

PHP-Implementation

DATABASKONSTRUKTION (IT354G)

WILLIAM LINDHOLM (A22WILLI)

Innehållsförteckning

Innehållsförteckning	1
1. Formulär (G)	2
2. Listbox (G)	6
3. Sökning (G)	7
4. Förändring av innehåll (G)	8
5. Visning av tabeller (G)	9
6. Generering av tabellrubriker (VG)	9
7. Exekvering av procedur (G)	10
8. Exekvering av procedur (VG)	11
9. Dynamiska länkar (G)	12
10. Formulär med dolda fält (VG)	13
11. Uppkoppling till SQL-databas	14
Bilagor	16
Bilaga A:	16

1. Formulär (G)

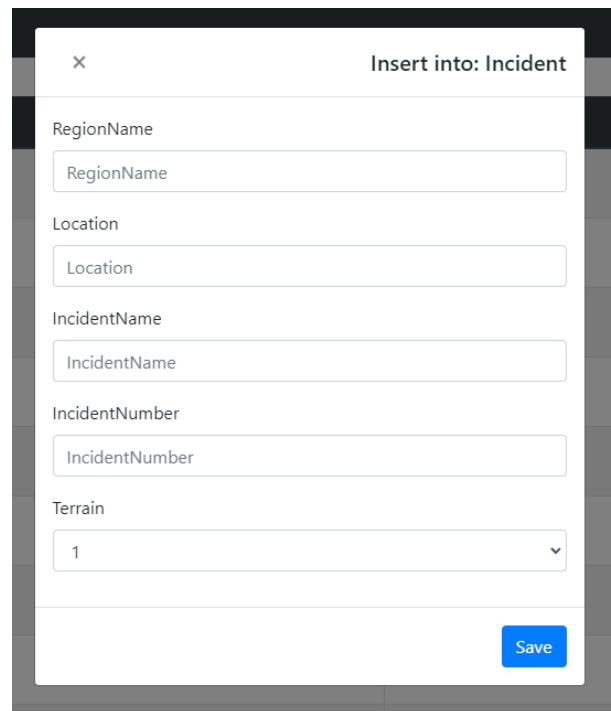
För att lägga till data i en databas genom ett webbgränssnitt används ofta formulär. Eftersom databasen kommer att ha många tabeller där användare kan lägga till data var det viktigt att skapa en generell funktion för datainmatning. Därför skapades klassen "ModalBuilder", som automatiskt genererar inmatningsrutor.

Innan data kan matas in i en SQL-databas måste vissa variabler bestämmas. Dessa inkluderar databasens adress (IP/domän, port, databas) och inloggningsuppgifter. I detta exempel är dessa uppgifter fördefinierade eftersom det inte finns något inloggningssystem. Därefter måste man bestämma vilken tabell data ska läggas till eller redigeras i. Slutligen kan användaren ange vilka kolumner som ska ändras eller skapas i, samt deras värden.

```
$modalBuilder = (new ModalBuilder())  
    ->setModalId('insertModal')  
    ->setTableName("Incident")  
    ->setPostHandler($handlerFactory->createHandler('Incident'))  
    ->addColumn("RegionName")  
    ->addColumn("Location")  
    ->addColumn("IncidentName")  
    ->addColumn("IncidentNumber")  
    ->addDropDownColumn("Terrain", getColumnValues("Terrain", "TerrainCode"));
```

Figur 1: Användning av "ModalBuilder".

Figur 1 visar hur "ModalBuilder" används. Ett konkret exempel visas där en modal skapas för inmatning i kolumnen "Incident". Den modalen innehåller fält för RegionName, Location, IncidentName, IncidentNumber och Terrain. Utifrån detta skapas motsvarande HTML med en form: "<form method='POST'>", se resultat i figur 2.



The image shows a modal dialog box titled "Insert into: Incident". It contains five input fields: "RegionName", "Location", "IncidentName", "IncidentNumber", and "Terrain". The "Terrain" field is a dropdown menu currently showing the value "1". A blue "Save" button is located at the bottom right of the dialog.

Figur 2: Resultat av "ModalBuilder".

Data som matas in skickas som HTTP POST-data till servern. Detta kan sedan hanteras på servern genom den globala PHP-variabeln "\$_POST", som kan ses som ett "dictionary" med all postdata servern tar emot. Till exempel kan "IncidentName" hämtas med \$_POST['IncidentName'].

För att göra processen enkel och återanvändbar har en mestadels generell lösning implementerats med hjälp av "PostHandler", ett interface. Det är i princip ett kontrakt som definierar vilka metoder en klass som hanterar postdata ska ha. Den viktigaste metoden här är handlePostData(), som tar emot och bearbetar inskickad data för att uppdatera databasen. Eftersom vissa tabeller kan ha mer komplexa krav på inmatningsfält, till exempel för sammansatta nycklar, skapades en fabriksklass. Den kan returnera specifika postdatahanterare baserat på behov. I exemplet (figur 1, rad 4) hämtas en hanterare för tabellen "Incident".

De flesta tabeller, inklusive "Incident" kräver ingen specifik datahanterare, och använder en generisk implementation. Denna klass, kallad "GenericInsertHandler" implementerar det tidigare nämnda "interfacet" "PostHandler" och hanterar i princip allt skapande av data i databasen.

```
$columns = array_keys($data);  
$placeholders = array_fill(0, count($columns), '?');  
  
$sql = "INSERT INTO {$this->tableName} (" . implode(', ', $columns) .  
      ") VALUES (" . implode(', ', $placeholders) . ")";
```

Figur 3: Generellt skapande av SQL-kommandon för insättning.

I figur 3 illustreras hur SQL-kommandot, som ansvarar för datainsättningen, skapas. För att säkerställa kompatibilitet med alla potentiella datainsättningar på webbplatsen tar funktionen emot namnet på den specifika tabellen som ska användas samt den data som ska infogas. I rad 1 extraheras kolumnnamnen. Istället för att direkt extrahera värdena i rad 2, initieras variabeln "\$placeholders" med frågetecken '?'. Antalet frågetecken motsvarar antalet värden som ska infogas. Detta möjliggör att värdena kan bindas säkert vid exekvering med hjälp av "prepared statements". SQL-kommandot konstrueras sedan genom att sätta ihop de olika komponenterna med strängkonkatenering. PHP-funktionen "implode" används för att omvandla kolumnnamn och värden till en sträng, i detta fall separerade med kommatecken, för att uppfylla SQL:s syntaxkrav.

Efter att SQL-kommandot har konstruerats, förbereds det för exekvering genom att använda PDO (PHP Data Objects) metoden prepare(). Detta skapar ett "förberett uttryck", vilket är en teknik som används för att säkerställa att SQL-kommandot exekveras på ett säkert sätt, särskilt när det gäller att skydda mot potentiella SQL-injektionsattacker. När det förberedda uttrycket är klart, försöker systemet exekvera det med de faktiska värdena från "\$data-arrayen" (Kopierad från \$_POST). Om exekveringen av någon anledning misslyckas, kastas ett undantag för att informera om felet. Detta undantag ger en tydlig indikation på att insättningen av data i den angivna tabellen inte lyckades, vilket möjliggör snabb felsökning och korrigering. Se figur 4.

```
$stmt = $this->pdo->prepare($sql);  
if (!$stmt->execute(array_values($data))) {  
    throw new Exception("Failed to insert data into {$this->tableName}.");  
}
```

Figur 4: Exekvering av "prepared statement".

Till sist, för att den data som skapats ska dyka upp på hemsidan, används funktionen "refreshTables()". Som uppdaterar hela sidan. Här hade det kanske varit värt att använda AJAX, för att möjliggöra uppdatering av endast det som ändrats på sidan. Detta är något som kan förbättras i framtiden.

```
function RefreshTables() {  
    $location = $_SERVER['PHP_SELF'];  
    if (!empty($_SERVER['QUERY_STRING'])) {  
        $location .= "?" . $_SERVER['QUERY_STRING'];  
    }  
    header("Location: " . $location);  
    exit();  
}
```

Figur 5: Omladdning av sida.

Varpå data ändrats, raderats eller skapats, bör detta visas på sidan, vilket inte är något som händer automatiskt. Därför skapades funktionen "RefreshTables()" som körs varje gång data har modifierats på något sätt. Först hämtar funktionen den nuvarande relativa adressen adressen användaren befinner sig på, som sparas i "\$location". Därefter konkateneras "query strings" på adressen ifall det finns några, alltså det som befinner sig efter frågetecknet (?), om det finns någon query. Till sist används "header("location: ") " för att förflytta klienten till den adressen som hämtats genom föregående operationer. Detta är ett bra sätt att uppdatera sidan på då det fungerar oberoende av sidans domännamn, och filens plats i filsystemet. Se implementation i figur 5.

2. Listbox (G)

Vid inmatning av data krävs ibland inmatning av främmande nycklar, dessa är fördefinierade utifrån de primärnycklar som redan finns i den främmande tabellen. Databasen kommer därför svara med ett fel om dessa inte är inmatade korrekt. Därför bör det finnas ett sätt att välja bland de främmande nycklar som existerar.

Listboxar kan användas i detta fall, då de enkelt låter användaren välja ett av de redan existerande värdena för att undvika "foreign key constraint errors". Detta hanteras genom att automatiskt generera listboxar för dessa. Se PHP-kod för generering av HTML i figur 6, HTML i figur 7, och resultat i figur 8.

```
foreach ($this->dropdownColumns as $column => $values) {  
    $options = '';  
    foreach ($values as $value) {  
        $options .= "<option value='{ $value }'>{ $value }</option>";  
    }  
    $modalBody .= "<div class='form-group'><label for='$column'>$column</label>  
<select class='form-control' id='$column' name='$column' required>  
    $options</select></div>";  
}
```

Figur 6: Generering av listbox.

Ovan; i figur 6 genereras listboxen utifrån den data som befinner sig i \$values, där varje element får en egen "<option>" tagg. Dessa värden hämtas från databasen genom en separat funktion, "getColumnValues()", som befinner sig i "/db/dbhelper.php", se bilaga A.

```
<select class="form-control" id="Incident" name="Incident" required>  
    ...  
    <option value="Arson, 108">Arson, 108</option>  
    <option value="Heist, 101">Heist, 101</option>  
    <option value="Ambush, 106">Ambush, 106</option>  
    ...  
</select>
```

Figur 7: Genererad HTML för en Listbox.

StartDate	EndDate	IncidentName
2023-04-01		Espionage
2023-05-01	2023-05-10	Breach
2023-03-10	2023-03-30	Sabotage
2023-02-05	2023-02-18	Heist
2023-02-05	2023-03-12	Heist
2023-02-05	2023-03-12	Heist
2023-02-21	2023-02-25	Heist
2023-09-28	2023-10-01	Kidnapping
2023-09-28	2023-09-28	Heist
2021-02-01	2021-03-08	Arson
2021-02-01	2021-03-08	Arson
2023-01-01	2023-02-05	Sabotage

×

Insert into: Operation

OperationName

OperationName

SuccessRate

SuccessRate

StartDate

2023-10-02

EndDate

2023-10-02

GroupLeader

K23

Incident

Breach, 105

213123asd, 234234
Arson, 108
Heist, 101
Ambush, 106
Kidnapping, 102
Assault, 109
Breach, 105
Hackning, 923
Sabotage, 103
Espionage, 104
Hijacking, 107
Kidnapping, 989995
inringning, 901
Kidnapping, 42023

Figur 8: Bild på resultat av genererat formulär.

3. Sökning (G)

För att söka i databasen kan SQL-frågor byggas där teckenmatchning används för att hämta värdena.

```
$searchQuery = $_GET['query'] ?? null;

$sqlQuery = $searchQuery ? "SELECT * FROM ArchivedReport WHERE Title LIKE
'%"{$searchQuery}%";" : "SELECT * FROM ArchivedReport;";
```

Figur 9: Generering av SQL-fråga för sökning.

I figur 9, hämtas först den söksträng som genomförts, om det inte finns någon tilldelas variabeln "\$searchQuery" null explicit för att undvika fel. Därefter skapas sökningen genom att, om \$searchQuery är sant, dvs inte är null, skapas en SQL-fråga där teckenmatchning "SQL LIKE"-kommandot. Här innebär "%\$searchQuery%" att alla värden där titeln innehåller sökvärdet kommer hämtas. Värt att tänka på är att denna inmatning inte är skyddad mot SQL injektion, och kan därför vara något som behöver förbättras vid den slutgiltiga implementationen av PUCKO:s hemsida. I figur 10 syns den HTML/PHP som används för att söka efter rapporter genom att skapa ett formulär med metoden "GET" och sedan skapa en "url-query" på det sökvärde som görs.


```
<form class="form-inline" action="" method="GET">

    <div class="form-group">

        <input type="text" class="form-control" id="query" name="query"
            placeholder="Search for old report" value="' . $_GET['query'] . '">

    </div>

    <button style="margin-left:0.5em;" type="submit" class="btn btn-
        primary">Search</button>

</form>
```

Figur 10: Formulär för sökning av värden.

4. Förändring av innehåll (G)

För att förändra innehåll som redan existerar på databasen kan flera olika tekniker användas, bland annat går det uppdatera, ta bort eller köra diverse procedur som kan göra diverse förändringar på redan existerande data. I detta fall valdes SQL-operationen "UPDATE".

```
$updateGroupLeaderModalBuilder = (new ModalBuilder())
    ->setModalId('updateModal')
    ->setTableName("Operation")
    ->setPostHandler($updateFactory->createHandler("Operation", $condition))
    ->addDropdownColumn("GroupLeader", getColumnValues("GroupLeaders", "CodeName"));
```

Figur 11: Formulär för uppdatering av gruppleddare.

Det formulär som skapas i figur 11, kommer då skapa en HTML "<form method='post'>" tagg. Där den nya gruppleddaren kommer skickas som nyckeln "GroupLeader". Och tabellen sätts till Operation där alla värden ska uppdateras, utifrån det villkor som matas in i "\$condition". Villkoret skapas i detta fall utifrån frågesträngar som skickas när en användare klickar på en operation. Varpå användaren valt en ny användare, kommer koden i figur 12 hantera detta.

```
$setClause = [];
foreach ($data as $column => $value) {
    $setClause[] = "$column = '{$value}'";
}
$setClause = implode(' , ', $setClause);
$sql = "UPDATE {$this->tableName} SET {$setClause} WHERE {$this->condition}";
$stmt = $this->pdo->prepare($sql);
```

Figur 12: Hantering av Postdata för "UPDATE".

Först hämtas alla värden som ska "sättas" med SQL-kommandot "SET" genom att iterera över den data som befinner sig i "\$data"-variabeln. På så vis skapas ny data utifrån de tabeller och värden som är satta. Sedan matas villkoret in i funktionen separat i detta fall, se figur 13.

```
$condition = "OperationName = '{$operationName}'  
            AND StartDate = '{$startDate}'  
            AND IncidentName = '{$incidentName}'  
            AND IncidentNumber = '{$incidentNumber}';
```

Figur 13: Villkor för uppdatering av gruppleddare (SQL).

5. Visning av tabeller (G)

För att skriva ut respektive tabeller, skapades först en klass, "tableFactory" som automatiskt kan generera olika typer av tabeller, beroende på vad som efterfrågas, exempelvis en tabell där varje rad går att klicka på, eller en tabell med en raderingskolumn. Detta blev i längden ohållbart, och därför skapades även en klass kalled "tableBuilder". Där avancerade tabeller kan byggas automatiskt genom "methodchaining", likt hur "modalBuilder" i figur 11 fungerar. Genom "TableFactory" går det därmed exempelvis att skriva "tableFactory::createCustomTable(\$queryGroupLeader)" för att generera en tabell som tar in en SQL-fråga som hämtar alla agenter av typen gruppleddare. För själva genereringen av tabellens rader körs koden i figur 14:

```
foreach ($this->db->getPdo()->query($this->query) as $row) {  
    $output .= $this->buildRow($row);  
}  
  
//nedan kod befinner sig i buildRow($row)  
$rowData = "<tr>";  
foreach ($row as $key => $value) {  
    if (!is_numeric($key)) {  
        $rowData .= "<td>" . $value . "</td>";  
    }  
}  
$rowData .= "</tr>";
```

Figur 14: Centrala delar för att bygga rader i "TableBuilder".

Först skapas raderna genom att iterera över alla rader som hämtas från databasen med den frågan som skickats in tidigare. Sedan itererar funktionen buildRow över varje rad och genererar <td> taggar för varje värde, då den data som hämtas innehåller ett numeriskt index, som med hjälp av denna sorteras bort.

6. Generering av tabellrubriker (VG)

Tabellbyggaren genererar även rubriker automatiskt genom följande kod i figur 15.

```
foreach ($row as $key => $value) {
    if (!is_numeric($key)) {
        $output .= "<th>" . $key . "</th>";
    }
}
```

Figur 15: Automatisk generering av kolumntitlar.

Istället för att använda värdena, och placera dessa i <td> taggar, används nu nyckeln, som hämtats genom samma SQL-fråga som tidigare använts till att skapa rader för att bygga tabellhuvudet, genom att placera "\$key" värdet som innehåller SQL-kolumnens namn, i en <th> tagg. Återigen ignoreras de numeriska indexvärdena som hämtas vid SQL-frågan. Se hela "TableBuilder" klassen här: [Github | TableBuilder](#), och Hela TableFactory klassen här: [Github | TableFactory](#). Se resultat i figur 16.

GroupLeaders

CodeName	FirstName	LastName	Salary	AvgSuccessRate
A1	John	Doe	50000	
D56	Emily	Brown	50500	1.0000
I92	Jan	Jansson	93	
K23	Förstanamn	Efternamn	945	
L29	William	Lindholm	100000	0.5000

Figur 16: Resultat av automatisk tabellgenerering.

Detta kan genereras automatiskt då den fråga som använts för att hämta alla värden i en tabell också hämtar kolumnerna, eller de "nycklar" som används för att identifiera var ett värde hör hemma. Se hur en rad ser ut när den hämtas från databasen i figur 16. Notera här att varje värde har en tillhörande "nyckel". Ex kolumnen "FirstName", har värdet "Jesper".

```
array(10) { ["CodeName"]=> string(3) "X03" [0]=> string(3) "X03" ["FirstName"]=>
string(6) "Jesper" [1]=> string(6) "Jesper" ["LastName"]=> string(5) "Molin" [2]=>
string(5) "Molin" ["Salary"]=> string(5) "23032" [3]=> string(5) "23032"
["AvgSuccessRate"]=> string(6) "1.0000" [4]=> string(6) "1.0000" }
```

Figur 17: Rådata (en rad) hämtad från databas.

Detta innebär indirekt att endast kolumnnamnen kan hämtas på samma sätt som värdena filtreras ut för användning vid konturering av resterande delar av tabellen. Fast för att bygga <tablehead> sektionen.

7. Exekvering av procedur (G)

Den första proceduren som skapats arkiverar en rapport när knappen "Archive" klickas på. Denna knapp genereras av en "callback" funktion där en rad matas in i funktionen. Denna funktion skapar därefter en knapp, som är "wrappad" av en <form> som skickar ett postmeddelande. Meddelandet hanteras därefter av funktionen i figur 18.

```
if (isset($postData['Function'])) {  
    $procedureName = $postData['Function'];  
    unset($postData['Function'], $postData['OperationType']);  
  
    $placeholders = array_fill(0, count($postData), '?');  
    $sql = "CALL {$procedureName}(" . implode(", ", $placeholders) . ")";  
  
    $stmt = $pdo->prepare($sql);  
  
    $stmt->execute(array_values($postData));  
}
```

Figur 18: Generell implementation av procedurshanterare.

Inom funktionen som visas i figur 18, görs initialt en verifiering för att säkerställa att det mottagna meddelandet är av typen "EXECUTE", vilket är den typ som "procedureHandlers" konfigurerats att lyssna på i detta fall. Därefter hämtas namnet på den procedur som ska exekveras från "Function". Denna nyckel, tillsammans med "OperationType", avlägsnas från \$postData-variabeln eftersom dessa inte behöver skickas till de procedurer som kommer köras av denna kod.

För att konstruera SQL-kommandot, fylls en \$placeholders-variabel med ett antal frågetecken (?), motsvarande det antal element som befinner sig i \$postData. Dessa frågetecken placeras sedan inom "\$sql" variabeln där själva kommandot för att anropa proceduren byggs upp.

Slutligen förbereds ett SQL-statement med hjälp av \$pdo->prepare(\$sql), varefter alla värden inom \$postData matas in med hjälp av funktionen array_values(). Denna funktion omorganiserar \$postData så att dess värden indexeras numeriskt istället för associerat, vilket säkerställer att data associerad med nyckeln 1 placeras i det första frågetecknet i SQL-kommandot.

8. Exekvering av procedur (VG)

Den andra implementerade proceduren skapades för att sortera operationer inom ett visst datumspann. Detta sker genom att anropa SQL-proceduren "GetOperationsInRange()", där tidigast startdatum, och senast slutdatum används som parametrar. Dessa variabler hämtas genom ett formulär med två <input type='date'> taggar. Dessa skapar en fin meny där användaren kan välja datum som automatiskt hamnar i ett format SQL-databasen kan läsa.

```
if ($_SERVER["REQUEST_METHOD"] == "POST" && isset($_POST["OperationType"]) &&
    $_POST["OperationType"] == "filter") {
    $startDate = $_POST["startDate"];
    $endDate = $_POST["endDate"];
    $query = "CALL GetOperationsInRange('$startDate', '$endDate')";
} else {
    $query = "SELECT * FROM Operation";
}
```

Figur 19: Hantering av postdata för sortering av operationer.

I koden i figur 19 sker följande: Först kollar koden att den data som befinner sig i "\$_POST" är ämnad åt just denna funktion genom att kolla att "operationType" är satt till "filter" och att metoden som använts är post. Därefter konstrueras SQL-frågan genom att lägga in startdatum och slutdatum i funktionen. Till sist konstrueras tabellerna med följande funktion: "tableFactory::createTableWithRedirect(\$query, ...)", där den genererade SQL-frågan matas in som källa.

9. Dynamiska länkar (G)

För att tillåta radering av rader i databasen, skapas dynamiska länkar för varje rad där radering är möjlig. Dessa länkar pekar mot filen delete.php. Länkarna innehåller information om vilken tabell datan ska raderas från, samt alla värden som den specifika raden består av. Med hjälp av denna information skapas sedan ett SQL-skript som raderar de värden från databasen där raden matchar alla de värden som skickats med länken. Se figur 20 för kodexemplet som används för att generera raderingslänken.

```
private static function addDeleteButton($row, $table) {
    $params = [];
    foreach ($row as $key => $value) {
        if (!is_numeric($key)) {
            if (isset($value)) {
                $params[] = "{$key}=" . urlencode($value);
            }
        }
    }
    $queryString = implode('&', $params);
    $tableQuery = "table=$table&";
    $deleteUrl = "./delete.php?" . $tableQuery . $queryString;
    return "<a href='{$deleteUrl}' class='btn btn-block btn-danger m-0'>Delete</a>";
}
```

Figur 20: Generering av knapp med dynamisk länk.

Koden i figur 20 representerar en metod med namnet `addDeleteButton`, som används som en "callback-funktion" av klassen `TableBuilder`. Denna metod anropas från `TableFactory` och matas in i `TableBuilder` för att generera "delete"-knappar för varje rad i tabellen. Metoden tar in två argument; `$row`, som representerar en rad av data, och `$table`, som representerar namnet på databastabellen.

Metoden inleds med en `foreach`-loop som går igenom varje kolumn i den aktuella raden (`$row`). Varje kolumn och värde kontrolleras så att kolumnnamnet inte är numeriskt med hjälp av `is_numeric()`-funktionen, samt att värdet inte är null med hjälp av `isset()`-funktionen. Om dessa villkor är uppfyllda, skapas en parametersträng där kolumnnamnet och värdet sammanfogas, separerade med ett likamed-tecken (=). För att säkerställa korrekt hantering av eventuella specialtecken, används PHP-funktionen `urlencode()` på värdet.

Efter att alla kolumn-värdepar har behandlats, sammanfogas de till en enda sträng med hjälp av PHP-funktionen `implode()`, där varje parameterpar separeras med ett &-tecken. En ytterligare parameter som anger tabellnamnet läggs till i början av strängen.

Slutligen konstrueras URL:en till `delete.php`-filen genom att sammanfoga bas-URL, tabellparametern och den nyss skapade parametersträngen. Sedan används en `<a>` tagg med en `"href=..."` där den genererade länken matas in för att möjliggöra den länk som genererats funktion.

10. Formulär med dolda fält (VG)

Vissa av de tabeller som visas på sidan har rader som går att klicka på. Dessa genereras genom att varje rad länkar till en ny sida där variabler som talar om vilken individuell tupel som länkats skickas med genom frågesträngar i URL:en. Varpå användaren navigerat till en sådan sida, och exempelvis ska ändra eller lägga till data, är förälderelementet redan förbestämt. Därför är det onödigt att användaren ska behöva mata in detta igen. Vid dessa tillfällen används dolda fält (hidden fields). Dessa fungerar precis som vanliga inmatningsfält, bara att användaren inte kan se dem. Dessutom måste ett förbestämt värde matas in i dem. Ett exempel där de används i den sidan som skapats i denna uppgift är när en användare klickar på en incident, för att visa vilka operationer och rapporter som skapats i denna incident.

```
//I modalBuilder:
->addHiddenColumn("IncidentName", $incidentName)
->addHiddenColumn("IncidentNumber", $incidentNumber);

//Motsvarande HTML:
<input type="hidden" name="IncidentName" value="Arson">
<input type="hidden" name="IncidentNumber" value="108">
```

Figur 21: Skapande av dolda fält.

De första två raderna (exklusive kommentaren) i figur 21, används i "ModalBuilder" för att generera dolda fält på de ställen där det finns data som redan är förbestämd, exempelvis då användaren valt denna information tidigare. Därefter genereras den HTML, som befinner sig på resterande rader utifrån detta. Se figur 22 för genereringskod.

```
foreach ($this->hiddenColumns as $column => $value) {  
    $modalBody .= "<input type=\"hidden\" name=\"$column\" value=\"$value\">";  
}
```

Figur 22: Generering av dolda fält i formulär.

11. Uppkoppling till SQL-databas

För att genomföra den uppkoppling som alla tidigare nämnda klasser, filer och funktioner använder sig av när de interagerar med databasen, skapades en "singleton"-klass, som ser till att endast en instans av denna klass kan skapas. Detta är fördelaktigt då det är onödigt att skapa en ny anslutning varje gång en del av programmet vill komma åt databasen.

Denna klass initieras i början av alla PHP-filer (genom koden i figur 23) som användarens klient direkt läser av. Alltså index.php, terrain.php, operations.php osv. Det vill säga Inte hjälpfiler såsom logging.php.

```
dbconnection::getInstance('host', 'a22willi', 'användare', 'lösenord');
```

Figur 23: instansiering av databasanslutning.

```
private function __construct($host, $dbname, $username, $password) {  
    try {  
        $this->pdo = new PDO("mysql:host=$host;dbname=$dbname",  
            $username, $password);  
    } catch (PDOException $e) {  
        die('Connection failed: ' . $e->getMessage());  
    }  
}
```

Figur 24: Konstruktör för klassen dbconnection.

I figur 24, skapas en anslutning till databasen, med de parametrar som matas in vid instansieringen av denna klass. Misslyckas klassen av någon anledning att upprätta en anslutning och skapa en pdo. Avslutas hela exekveringen av hemsidan, och skriver ut ett felmeddelande genom "die()" funktionen. Eftersom detta är det första som exekveras på hela sidan, kommer vi då få en blank sida med ett felmeddelande, exempelvis "authentication error".

En pdo som genererats här kan sedan hämtas varsomhelst i programmet genom följande kod i figur 25.

```
$db = dbconnection::getInstance();  
$pdo = $db->getPdo();
```

Figur 25: Hämtning av PDO.

Bilagor

Bilaga A:

Länk till Github repository: <https://github.com/LindholmLabs/PHP-databaskonstruktion>