CAPESTONE DESIGN DOCUMENT

Capestone is a JavaScript derived game that will rely on database data to operate different for every user and every use. The user will sign in and create a unique user account. From this point they will be able to sign in from anywhere and have the same game data available to them. After the user has logged in they will begin their Capestone adventure. Prior to page load the server will run an assortment of PHP functions to prepare the game data for use. With the use of PHP userData, heroData, itemData, dungeonData, and monsterData will all be gathered from the database in arrays then will fill JavaScript variables. Game.js will then compile the game environment and draw the game to a canvas on the page. As the user descends a staircase in game the state of the player will be saved back to the database with user stats and inventory list. This way the next time the user logs in the most recent game data for their game experience will be loaded and they may continue their adventure.

Arrays derived from database

heroData = armorClass, attackBonus, currentHP, damage, description, imagePath, maxHP, pass, userID,

username, x, y

(*)monsterData = monsterID, armorClass, attackBonus, currentHP, damage, description, dungeonLevel,

imagePath, maxHP, monsterName, pass

(*)dungeonData = dungeonID, code (**deprecated)**

(*)itemData = itemID, action, armorClass, attackBonus, damage, description, health, itemName, itemType

inventoryData = incriminator, userID, itemID

(*)Each of these is a multidimensional array. Each index is filled with the variables listed, which is an associative array.

Functions in game.js

Game.js is the main code for the Capestone game. Many functions are contained within which I will document to familiarize others with the structure of how it works up to this point. This is a living document.

attack() – A dynamic function that takes two arguments, assailant and defender, which is used to handle all combat within the game. If the user attempts to move into a square that is occupied and has an armorClass and a currentHP value, the assailant will attack that spot, whose values get passed as defender. This works with environment objects as well; very handy. Various messages are displayed to the user to explain what is happening during combat.

CAPESTONE DESIGN DOCUMENT

anItemIsAt(x, y) – Checks to see if there's an item on the ground. Works with the pickUpItem function.

autoLoader() – Main function for the setup of the game. Automatically creates the canvas to draw on, draws the quadrants, creates a single instance of an enemy and an instance of the hero (the user's character), and runs the startGame() function.

being() – Main constructor for any creatures (enemies) or heroes within the game. The object constructor requires three arguments to be passed with an optional fourth. You must pass the image path of the creature you want to instantiate. If you opt to use the options argument, you can pass it values as well, such as description and damage values. This also handles all movement.

canvasBackground() – Simply adds default values for the canvas background, including color and size.

changeLevel() – Works similar to autoloader(), only loads what we need when we change the level. Formats the enemyList, monstersOnThisLevel, and enemyIncrementer, clears out the coordinates and actorcoordinates, and loads a new dungeon up. Also handles the final level.

checkCollision(actor) – Checks for collision when the hero or a random monster is placed on the board. If a collision is detected (either another being object or the environment) it will generate a random position until it finds clear ground.

checkDeath() – A function which checks for the death of the creature or user. Used in conjunction with the attack() function. A creature is deemed dead when it has zero or negative currentHP (current hit points).

clearConsole() – A function that clears the console. Somewhat unused at this point, just preparing for the future.

destroyItem(index) – Works with the dropItem and quaffPotion functions to remove the item from the inventory.

displayControls() – Displays to the user the control scheme of the game. If we add any controls to the game, we must update this function. displayControls() is also called in the startGame() function, so the user should know the controls when starting the game. This can also be called by pressing 'c'.

displayInventory() – Displays the inventory of the user. In the future this will be used in conjunction with equipItems() to show you the inventory you can equip.

dropItem(keyPressed) – Handles dropping of items. Clears the inventory array and updates it when you do so. keyPressed is a variable you pass with the keyhandler.

enemyBehavior() – This function determines the behavior of every creature in the game. As of this point, the enemy will only move towards the user if it is within a square radius of ten spaces. Once it has 'seen' the user, it will move toward them no matter where they go, unless they fool them into getting behind an object they can't find a way around. We plan to improve this function using an 'a star' topology so the creatures can find their way around objects or other structures.

CAPESTONE DESIGN DOCUMENT

enemyLoader() – Every fifty turns a random monster will be added to the board with random x and y values using this function.

environment() – This function is the default constructor for any environment objects contained within the game. If we wish to instantiate an object that has no movement, we can make it through this constructor. All environment objects are used to create destructible objects (trees, doors, etc.), impassible objects (rock wall, candelabra, etc.), and in the future we will use it to create objects the user can manipulate, such as doors or treasure chests.

equipCheck() – This checks whether or not something is equipped or not. Does not allow the equipping of more than one weapon or armor.

equipItems() – Allows the user to choose which item they want to equip from their inventory. This function will change the user's attributes to reflect the item in which they are equipping.

fenceLoader() – **Depricated**. Originally used when the playfield was 10 x 10. It would draw a fence with an arrow inside of it which could be used to throw at an enemy.

getChestItem(keyPressed) – Handles the grabbing of items from any chests. You can leave the item in the chest or take it. It will remain there when you come back.

getRandomDungeon() – Handles the random dungeon placement. Grabs data from dungeons.js (all of the dungeon quadrants)

heroLoader() – Creates the hero object which gets values passed from PHP. Default values are stored within our database upon signing up and can be saved by pressing 's' and running the saveData() function.

item() – Main object constructor for the item() object. Holds all the values we would ever need for items that can be held by the player (inside inventory).

interactive() – This function is the main constructor for an interactive state. This would be something like checking a chest, opening a door, moving down stairs, etc.

openChest(keyPressed) – Handles chest opening. Works similar to openDoor, but adds more functionality for grabbing an item inside.

openDoor(keyPressed) – Handles opening of doors. Works similarly to openChest but with less functionality.

pickUpItem(keyPressed, x, y) – Handles picking up of items. We didn't actually implement this, but it does work. There is a function called placeItem() which you can use to test this with. This was going to be used but we didn't have enough time. A shame, because we could have definitely implemented it now that we have both actorCoordinates and coordinates separating the two entities.

CAPESTONE DESIGN DOCUMENT

placeStairs() – Function that places stairs randomly throughout the map. One up staircase and one down staircase are placed on each floor.

quadrantLoader() – A series of quadrant loaders numbering from one to four which draws the environment objects from a dynamically coded array. This allows us to place randomly the quadrants so that we can have semi-randomized environments.

quaffPotion(keyPressed) – Handles the drinking of potions. Works very similar to the dropItem(keyPressed) function.

randomBarrier() – Creates random, destructible barriers which are placed after the quadrants have been loaded. This adds another level on randomness to the levels. Between 2 and 30 barriers are created when this function is called.

randomInventory() – There is one argument for this function which gives the possessor a random weapon and a random armor. This will need to be changed to reflect the values given to me by the database.

redrawCoordinates() – When creating a turn based game using canvas, you need to clear out the canvas and re-draw everything whenever a creature or the user moves. This function does exactly that.

RNG() – Random number generator between one and the max number you decide when calling the function. Very simple but used often.

rollDice() – Does exactly the same thing as RNG() but the random number is between one and the max number.

saveData() – Makes an AJAX call using jquery to send data back to PHP so it can be processed and placed back within the database. At this point, only saves the hero object.

startGame() – Doesn't do much. Displays the controls and a welcome message.

traverseStairs(keyPressed) – Handles moving up and down stairs.

updateItemData() – Takes the data being passed to me from PHP and turns them into item() objects.

updateHTMLStats() – This updates the statistics for the graphical user interface.

updateMonsterArray() – This pushes all monster data for the current level. Only appropriate monsters for that particular level will be pushed to this.

updateStats(equipper, index) – Works with the equipItem function. Updates all of the equippers statistical values. I made this dynamic so we could have enemies wield items. We never got around to it unfortunately, not enough time.

document.onkeypress = function(e) – This is a function which gets called on the onkeypress event of the document. This allows the user to press any key on the keyboard and the game will be able to

interpret it if logic is applied to that key press. Extremely useful. There is a console.log() contained within that shows the key code of the key that is pressed, so if we decide that we want to implement more functionality we can find the key code easily. This function also causes the enemies to move and the coordinates to be redrawn. Every time a key is pressed, everyone on the game field gets a turn, so therefore this determines the global counter of the game.

Functions in PHP classes. Functions are separated by what PHP file they are located in.

Signup.php Class

usernameIsTaken($username) : will take the username provided on signup and compare it to usernames in the database to make sure there are no duplicates.

emailEntryIsValid() : will compare email provided on signup form and make sure it is a real email

usernameEntryIsValid(): will make sure that the username is a valid username

passwordEntryIsValid(): will make sure the password is long enough and valid characters

saveEntry(): this will save the user information to the database after all information provided has been validated. This is a simple insert SQL query that relies on the default values for hero table to create new character.

Login.php Class

getUserID($username, $password): will be used to get the user ID based on the username and password given as long as they match values in the user table.

loginIsValid($username, $password): this makes sure that the information provided on the Login form is a real user retained in the database.

HeroDB.php class

getHeroData($userID): this is a SQL query to get all the user data from the database based on the user ID Provided.

getUsername($userID): this will get the username for display in the game based on the user ID

getUserID(): this will get the user ID based on the email username and password retained in the $_Post super global variable

defaultEntry(): this is used on character creation to save username and user ID to the Hero table and relying on default values in the table to fill in the rest.

saveHero($userID, $armorClass, $attackBonus, $currentHP, $damage, $description, $maxHP, $x, $y, $dungeonLevel): this will take all the current hero information in the game and save it back to the database so the user can pick up where they left off. It is an update SQL query.

CAPESTONE DESIGN DOCUMENT

ItemDB.php

getItemData(): this will get all the items out of the database to be used in the game. It is a Select SQL Query.

getInventoryItemData($itemID): this will be used to get specific item data based on item ID

InventoryDB.php class

getInventoryData($userID): this will get all the items that a specific user has in their inventory. This will return the Item ID to be used to get individual item data.

clearInventory($userID): this will clear all inventory that a user has to clean up the table before saving updated inventory data back to the table

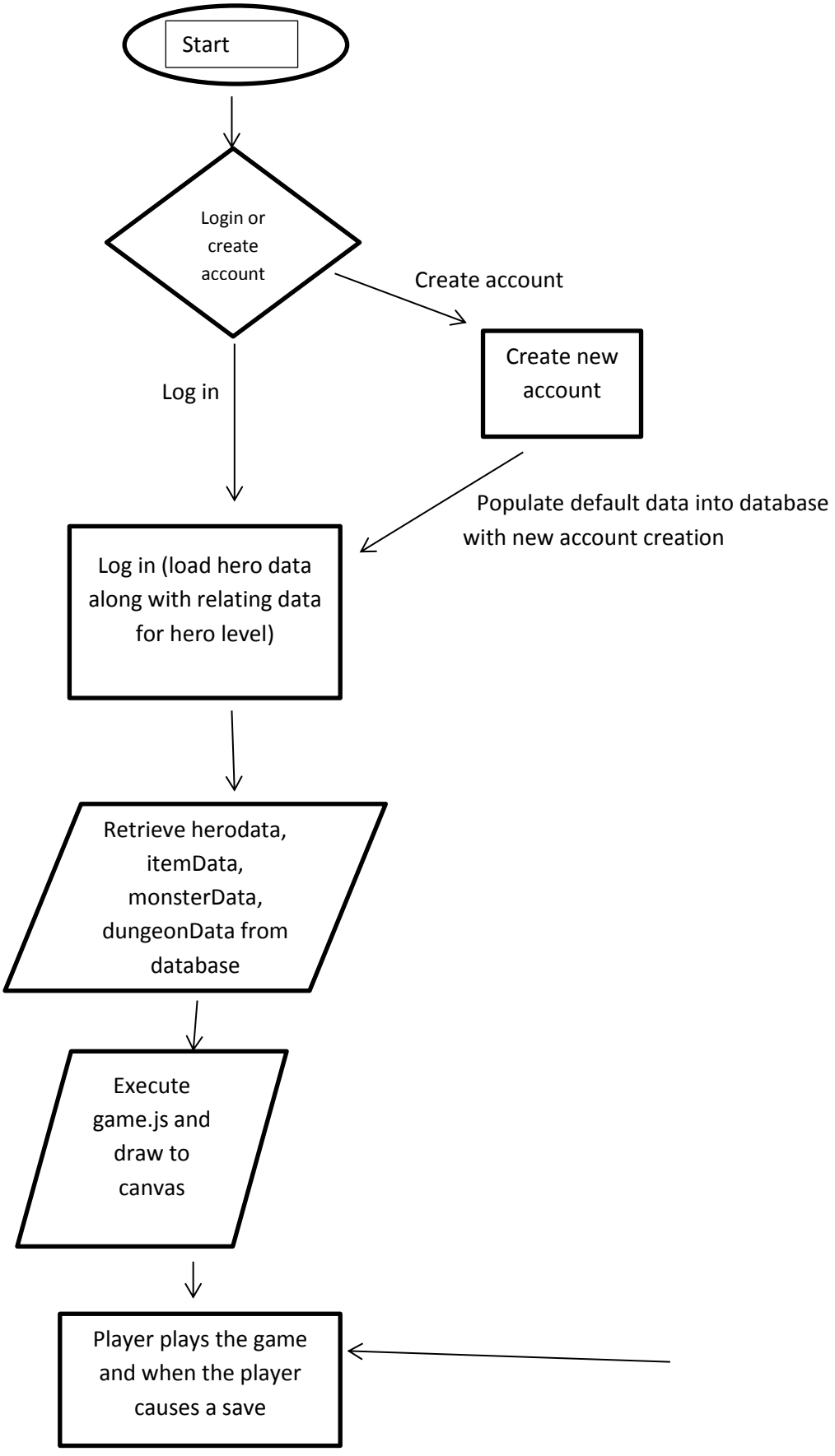
saveInventory($userID, $itemID, $equipped): this will save the inventory back to the table passing it the user ID, item ID, and if it is equipped or not.

defaultInventory(): this will save the default inventory to the table based on the user that is creating a character. This way the hero will have an inventory when starting.

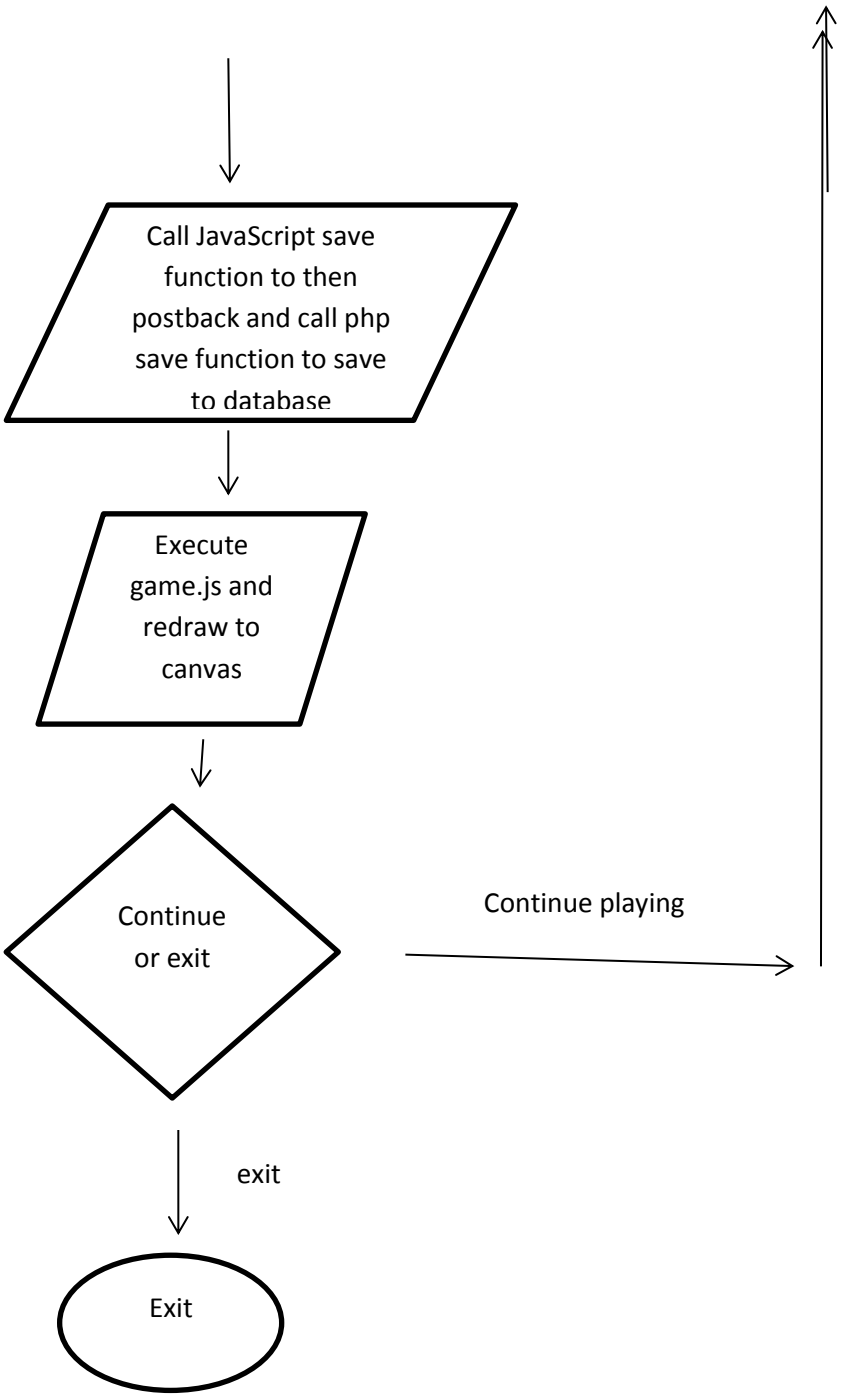
MonsterDB.php class

getMonsterData(): this will get all monsters data from the table

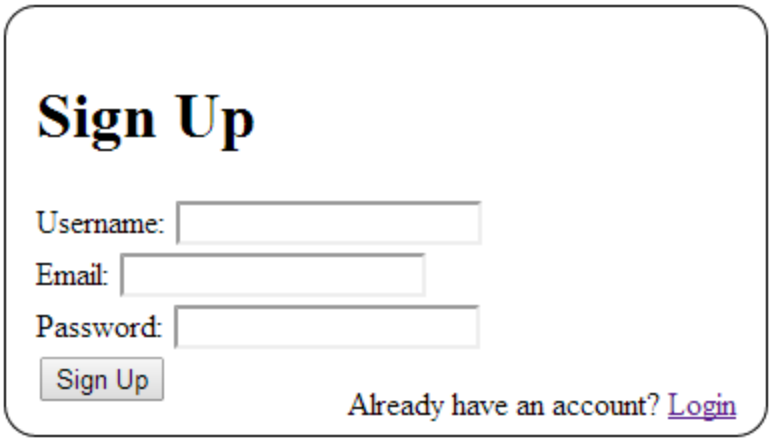
CAPESTONE DESIGN DOCUMENT

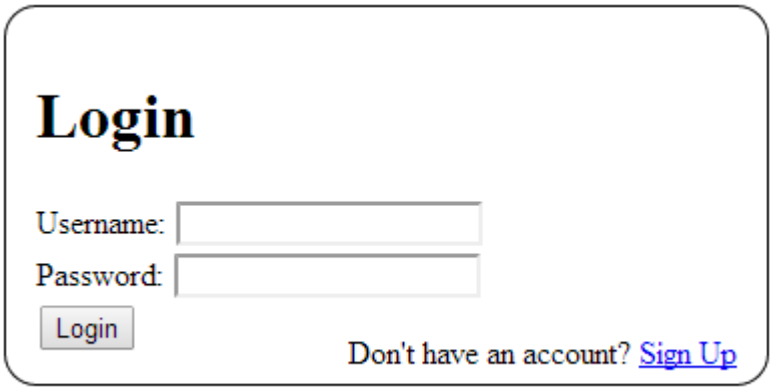


CAPESTONE DESIGN DOCUMENT

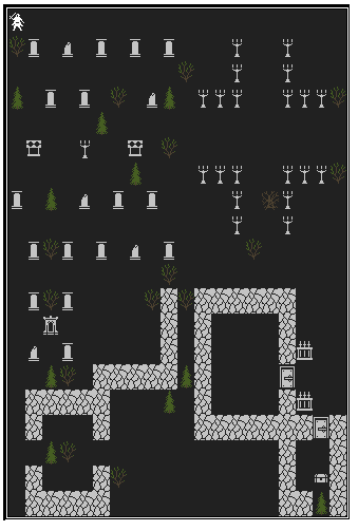


CAPESTONE DESIGN DOCUMENT





The user will first signup and must input a valid and unique username that will be displayed as their character name in game. After having an account they can sign in with that username and password.



CAPESTONE DESIGN DOCUMENT