**ACNH: Turnips**

**Documentation**

**Bueti Serafino Giovanni**

**189709**

1. Project Description

ACNH: Turnips is a single-page web application that offers a service to Animal Crossing: New Horizons players.

The game is centred on city-building (or island-building, in this specific scenario). The economy of the game orbits around *bells*. Bells are the game’s currency. To be able to build, move around, modify and demolish buildings and structures alike, *bells* are needed.

“*Stalk Market*” is a term associated with the game mechanic that allows the player to buy *turnips* on Sunday (in exchange of bells) for a variable, random price, and sell them on any other day for the highest price possible to maximize profits. The mechanic is inspired to financial markets: turnips play the role of stocks or shares into which the ownership of a company is divided.

Players can fly to other players’ islands using a Dodo Code, a 5-digit alphanumeric code.

The project exposes a queueing service to players, allowing them to share their Dodo Code in an orderly fashion with players from all around the world.

Users are able to host their island, join an existing one using a *Turnip Code*, or filter through the top 30 islands by turnip price (ascending or descending) and hemisphere (Northern, Southern, either).

1. Architecture

The website is built using a **Java**-**JDBC**-**PostgreSQL**-**JSP** tech stack. Front-end technologies involve pure **HTML5**, **CSS3** (abundant Grid and FlexBox layouts usage), **JavaScript** (ES6 syntax + dynamic module loading), **jQuery**, **Sock.js** (for WebSocket-based real-time communication using STOMP messaging).

**Back-End**

It is mostly MVC-based.

Requests are handled by *routers*: index, REST, error and socket. Responsibility is delegated to *entities*, Singleton-controllers. The database itself is an entity.

Being a single-page application, the only route and only view that gets rendered to the user is */index*. Every other route corresponds to a RESTful endpoint.

Each queue created by the user (through an AJAX [POST] request) is separately stored on the DB and in memory: static information less frequently accessed is stored on the DB which is easily accessed through the *QueueDAO* fully static object, returning *QueueMeta* instances – visitors and queued users are instead stored in the server’s RAM.

A single queue, then, is an aggregation of *QueueMeta* information and *Queue*. Visitors (players that have access to the Dodo Code), and queue users are thought to be stored on Redis Cache, future feature. Synchronization and inconsistent states are automatically handled at startup.

The reason behind this decision is efficiency: databases are architecturally built around the necessity to persist information that needs “on-demand” access. Resources that need instantaneous, real-time access are better stored in a caching system where the significantly reduced request overhead allows for much faster access.

Users and admin users are uniquely identified throughout a session. Their data – *userId* and *username* in the case of a visiting user – is persisted on the database with respect to the existence of a queue: when a queue is destroyed, all the respective user data is destroyed along with it.

Authentication and authorization are handled with *LocalStorage* and *UUID* tokens generated after a user creates a queue, and after a user joins one. This approach is thus very susceptible to cross site scripting and cross site request forgery. No sensible data is in any case at stake. The attacker could impersonate a user and leave a queue, or worst case, impersonate the admin and destroy the queue.

A socket service has been additionally implemented to communicate a state-change to all users that joined a queue: a queue can be locked, edited, destroyed. Users are notified in real-time. This approach dramatically reduces the number of requests that would have been otherwise necessary in a AJAX-polling method, given WebSockets take full-on advantage of a TCP connection tunnelled through the server.

The **Database** consists of a single entity: **Queue**.

Each queue has the following properties:

* **turnipCode** (varchar): unique identifies the queue.
* **adminId** (varchar): UUID token identifying the player that created the queue.
* **islandName** (varchar): the name of Animal Crossing island that the admin is hosting.
* **nativeFruit** (varchar): the island’s native fruit. Possible values are: apple, orange, cherry, peach, pear.
* **private** (boolean): whether the island is visible to everyone or exclusively to users in possess of the Turnip Code.
* **turnips** (int4): the price at which Nooklings buy turnips. Determines profit.
* **hemisphere** (varchar): either Northern or Southern, indicates if the island host lives in the Northern or Southern hemisphere (seasons are also at play).
* **maxLength** (int4): the queue’s maximum length.
* **maxVisitors** (int4): the maximum number of users concurrently allowed to join the island (and thus receive the Dodo Code).
* **dodoCode** (varchar): the alphanumeric Dodo Code needed to fly to the host’s island.

Each queue instance persisted in memory has:

* **turnipCode** (String): needed to join the database’s queue object.
* **maxLength, maxVisitors** (Integer): duplicated for their need to be accessed both from the DB (filter search) and memory (join, leave etc.).
* **treasury** (ArrayList<User>): an array used to store the partition of users that have current access to the Dodo Code.
* **queuedUsers** (ArrayList<User>): an array used to store users that are waiting to get access to the island.

Queues can be RESTfully created, read, updated and removed. The Statelessness constraint is however somewhat violated: an admin destroying an island modifies the queue state on the server. The state change is communicated in real-time with all users that subscribed to queue-related messages. This means that the state doesn’t exclusively depend on the client - a subsequent request from another user would be different and cannot thus be cached.

1. Front-End

The front-end uses exclusively HTML5, CSS3, JavaScript, jQuery and Sock.JS. The single *index.jsp* view statically includes the “pages” JSPs, refactored in single files for readability and cleanliness. As per JavaScript, ECMAScript 6 syntax was used.

The application is structured in different “pages”, statically preloaded by the index route. Navigating through pages requires user-interaction.

Navigation can also occur automatically, by specifying the unique *pageName* as part of the hypertext reference’s pathname.

The different pages are:

* **Main Page**: factually the landing page. Display the logo, application description and *notes* that allow the user to navigate to all the other pages.
* **Host Page** (step 1, step 2): allows the user to create an island. This is a 2-step process – on step 1 the user specifies the Dodo Code, on step 2 all the island information, plus listing description.
* **Islands Page**: displays a list of maximum 30 islands that are being currently hosted. Allows the user to filter by turnips price and hemisphere, dynamically requesting a new set of results from the server.
* **Individual Island Page**: allows users to join and leave islands.

User-interaction has been implemented using a completely custom approach. Below is a brief description of all the components that have been developed for this purpose.

* **Index**: entry-point and single file specified on the index JSP. Orchestrates initialization and handles the audio external API.
* **Animator**: class that allows to hide and show custom-built Promise-based modal interfaces and to slide-on and off pages. It has been implemented such that Flex containers are also supported with jQuery’s *.hide* and *.show* methods.
* **Page Router**: routes the user to the correct page depending on the current location’s href. It also holds and loads all page references dynamically. It is the only responsible for navigation, pushing new states to the browser history.
* **Page Handler**: handles page swapping internals and animations. Triggers a custom jQuery event on page swap. Holds the HTML root nodes of each page and information about the current page being shown.
* **Socket Handler**: responds to state-change communicated by the server, such as island destroyed, updated, new user joined etc.
* **Extensions**: a set of utility methods injected to the prototypes of the jQuery object and native String object. Most importantly, a method for handling asynchronous forEach loops on Array prototypes.

Respecting the *single responsibility principle*, each page is individually responsible for responding to user interaction. jQuery *click* event bindings are thus declared on each page. Below is a very brief description of each individual page.

* **Main Page**: handles user interaction with notes triggering navigation with all other pages.
* **Host Page**: it is comprised of two pages linked with each other. This module validates user input for both pages and shows error messages along examples were the data submitted to be invalid. On step 2, assuming data is in fact valid, an AJAX request is made to the server to create a new queue. It also used to update the island information: if the host were to be updating the island, the *Next*button makes an AJAX request to the server to update the queue rather than creating a new one.
* **Islands Page**: upon swap, data is fetched from the server and preloaded using a template HTML element. This element – the individual island note – is then cloned, updated with the fetched data, and appended to the list of islands. If no islands were to be fetched from the server, a default note would be shown notifying the user that no islands were found that satisfy their filter criteria. As of right now, the filter operates exclusively on the islands’ hemisphere. In the future many other could be implemented.
* **Island Page**: responsible for the pulsating heart of the application. Handles user join, leave, admin locking, updating and destroying in real-time for all users. This module is comprised of many asynchronous methods responsible for interacting with the server for validating user status (anonymous, queued, admin), queue status temporarily persisted in *LocalStorage*. It has a *preload* method that allows it to fetch data from the server and load it before showing it to the user. Interface updates are done instantly, with the intention to be seamless – the reduced average attention span of users and instant gratification playing a major role in an app’s marketization and branding, seamless updates are becoming more and more required (<https://www.researchgate.net/publication/327367023_Myth_and_Mystery_of_Shrinking_Attention_Span>). It is additionally responsible to respond to WebSockets updates.

The interface is responsive and has been adapted for a good range of resolutions: 320px (iPhone SE) – 1920px+ (desktop & tablets). This was achieved with heavy use of FlexBox and Grid layouts, shrinking and expanding depending on content and screen resolution. In particular, the islands page is a (3 x auto) grid layout – depending on screen resolution, the layout changes to (2 x auto) and (1 x auto), reducing to a single coordinate FlexBox layout, column-directed. CSS3 variables have additionally been used to aggregate identical RGB values and render extremely easy changing them on the go (for future themes support).

It is also optimized for WebKit, thus Chrome and Safari, embracing more than 89% of the browser market share. It does not willingly support Microsoft Edge or Opera. Partially compatible with Firefox.

The chosen external API is: <http://acnhapi.com/>

It is used to load random game music tracks directly from the server using AJAX playable using HTMLAudioElement’s JS API.