Architecture Document

Universal Switchboard API

version 7.0

Company: Culture Centre “De Klinker”, Winschoten

Customers: Patrick Binee (head IT) & Thomas Hoeksema (technical consultant)

Team: Team 5
Milton Antonis
Patrick Beuks
Carolien Braams
Barnabas Busa
Floris Cornel
Leon Kielstra

Who	When	Which Section	What
Dimitris Aktsoglou	19-3-2017	Title/Intro	Fixed some of the things suggested in feedback.
Barnabas Busa	21-3-2017	Overview of whole document	Updated front end part, fixing typos
Razvan Andrei Poinaru	21-3-2017	Overview of whole document	Updated back end part
Milton Antonis	21-3-2017	Architectural Overview	Added login functionality description.
Razvan Andrei Poinaru	28-3-2017	Backend Design	Replaced Connecter with Link.
Floris Cornel	31-3-2017	Architectural overview	Updated module implementation with Hooks and Actions
Milton Antonis	05-04-2017	Architectural overview	Added login sequence diagram
Barnabas Busa	05-04-2017	Front end	Update front end
Milton Antonis	13-05-2017	Database Design	Updated Database Design
Floris Cornel	14-05-2017	Overview of whole document	Feedback of document 4
Milton Antonis	18-05-2017	Database Design	Updated Database Design and resolved comments
Milton Antonis	18-05-2017	Login System	Added use case for logging in. Removed logging in steps from “Defining the connections”

Milton Antonis	26-05-2017	Database Design	Added extra table for UpdateConnection
Milton Antonis	28-05-2017	Database Design	Removed ORM diagram
Milton Antonis	28-05-2017	Defining the connections	Updated use case and included sequence diagram
Leon Kielstra	28-05-2017	Back-end Design	Updated the class diagram
Leon Kielstra	28-05-2017	Back-end Design	Rewritten the design and Updated usecases
Milton Antonis	03-06-2017	Back-end Design	Updated SSDs, some use cases.
Milton Antonis	03-06-2017	Introduction	Rewritten Introduction
Milton Antonis	03-06-2017	Back-end Design	Rewritten Use Cases
Milton Antonis	03-08-2017	Back-end Design	Added all sequence diagrams and restructured sections

Introduction

The product we are developing is a general purpose connector of API's. The specialized goal was to connect two specific API's : YesPlan and TicketMatic, which are used by our customer's company. However, since none of the two APIs support webhooks, we use Github and Trello APIs to test our core application. If YesPlan and TicketMatic decide to provide support for webhooks, the client can then be able to connect the two APIs.

The application has two main focuses. The first being that the connections between the API's can be defined and the other one being able to use these connection definitions to actually apply them to the connected APIs.

The front-end is more focused on being able to define the connections. The user has to be able to add a connection between one API and another API, specifying which fields should be connected.

The back-end focuses on storing these connections, and managing the connections between APIs. This means that the backend is be able to receive messages(webhooks) from external APIs and perform actions on external APIs.

Architectural overview

Back-end Design

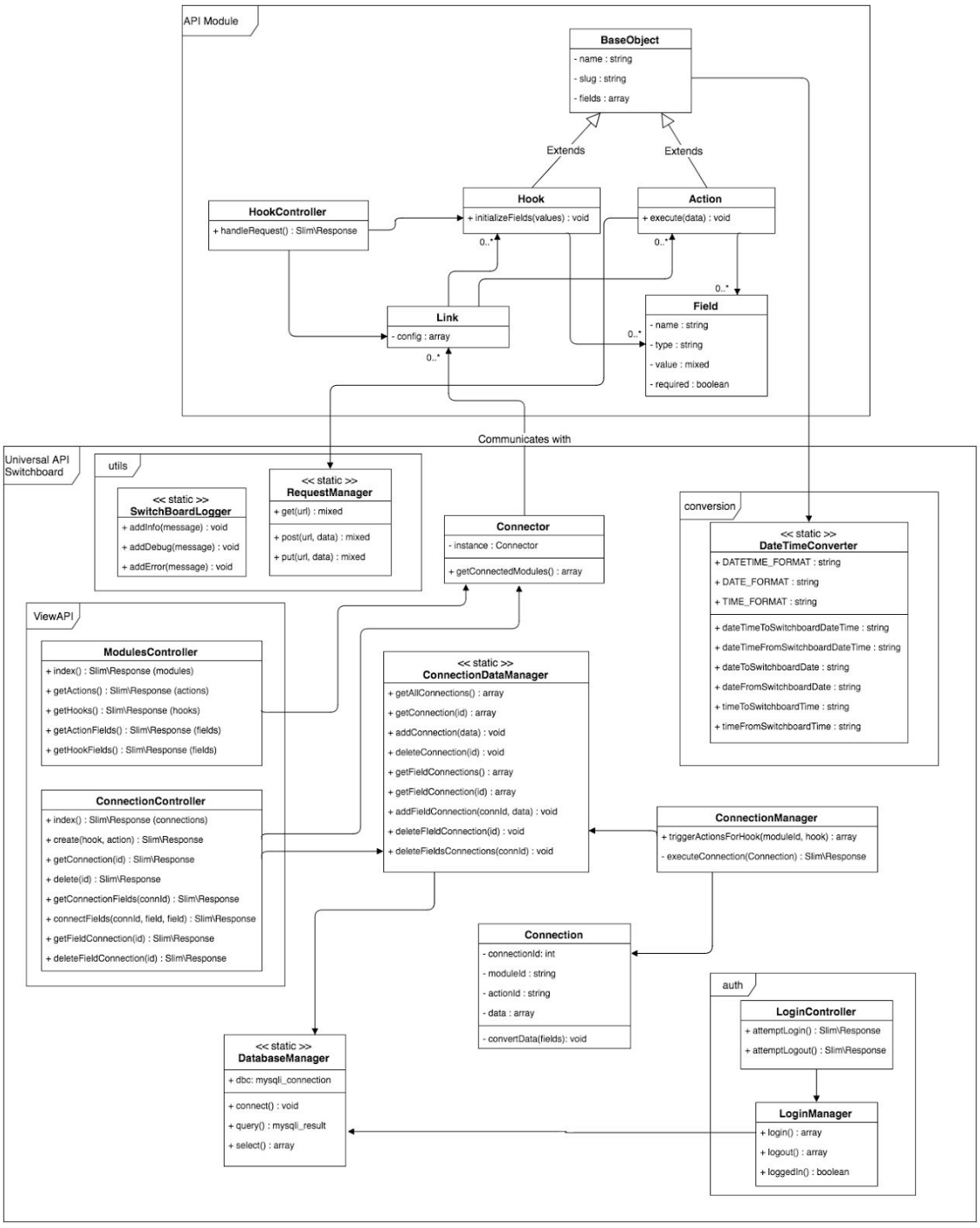


Figure 1: Classes and class interactions in the UAS and a generic API module.

Design explanation

Our design consists of our core application, the Universal API Switchboard, and a number of API Module's that have implementations for a specific API. The class diagram above only covers the most important fields and methods of each class. We left out the middleware that we are using.

In our back-end design we have created different types of classes. All **Controller** classes handle incoming requests, the **Manager** classes manipulate our model or have specific tasks like doing outgoing requests or handle the connection with the database. **Middleware** classes are invoked on every request to handle a specific task. Most of the other classes are part of the model.

Universal API Switchboard.

The Switchboard is essentially our main application. It has a **Connector** that holds a 'list' of all connected modules. This is done by storing all **Link** objects of connected modules in an array in the **Connector**. This class can be seen as the 'main class' of our application and since we want it to be accessible from every class and given the fact that there should only be one instance of we made it a singleton class.

For the connections we have created a **ConnectionManager** which creates **Connection** objects and uses that to execute a certain connection on an incoming hook. The **ConnectionDataManager** manipulates the actual connection data, hence it retrieves and deletes connections.

The **database** namespace

For the Switchboard we are using a MySQL database because of the support it has from PHP. The interaction between this database and our application goes through the **DatabaseManager**. The database holds the users and connections between hooks and actions and their fields.
In the database we have a table that stores connections between a hook and an action and a table that stores each connected field for a certain hook/action connection.

The **api** namespace

The Switchboard has a view API for the frontend which features APIs for all information of a modules and its hooks and actions through the **ModulesController**. It also contains a **ModuleConnectionsController** to provide and manipulate the connections between hooks and actions. **FieldConnectionsController** is used to manipulate the connections between the fields of a hook and an action.

The **auth** namespace

We have added authorization in the auth namespace which features the **LoginController** and **LoginManager**. Authentication is explained more thoroughly in the Login System section.

The **utils** namespace

This namespace contains **RequestManager** which can send HTTP requests to external APIs, **SwitchboardLogger** which is used as a logger, and **Validation** which is used to validate the integrity of connection data provided by the user and log any errors from incorrect data.

The **converters** namespace

Since different APIs have different conventions on how to store for example dates, we have added a conversion namespace. This holds the converters that convert the incoming data to an internal representation and convert it to another API's standard when executing an action.

The **middleware** namespace

This namespace contains classes that intercept requests to check for authentication and log incoming HTTP requests.

The **config** namespace

This namespace contains settings for the database host, the logger, and the environment (i.e. production, development or test).

API Module

Any module's connection to the Switchboard is done through the **Link**. The **Link** can communicate with the Switchboard's **Connector**. The **Link** has some properties regarding the API it has contact with(e.g. the name and the base url of the API).

A module contains two different types of interactions with an external system. **Hooks** are incoming requests from the external system. These are HTTP requests containing some data and are handled by the **HookController**. The other interactions are outgoing requests, called **Actions**, these requests result in an external API call to another system.

A module contains an array of zero or more **Hook** and **Action** classes. They all have a list of **Field** objects, which are the possible fields that can be used for a hook or action.

The **Action** requests will be performed by the **RequestManager**. Before doing the request the **Action** class may use one or more converter classes to convert it's data to the appropriate format.

For the **Link**, **Hook**, **Action** and **Field** classes we have created superclasses in the 'ModuleData' namespace. It is up to the user of the application to extend these classes for the API he/she wants to implement.

Defining the connections

All the connections that are created by our user are stored in the database.We only allow for connections between a hook of type CREATE and an action of type CREATE, and between a hook of type UPDATE and an action of type UPDATE. Below is an example on how interaction goes for our application:

Use Case: Creating a Connection

Preconditions:

1. The user should be logged in to our system.

Postconditions:

1. There should be a new hook/action connection in our database.
2. There should be 0 or more new field connections in our database.
3. The system should have responded whether the action is successful or not.

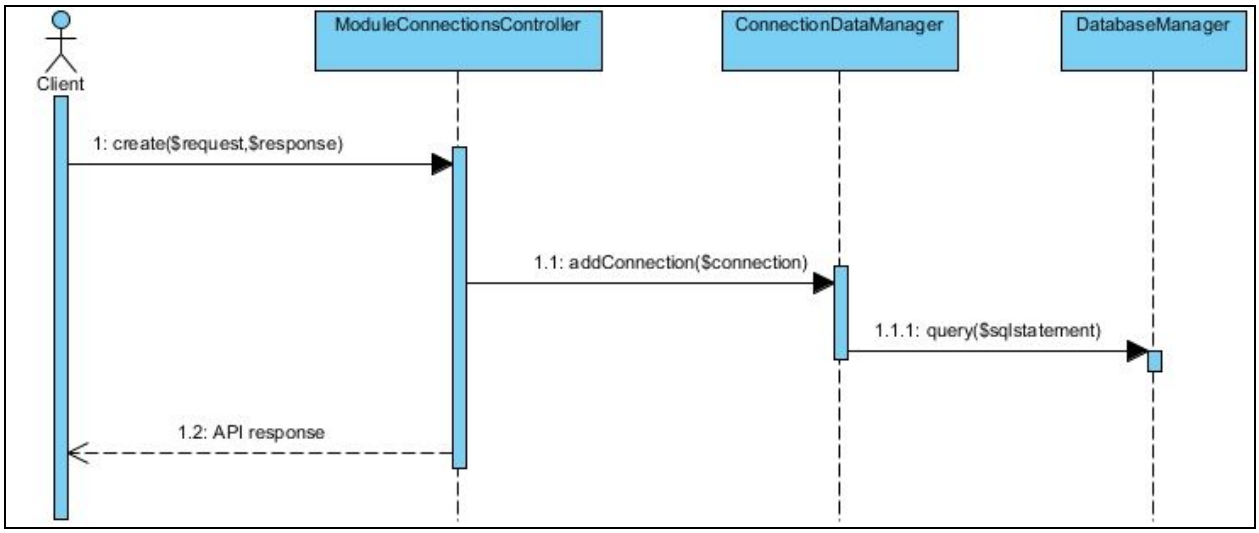
Main success scenario:

1. The front-end asks the back-end for a list of modules (User will pick two modules to connect, we assume a connection from A to B).
2. The front-end asks the back-end for a list of Hooks for module A, the user will pick one **Hook**.
3. The front-end asks the back-end for a list of **Fields** for the selected **Hook**.
4. The front-end asks the back-end for a list of Actions for module B, the user will pick one **Action**.
5. The front-end asks the back-end for a list of **Fields** for the selected **Action**.
6. The user connects fields from the selected **Hook** to fields from the selected **Action**.
7. The user sends the request for the new connection to be registered.
8. The frontend makes the API call to add a connection between a hook and an action for the user's choice.
9. **ModuleConnectionsController** receives the request.
10. **ModuleConnectionsController** verifies the integrity of the connection data.
11. **ModuleConnectionsController** asks **ConnectionDataManager** to store the new connection.
12. **ConnectionDataManager** stores the new connection between the **Hook** and the **Action**.

Loop 12-15 : for all new field connections defined :

13. The frontend makes the API call to add a connection between the two fields.
14. **FieldConnectionsController** verifies the integrity of the field connection's data.
15. **FIeldConnectionsController** asks **ConnectionDataManager** to store the new field connection.
16. **ConnectionDataManager** stores the connection between the field of the **Hook** and the field of the **Action.**

The way API calls work to add connections can be illustrated through the following diagram for adding a connection between a hook and an action.



Sequence Diagram : Adding a connection between a hook and an action.

Handling incoming webhooks

All the connections that are created by our user are stored in the database.We only allow for The following use case illustrates what happens when our system is receives an incoming webhook:

Use Case: Handling an incoming Hook

Preconditions:

1. The incoming request is authenticated.

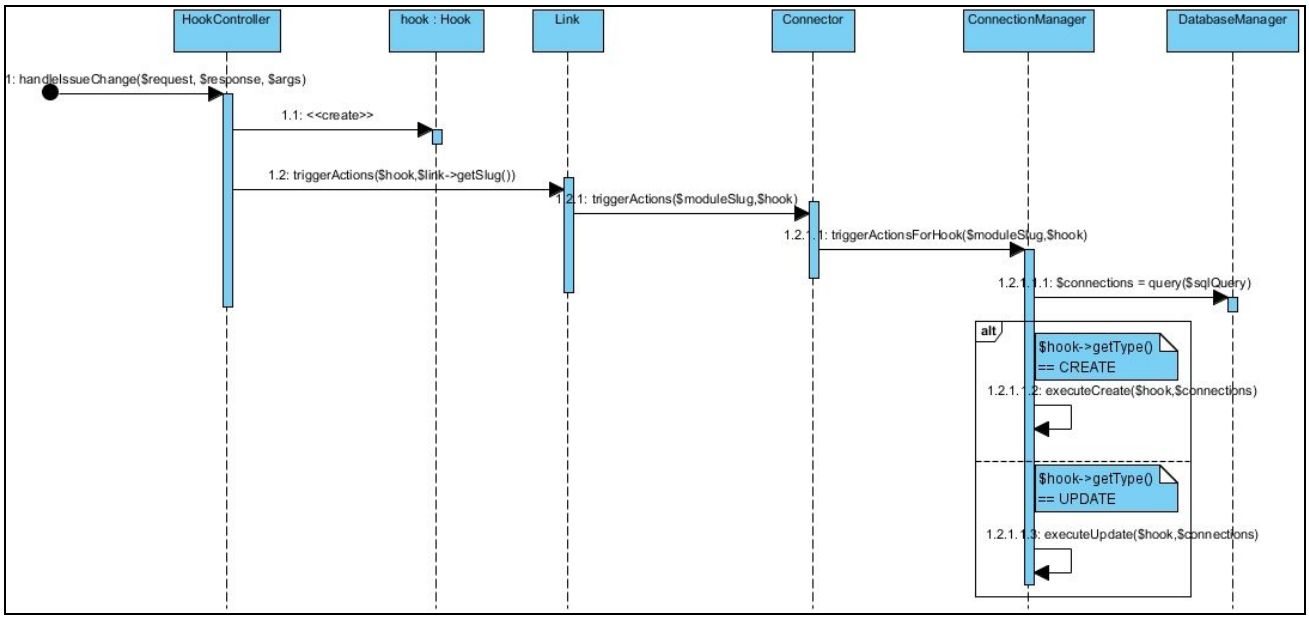
Postconditions:

1. The requested action (create or update) should be executed.
2. All connected actions are executed.
3. The system responded on whether it is successful or not.

Main success scenario:

1. The incoming requests is catched by **Slim**'s routes and a method in the corresponding **HookController** is executed.
2. The **HookController** creates a hook with the incoming data
3. **HookController** passes the **Hook** to **Link.**
4. **Link** passes the data to **Connector**.
5. **Connector** passes the data to **ConnectionManager**.
6. **ConnectionManager** asks the **DatabaseManager** for all **Connection**s related to this **Hook**.
7. IF the type of the hook is CREATE THEN **ConnectionManager** executes connections where hook and action are of type CREATE ELSE **ConnectionManager** executes connections where hook and action are of type UPDATE (see use cases below)

This use case can be illustrated with the following sequence diagram which shows what happens upon receiving a hook from Github:



Sequence Diagram: Handling an incoming hook from Github

Use Case : Executing Connections of type CREATE

Preconditions:

1. The hook and all the actions it is connected to are of type CREATE.

Postconditions:

1. The actions of these connections have been executed.
2. The combinations between the hook object's id and the objects' id's from the executed actions have been stored in the database, so that they can be referred to when the equivalent connections of type UPDATE are triggered.

Main Success Scenario:

1. **ConnectionManager** obtains the id of the hook's object (i.e. The id of the object from the external API which we will call **hookObjectId**).

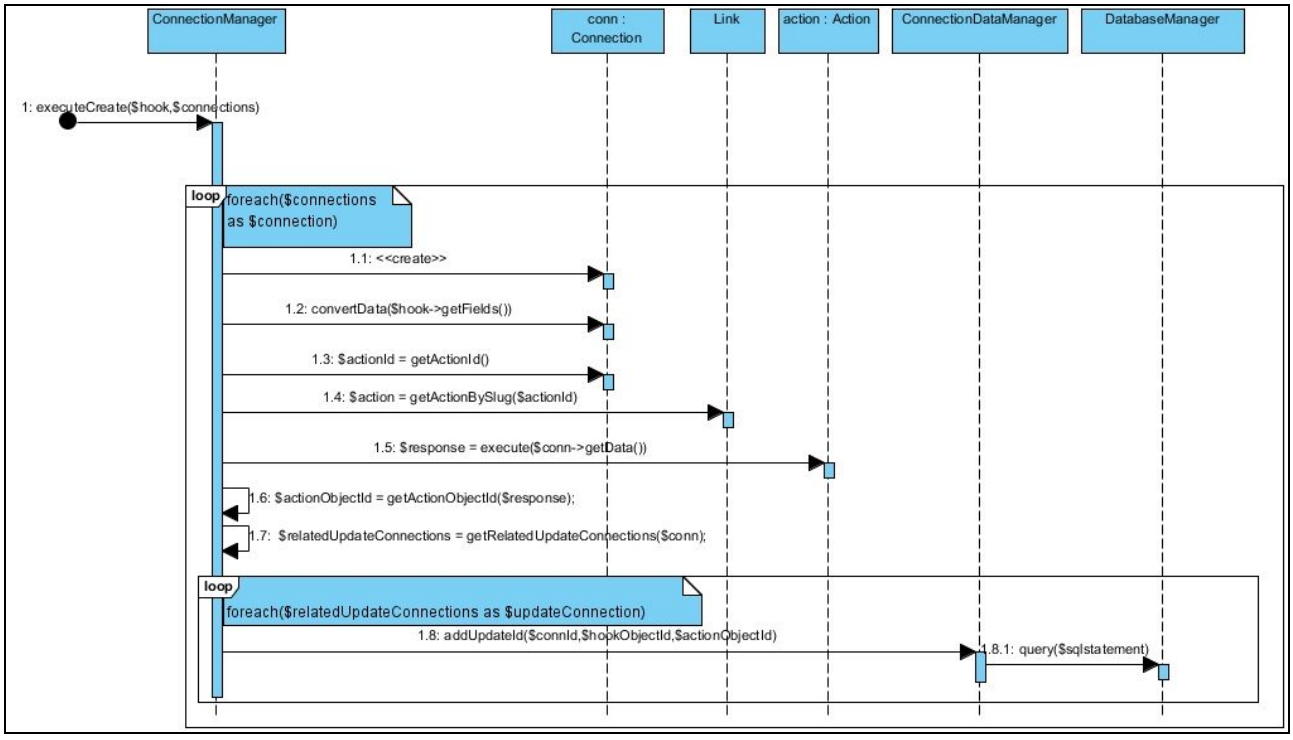
Loo 2-8: for all connections rows related to this hook:

2. **ConnectionManager** creates a new **Connection** object from the row.
3. **ConnectionManager** converts the field names of the **Connection** to the ones of the **Action** in this **Connection.**
4. **ConnectionManager** executes the **Action** contained in this **Connection.** (See use case "Executing an Action" below)
5. **ConnectionManager** receives the response received from executing the **Action**.
6. **ConnectionManager** grabs the value of the **id** field which we call the **actionObjectId** from the response.
7. **ConnectionManager** gets all connections of type UPDATE related to this **Connection.**

*Loop 8 : for all connections of type UPDATE related to this **Connection***

8. **ConnectionManager** uses **ConnectionDataManager** to store the connection between the **hookObjectId** and the **actionObjectId**.

This use case can be illustrated by the following sequence diagram:



Sequence Diagram: Executing a CREATE connection.

Use Case : Executing Connections of type UPDATE

Preconditions:

1. The hook and all the actions it is connected to are of type UPDATE.

Postconditions:

1. The actions of these connections have been executed.

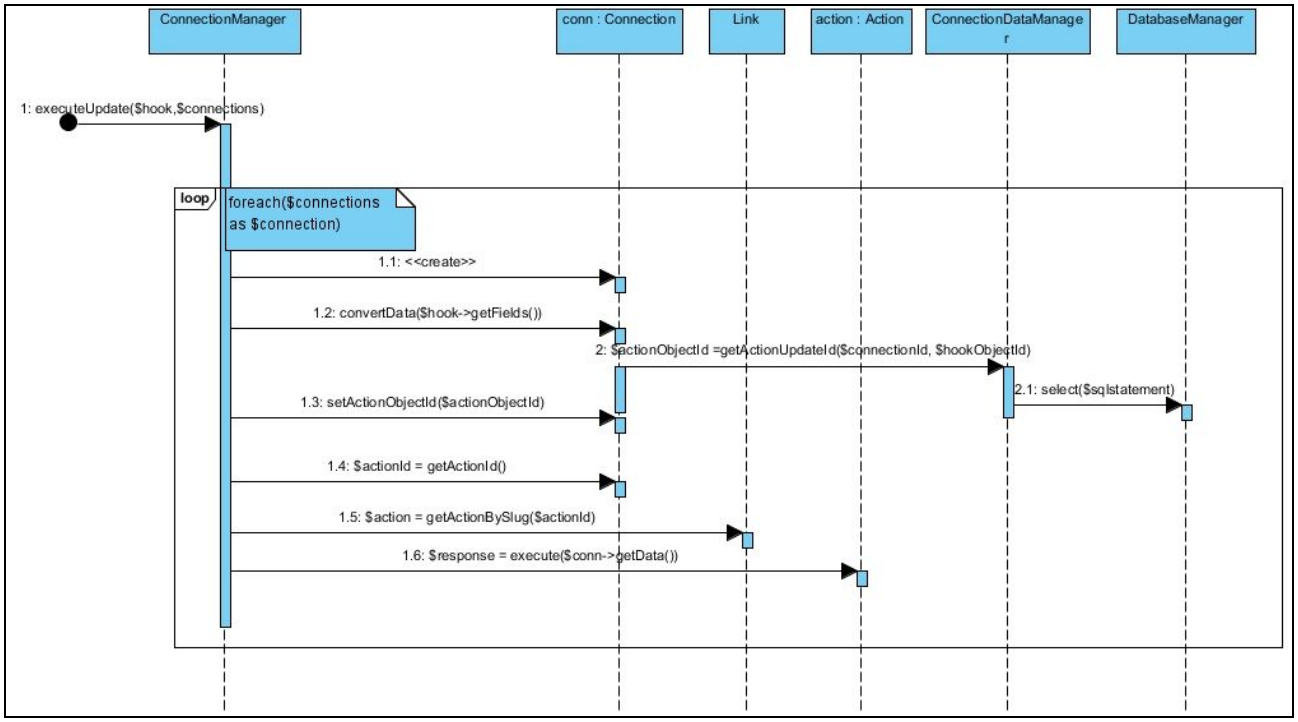
Main Success Scenario:

1. **ConnectionManager** obtains the hook's **hookObjectId**.

Loop 2-6: for all connections rows related to this hook:

2. **ConnectionManager** creates a new **Connection** object from the row.
3. **ConnectionManager** converts the field names of the **Connection** to the ones of the **Action** in this **Connection.**
4. **ConnectionManager** uses **ConnectionDataManager** to get the **actionObjectId** for the **Action** of this **Connection.**
5. **ConnectionManager** adds the **actionObjectId** to the parameters passed to the **Action.**
6. **ConnectionManager** executes the **Action** contained in this **Connection.** (See use case "Executing an Action" below).

This use case can be illustrated by the following sequence diagram:



Sequence Diagram: Executing an UPDATE connection.

Use Case : Executing an Action

Preconditions:

1. The array of parameters passed to the **Action** is filled correctly.

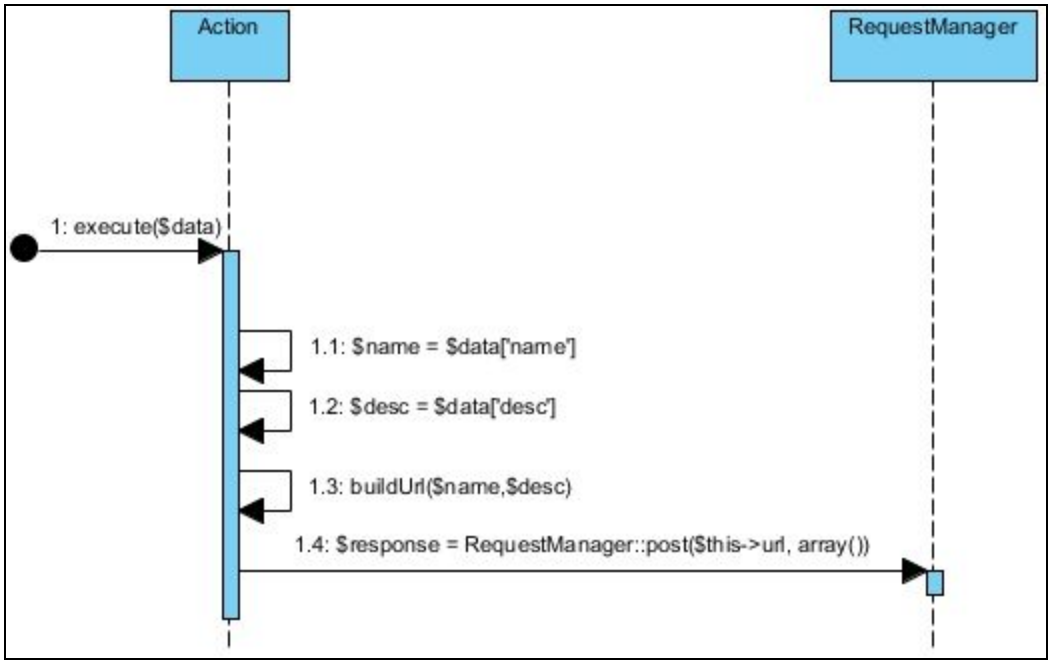
Postconditions:

1. The **Action** has been executed.
2. The **Action** has returned the response received from the external API call it has made.

Main Success Scenario:

1. The **Action** grabs the data to make the external API call it has been provided with.
2. The **Action** formulates an HTTP request to its external API.
3. The **Action** uses **RequestManager** to execute the formulated request.
4. The **Action** returns the response received from **RequestManager.**

This use case is illustrated by the following sequence diagram which shows what happens when the action creates a new Trello card:



Sequence Diagram: Executing an action

Database Design

The database model presented here does not show the authentication for the Switchboard account since the application has only one user. The design represents how the connections between fields of APIs will be stored.

The first and foremost connections to be stored in the database are those between hooks and actions. Hooks and Actions are identified by an id which corresponds to their name, and their corresponding API. The next connections to be stored are those between the fields of a hook and the fields of an action. These are stored in relation to the connection between the hook and the action.

We store the connections between hooks and actions in the following table:

HookActionConnection

id	**connectionName**	**hookModuleId**	**hookId**	**actionModuleId**	**actionId**
1	Create card when issue is created	github	newIssue	trello	newCard
2	Update card	github	issueChange	trello	updateCard

We store the connections between the fields of a hook and the fields of an action in the following table:

FieldConnection

id	**connectionId**	**hookField**	**actionField**
1	2	title	description
2	1	body	title

Here **connectionId** references **id** in HookActionConnection.

To know where updates should be made we introduce the table:
UpdateIds

id	connectionId	hookObjectId	actionObjectId
1	2	53454563	1cDelTam4
2	2	929284762	0xRW4zMw

connectionId again references **id** in HookActionConnection. The first entry tells us that whenever github issue with id 53454563 is updated, trello card with card id 1cDelTam4 will be updated.

Login System

The login system consists of two classes: **LoginManager** and **LoginController.**
The difference between the **LoginManager** and **LoginController** is that the Manager has to handle all the logic that comes with logging in. The Controller is used to answer requests that have to do with logging in and out over our API.

The helper class **DatabaseManager** is also used to perform database operations.
LoginManager queries the database for a given username and password in order to login a user. The **LoginController** uses the **LoginManager** to obtain the result of logging in the user. According to the result given by the **LoginManager,** the **LoginController** responds with an HTTP status and a JSON body. In case of a successful login the JSON body contains the session id of the session that was started when the user was successfully logged in. The session id can then be sent in headers for all subsequent requests that need authorization. The JSON body also contains a message to indicate what went wrong, or if the login was successful.

Logging out, is straightforward since, **LoginManager** destroys the session it initiated. In case the user was not logged in, nothing happens.

The use case for logging in is the following:

Use Case : Logging in

Preconditions: None

Postconditions:

1. The user is logged in the system.
2. The user has obtained session id that can be used for authorized requests to the system.

Main success scenario:

1.The user provides the username and password combination.

2.The system checks if there is a combination of given username and password in the database. (we assume there is for this use case)

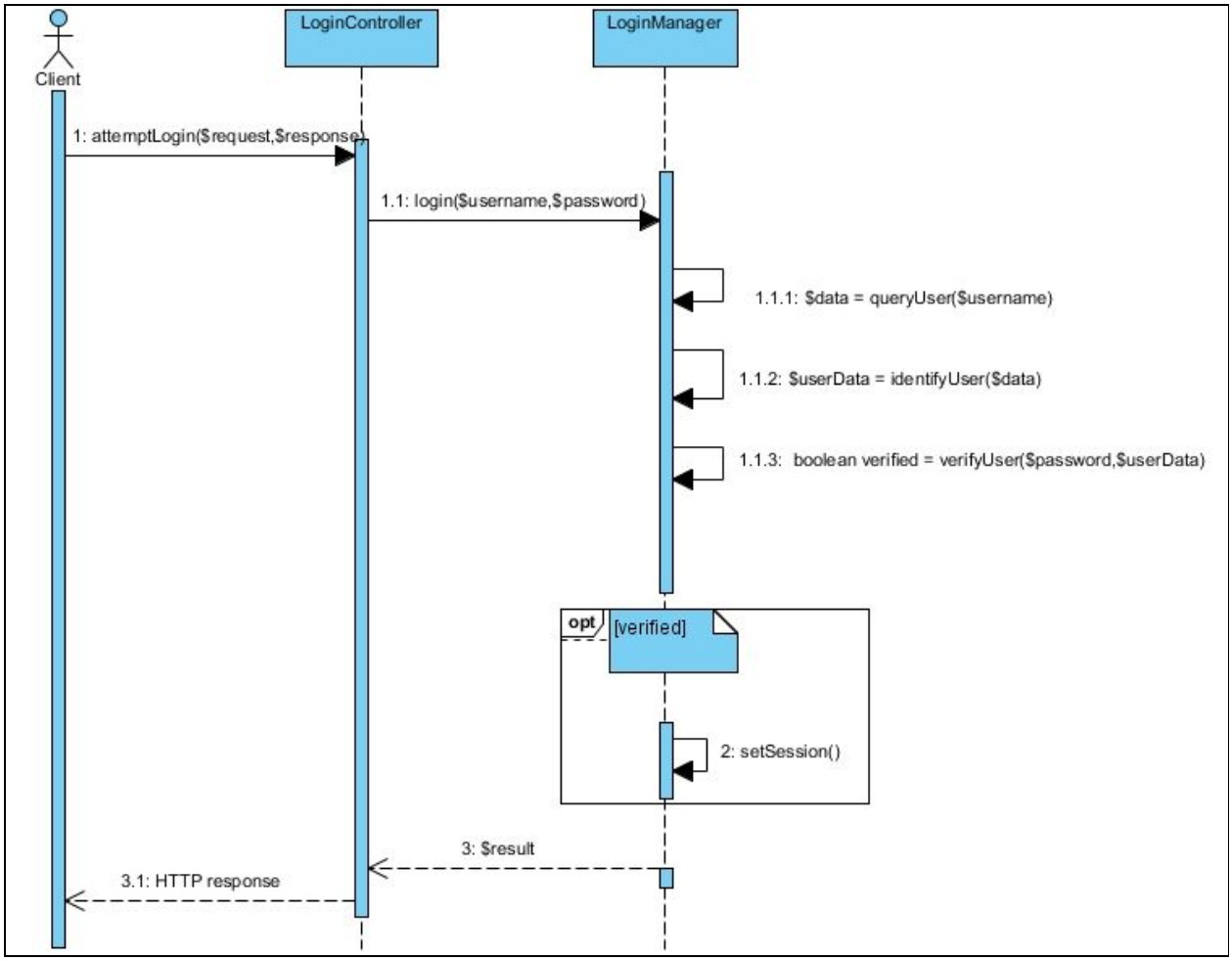
2.The system starts a new PHP session.

3.The system returns a message in JSON where the session id of the initiated session is included.

4.The frontend grabs the session id.

From then on every authorized request requires that the session id is also sent.

The login procedure is illustrated by the following sequence diagram:



The Login process

Route Authentication

Access to certain routes should be restricted unless the user is logged in. This is done by using adding a Slim Middleware to the application. The middleware used is called **AuthenticationMiddleware** which intercepts the required routes by checking for authentication. This process involves calling the function that checks if the user is logged in **LoginManager** and forbidding access to the route if the user is not logged in.

Front-end Design

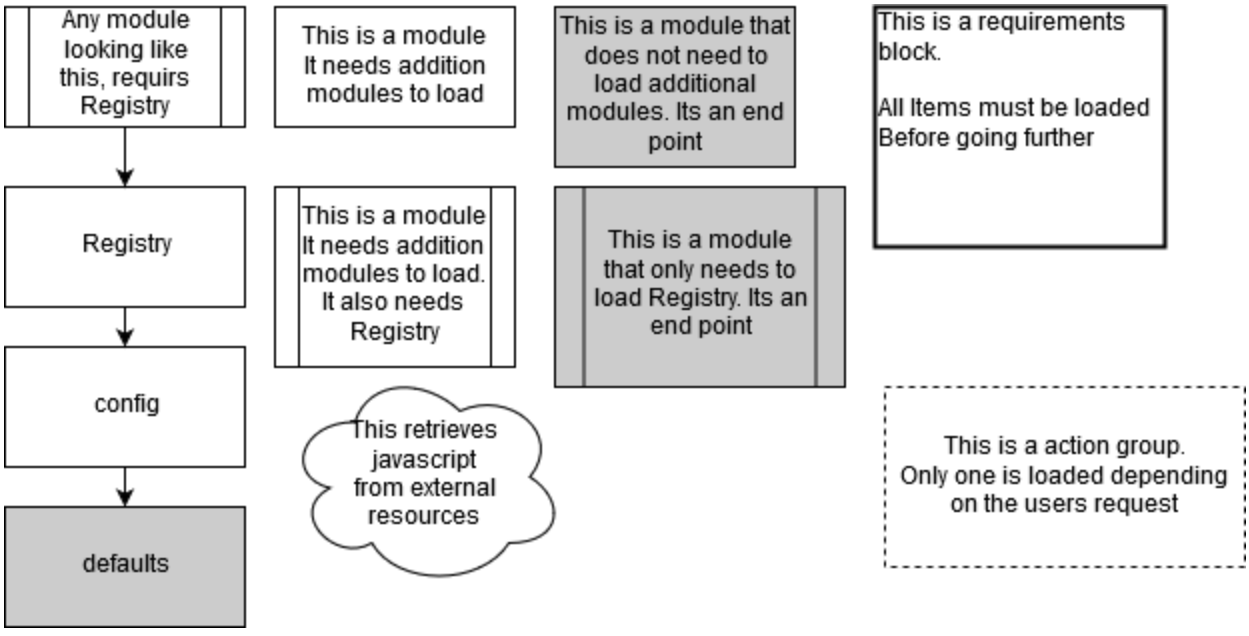
The front end is a webpage that will be accessible by the client to be able to edit connections between API's from other applications.

The front end is built on AngularJS. This is a framework that makes creating dynamic web-pages require little effort.

Below here is a diagram on how the front-end require stack is loaded:

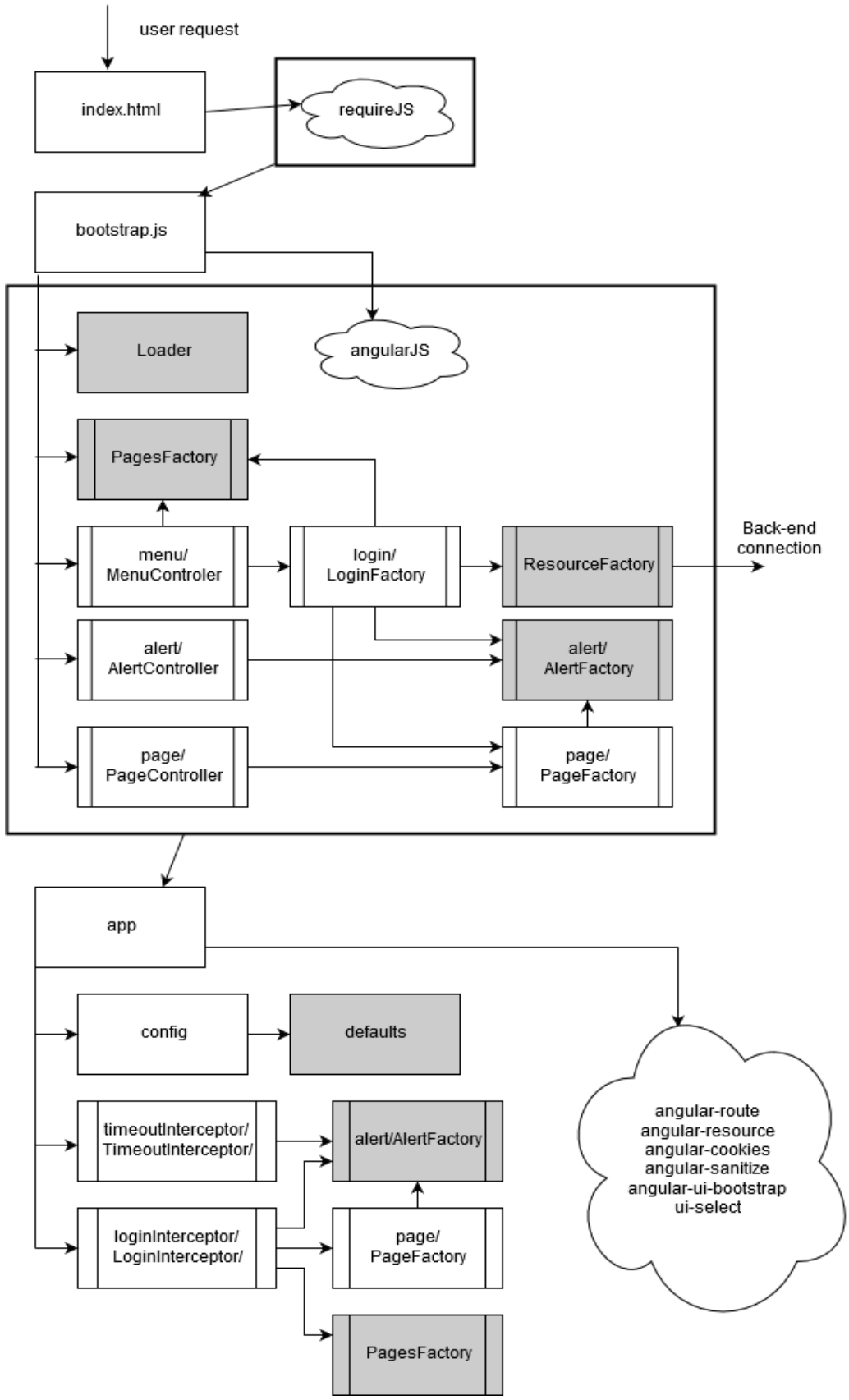
Legenda

Below here is the different symbols we use to show how the connections are loaded
The javascripts are loaded as a stack. This means that it opens the file, looks what dependencies this has, opens them and continues until a file without dependencies. It then executes the code in reverse order. This way this code is available for the file that depend on it. The only exception to this order is for the blocks with a thick border. It will first load the code inside that before continuing.



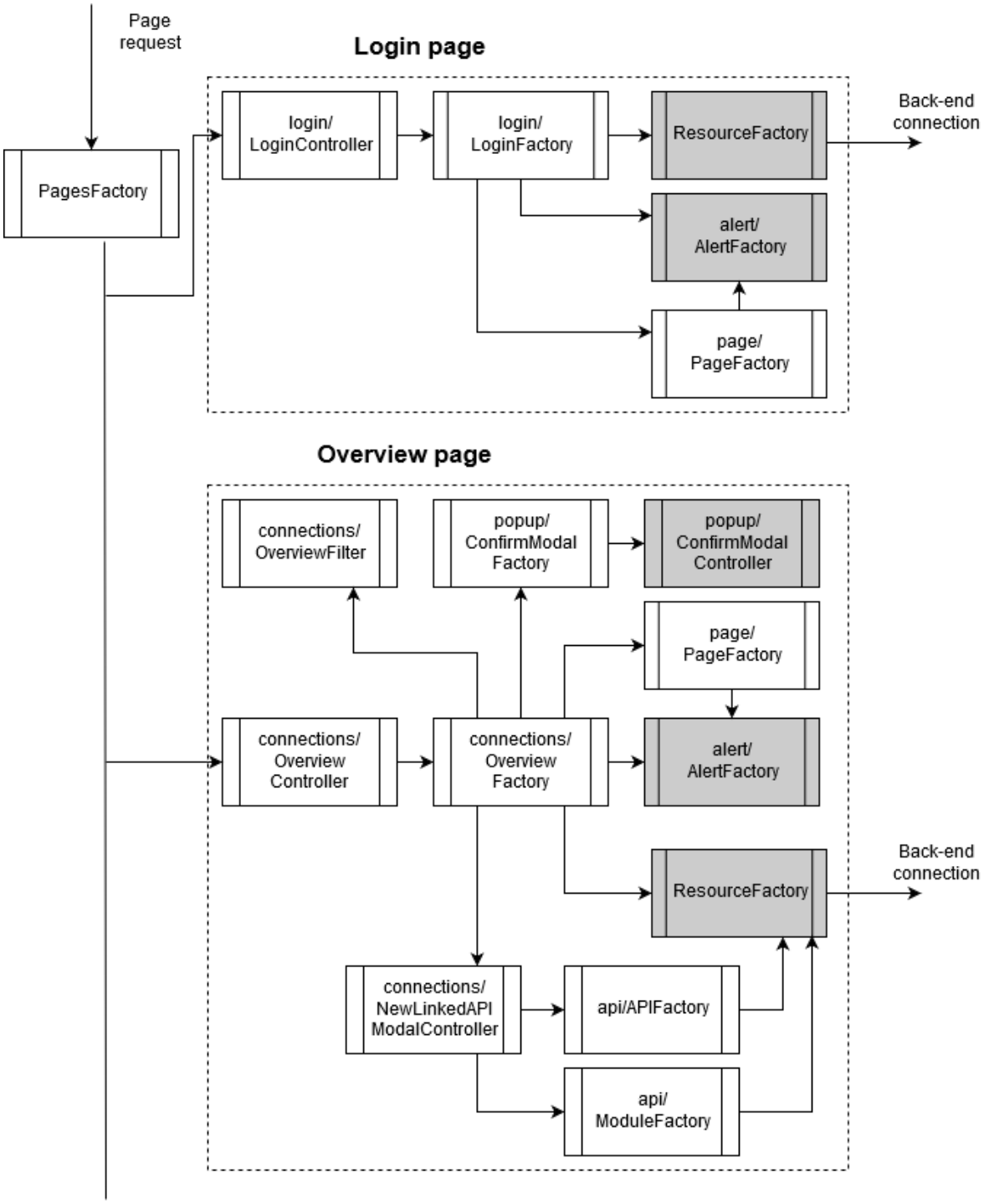
Initialization

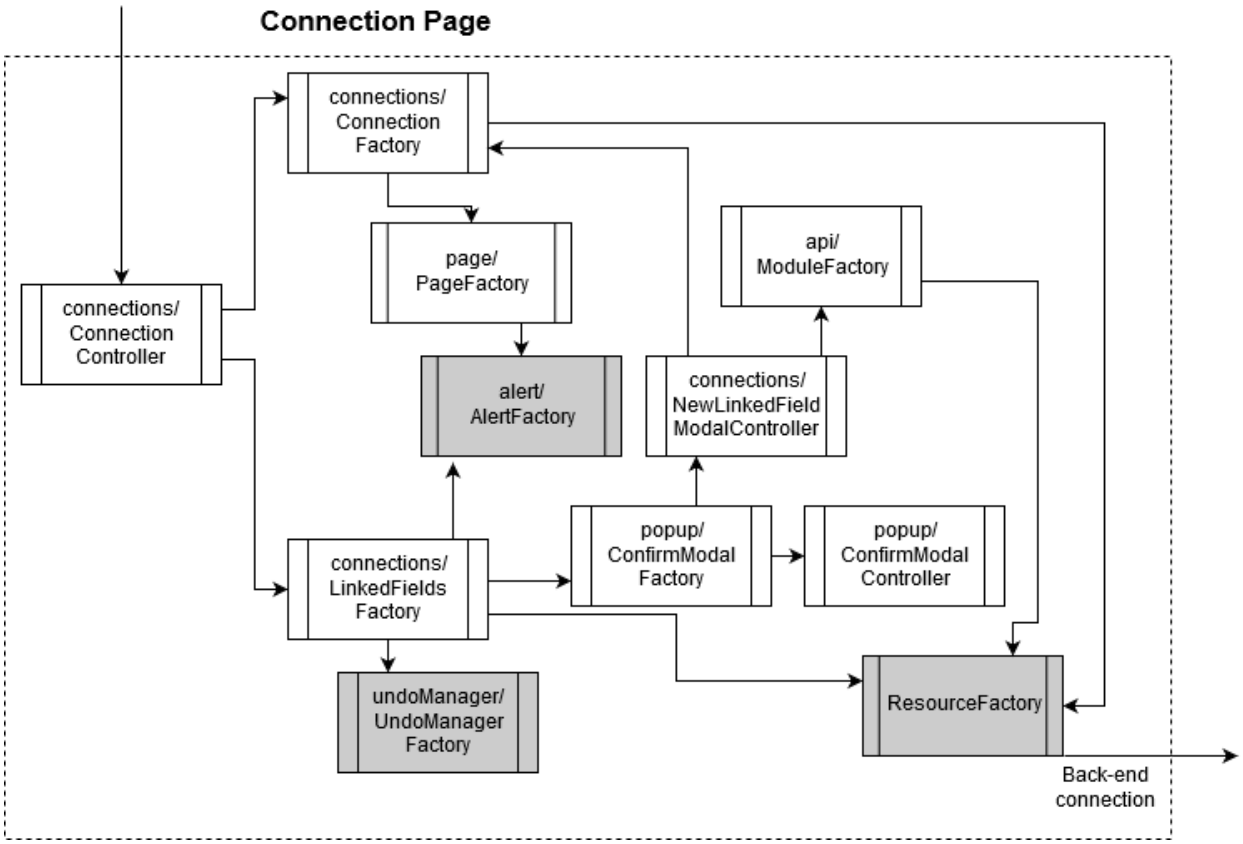
This code is always included. This has for example the menu bar, the page controller to switch pages and change the title, and registries and loaders to load everything correctly.


user request
index.html
requireJS
bootstrap.js
Loader
angularJS
PagesFactory
menu/
MenuControler
login/
LoginFactory
ResourceFactory
Back-end
connection
alert/
AlertController
alert/
AlertFactory
page/
PageController
page/
PageFactory
app
config
defaults
timeoutInterceptor/
TimeoutInterceptor/
alert/AlertFactory
loginInterceptor/
LoginInterceptor/
page/
PageFactory
PagesFactory
angular-route
angular-resource
angular-cookies
angular-sanitize
angular-ui-bootstrap
ui-select

Page request

To split up the code, we use different pages that do one task on their own. This way only a part of the code is loaded per time and make the code more manageable. Beside the javascript that is loaded with these page request that you can see below. Another part that is loaded with this is the html for the page and the css.




Connection Page
connections/
Connection
Factory
page/
PageFactory
api/
ModuleFactory
connections/
Connection
Controller
alert/
AlertFactory
connections/
NewLinkedField
ModalController
popup/
ConfirmModal
Factory
popup/
ConfirmModal
Controller
connections/
LinkedFields
Factory
undoManager/
UndoManager
Factory
ResourceFactory
Back-end
connection

Technology Stack

For the backend the following languages and technologies are used:

- PHP with Slim 3 Framework
- MySQL database
- Phinx
- PHPUnit
- Travis CI

Instructions to setting up your environment can be found here.

The front-end is using the following technologies and languages:

- Dgeni
    - Dgeni
        - Used for creating a documentation from the specified comments in the code.
    - Dgeni-packages
        - Default packages you can use for creating dgeni documents.
- HTML
        - Used by browsers as information to display.
- CSS
        - Works with HTML to style the information displayed.
- Javascript
        - A client side scripting language used by browsers.
- angularJS
        - AngularJS is build on top of javascript.
    - AngularJS
        - The core module of angular that contains most functionality.
    - Angular-mocks
        - A module to test angular without having a webpage open.
    - Angular-resource
        - A module for Angular that allows for retrieving remote content.
    - Angular-route
        - A module for Angular that allows for rewriting urls so that specific content is loaded on certain url's.
    - Angular-sanitize
        - A module with filters for angular
    - Angular-ui-bootstrap
        - A combination of CSS, HTML and JavaScript. This is a library that contains a bunch of pre formatted HTML that are compatible with angular.
    - UI Select
        - A module for angular UI that gives an easy searchable dropdown select

- Babel-core
    - A module with a few functionalities needed for dgeni to work
- Chai
    - A assertion module for testing used by mocha
    - Chai
    - Chai-as-promised
- Browser-sync
    - A easy browser server we use to display the generated documentation
- Canonical-path
    - A module to convert paths to the right addresses
- Del
    - Used to delete generated documents before creating them again, for a clean slate
- Eslint
    - A tool to verify the document structure and style
    - Eslint
    - Eslint-plugin-angular
- Jsdom
    - A nodeJS module that can simulate a document. Used for testing.
- Mocha
    - A NodeJS testing module.
- requireJS
    - A tools that allows for dynamically loading javascript files.
- Sinon
    - A testing tool for spying on data in javascript object. Used for testing.
    - Sinon
    - Sinon-chai
- Gulp
    - A module that allows you to run nodeJS scripts
- Lodash
    - A nodeJS module for utility functions
- nodeJS
    - A program that allows for running javascript on the computer instead of only in the browser.
- Npm
    - A package library installer for nodeJS modules.
- Travis CI
    - Continuous integration tool to test our pull requests

Team Organization

There are two teams: back-end and front-end team.
The back-end team is responsible for the incoming and outgoing HTTP requests and the database.
The front-end team is responsible for the user interface and sending user input to the back-end application.

API Documentation

The back-end documentation can be found here
The front-end documentation can be found here.