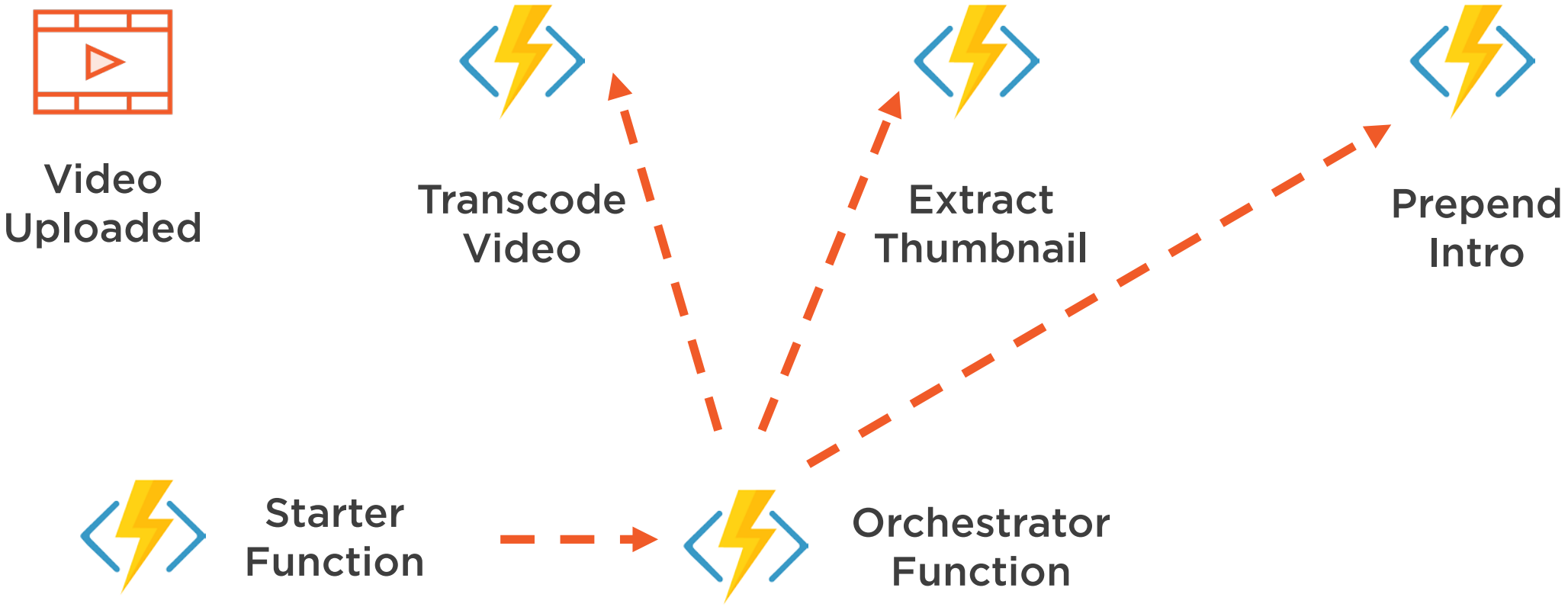
Demo Scenario

Video Publishing Workflow

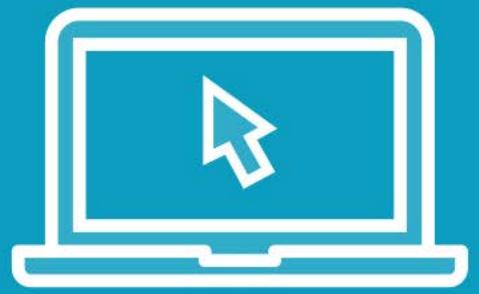


Demo Scenario

Video Publishing Workflow



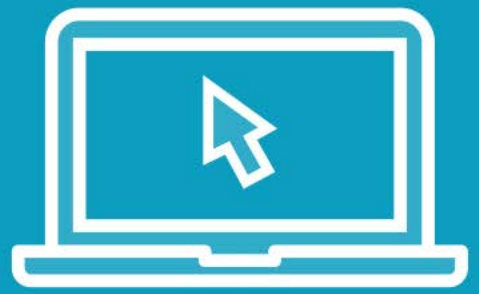
Demo



Create a workflow starter function

- OrchestrationClient binding
- Orchestrator input data
- CreateCheckStatusResponse

Demo



Create an orchestrator function

- Calls each activity in turn
- Receive input data from the starter function

Orchestrator code constraints

The replay behavior creates constraints on the type of code that can be written in an orchestrator function:

- Orchestrator code must be **deterministic**. It will be replayed multiple times and must produce the same result each time. For example, no direct calls to get the current date/time, get random numbers, generate random GUIDs, or call into remote endpoints.

  If orchestrator code needs to get the current date/time, it should use the CurrentUtcDateTime API, which is safe for replay.

  Non-deterministic operations must be done in activity functions. This includes any interaction with other input or output bindings. This ensures that any non-deterministic values will be generated once on the first execution and saved into the execution history. Subsequent executions will then use the saved value automatically.

- Orchestrator code should be **non-blocking**. For example, that means no I/O and no calls to `Thread.Sleep` or equivalent APIs.

  If an orchestrator needs to delay, it can use the CreateTimer API.

- Orchestrator code must **never initiate any async operation** except by using the DurableOrchestrationContext API. For example, no `Task.Run`, `Task.Delay` or `HttpClient.SendAsync`. The Durable Task Framework executes orchestrator code on a single thread and cannot interact with any other threads that could be scheduled by other async APIs.

- **Infinite loops should be avoided** in orchestrator code. Because the Durable Task Framework saves execution history as the orchestration function progresses, an infinite loop could cause an orchestrator instance to run out of memory. For infinite loop scenarios, use APIs such as ContinueAsNew to restart the function execution and discard previous execution history.

While these constraints may seem daunting at first, in practice they aren't hard to follow. The Durable Task Framework attempts to detect violations of the above rules and throws a `NonDeterministicOrchestrationException`. However, this detection behavior is best-effort, and you shouldn't depend on it.

> ⓘ **Note**
>
> All of these rules apply only to functions triggered by the `orchestrationTrigger` binding. Activity functions triggered by the `activityTrigger` binding, and functions that use the `orchestrationClient` binding, have no such limitations.

http://bit.ly/orchestrator-rules

Orchestrator Function Rules

Must be deterministic

- The whole function will be “replayed”

Don’t

- Use current date time
- Generate random numbers or Guids
- Access data stores (e.g. database, configuration)

Do

- Use DurableOrchestrationContext.CurrentUtcDateTime
- Pass configuration into your orchestrator function
- Retrieve data in activity functions

More Orchestrator Function Rules

Must be non-blocking

- No I/O to disk or network
- No Thread.Sleep

Do not initiate async operations

- Except on DurableOrchestrationContext API
- No Task.Run, Task.Delay, HttpClient.SendAsync

Do not create infinite loops

- Event history needs to be replayable
- ContinueAsNew should be used instead

Logging in Orchestrator Functions

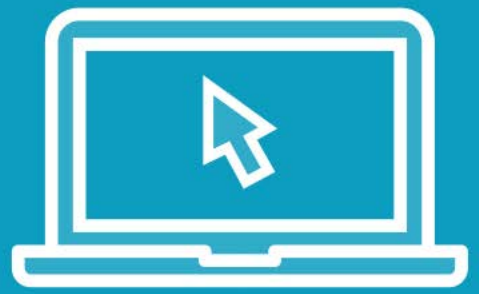
Use the built-in TraceWriter

Log messages get written on every replay

- Avoid with DurableOrchestrationContext.IsReplaying

```
if (!ctx.IsReplaying)
{
    log.Info("Calling TranscodeVideo activity");
}
```

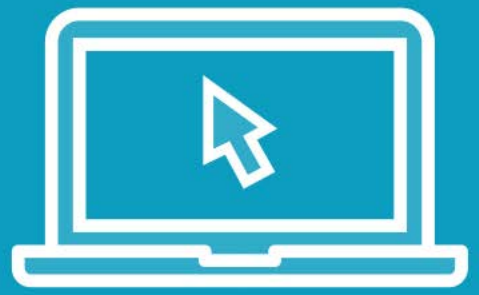
Demo



Creating activity functions

- ActivityTrigger
- Receiving input data
- Returning output data

Demo



Running an orchestration

- Initiate with starter function
- Orchestrator function calls activities in turn
- Orchestrator can provide output data

Starter Function

```
[FunctionName("ProcessVideoStarter")]
public static async Task<HttpResponseMessage> ProcessVideoStarter(
            [HttpTrigger] HttpRequestMessage req,
            [OrchestrationClient] DurableOrchestrationClient starter)
{
    var video = req.GetQueryNameValuePairs()
      .FirstOrDefault(q => string.Compare(q.Key, "video", true) == 0)
      .Value;
    var instanceId = await starter
                .StartNewAsync("O_ProcessVideo", video);
    return starter.CreateCheckStatusResponse(req, instanceId);
}
```

Orchestrator Function

```
[FunctionName("O_ProcessVideo")]
public static async Task<object> ProcessVideo(
    [OrchestrationTrigger] DurableOrchestrationContext ctx)
{
    var videoUri = ctx.GetInput<string>();
    var transcodedUri = await ctx.CallActivityAsync<string>
        ("A_TranscodeVideo", videoUri);
    var thumbnailUri = await ctx.CallActivityAsync<string>
        ("A_ExtractThumbnail", transcodedUri);
    var withIntroUri = await ctx.CallActivityAsync<string>
        ("A_PrependIntro", transcodedUri);
    return new { videoUri, thumbnailUri, withIntroUri };
}
```

Orchestrator Function Rules

- Must be deterministic
- Must be non-blocking
- No async operations
- No infinite loops

Activity Function

```
[FunctionName("A_TranscodeVideo")]
public static async Task<string> TranscodeVideo(
    [ActivityTrigger] string inputUri)
{
    await Task.Delay(5000); // simulate some work
    return $"{inputUri}-transcoded.mp4";
}
```

Up next ...

Tracking workflow progress and handling errors