**LockdownController.java** the class is a spring boot-based REST controller made to control user login and cause a lockdown mode upon correct user authentication. The web front end is caused to go into lockdown mode in full-screen kiosk mode and preventing access to other applications. The class is a RESTful web controller(@RestController) which exposes API endpoints. The @SpringBootApplication annotation enables auto-configuration in addition to component scanning. An HTTP Session object is used to control session attributes, ensuring state persistence during the user interactions.

**loginPart()** the method maps HTTP POST requests to /api/login. The username and password are received as request parameters. The userAuthentication is called to validate the login credentials. The lockdownTrigger(sessions) method is called to activate lockdown mode. If the login is valid, the success message is returned. Otherwise, an error is returned. **userAuthentication()** user authentication is simulated through this private helper method. This is a dummy application for admin for the username and password 123 for the password. These need to be changed according to what is used for the real application. True is returned if the authentication is successful. Otherwise, the Boolean false error is produced. Ideally this method is just for simulating user authentication, another team member has probably done the part of assigning a username and password. **lockdownTrigger()** an attribute is first set to the session to track if the lockdown mode is active. The screen is forced to full-screen mode in the browser using the code in the try block statement. error traces will happen if an error occurs whilst executing the command. The command itself will launch google chrome in kiosk mode(which is a full screen without the user accessing the browser controls. The webfront is opened at the hosted <https://localhost:8080> which is the case since the webfront will be hosted locally on a laptop as per the requirements of the project. **checkIfLockdownIsActive().** The lockdown attribute is retrieved from the session. the methods output depends on if the lockdown is null or false which will thus return false, meaning lockdown mode is inactive.

**LockdownIntercepter.java**  This class is responsible acting as an interceptor for incoming HTTP requests in a Spring Boot web application. The goal is to enforce a lockdown mode where the user is restricted to specific endpoints of an API when the lockdown is active. If the user attempts to access any other page or another API whilst the lockdown is in full effect, the user is taken back to the lockdown page. First off the class is annotated with @RestController and @RequestMapping(“/api”) making it a RESTful web controller in charge of handling API requests. The @SpringBootApplication annotation activates the Spring Boot auto-configuration and component scanning. Three interceptor style methods are contained in the class. **preHandle()**  the first line in the method is responsible for retrieving the existing session(false will prevent a fresh one being created if none exist). Only if a session exists does it check if the lockdown attribute is set( the condition inside the if block). The lockdown mode is active if the session exists and the lockdown attribute is true. The string variable inside the first if block extracts the endpoint which the user is attempting to access. The nested if statement has the condition where if the request is not for /api/checkLockdown and /api/login then the user is blocked from signing in. the code inside the nested if block is where the enforceLockdown() method gets called which has built in commands to restrict access to things such as firefox, taskmgr and explorer. the commands close unauthorised apps such as firefox and task manager and task manager is prevented from reopening by modifying the windows registry.  **postHandle()** this simply runs once the request is sorted but before the response is sent to the client. No modifications needed for this section. **afterCompletion()** This method runs once the response is fully completed. It issued for cleaning up resources and handling exceptions but not needed here.

**WebMvcConfig.java** this class is a spring web MVC configuration class responsible for customizing how the web application controls which external domains can make requests(Cross-Origin Resource Sharing). **addingCorsMapping()** this method configures the CORS settings to allow requests from the front end. The CorsRegistry is taken as a parameter. The parameter will be used to define which domains can interact with the backend. Thr “\*\*” means all endpoints will have CORS settings applied to them which includes any API request made to the backend. Only requests from the local host link are permitted. Simply replace the example with the link when loading up XAMP and accessing the database. The HTTP requests such as GET(fetching data), POST(submitting new data), PUT(updating existing data) and DELETE(removing data) are all accepted by the backend. Cookies and authentication headers are also enabled to be shared between the frontend and the backend in the. allowCredentials() built in property. This is essential to prevent requests getting blocked from the frontend due to security policies. **addingResourceHandlers()** this method configures how static resources are served. Static resources are the images, CSS and JavaScript files. These files are made accessible in the web application. The .addResourceLocation() are responsible for asking spring boot to check inside the /static/ folder in the class path which is inside the project. E.g. <http://localhost:8080/static/logo.png> is accessed if the folder has static/logo.png. this is essential to enable frontend assets such as images, styles, js files to be served without the necessity of an extra backend API. **addingViewControllers()** this method creates a simple URL to view the mapping without requiring a separate controller. It also allows a URL to be a mapped to a view template. When the user visits the https://localhost:8080/lockdown , the lockdown view is loaded. A dedicated controller method is therefore not needed to return this view. The routing to static pages is therefore simplified. **startKiosk()** @GetMapping(“/startKiosk”) initiates google chrome in full-screen kiosk mode where the browser is ran in restricted mode (no navigation buttons, no address bar). The REST endpoint(<http://localhost:8080/startKiosk>) is defined and ran when it is called. So in the try block of the method, the Runtime class is used to execute a command-line instruction. i.e. the cmd /c start/max maximises the window after opening a new command prompt. The chrome –kiosk <http://localhost:8080> is where chrome is launched in kiosk mode and the web application homepage is displayed.

**disableshortcuts.js** the javascript file enforces a lockdown mode by restricting user actions in a web browser. Keypress events are listened for and certain keyboard shortcuts are blocked which could be used to get bypass restrctions or accessing the developer tools. The first line inside the file listens for when a key is pressed on the keyboard. Details are stored inside the event object regarding which key was pressed. The first if statement stops common keys users may use to open developer tools, save pages or copy content. i.e. “event.ctrlKey” blocks Ctrl shortcuts. “event.metaKey” blocks command key shortcuts on macOS. “event.altkey” blocks the Alt key shortcuts. “event.preventDefault()” prohibits the default browser behaviour such as saving the page. The second if statement checks if the F11 key is pressed which toggles the full screen mode. The event.prevenDefault() stop the browser from entering the fullscreen mode. The third statement blocks the F12 key which opens the browsers developer console. The second parentheses condition block stops the developer tools being opened and the developer console from being ooened also. The third set of parentheses for the condition block stops Ctrl+U which opens the view page source.