TASKZERO – **STEP 3**

At this point, after completing Step 2, you should have a solution that allows you to add a new item to the to-do list and edit it back. The final step is adding the code to delete an existing item.

- As first thing, let's add some JavaScript code to trigger the server-side code that will delete the selected to-do item. Open the **pv_pendingtasks.cshtml** file and edit the button element in the ta ble. The folder is Views/Dashboard. Edit the code as shown below.

```
<button class="btn btn-danger"
        data-action-delete="@task.TaskId">
        <i class="fa fa-fw fa-trash"></i>
</button>
```

- Who's going to handle the click on the button? Add the following code to the parent Razor file for the partial view. Open the file **index.cshtml** in Views/Dashboard and add the following at the bottom.

```
<script>
    $("button[data-action-delete]").click(function (evt) {
        evt.stopPropagation();
        var id = $(this).data("action-delete");
        if (confirm("Are you sure?")) {
            var url = "/task/delete/" + id;
            Ybq.post(url,
                "",
                function(response) {
                    if (response.Success) {
                        // Refresh list of tasks
                        Ybq.invoke("/dashboard/tasklist", function(html) {
                            $("#task-dashboard").html(html);
                        });
                    }
                });
        }
    })
</script>
```

If you run the application now and click on the red trash button a message box will ask to confirm the deletion. The next step is adding the necessary code to delete a to-do item.

- Now let's proceed creating a new controller action to process the request to delete a to-do item. In the **TaskController** class let's add a new method.

```
#region DELETE TASK
public ActionResult Delete(string id, string signalrConnectionId)
{
    Guid guid;
    var outcome = Guid.TryParse(id, out guid);

    if (!outcome)
        throw new InvalidGuidException("Could not find specified task");

    try
    {
        _service.QueueDeleteTask(guid, signalrConnectionId);
    }
```

```
    catch (Exception exception)
    {
        return HandleException(exception);
    }

    // Message delivered
    var response = new CommandResponse(true)
        .SetPartial()
        .AddMessage("Delivered");
    return Json(response);
}
#endregion
```

- You miss a method **QueueDeleteTask** on the **TaskService** class. Let's add it in the Application folder.

```
public void QueueDeleteTask(Guid id, string signalrConnectionId)
{
    var command = new DeleteTaskCommand(id, signalrConnectionId);

    Bus.Send(command);
}
```

- Create also a file named **DeleteTaskCommand.cs** in the Commands folder of the CommandStack project.

```
using System;

namespace TaskZero.CommandStack.Commands
{
    public class DeleteTaskCommand : NotifyCommand
    {
        public DeleteTaskCommand(Guid id,
            string connectionId) : base(connectionId)
        {
            TaskId = id;
        }

        public Guid TaskId { get; set; }
    }
}
```

- At this point, everything is up and running for deleting an existing to-do item. The only missing part is handling the command that orders to delete. Let's then open the **ManageTaskSaga.cs** file and edit as below.

```
public class ManageTaskSaga : Saga,
        IAmStartedBy<AddNewTaskCommand>,
        IHandleMessages<UpdateTaskCommand>,
        IHandleMessages<DeleteTaskCommand>
{

}
```

Now add some more code in the body of the class to implement the **IHandleMessage** interface.

```
public void Handle(DeleteTaskCommand message)
{
```

```
    var task = Repository.GetById<Task>(message.TaskId);
    task.MarkAsDeleted();
    Repository.Save(task);
}
```

- You need also to make some changes to the **Task** class to make it support the deletion. Open the **Task.cs** file in the Model folder of the CommandStack project. As first thing add the following method:

```
public void MarkAsDeleted()
{
    var deleted = new TaskDeletedEvent(TaskId);
    RaiseEvent(deleted);
}
```

Next, edit the **Task** class definition to support another closed version of the **IApplyEvent** I interface. (Add **IApplyEvent<TaskDeletedEvent>**.)

```
public void ApplyEvent(
            [AggregateId("TaskId")] TaskDeletedEvent theEvent)
{
     Deleted = true;
}
```

- Finally, add a **TaskDeletedEvent** class in the Events folder of the Shared project.

```
using System;
using Memento;

namespace TaskZero.Shared.Events
{
    public class TaskDeletedEvent : DomainEvent
    {
        public TaskDeletedEvent(Guid id)
        {
            TaskId = id;
        }

        public Guid TaskId { get; set; }
    }
}
```

- This is enough to update the Command stack. We won't have in this case any additional code that refreshes the UI. The UI will be refreshed care of the JavaScript we added earlier. All that remains to do is updating the Query stack. Open the **Denormalizers** folder in the ReadStack project and pick up the **ManageTaskDenormalizer** class. Edit as below.

```
public class ManageTaskDenormalizer :
    IHandleMessages<TaskCreatedEvent>,
    IHandleMessages<TaskUpdatedEvent>,
    IHandleMessage<TaskDeletedEvent>
{
}
```

- In the same file, also add an implementation for the **IHandleMessage** interface.

```
public void Handle(TaskDeletedEvent message)
{
```

```
    using (var context = new TaskContext())
    {
        var task = (from t in context.PendingTasks
            where t.TaskId == message.TaskId
            select t).SingleOrDefault();
        if (task == null)
            return;

        context.PendingTasks.Remove(task);
        context.SaveChanges();
    }
}
```

- All done. Your application now should be able to pick and delete an existing to-do item.