KTH EECS

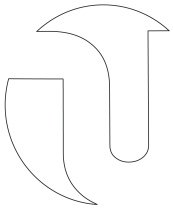
FlowerPower Project
Backend Component Description

Abstract

This document describes the required behavior and interfaces of the Backend component.

Version History

Date	Version	Author	Description
27/05/2018	1.4	Nicole Othman	Final update
18/05/2018	1.3	Nicole Othman	Updated with component description
12/05/2018	1.2	Nicole Othman	Updated with component description
10/05/2018	1.1	Nicole Othman	Layout established



IVAR JACOBSON CONSULTING

An Essential Unified Process Document

Table of Contents

1 Introduction

1.1 Document purpose

The purpose of this document is to describe the behavior and interfaces of the backend component in order to:

- Enable the implementation effort to be sized and managed
- Establish the dependencies that exist with other components
- Support the independent development of this component
- Provide a basis for testing that the component operates as required.

1.2 Document Scope

The scope of this document is limited to consideration of:

- The specification of the required behaviors for this component
- The specification of the interfaces that this component must implement
- Listing the required interfaces that this component depends upon
- Optionally specifying the internal structure and design of the component.

This scope of this document does not include consideration of:

- The unit tests that required to verify that this component conforms and performs to its specification
- The specification or implementation of any other components that depend upon this component or that this component depends upon.

1.3 Document Overview

This document contains the following sections:

- **Required Behavior** – behavior that is required from the component in terms of the responsibilities that it must satisfy and related requirements
- **Implementation Constraints** – constraints like implementation environment, implementation language and related patterns, guidelines and standards
- **Internal Structure** – optionally specifies the internal design structure of the component in terms of its constituent components and their relationships
- **References** – provides full reference details for all documents, white papers and books referenced by this document.

2 Required Behavior

Every user interaction on the website has to spark an event, this is where backend coding plays an important role. There are currently three different user actions that can be performed on the website, these required functions are:

- Login functionality – a user must be able to login
- Watering functionality – a user must be able to start the watering of a plant
- Set minimum dryness level functionality – a user must be able to update the minimum dryness level

Among these functions there's a current dryness level displayed and the user is able to view different pages on the website. All these actions require a backend component that can check whether or not an action has been made.

Whenever a user navigates to the login site and performs a login, typed user credentials such as username and password needs to be fetched and matched with a record in the database. If the entered data matches with the database record, the user is routed to a logged in site, from here the user can perform actions such as water a plant and set the minimum dryness level.

If the user presses a button specifically designed for watering a plant, it'll activate the watering function and a notice will be shown indicating that the plant has been watered. If the user presses a button specifically designed for setting the minimum dryness level, it'll update the current value and a notice will be shown indicating that the minimum dryness level has been set.

3 Implementation Constraints

For this project, we've used an architectural pattern called MVC (**M**odel-**V**iew-**C**ontroller). The view is responsible for the UI (**U**ser **I**nterface), i.e. responsible for displaying information for the user. The controller is responsible for managing the user inputs, i.e. interpreting them in order to understand what should be done and act accordingly. Finally, the model is the layer in which all the business logic is added.

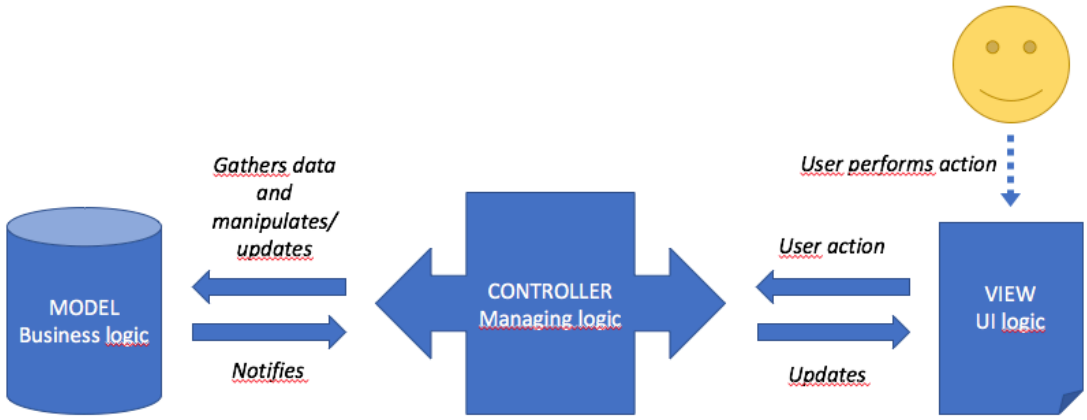


Figure 1: MVC design pattern, design Nicole Othman

The backend component consists therefore of four different packages with their belonging classes:

- database
  Consists of a class named "DatabaseHandler"
- websitecontroller
  Consists of a class named "WebsiteController"
- websitemodel
  Consists of a class named "WebsiteModel"
- websiteview
  Consists of a class named "WebsiteView"

The source code of the backend component is written in Python. In order to build the web application and running the server I've used Flask [1], a micro web framework for Python.

The database for storing user credentials is created in MySQL [2]. Our database is called "flowerpower" and the table in which we've stored the user credentials is called "Användare" and it consists of two columns named "namn" and "lösenord". The column named "namn" is the primary key, this value is unique and therefore used to uniquely identify the rows in the table. AN1 in the conceptual model (figure 2 below) indicates that the two columns together are a composite key, this means that the combination of the two columns can also be used to uniquely identify a row.

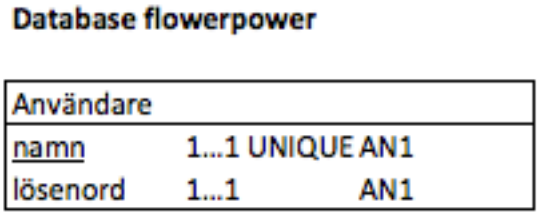


Figure 2: Conceptual model of the database

By installing and configuring the MySQL Connector/Python, which is a self-contained Python driver used for communication with MySQL servers [3], we're able to create SQL-queries towards our database.

4 Required/Provided Interfaces

In order to get the backend component to work completely it has to establish a connection with a database (MySQL), the database is used to fetch user records whenever a user performs a login. It also has to be able to establish a connection to the web server, which is a Python-powered web server built with the framework Flask. Finally, in order to react to user actions and perform accordingly, the backend component must be able to synchronize with the source code of the frontend component as well as the source code in the externalscontroller-package.

As this project is built using the MVC-structure, the backend component provides an interface between the frontend component and the more internal components such as the externalscontroller-class, these components are described more further in their respective documents.

5 Internal Structure

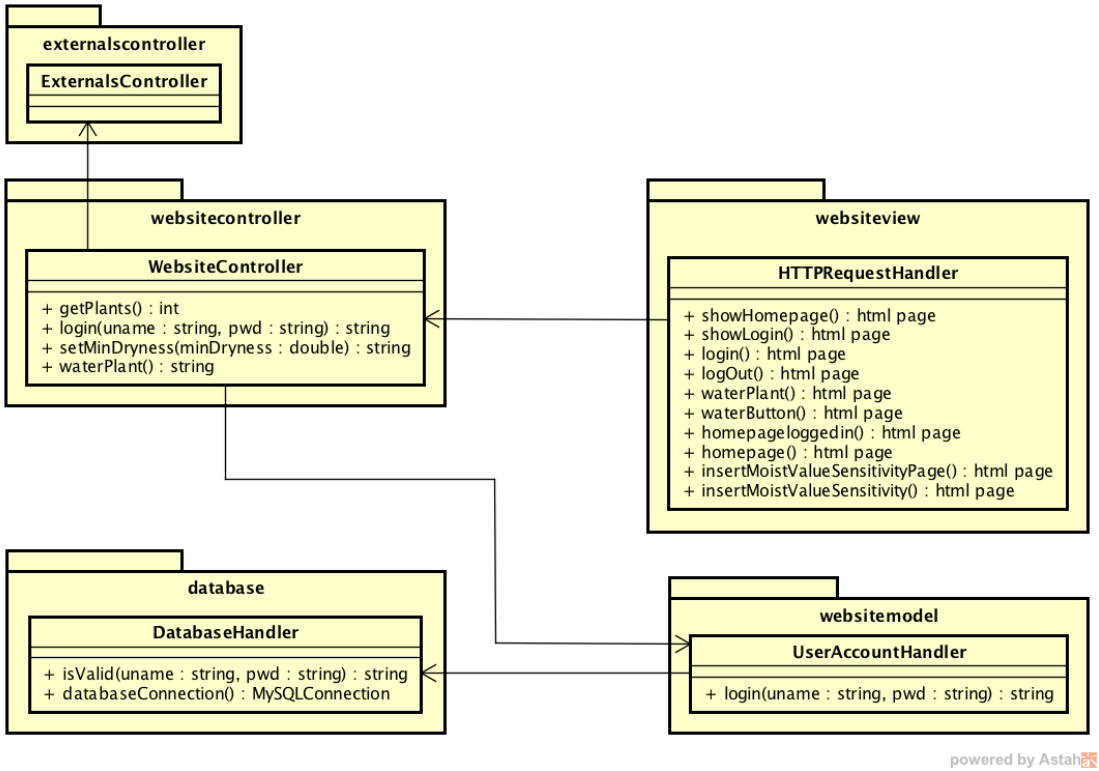


Figure 3: UML design of the backend component

HTTPRequestHandler-class

The key package in the backend component is the "websiteview", this package contains a class called "HTTPRequestHandler" and whenever a user performs a user action on the website it sparks an event in that specific class. As mentioned in section 4 Implementation constraints, the framework Flask is used for managing the website, which is designed by our frontend-developer.

By the use of the route()-decorator we're able to bind a function to a specific URL, i.e. the route()-decorator tells the Flask-framework what URL should trigger our function. In order to be able to shift between the different web pages I've used a Flask-function called render_template() which, instead of returning hardcoded HTML, returns a static HTML-file. This function also enables data to be passed to the specific HTML-file which we're taking advantages of by passing values such as the current moistness value to be displayed on the website. The return_template()-function is used in all methods inside the "HTTPRequestHandler"-class.

As mentioned in section 2 Required Behavior, we've three main functions that can be performed by a user. In the source code of this class I've added an if-statement that checks if a specific login-button has been pressed. When the button is pressed, the typed username and password are stored in variables which in return are passed as arguments in a method-call to the WebsiteController-class. If the result from the method call matches with the typed username it indicates on a successful login and a session-

variable is set to true, when the user then presses the logout-sign on the website, the session-variable is set to false. Similarly, there's an if-statement that checks whether or not the watering-button and the button for setting the minimum dryness value has been pressed.

In order to implement some security-aspects I've added a session-variable (mentioned above) which is set to true whenever a user is logged in. With the help of an if-statement, a control of the current value of the session-variable is made before rendering the correct web page. This prevents unauthenticated users from navigating to pages from where they can perform tasks such as watering the plant and setting the minimum dryness value.

All method calls in this class are made towards methods in the WebsiteController-class.

WebsiteController-class

This class is responsible for interpreting the user actions in order to understand what should be done and act accordingly. As there are currently three functions a user can perform from the website and a value indicating a plant's current moistness value, naturally there are four corresponding methods in this class.

The getPlants()-method calls upon a method in the ExternalsController-class which returns the current moistness value, this value is returned to the view which in turn renders a HTML-file with the specific data.

The login()-method takes two arguments, a username and a password, and makes a method call to a method in the UserAccountHandler-class. The result from this call is returned to the view which in turn renders a HTML-file with the specific data.

The setMinDryness()-method takes one argument, which is the value that the user has typed in, and calls upon a method in the ExternalsController-class. setMinDryness()-method returns a string value to the view which in turn renders a HTML-file with the specific data.

The waterPlant()-method calls upon a method in the ExternalsController-class. waterPlant()-method returns a string value to the view which in turn renders a HTML-file with the specific data.

UserAccountHandler-class

The UserAccountHandler consists of a method called login() which takes two arguments, a username and a password. These are passed as arguments to a method call in the DatabaseHandler-class which returns a string value. This string value is returned to the view which in turn renders a HTML-file with the specific data.

DatabaseHandler-class
This class is separated from the model-layer since it makes calls to an external database. The class consists of two methods, the isValid()-method and the databaseConnection()-method. The databaseConnection()-method tries to set up a connection and establish a session with the MySQL server. If the connection is successful, a MySQLConnection object is returned.

What the isValid()-method does is the following: It tries to establish a connection to the MySQL server, with the returned object from the databaseConnection()-method call it makes a method call to the cursor()-method in the MySQLCursor class. The MySQLCursor class instantiates an object that can execute SQL-queries, this object is then used when creating a SQL-query towards our database. The SQL-query fetches a username for the given password and if the fetched username matches with the typed username then the method will return the username in string format, otherwise it'll return a string with the value 'False'. Before returning the result, the connection is closed.

6 Tests

Throughout this project, we've had the ambition of writing code according to TDD (**T**est **D**riven **D**evelopment) [4]. This simply means that the tests are written before the actual source code and as the backend component consists of four classes, naturally there are four test-files for each class and their belonging methods.

When working with the tests we've used a Python framework specially designed for tests called pytest [5]. Pytest makes it easy to build simple and scalable tests, the framework also enabled us to implement automated tests which was extremely time saving when performing regression tests whenever new functionality was added.

Appendix A - References

[1] *Flask* (fetched 27/05/2018)
http://flask.pocoo.org/docs/1.0/
[2] *MySQL* (fetched 27/05/2018)
https://www.mysql.com
[3] *MySQL-connector* (fetched 27/05/2018)
https://dev.mysql.com/doc/connector-python/en/
[4] Ian Sommerville, chapter 8.2 "Test-driven development", Software Engineering, Ninth Edition, Addison Wesley, 2011
[5] *Pytest* (fetched 27/05/2018)
https://docs.pytest.org/en/latest/contents.html