rijksuniversiteit groningen

Hestia for iOS

Hestia

Architecture Document

Client: Lars Holdijk

iOS Team:

Andrei Mădălin Oancă

Daan Groot

Geanne Barkmeijer

Juan José Méndez Torrero

Jur Kroeze

Marc Fleurke

Tom den Boon

Zino Holwerda

TA: Feiko Ritsema

Iteration: 3

Introduction

This project aims at extending the existing Hestia system, which is developed to be an extendable home automation platform. The current system consists of a server and an Android client. We extend it with an iOS client. A web server/site is also currently in development by a separate team of software engineers.

In this document we describe the design- and architectural decision that were made creating the front- and back end of the iOS application.

Architectural overview of the entire system

The Hestia system contains various parts:

- Two clients, the iOS app, which is the main focus of this document, and an Android app.
- The peripherals, which can perform home automation tasks.
- A server, which provides a general interface for communication between the client and the peripherals.
- A web server/site, which will supply the login functionality as well as function as a bridge between the server and client.

iOS client

The app is designed to be the skin that can be used to interact with devices. The functionality is nothing more than a wrapper around the computation that is performed on the server. The client communicates with the server through HTTP requests over an HTTPS connection. This connection follows the REST protocols. The messages that the server and client use to interact are JSON objects, this way the interaction remains uniform across different platforms.

The current login system is a temporary solution, this will be changed to use Google Firebase. The login will also not happen locally anymore as cooperation with the web server will enable logging in from everywhere using said system. After login, the user will be shown a menu from which navigation is possible to different sections of the app. The main focus of the application lies on the device's screen. From here the user can change and add devices. These operations will use the API provided by the server and are performed on the server itself.

Server

A quick reiteration of the functionality of the server. It functions on the local network as a REST API, which means it can be interfaced using the HTTPS methods: GET, PUT, POST and DELETE. The server keeps track of the different devices for the client to interact with.

iOS app

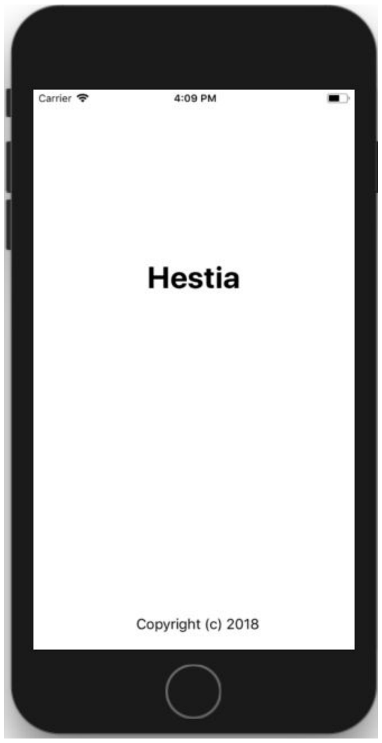
Architectural overview

The Hestia iOS app is split, very similarly to Android app, into a front- (No separate yet) and back end (package backend). The back end section is concerned with sending and receiving JSON objects from the server, it also serializes and deserializes said objects. The frontend is concerned with formatting and presenting the gathered data to the user and making it editable. The back end also contains the needed code and data for interacting with the server, saving its address and other data and maintaining a list of devices with their activators.

User interface design

Following the demands of the customer, we tried to design a “native” iOS look and feel for the application. We took inspiration for the design from the following Apple developed iOS apps: Contacts, Remote, Music, Settings and Clock. Examining the way Apple designs its apps should be a good starting point to achieve the desired iOS look.

Loading screen



Apples build-in apps do not feature a loading screen, but third-party apps often do. We decided to keep things simple and use a white screen with the Hestia logo.

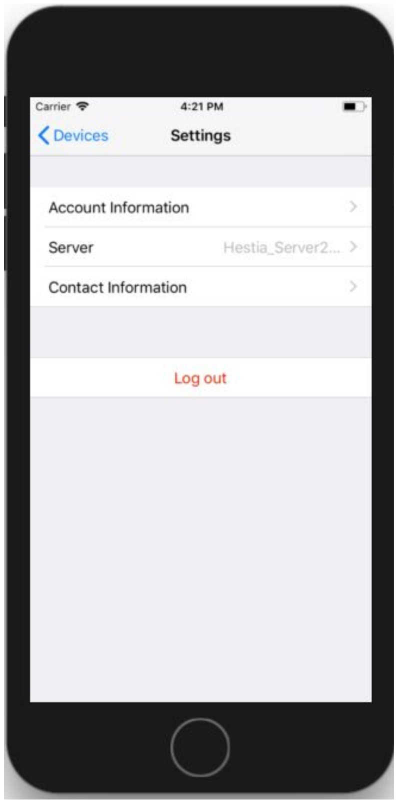
Login screen

To be able to control devices, the user should login by choosing to use one of the users accounts. For this login system Google Firebase is used. The Firebase Authentication component enables the user to login by using a Google-, Facebook-, Twitter- or Github-account. This screen contains four buttons, each of the buttons contains the logo of the specific platform. When choosing to use a Google-account, the user is taking to the accounts.google.com-page. Here the user can sign in with the users Google-account. After specifying the username and password, the user will return to the Hesta iOS app. The same procedure follows after choosing to sign in with a Facebook-, Twitter- and Github-account.

Settings screen

As any application in use, the Hestia application includes a Settings menu. This menu contains 4 buttons:

- Account Information
- Server
- Contact Information
- Log out



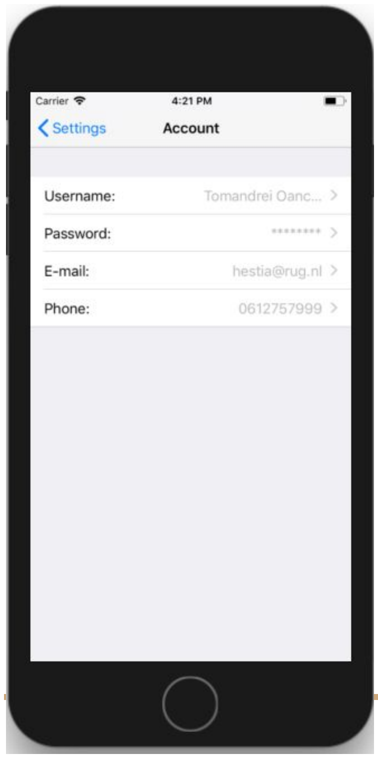
Server: Hestia_Server2...

In the Server Settings, the user will be able to change the server by adding the IP address of the new server that he wants to connect to.

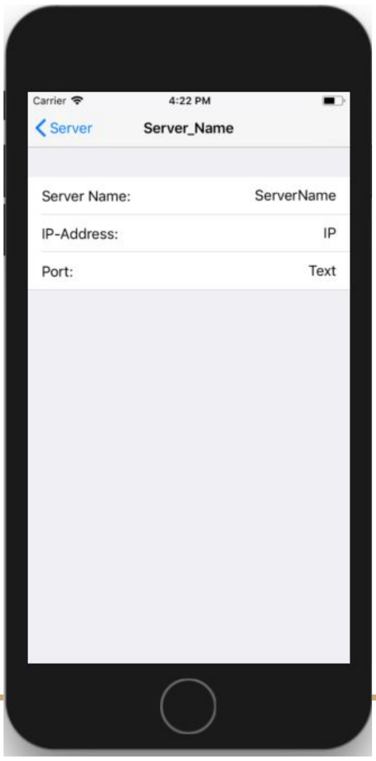
In the Account Settings, the user will be able to change his account's password by introducing the old one and the new one.

In the Contacts, there will be information about how to get contacts information of our development team.

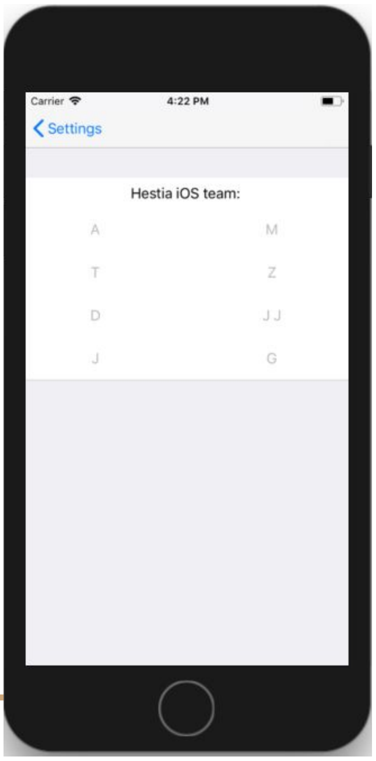
Log out just terminates the session for the current user and goes back to the Login screen.



Username: Tomandrei Oanc...
Password: ********
E-mail: hestia@rug.nl
Phone: 0612757999



Server Name: ServerName
IP-Address: IP
Port: Text

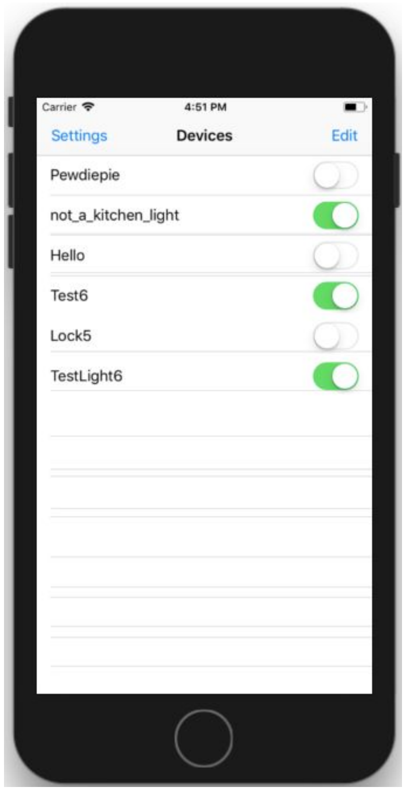


Hestia iOS team:

A	M
T	Z
D	J J
J	G

Devices screens

Devices main screen



The devices screen has a central role in the app. The user arrives at this screen after logging in. The screen contains a list with devices. Following the device name is an on/off button in the style that is abundant in iOS, for example in Settings. Some devices support a “floating” activator, for example, a dimmable light. For these kind of devices, a slider appears when the device name is tapped. If there are multiple “floating” activators for a device, multiple sliders will appear. If tapped again, they will disappear. So, initially the screen will be free from any sliders, which gives a clean overview of the devices.

Considered alternatives

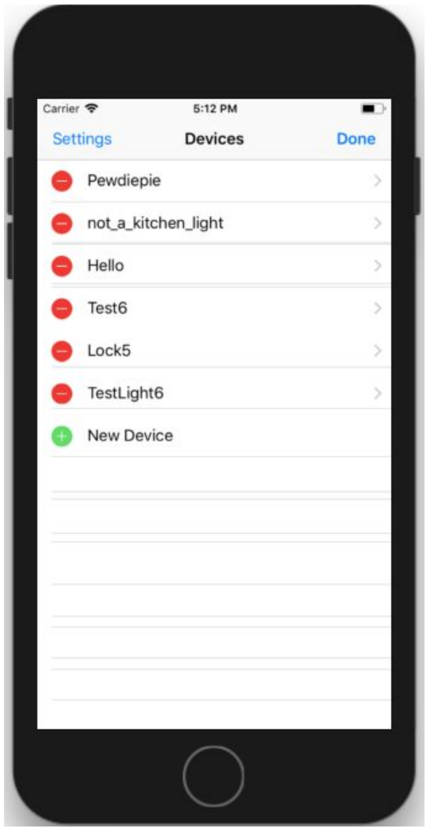
First we considered a different approach for showing sliders. The idea was as follows. If a device is switched on, the slider(s) appear(s). If switched off, the slider(s) disappear(s). This keeps the screen also free from elements that are not functional and has the extra benefit that it gives the user a clear suggestion on turning on the devices, that the device supports finer grained control. After discussion with the customer, we found that he did not like the sliders to be present if a device is turned on. Hence, that is why we decided to implement the alternative described in the section above.

In the top left of the devices main screen, one can find a settings button. This leads the user to the settings screen. In the top right of the screen contains the edit button. On tap the screen turns to edit devices mode. The placement of the edit button is not consistent among iOS apps: sometimes it on the right, sometimes on the left. We decided that the right would make more sense, as our application does not have a “+” button in the devices main screen, that is found in many apps the have the edit button placed in the left corner. Moreover, it is more easy to reach for right handed people.

Considered alternatives

We considered adding a “+” button in the devices main screen to add devices, but rejected this idea in favor of the way we add devices as explained in the next section. The main reason for this choice is that adding the “+” button instead of the settings button, would force us to find another way of accessing the settings screen. This would lead to a less natural solution than the one established now.

Edit devices screen



On tap of the edit button, the main devices screen evolves in the edit devices screen. Before each listed device appears the standard iOS delete icon.

Besides the red delete icon, a right arrow “>” appears to the right of the device name, indicating that tapping it takes the user to a new screen (the (individual) device edit screen).

At the bottom of the devices list, a new “list item” appears. A green add icon and the text “New device” indicate that the button should be tapped if the user wants to add a new device. Indeed, on tap, the add device screen appears.

The way that the edit button responses will seem natural to iOS users. A similar response can be found in the Clock and Music app.

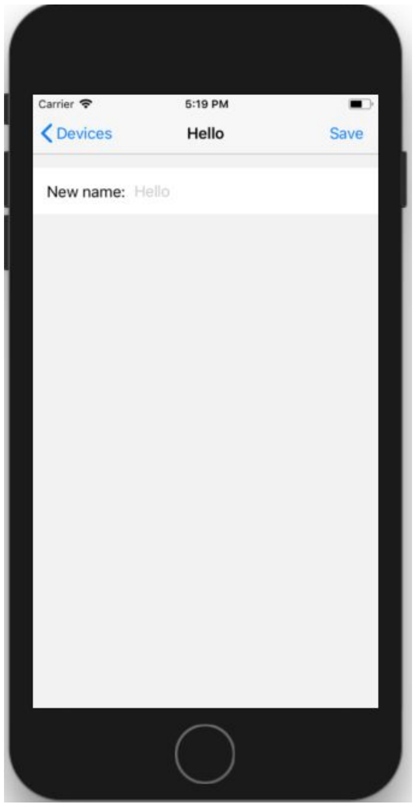
If the user taps the red deletion icon, a confirmation button slides in at the right side of the cell. If the taps this the device is deleted from the list.

Considered alternatives

We had the idea to implement the deletion confirmation as a pop-up window, but this turned out to be non-standard and very difficult to implement.

Device edit screen

This screen is entered when the user taps on the right arrow next to a device name in the edit devices screen. Here the user can change the name of the device. The user can confirm changes by pressing save in the top right corner, which lead the user back to the devices main screen.



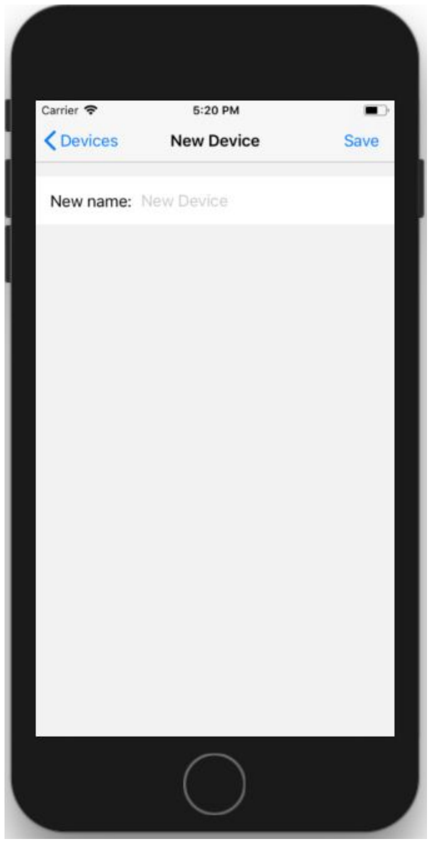
Add device screen - In Progress

The screen is entered on tapping New device in the edit devices screen. The user has to provide four properties for device to be added. First, he should tap the Manufacturer button, which leads the user to the Manufacturer list screen as described below.

The second button is the Device Type button, which behaves similarly to the Manufacturer button.

Then two text input fields follow, where the user can enter a name for the device and the IP address at which it can be found by the server.

The additions can be undone by tapping cancel, or saved by pressing save.



Front end architecture

We used a combination of the Storyboard designer and coding in C# using Xamarin.iOS to implement the front end.

Devices screens

The devices main screen shows a list of devices. A natural choice to implement this in Xamarin.iOS is a (UI)TableView. The TableView is dynamically filled with UITableViewCells, which contain the device name and on/off switch if present. This switch in placed in the AccessoryView. This is the place on the right of a cell, where normally for example a right arrow is shown to indicate that one can move to the next screen in a settings menu. The TableViewController (in which the TableView lives) is tied to a NavigationViewController to be able to show the Edit/Done button in the top right corner. Which button is shown depends on the mode of the TableView. If the TableView is in Editing mode, the button shows Done and Edit otherwise. The 'New Device' is also a UITableViewCell, which is shown if the TableView is in Editing mode. The actual behavior of the TableView is defined in a file that is subclassed from the UITableViewSource class. This class provides a number of method overrides that determine, for example, how a row behaves when selected.

The screen in which the user can edit the name of a device, is not a UITableView, but a combination of simple UIViews for the label and the input text field. This is because the amount of content of this screen is fixed.

Back end architecture

The backend consists of the import network handler and server interactor which do most of the lifting for connecting to the server.

The network handler and server interactor are separate classes. The network handler class is used for low level http requests. It has methods for GET, POST, PUT and DELETE requests. For handling the http requests we use a third party rest client package named RestSharp. Another option for doing http requests is with the build in HttpClient class.

We have chosen RestSharp over HttpClient, because HttpClient requires the user to work with asynchronous tasks. RestSharp doesn’t and therefore it’s easier to use. For sending and retrieving data we use JSON, since the Hestia server only accepts JSON. We use the Newtonsoft framework to parse JSON, because it’s an easy to use JSON framework and it comes with deserialization and serialization functionality.

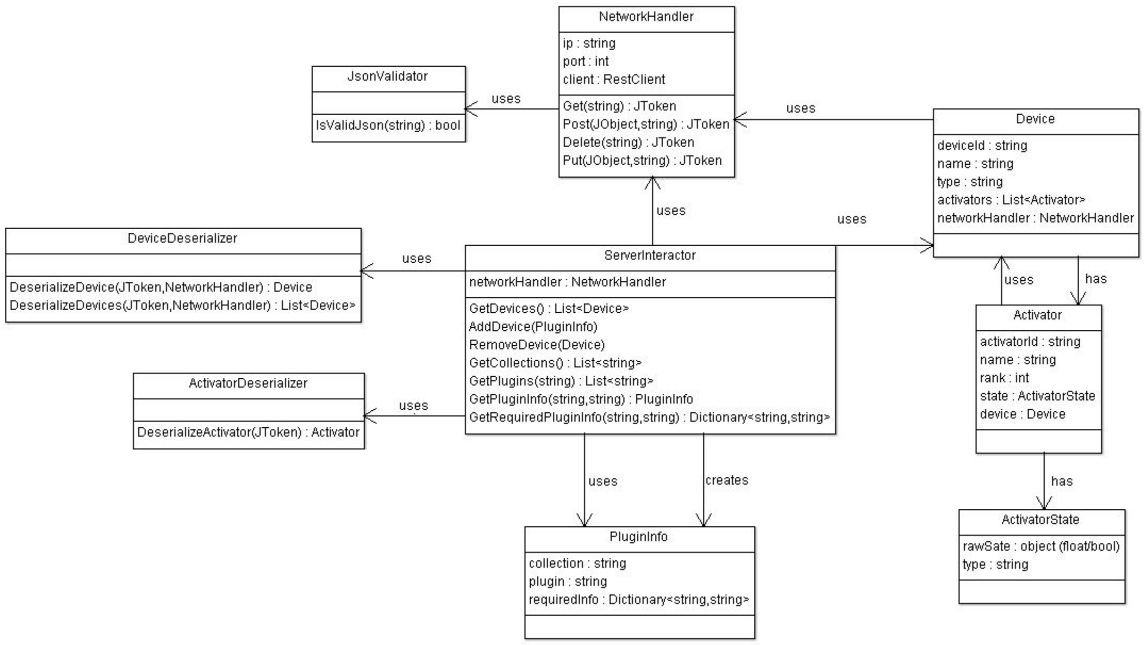
Next there is the server interactor class. The server interactor class depends on the network handler, because it uses its methods for executing http requests. The server interactor class contains all necessary methods for interacting with the server, e.g. getting a list of devices. The front end will use this class to interact with the server.

Besides the network handler and server interactor we have three packages: exceptions, models and utils. Utils is rather self explanatory it contains some essential utilities which have various appliances. Furthermore we have the exceptions class, currently we have but one exception; the Com exception, is thrown when the network handler fails on connection, however not yet implemented. Lastly we have the models package. The model package contains four important classes which define the models for a device and activator and how they should be read and written in JSON format.

When receiving JSON objects containing a Device, we use deserializers to turn them back into c# objects that the app can use. The ActivatorDeserializer turns the Activator object inside a Device object into an Activator first. The Activator type is read before making the Activator model, to make sure it has the functionality like an on or off switch or a floating point slider. Then the DeviceDeserializer deserializes a Device object and adds the Activator to the Device. The DeviceDeserializer also has the functionality to deserialize an array of Device models at once, which can be helpful when the list of Device models is required in the network handler.

Our network handler uses two things id and port. This will return the network handler object which can be passed to the Server interactor such that it will connect to the server that is described in the handler. This splits functionality over multiple classes such that it remains clean code. The interactor also can get devices, plugins and required info. The devices can also be changed.

The following diagram shows the interactions between all back end classes.


NetworkHandler
ip : string
port : int
client : RestClient
Get(string) : JToken
Post(JObject,string) : JToken
Delete(string) : JToken
Put(JObject,string) : JToken
JsonValidator
IsValidJson(string) : bool
uses
uses
Device
deviceId : string
name : string
type : string
activators : List<Activator>
networkHandler : NetworkHandler
uses
uses
DeviceDeserializer
DeserializeDevice(JToken,NetworkHandler) : Device
DeserializeDevices(JToken,NetworkHandler) : List<Device>
uses
ServerInteractor
networkHandler : NetworkHandler
GetDevices() : List<Device>
AddDevice(PluginInfo)
RemoveDevice(Device)
GetCollections() : List<string>
GetPlugins(string) : List<string>
GetPluginInfo(string,string) : PluginInfo
GetRequiredPluginInfo(string,string) : Dictionary<string,string>
uses
has
Activator
activatorId : string
name : string
rank : int
state : ActivatorState
device : Device
ActivatorDeserializer
DeserializeActivator(JToken) : Activator
uses
uses
creates
has
PluginInfo
collection : string
plugin : string
requiredInfo : Dictionary<string,string>
ActivatorState
rawSate : object (float/bool)
type : string

Team Organisation

When we started working we took from our meeting with teaching assistant and customer that it might be wise to work in a Scrum like fashion. Besides this we divided our group up into two departments: frontend and backend. This way we could efficiently work on our separate parts. However we were not meeting on a daily basis, we made it weekly basis and separate meetings during the week in which we worked together.

To keep track of our tasks we used Trello to pass them to specific people and help planning through the sprints.

Change Log

Who	**When**	**Section**	**What**
M. Fleurke	February 23, 2018	Document	Created document + headings

Z. Holwerda	March 1, 2018	Document	Modified the front page to match other document.
M. Fleurke	March 11, 2018	User Interface design	Section about device screens
M. Fleurke	March 12, 2018	User interface design	Adding more on device screens + screenshots
A.M Oanca	March 12, 2018	User interface Design	Added Settings description + screenshots
Z. Holwerda	March 13, 2018	Document	Changed version style, added a small general overview.
M. Fleurke	March 25, 2018	Front end architecture	Added architecture of devices screen
Z. Holwerda	March 27, 2018	Back end	Added overview of the app.
J. Kroeze	April 17, 2018	Back end	Added description of deserializers
A.M. Oanca	April 17, 2018	Front End	Changed Screenshot according to the application. Changed description for some of the subparagraphs.
M. Fleurke	April 17, 2018	Introduction	Wrote short introduction + small fixes and additions in other sections
G. Barkmeijer	April 17, 2018	Front end	Updated the section of the login screen.
Z. Holwerda	April 17, 2018	Team organisation and document.	Changed the style of the document slightly. Small team organisation.
D. Groot	April 17, 2018	Back end	Added info about the network handler and server interactor.
D. Groot	May 1, 2018	Back end	Added a class diagram for back end.