Designing a Workflow Engine Database Part 5: Actions and Activities

exceptionnotfound.net/designing-a-workflow-engine-database-part-5-actions-and-activities

April 22, 2015

This is Part 5 of an eight-part series describing how to design a database for a Workflow Engine. Click here for Part 1

Having already defined our Process Infrastructure, our Request structure, and our States and Transitions, we can now start to design tables for what a User can actually *do* to a Request in this engine. For that, we will be defining two terms:

- **Actions**: Things a user can perform on a Request.
- **Activities**: Things that result from a Request moving to a particular State or following a particular Transition.

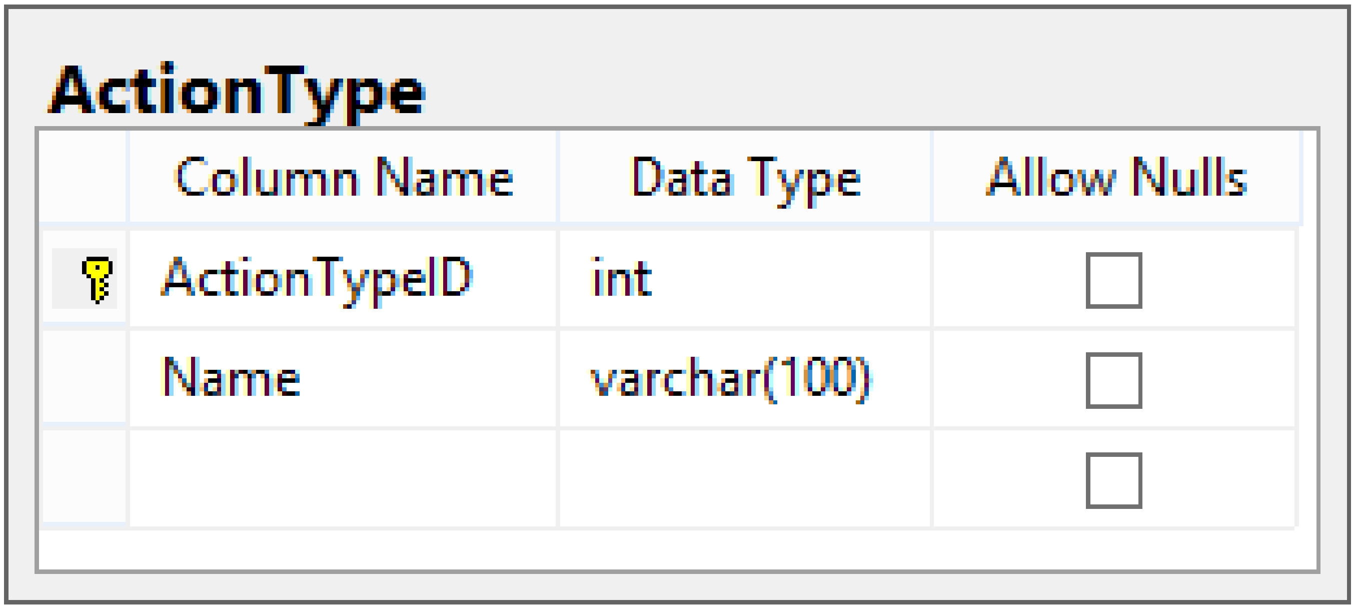
Let's start by creating Actions and Action Types.

Actions

Actions are things a user can perform upon a Request.

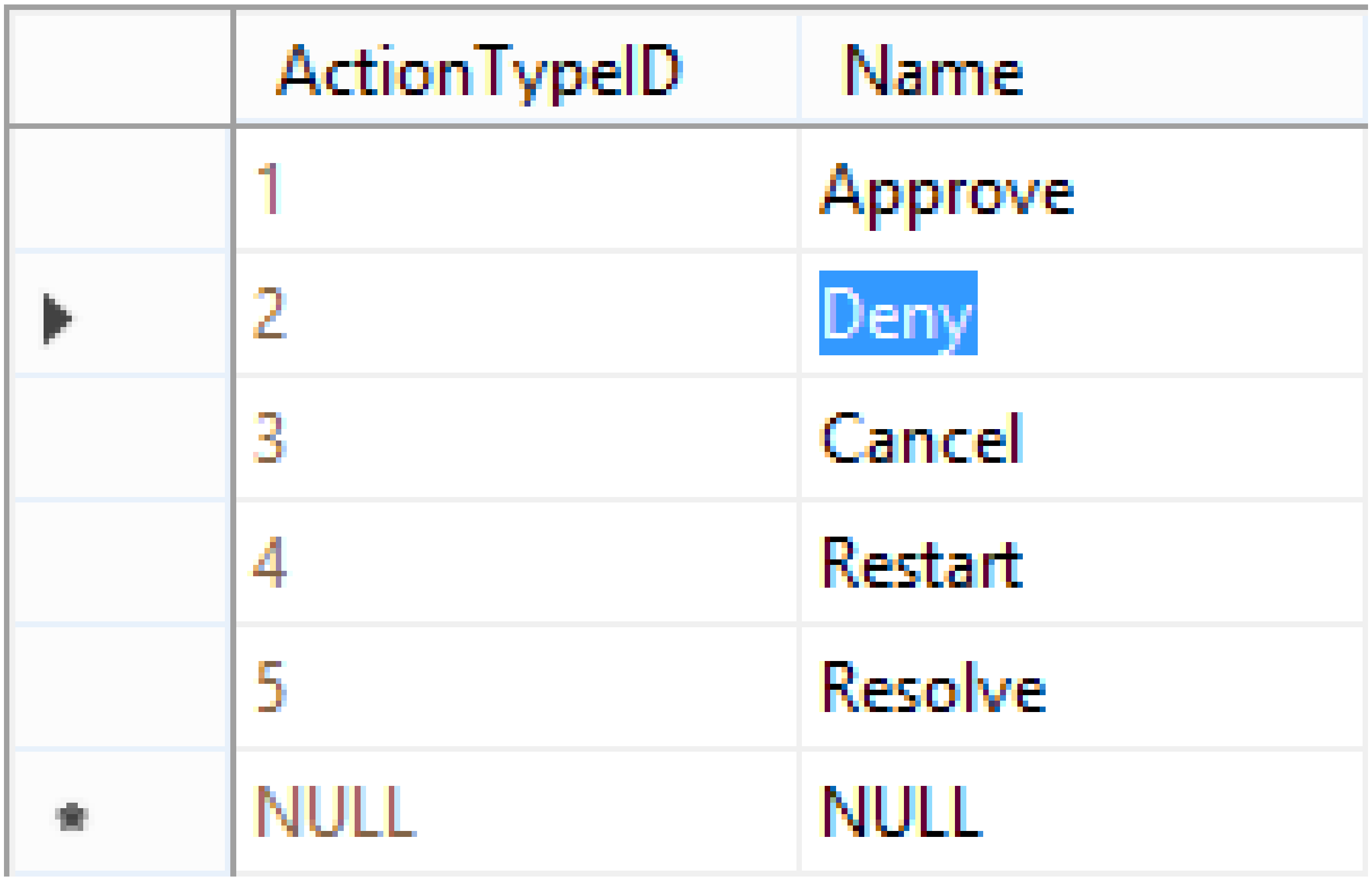
Say we've got a request to build a new grocery store, and that request includes the address of the development where it should be built. The person who is in charge of approving new store construction, John, takes a look at the request and decides that, yeah, it's a good idea to build a store here. He submits an Approval to the Request, which can cause the Request to go to the next state. John has submitted an Action.

Since we don't want to allow an infinite number of kinds of actions that can be performed, we are going to group Actions together via an ActionType table that looks like this:


ActionType

	Column Name	Data Type	Allow Nulls
🔑	ActionTypeID	int	☐
	Name	varchar(100)	☐
			☐

Just like the StateType table, this table is independent of the Process and will be considered static. For our database, we'll use the following Action Types:



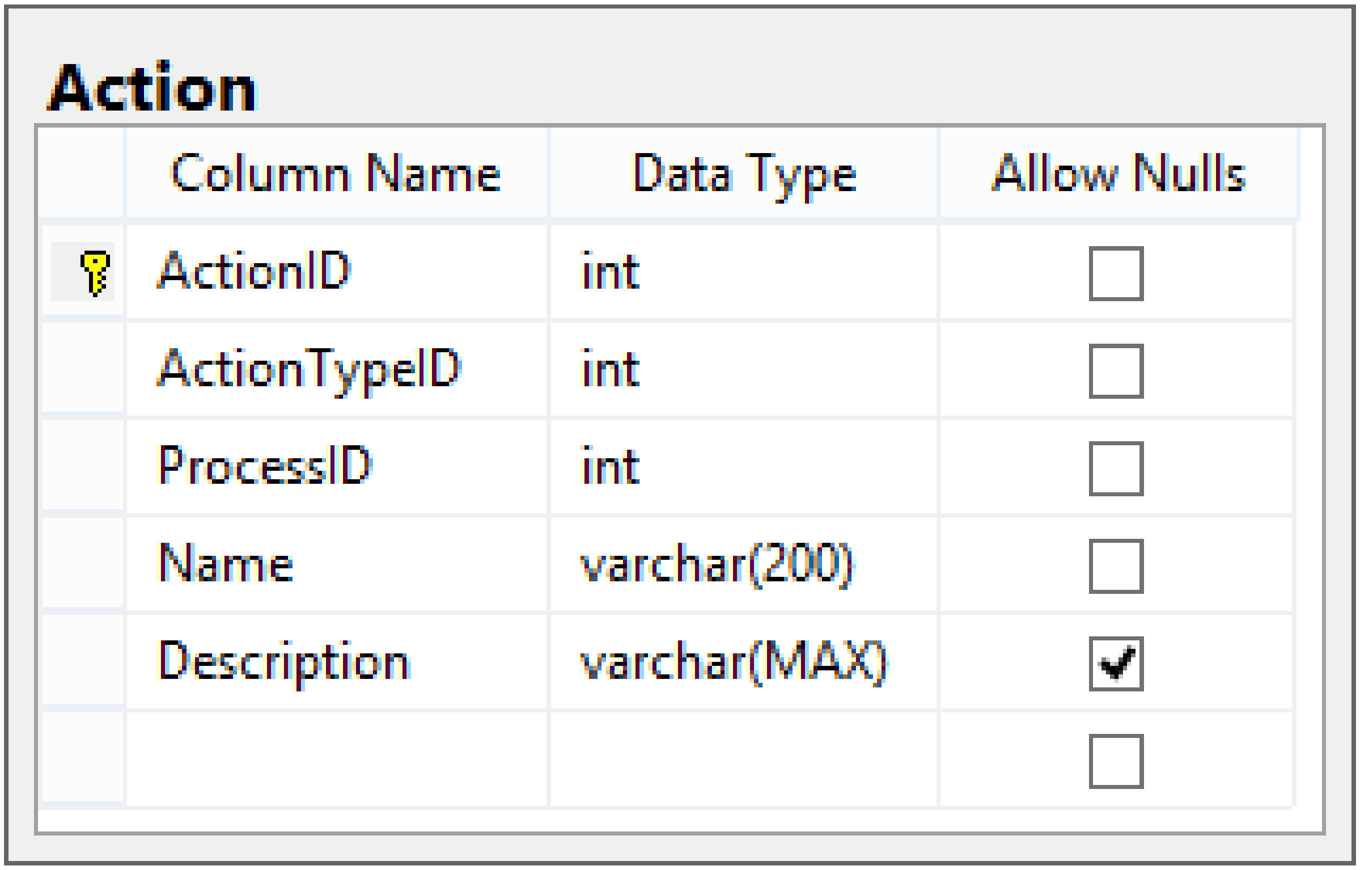
	ActionTypeID	Name
	1	Approve
▶	2	Deny
	3	Cancel
	4	Restart
	5	Resolve
✱	NULL	NULL

Why are we using these action types?

- **Approve**: The actioner is suggesting that the request should move to the next state.
- **Deny**: The actioner is suggesting that the request should move to the previous state.
- **Cancel**: The actioner is suggesting that the request should move to the Cancelled state in the process.
- **Restart**: The actioner suggesting that the request be moved back to the Start state in the process.
- **Resolve**: The actioner is suggesting that the request be moved all the way to the Completed state.

The reason we say the person is "suggesting" that the request be moved is that **we want to allow for a process to require multiple Actions to invoke a Transition**. It is possible that a request will need multiple things to happen before it can continue in the process, and we want to allow for that scenario.

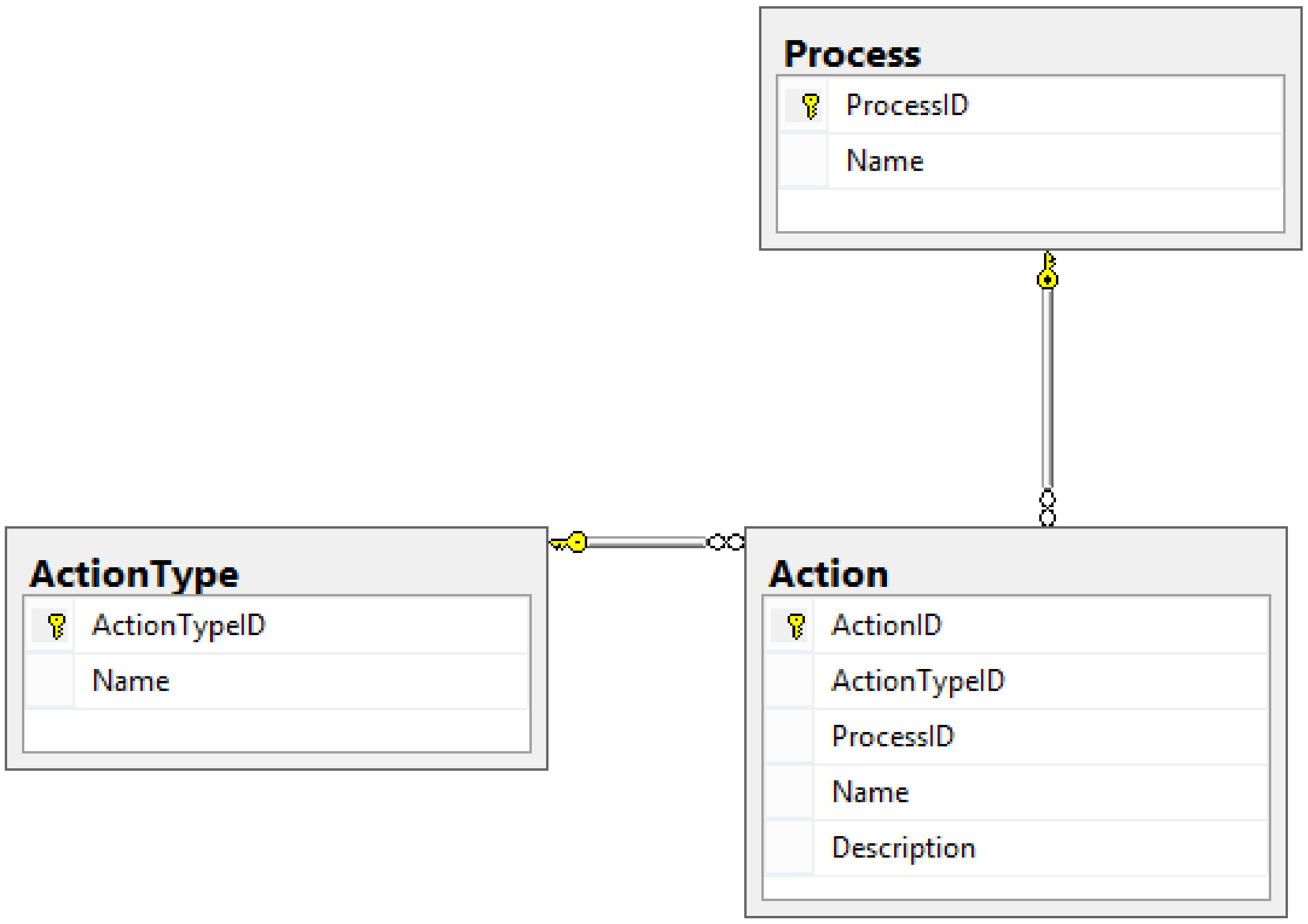
Now we need the table for the Actions themselves. Actions are unique to Processes, and each have an ActionType, so our table will look like this:



Action

	Column Name	Data Type	Allow Nulls
🔑	ActionID	int	☐
	ActionTypeID	int	☐
	ProcessID	int	☐
	Name	varchar(200)	☐
	Description	varchar(MAX)	✓
			☐

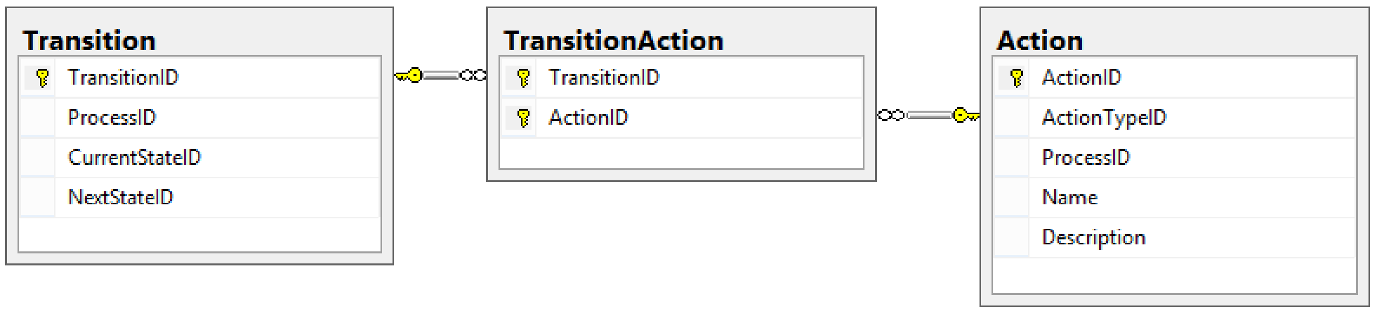
Our design for ActionTypes and Actions looks like this:



Transition Actions

Now that we've defined what Actions could ever be performed, we need to get more specific: which Actions can be performed for a particular Transition?

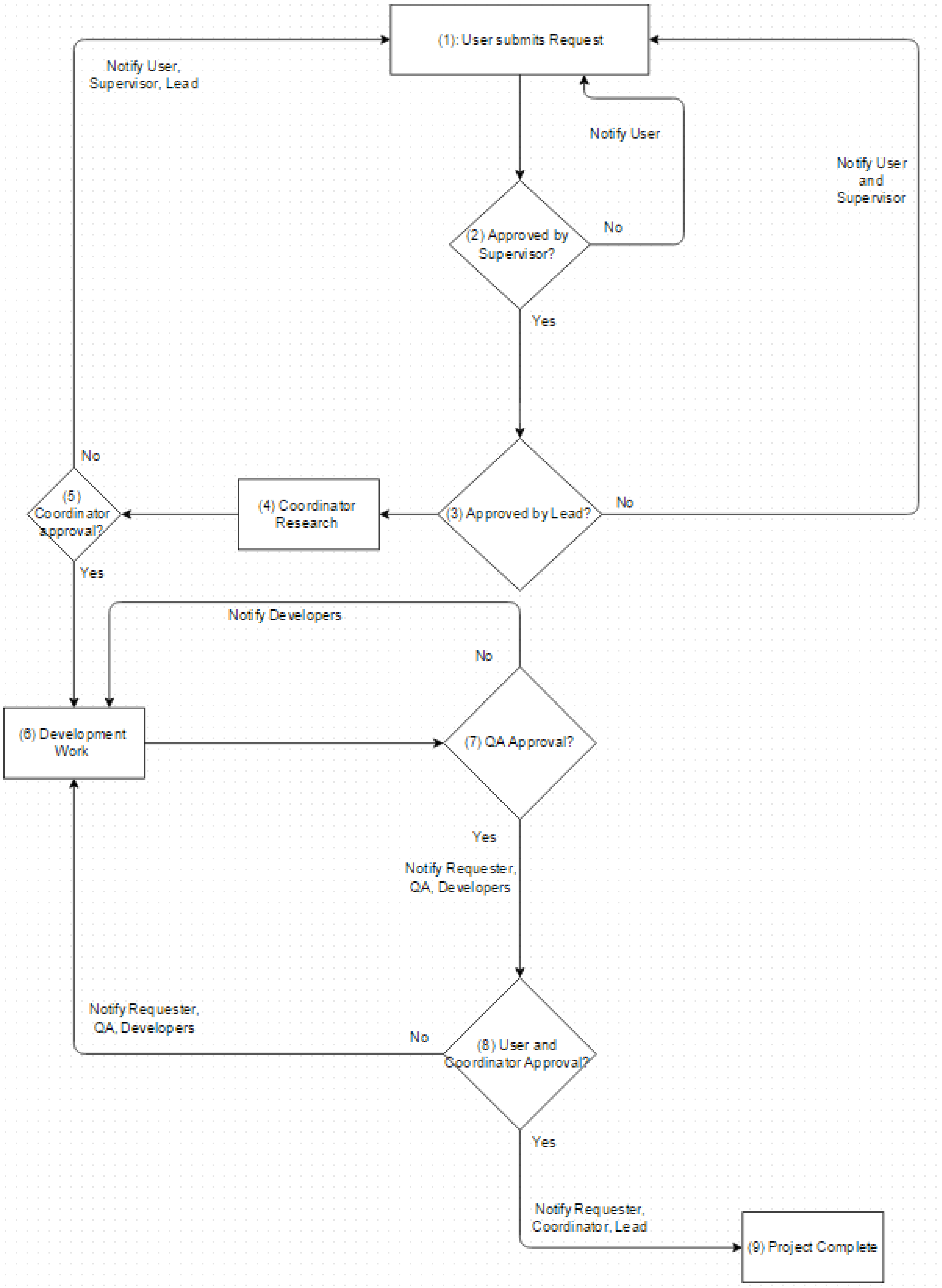
The relationship between Transition and Action is many-to-many:



Activities

Activities are things that can happen as a result of a Request entering a State or following a Transition.

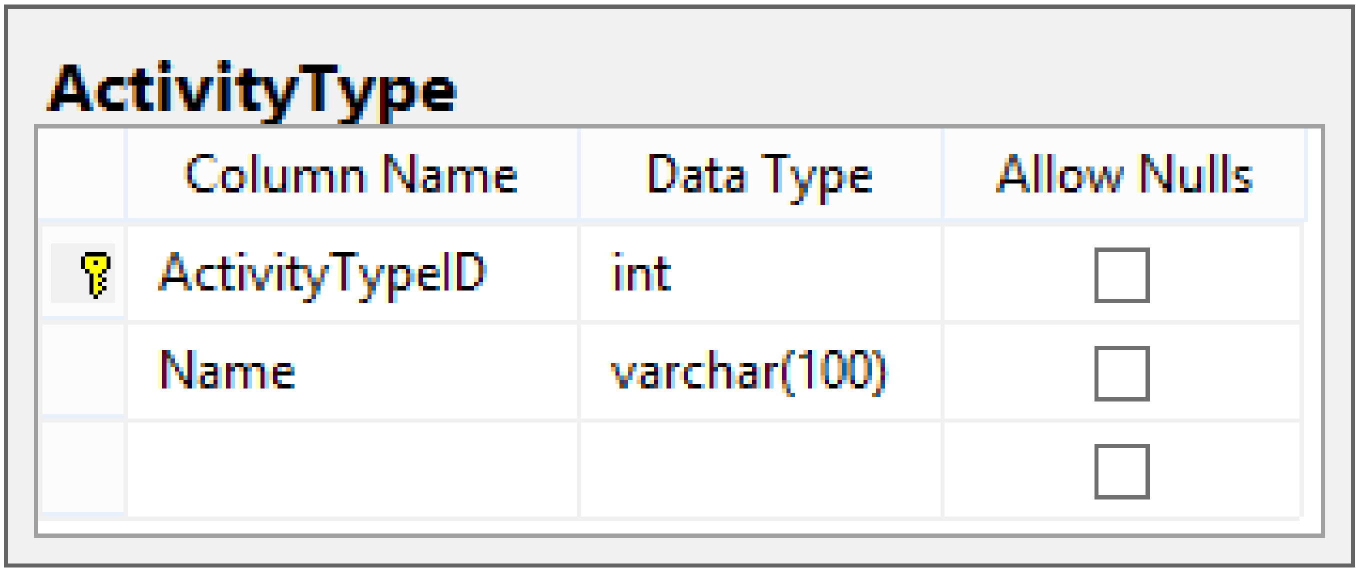
For example, let's see the diagram from Part 1 again.


(1): User submits Request
Notify User, Supervisor, Lead
Notify User
Notify User and Supervisor
(2) Approved by Supervisor?
No
Yes
(3) Approved by Lead?
No
(4) Coordinator Research
(5) Coordinator approval?
No
Yes
Notify Developers
No
(6) Development Work
(7) QA Approval?
Yes
Notify Requester, QA, Developers
Notify Requester, QA, Developers
No
(8) User and Coordinator Approval?
Yes
Notify Requester, Coordinator, Lead
(9) Project Complete

In Step 3 of this flowchart, we may want to add the Lead as a stakeholder on a request, so that s/he will receive automatic emails about the status of that request. However, if the Lead denies the request we will want to notify the Requester, but if s/he approves the request we need to notify the Coordinators.

In other words, in this example adding a stakeholder is an activity that we want to happen when a Request reaches a certain state, and sending email is an activity that we want to happen when a certain transition is followed. We need to design for both scenarios.

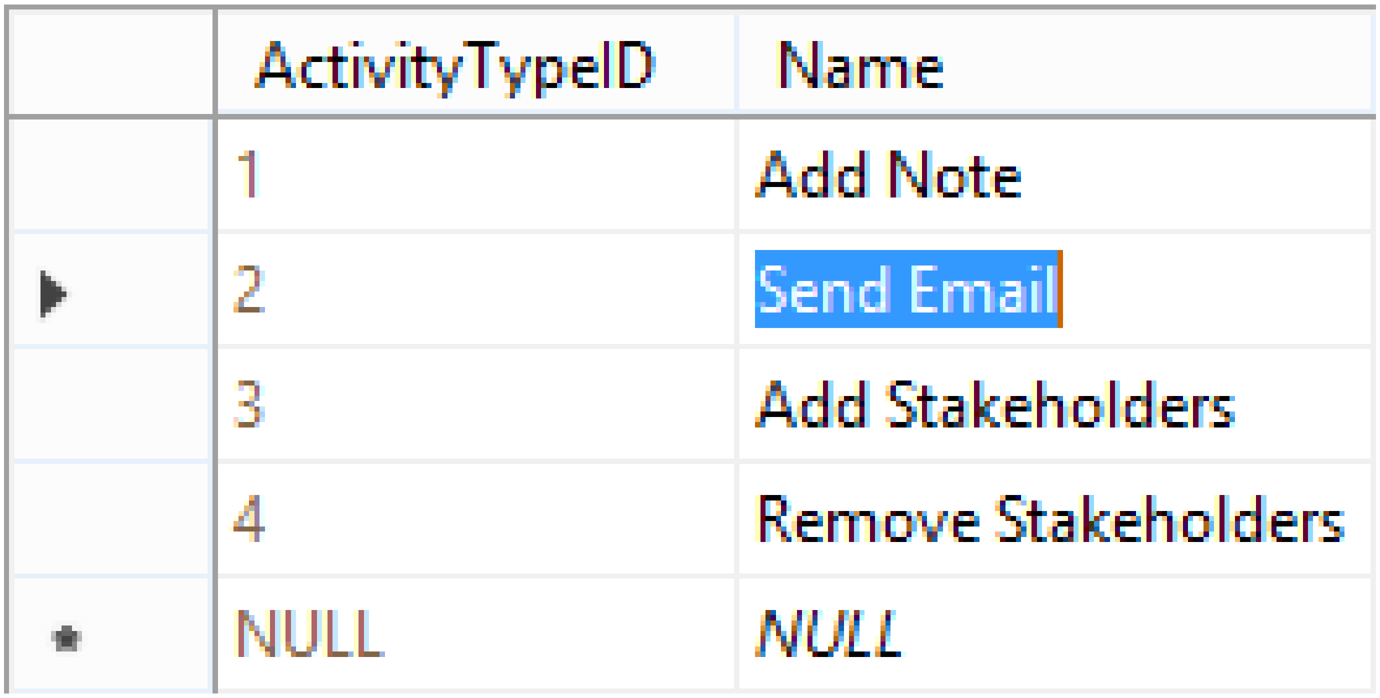
First, we need to know what kinds of actions we can do. This table is just like StateType and ActionType in that it is not unique to the Process and can be considered static. Here's the design for the ActivityType table:



ActivityType

	Column Name	Data Type	Allow Nulls
🔑	ActivityTypeID	int	☐
	Name	varchar(100)	☐
			☐

Just like ActionType, the values for ActivityType are static. We'll use the following values:

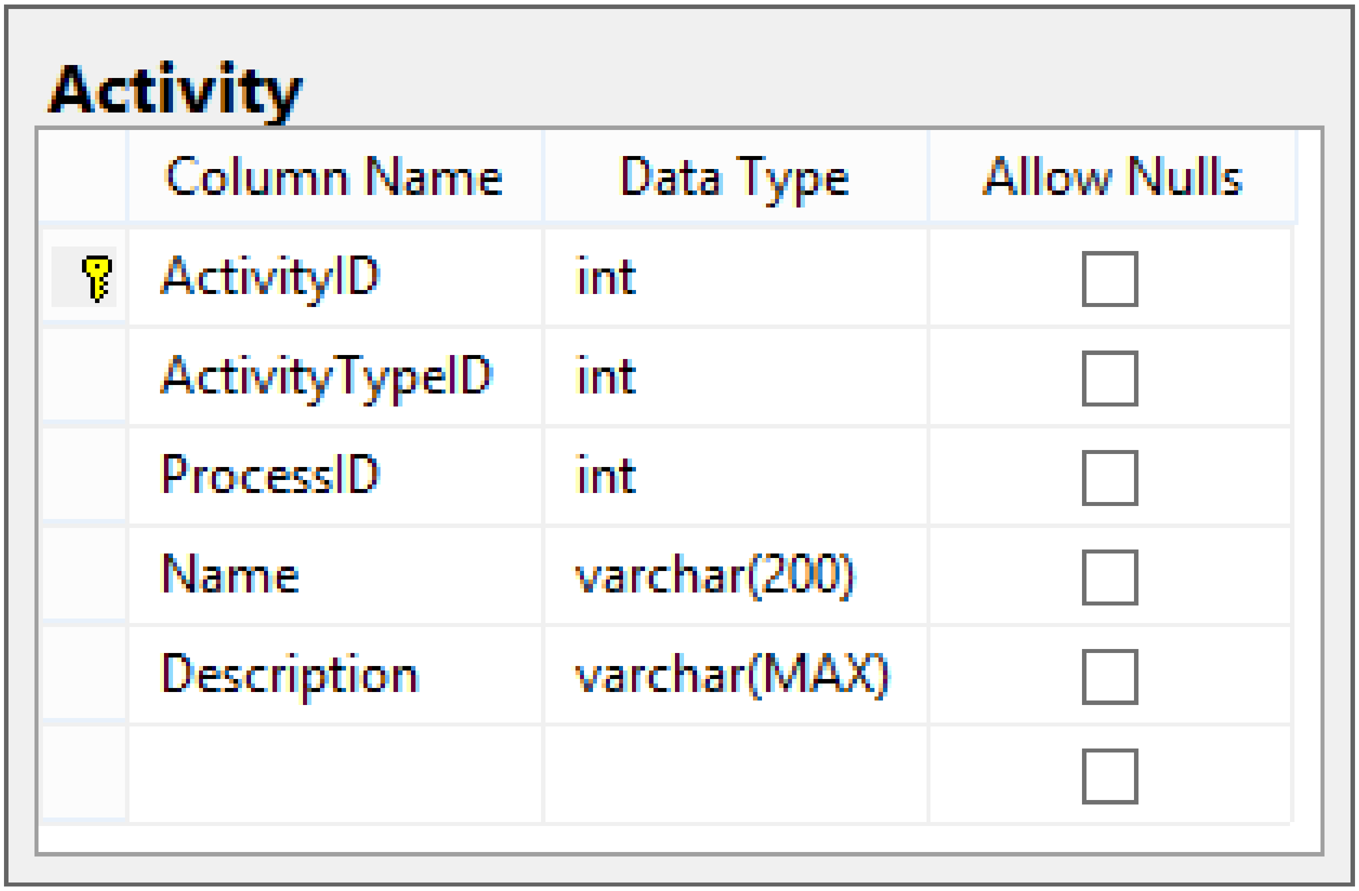


	ActivityTypeID	Name
	1	Add Note
▶	2	Send Email
	3	Add Stakeholders
	4	Remove Stakeholders
*	NULL	NULL

- **Add Note**: Specifies that we should automatically add a note to a Request.
- **Send Email**: Specifies that we should send an email to one or more recipients.
- **Add Stakeholders**: Specifies that we should add one or more persons as Stakeholders on this request.
- **Remove Stakeholders**: Specifies that we should remove one or more stakeholders from this request.

You could define quite a few more kinds of ActivityTypes, but for now we'll just use those four.

The last thing we need to do is design our Activity table, which will look a lot like the Action table:



Activity

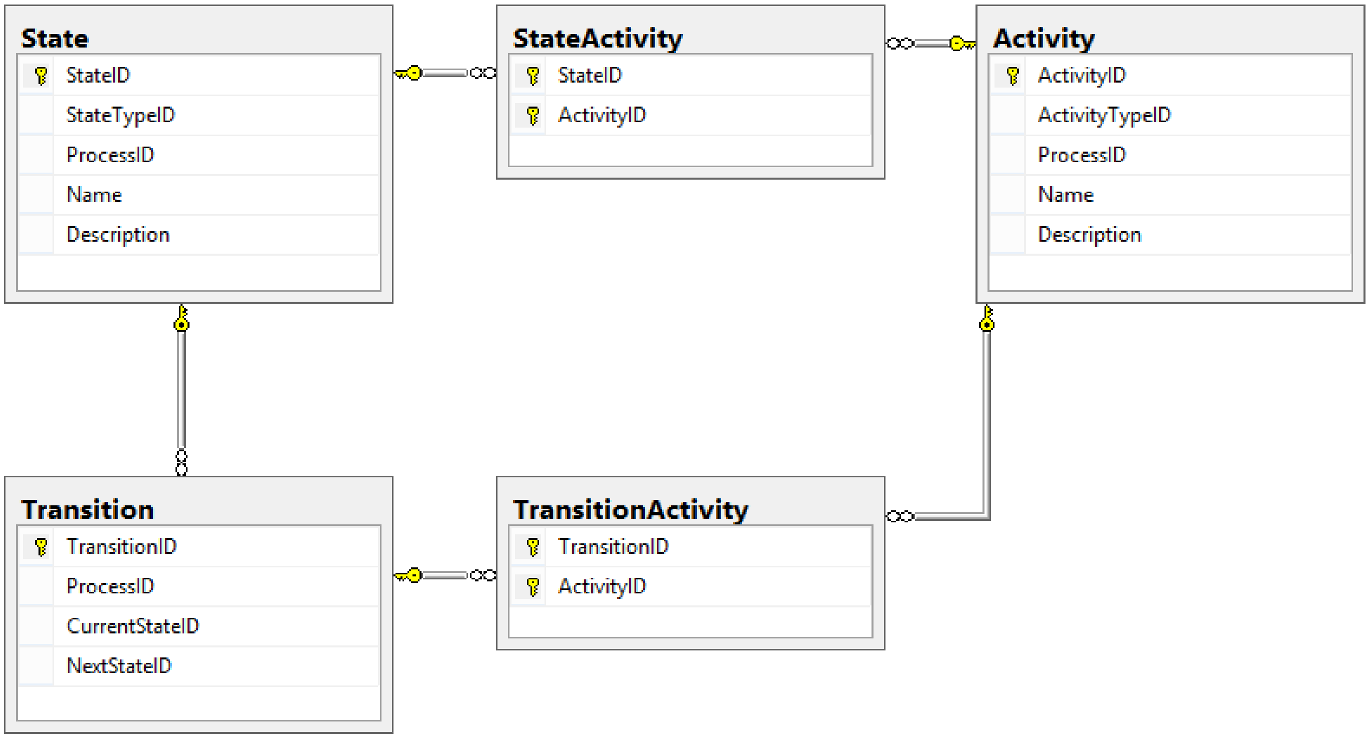
	Column Name	Data Type	Allow Nulls
🔑	ActivityID	int	☐
	ActivityTypeID	int	☐
	ProcessID	int	☐
	Name	varchar(200)	☐
	Description	varchar(MAX)	☐
			☐

State and Transition Activities

Once we've got the base Activity table defined, we can start designing how the Activities are associated to States and Transitions. As a reminder, we want to be able to kick off Activities in two situations:

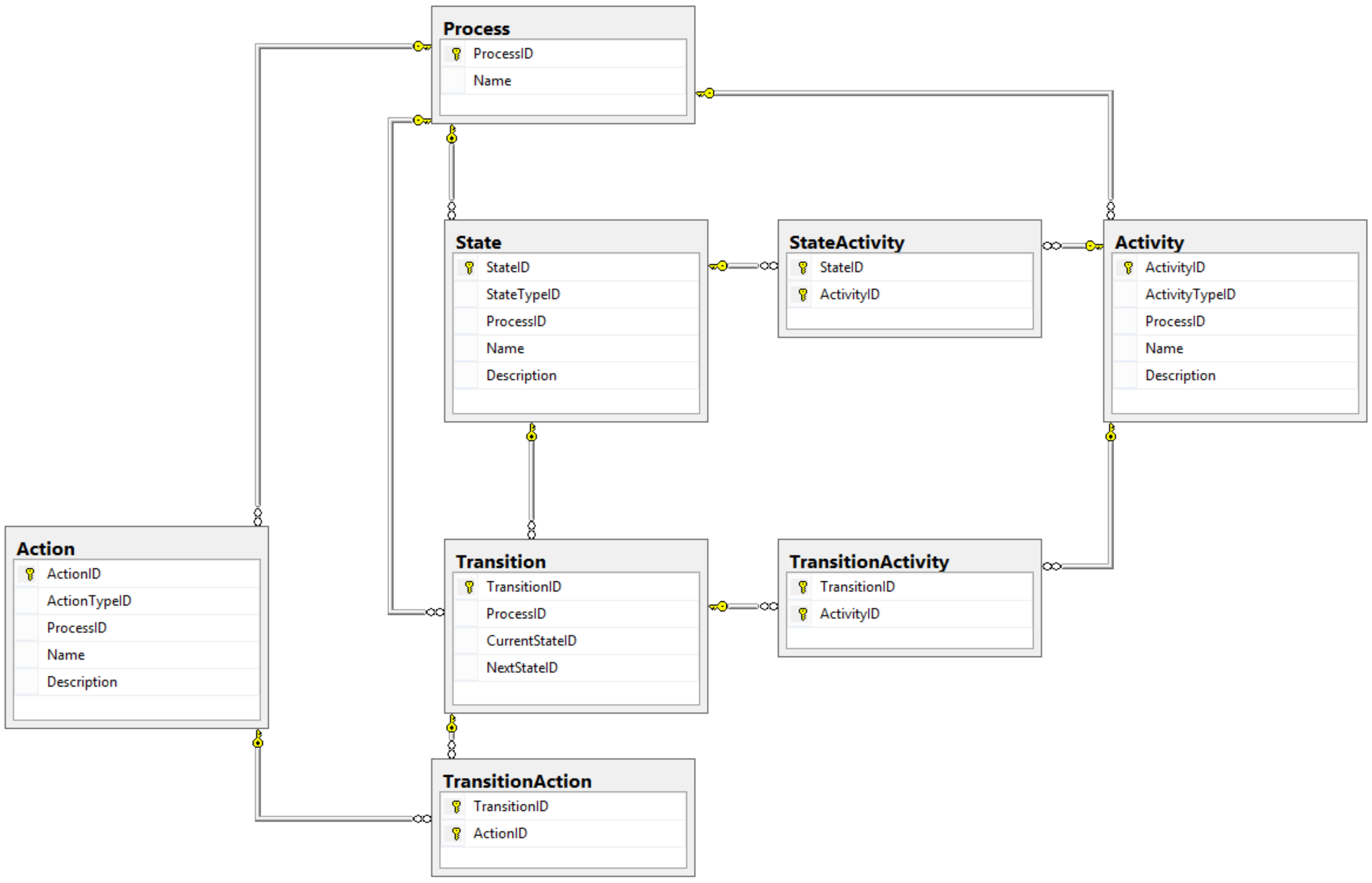
- When the Request enters a State
- When the Request follows a Transition

This means that we still need to associate Activities with States and Transitions, like so:



What did we accomplish?

Our database diagram (showing the Process, States, Transitions, Actions, and Activities) looks like this:



In this post, we demonstrated how we can store what Actions can be performed by Users, and what kinds of Activities can be kicked off by certain States or Transitions. In essence, we showed what Users can do to Requests, and what happens to the Users as a result.

We still have a piece of all this missing, though: exactly who can actually perform the Actions or receive the Activities? We'll answer that question in the next post, Part 6 of this series, where we will discuss **Groups and Targets**.