Streamframe Recruitment Test

In this test, you will implement a legacy tasking system that features dependencies. Each task will have an ID, title, status, and parent task ID. A task's status is **IN PROGRESS** until it is marked as **DONE**. While any task can be marked as **DONE**, a "parent" task (one with dependencies) must be marked as **COMPLETE** automatically when all of its dependencies (and sub-dependencies) are likewise marked as **COMPLETE.** A task is only considered **COMPLETE** when it's marked as **DONE** and either has no dependencies or all of its dependencies are also **COMPLETE**. Any individual task may have any number of dependencies but should never result in a circular dependency. For example, if Task A depends on Task B, then Task B cannot also depend on Task A.

You must build a web application that implements the above logic with following requirements. Complete as many requirements possible with the highest priority being those at the top of this list:

- Create a task listing page that shows a flat list of individual tasks and:
  - Each task's ID, description, and **IN PROGRESS** / **DONE** / **COMPLETE** status,
  - A status checkbox for each task to mark/unmark it as **DONE** when clicked.
- The task listing page should feature a status filter (**IN PROGRESS**, **DONE**, **COMPLETE**).
- Create a task creation form that takes the following inputs:
  - Task name (required),
  - Parent task ID (optional).
- Check for and prevent circular dependencies when creating a task that specifies a parent.
- Upgrade the task listing page so that parent tasks also show:
  - The total number of dependencies,
  - The number of dependencies marked as **DONE**,
  - The number of dependencies marked as **COMPLETE**.
- Upgrade the task listing page so that:
  - Marking a task as **DONE** will also check the status of all dependencies. When all dependencies are **COMPLETE**, mark the task as **COMPLETE** (instead of **DONE**).
  - Marking a task as **IN PROGRESS** (by clearing the status checkbox) should update its parent task (if it has one) so that the parent's status changes from **COMPLETE** to **DONE**. A parent task must not revert to **IN PROGRESS** from **DONE** or **COMPLETE**.
  - Repeat these two processes on the task's parents (if any) until the status change has propagated all the way to the top of the hierarchy.
- Upgrade the task listing page from a flat list to a nested hierarchal list. That is, all of the dependencies for a task should appear in a separate list within the parent task.
- Upgrade the task listing page to allow tasks to be edited:
  - Allow a task's name to be changed.
  - Allow a task's parent task to be changed. Doing this must trigger a status change propagation behaviour as described above.
- Optional / bonus features:
  - The task listing page should use pagination to show only 20 tasks at a time. This applies to the root level as well as for dependencies within a task.
  - Real time update of task listing page when changes are made by others.
  - Use AJAX wherever possible to reduce or eliminate page loads.

Database Structure

Below is a description of the Task table's database structure. Due to external dependencies on this table, its structure cannot be modified, however, you are free to create any additional tables to support the application's requirements.

Task Table

```
CREATE TABLE IF NOT EXISTS `tasks` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `title` varchar(255) NOT NULL,
    `status` tinyint(1) NOT NULL DEFAULT '0',
    `parent_id` int(11) NOT NULL DEFAULT '0',
    PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;
```

Column	Description
id	This is the unique identifier for each task.
title	This is a title for each task.
status	This is the task's status. Accepted values are: 0 = **IN PROGRESS** 1 = **DONE** 2 = **COMPLETE**
parent_id	If non-zero, this is the parent task's unique **id**.

Judgment Criteria

Your submission will be judged according to the following criteria:

1. **The correctness of the application** – Your ability to understand the requirements and implement bug-free solutions that fulfil those requirements is the first and foremost priority.
2. **The structure of the code** – The code patterns and data structures you employ tell us a great deal about your knowledge and experience.
3. **The algorithms utilized** – Your choice of algorithms and the degree of success when implementing them informs us about your logical abilities. We'll also be looking for memory and / or CPU performance issues.
4. **The cleanness and clarity of the code** – Consistent style and documentation are important for large-scale software projects.
5. **The application's front-end UI/UX** – All applications have users, and we want to see what considerations you make on their behalf.

Frequently Asked Questions

Q: What is the difference between **DONE** and **COMPLETE**?

A: The **DONE** status is specific to a single task by itself, while the **COMPLETE** status reflects the statuses of all dependencies. If a task is marked as **DONE** but has no dependencies, it must be automatically marked as **COMPLETE** instead. A task that's being marked as **DONE** that does have dependencies can only be marked **COMPLETE** when all of its dependencies are also **COMPLETE**.

Q: Can I use any libraries or frameworks to help me?

A: Yes, you can use any library or framework as long it's free or open source, however, all back-end application code must be written PHP or NodeJS and the front-end must be HTML/CSS and JavaScript.

Q: What database software must I use for this application?

A: We prefer MySQL, but you can use any database as long as there's a free version available.

Q: Will I fail this test if cannot finish all of the requirements?

A: We use this test to assess your skill level, so there's no way to fail short of not doing the test at all. How you perform on the test will be compared against the availability of junior, mid-level, and senior positions.

Q: I'm stuck on one of the requirements. Can I skip it and move on to next requirement?

A: You can, however some requirements depend on earlier ones, so skipping one may make another requirement impossible to complete.

Q: What is a circular dependency?

A: A circular dependency is a chain of dependencies that loops back on itself. This occurs when a task at one point in the hierarchy refers to a task further down in the hierarchy as its parent. This structure makes it impossible to trace a task's parents toward the highest (root) level in the hierarchy and can lead to an infinite loop in code. With a working circular dependency check, there should never be a circular dependency stored in the tasks table.

Q: How do I submit my application for review?

A: You can zip it up and email it to our recruiters, or give them access to any source control software you're using so we can download and examine it. If any special setup steps are required to run your application, please include instructions.

Q: How will you test my application?

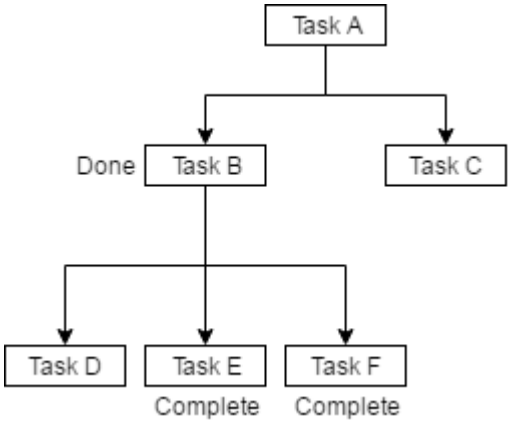
A: We will run the application internally and enter our own test data. Your application will be judged according to the Judgment Criteria section on the previous page.

Q: I have a question that this FAQ doesn't answer. Who do I ask?

A: Feel free to email to us if you have any questions regarding this task, however, you must do all of the work yourself. If you're invited for an interview, we will be discussing your code with you.

Example 1

In this scenario, Task A has two dependencies: Task B and Task C. However, Task B has three dependencies: Tasks D, E, and F. Tasks B, E and F were marked as **DONE**, but only tasks E and F are considered **COMPLETE** because they have no further dependencies. This is diagrammed below.

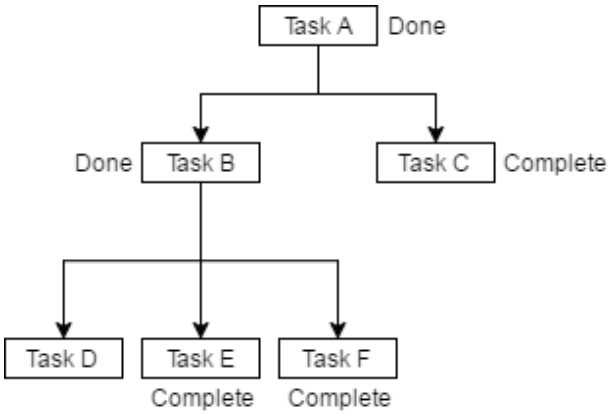


Task table data representation:

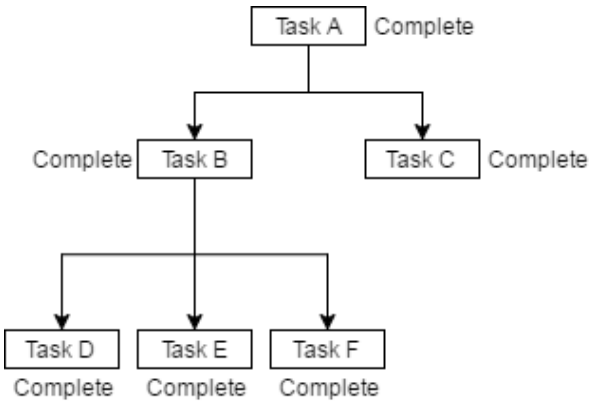
id	title	status	parent_id
1	Task A	0	0
2	Task B	1	1
3	Task C	0	1
4	Task D	0	2
5	Task E	2	2
6	Task F	2	2

Example 2

In this scenario, Task D is the only incomplete task while the rest are marked as either **DONE** or **COMPLETE** as shown in the diagram below.



When Task D is marked as **DONE**, it will automatically be set to **COMPLETE** and triggers a chain reaction that first marks Task B as **COMPLETE** followed by Task A as shown in the diagram below.



Task table data representation before:

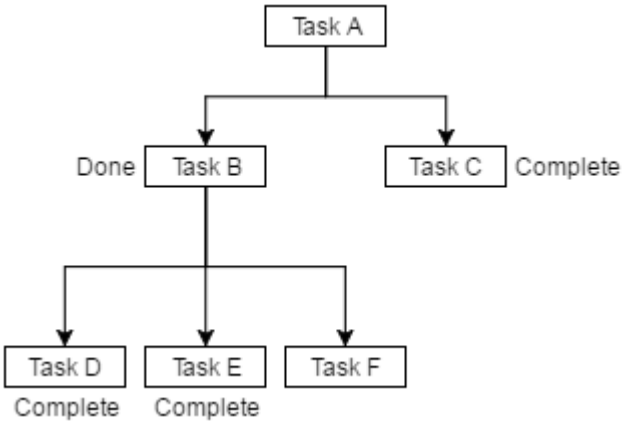
id	title	status	parent_id
1	Task A	1	0
2	Task B	1	1
3	Task C	2	1
4	Task D	0	2
5	Task E	2	2
6	Task F	2	2

Task table data representation after:

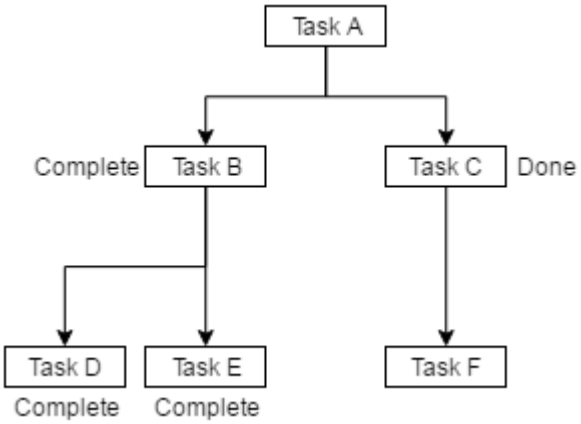
id	title	status	parent_id
1	Task A	2	0
2	Task B	2	1
3	Task C	2	1
4	Task D	2	2
5	Task E	2	2
6	Task F	2	2

Example 3

In this scenario, Task F is incomplete and also a dependency of Task B as shown in the diagram below.



Task F will be removed from Task B's dependency list and instead added to Task C. Since Task F was the only incomplete dependency of Task B, moving it will cause Task B to be marked as **COMPLETE**. Likewise, Task C was previously marked as **COMPLETE** but now has the incomplete Task F as a dependency, so Task C must be marked as **DONE**. The expected result is shown in the diagram below.



Task table data representation before:

id	title	status	parent_id
1	Task A	0	0
2	Task B	1	1
3	Task C	2	1
4	Task D	2	2
5	Task E	2	2
6	Task F	0	2

Task table data representation after:

id	title	status	parent_id
1	Task A	0	0
2	Task B	2	1
3	Task C	1	1
4	Task D	2	2
5	Task E	2	2
6	Task F	0	3