**CS673 Software Engineering** 

**Team 3 - ZicZac**

**Software Design Document**

|  |  |  |  |
| --- | --- | --- | --- |
| Team Member | Role(s) | Signature | Date |
| Elijah Curme | Team Leader | *EC* |  |
| Dinara Tiyekbayeva | Configuration.Backup Leader | *DT* | 3/27/21 |
| Chenghao Feng | Security Leader | *CF* |  |
| Pelin Akbiyik | QA Leader | *PA* | 3/21/21 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Revision history**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Author** | **Date** | **Change** |
|  |  |  |  |
|  |  |  |  |

[Introduction](#_87t9hln2vjz0)

[Software Architecture](#_buttcq9i221r)

[Design Patterns](#_x18fj36s1121)

[Key Algorithms](#_mtfbusfb0eq3)

[Classes and Methods](#_7ucksmkf6rzx)

[References](#_15tmymhipvdv)

[Glossary](#_8n34lvocupub)

# Introduction

This document provides details of software design for the Ziczac web application. Ziczac is a full stack application built on modern frameworks to allow fast coding and implementation.

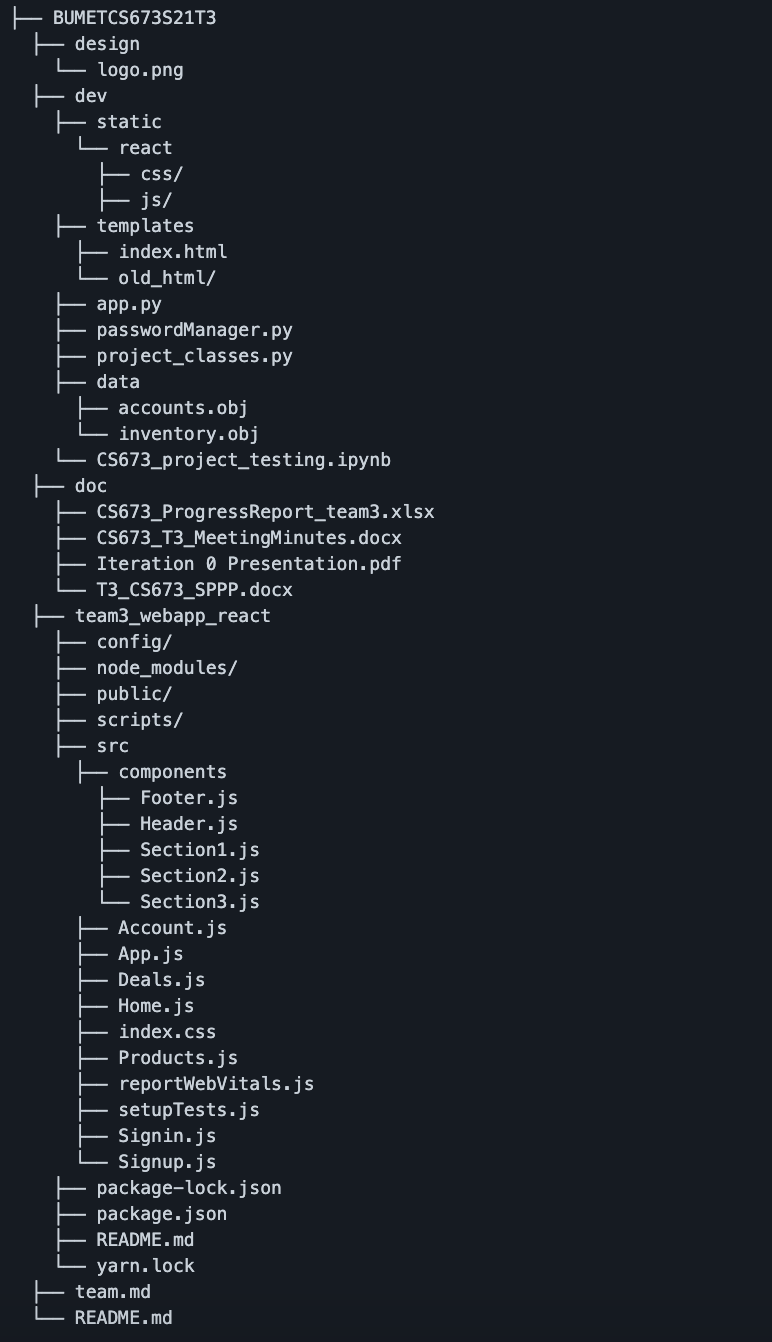
The Front end of the app is built on React framework. All communications with backend are performed via different REST API endpoints.

The Back end for the app is built as a REST API. It provides different endpoints to perform various actions such as posting an item, adding new users, filter products by category, etc. The REST API is built using Flask. CORS is enabled for the API so it only accepts requests form the React front end.

Data Layer in this application implemented by using SQLite relational database to persistently store data. It is stable, cross-platform, and backwards compatible. It implements a small, fast, self-contained, high-reliability, full-featured SQL database engine. The Flask API backend will be performing all DB operations connecting to the DB and getting requests from the frontend.

# Software Architecture

The home page is composed of 3 primary section components and is bookended by header and footer components. Each React page includes a function that handles user input and returns html that is displayed by the browser. Backend tasks such as account verification or product filtering are performed by Flask, which cross-references application data in the data folder and returns results to React in JSON format. The project directory structure is shown in the below figure.



The static files generated from the React Production build combined with backend files will be stored on main github repository, from there they will be pushed to Heroku github and deployed on heroku. The domain is <https://ziczac3.herokuapp.com/>.

# Database Design (if applied)

We use SQLite relational database in our application that will hold persistent data. We have following schema

|  |
| --- |
| INVENTORY |
| title STRING (30) |
| price NUMERIC |
| description TEXT |
| category STRING |
| date\_added DATETIME |
| seller STRING |
| state STRING |
| photo BLOB |

|  |
| --- |
| ACCOUNTS |
| username STRING (30) |
| email STRING (30) |
| salt STRING |
| password\_enc STRING |

|  |
| --- |
| {username}\_INBOX |
| message\_id INTEGER PK |
| sender STRING |
| body TEXT |
| date DATETIME |

|  |
| --- |
| {username}\_OUTBOX |
| message\_id INTEGER PK |
| sender STRING |
| body TEXT |
| date DATETIME |

The photos were previously stored in the public folder, as this allowed the browser to access files in the project repository. Upon implementation of the “post a product” feature, it became apparent that all item data needed to be stored together in the database, as users should not be able to modify the project directory other than the database.

Recently the database schema have been altered in preparation for the private message system. When an account is created, tables for the account’s inbox and outbox table are also created. This way, when a user goes to message the seller about an item, the system has tables for the seller’s inbox, and the buyer’s outbox, in which to store the message.

# Security Design

User’s account password follow the password standard such as minimum of eight upper- and lowercase alphanumeric characters, must include at least one special character (\*, &, $, #, !, or @).

Hash algorithms adopt SHA-3 and pbkdf2. Also add other features like times of hash into the secret.

Additionally, using salt to help encrypt the password while signing up the account and checking the database.

# Design Patterns

In ReactJS we mainly use “Container components” design pattern. It is a design pattern that has logic to set state or have functions to emit events up to a parent component. Usually the general rule of thumb is to keep component as simple as possible with a Single Responsibility Principle design principle in mind, which essentially means that a component must do one thing. By following this design pattern we reduce overall application complexity. So we essentially use components as building blocks which sometimes are presentation and have no functionality while other times a component has a functionality which does one thing. For example, one of our app components “CardCategory” filters product per category and has only that one functionality.

In Flask we tend to use Class Adapter design pattern. Class adapter uses inheritance instead of composition. This design pattern allows us to use the interface of an existing class as the interface for another class. For example, in our flask code we have class Item, we then inherit that class in a different class and perform some operations.

# Key Algorithms

In our web application we are not inventing or manually writing algorithm solutions.

Instead of reinventing the wheel we are using ready-to-use solutions such as frameworks, external libraries, etc. For example, product sorting uses python built-in sort function.

socket.IO is used in our chat system. Sockets have traditionally been the solution around which most real-time chat systems are architected, providing a bi-directional communication channel between a client and a server.This means that the server can *push* messages to clients. Whenever you write a chat message, the idea is that the server will get it and push it to all other connected clients. Socket.IO is composed of two parts:

1.A server that integrates with (or mounts on) the Node.JS HTTP Server [socket.io](https://github.com/socketio/socket.io).

2.A client library that loads on the browser side [socket.io-client](https://github.com/socketio/socket.io-client)

The main idea behind Socket.IO is that you can send and receive any events you want, with any data you want. Any objects that can be encoded as JSON will do, and [binary data](https://socket.io/blog/introducing-socket-io-1-0/#binary) is supported too.

# UI Design

Our goal was to make UI as easy as possible for users to interact with our application.

For designing User Interface we were using user stories from which we developed visual representations in justinmind wireframing tool.

Please see wireframing diagrams for complete understanding of UI used in this application.

# Classes and Methods

For the application API document we use **Flask-RESTful. It** is an extension for **Flask** that adds support for quickly building REST APIs. It is a lightweight abstraction that works with your existing ORM/libraries. **Flask-RESTful** encourages best practices with minimal setup. We are also using Marshmallow Flask plugin. **Marshmallow** is an ORM/ODM/framework-agnostic library for converting complex datatypes, such as objects, to and from native Python datatypes. It is accessed with /swagger (Json) and /swagger-ui (visualized Json) URI once hosted on heroku.

For reactJS we haven’t decided yet but looking into react-docgen library.

# References

<https://en.wikipedia.org/API>

<https://en.wikipedia.org/wiki/Representational_state_transfer>

<https://en.wikipedia.org/wiki/Adapter_pattern>

<https://swagger.io/>

<https://www.npmjs.com/package/react-docgen>

# Glossary

API - **application programming interface** (**API**) is an [interface](https://en.wikipedia.org/wiki/Interface_(computing)) that defines interactions between multiple [software applications](https://en.wikipedia.org/wiki/Software_application) or mixed [hardware](https://en.wikipedia.org/wiki/Computer_hardware)-software intermediaries.It defines the kinds of [calls](https://en.wikipedia.org/wiki/System_call) or requests that can be made, how to make them, the [data formats](https://en.wikipedia.org/wiki/Data_type) that should be used, the conventions to follow, etc. It can also provide extension mechanisms so that users can extend existing functionality in various ways and to varying degrees. An API can be entirely custom, specific to a component, or designed based on an industry-standard to ensure [interoperability](https://en.wikipedia.org/wiki/Interoperability). Through [information hiding](https://en.wikipedia.org/wiki/Information_hiding), APIs enable [modular programming](https://en.wikipedia.org/wiki/Modular_programming), allowing users to use the interface independently of the implementation.

REST - **Representational state transfer** (**REST**) is a [software architectural style](https://en.wikipedia.org/wiki/Software_architecture) which uses a subset of [HTTP](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol). It is commonly used to create [interactive](https://en.wikipedia.org/wiki/Interactivity) applications that use [Web services](https://en.wikipedia.org/wiki/Web_service). A Web service that follows these guidelines is called *RESTful*. Such a Web service must provide its [Web resources](https://en.wikipedia.org/wiki/Web_resource_framework) in a textual representation and allow them to be read and modified with a [stateless protocol](https://en.wikipedia.org/wiki/Stateless_protocol) and a predefined set of operations. This approach allows interoperability between the computer systems on the [Internet](https://en.wikipedia.org/wiki/Internet) that provide these services. REST is an alternative to, for example, [SOAP](https://en.wikipedia.org/wiki/SOAP) as way to access a Web service.

**Adapter pattern** is a [software design pattern](https://en.wikipedia.org/wiki/Software_design_pattern) (also known as wrapper, an alternative naming shared with the [decorator pattern](https://en.wikipedia.org/wiki/Decorator_pattern)) that allows the [interface](https://en.wikipedia.org/wiki/Interface_(computer_science)) of an existing [class](https://en.wikipedia.org/wiki/Class_(computer_science)) to be used as another interface.[[1]](https://en.wikipedia.org/wiki/Adapter_pattern#cite_note-HeadFirst-1) It is often used to make existing classes work with others without modifying their [source code](https://en.wikipedia.org/wiki/Source_code).

**Pbkdf2** - key derivation functions with a sliding computational cost, used to reduce vulnerabilities to brute-force attacks

**SHA-3** -(Secure Hash Algorithm 3) is the latest member of the Secure Hash Algorithm family of standards, released by NIST on August 5, 2015