Codes

iot_hub repository

MockEnvironment.java

```
package ece448.iot_hub;


import java.util.HashMap;

import java.util.Map;


public class MockEnvironment {

  private final Map<String, String> properties = new HashMap<>();


  public boolean containsProperty(String key) {

    return properties.containsKey(key);

  }


  public String getProperty(String key) {

    return properties.get(key);

  }


  public String getProperty(String key, String defaultValue) {

    return containsProperty(key) ? getProperty(key) : defaultValue;

  }


  public void setProperty(String key, Object value) {
```

```
        properties.put(key, String.valueOf(value));
    }

    public void put(String key, Object value) {
        setProperty(key, value);
    }

    public String[] getActiveProfiles() {
        return new String[0];
    }

    public String[] getDefaultProfiles() {
        return new String[0];
    }

    public boolean acceptsProfiles(String... profiles) {
        return true;
    }
}
```

App.java

```
package ece448.iot_hub;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.beans.factory.annotation.Value;

@SpringBootApplication
public class App {
    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }

    @Configuration
    public static class AppConfig {
        @Value("${mqtt.broker}")
        private String mqttBroker;

        @Value("${mqtt.clientId}")
        private String mqttClientId;

        @Value("${mqtt.topicPrefix}")
        private String mqttTopicPrefix;

        @Bean
        public MQTTController mqttController() throws Exception {
            return new MQTTController(mqttBroker, mqttClientId, mqttTopicPrefix);
        }
    }
}
```

MQTTController.java

```
package ece448.iot_hub;

import java.nio.charset.StandardCharsets;
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;

import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MQTTController {
   private final String broker;
   private final String clientId;
   private final String topicPrefix;
   private final MqttClient client;
   private final Map<String, String> states = new HashMap<>();
   private final Map<String, String> powers = new HashMap<>();
   private static final Logger logger = LoggerFactory.getLogger(MQTTController.class);

   public MQTTController(String broker, String clientId, String topicPrefix) throws Exception {
      this.broker = broker;
```

```
    this.clientId = clientId;
    this.topicPrefix = topicPrefix;
    this.client = new MqttClient(broker, clientId, new MemoryPersistence());

    // Initialize all plugs to OFF by default
    for (char c = 'a'; c <= 'g'; c++) {
        String plugName = String.valueOf(c);
        states.put(plugName, "off");  // Use "off" instead of "oZ"
        powers.put(plugName, "0.000");
    }

    // Start the connection during initialization
    this.start();
}

public void start() throws Exception {
    MqttConnectOptions mqttopt = new MqttConnectOptions();
    mqttopt.setCleanSession(true);
    mqttopt.setAutomaticReconnect(true);
    mqttopt.setConnectionTimeout(10);
    mqttopt.setKeepAliveInterval(60);
    client.connect(mqttopt);
    client.subscribe(topicPrefix + "/update/#", this::handleUpdate);
    logger.info("MqttCtl {}: {} connected", clientId, broker);
}
```

```
  public void close() throws Exception {
    client.disconnect();
    logger.info("MqttCtl {}: disconnected", clientId);
  }

  synchronized public void publishAction(String plugName, String action) {
    String topic = topicPrefix + "/action/" + plugName + "/" + action;
    try {
      logger.info("Publishing action: {}", topic);
      client.publish(topic, new MqttMessage());
    } catch (Exception e) {
      logger.error("MqttCtl {}: {} fail to publish", clientId, topic, e);
    }
  }

  synchronized public String setState(String plugName, String value) {
    // For MQTT, use "on" and "off"
    String normalizedValue = "on".equals(value) ? "on" : "off";
    logger.info("Setting state for {}: {}", plugName, normalizedValue);

    // Publish state update
    try {
      String topic = topicPrefix + "/update/" + plugName + "/state";
      MqttMessage message = new
MqttMessage(normalizedValue.getBytes(StandardCharsets.UTF_8));
      client.publish(topic, message);
```

```
        } catch (Exception e) {
            logger.error("Failed to publish state update for {}: {}", plugName, e.getMessage());
        }

        return states.put(plugName, normalizedValue);
    }

    synchronized public String setPower(String plugName, String value) {
        // Publish power update
        try {
            String topic = topicPrefix + "/update/" + plugName + "/power";
            MqttMessage message = new
MqttMessage(value.getBytes(StandardCharsets.UTF_8));
            client.publish(topic, message);
        } catch (Exception e) {
            logger.error("Failed to publish power update for {}: {}", plugName, e.getMessage());
        }

        return powers.put(plugName, value);
    }

    synchronized public String getBroker() {
        return broker;
    }

    synchronized public String getState(String plugName) {
```

```
    String s = states.get(plugName);
    return (s == null) ? "off" : s;  // Default to "off" if no state stored
}

synchronized public Map<String, String> getStates() {
    return new TreeMap<>(states);
}

synchronized public Map<String, String> getPowers() {
    return new TreeMap<>(powers);
}

synchronized public String getPower(String plug) {
    String p = powers.get(plug);
    return (p == null) ? "0.000" : p;
}

synchronized protected void handleUpdate(String topic, MqttMessage msg) {
    logger.debug("MqttCtl {}: {} {}", clientId, topic, msg);
    // strip off "<prefix>/" and split -> [ "update", plugName, field ]
    String[] parts = topic.substring(topicPrefix.length() + 1).split("/");
    if (parts.length != 3 || !"update".equals(parts[0])) {
        return;
    }

    // decode actual payload bytes as UTF-8 text
```

```
        String payload = new String(msg.getPayload(), StandardCharsets.UTF_8).trim();
        if ("state".equals(parts[2])) {
            // only allow "on" or "off"
            states.put(parts[1], payload.equals("on") ? "on" : "off");
            logger.info("Updated state for {}: {}", parts[1], states.get(parts[1]));
        }
        else if ("power".equals(parts[2])) {
            // record the numeric power string
            powers.put(parts[1], payload);
            logger.info("Updated power for {}: {}", parts[1], payload);
        }
    }
}
```

PlugsModel.java

```
package ece448.iot_hub;

import java.util.HashMap;
import java.util.Map;
import org.springframework.stereotype.Component;


@Component
public class PlugsModel {
    private final MQTTController mqttController;
    private final Map<String, String> states;
    private final Map<String, String> powers;


    public PlugsModel(MQTTController mqttController) {
```

```
    this.mqttController = mqttController;
    this.states = new HashMap<>();
    this.powers = new HashMap<>();

    // Initialize all plugs to OFF by default
    for (char c = 'a'; c <= 'g'; c++) {
        String plugName = String.valueOf(c);
        states.put(plugName, "off");
        powers.put(plugName, "0.000");
    }
}

public String getPlugState(String plug) {
    // Return state from local cache, or fall back to MQTT controller state
    String state = states.get(plug);
    if (state == null) {
        // Try to get state from MQTT controller
        state = mqttController.getState(plug);
        // Normalize state values - convert "oZ" to "off" for API
        state = "on".equals(state) ? "on" : "off";
        // Cache the state
        states.put(plug, state);
    }
    return state;
}
```

```
public String getPlugPower(String plug) {
    // Return power from local cache, or fall back to MQTT controller power
    String power = powers.get(plug);
    if (power == null) {
        // Try to get power from MQTT controller
        power = mqttController.getPower(plug);
        // Cache the power
        powers.put(plug, power);
    }
    return power;
}

public void setPlugState(String plug, String state) {
    // Normalize state values for the API
    String normalizedState = "on".equals(state) ? "on" : "off";
    // Update local cache
    states.put(plug, normalizedState);
    // Update MQTT controller
    mqttController.setState(plug, normalizedState);
}

public void setPlugPower(String plug, String power) {
    // Update local cache
    powers.put(plug, power);
    // Update MQTT controller
    mqttController.setPower(plug, power);
```

```
    }

    public Map<String, String> getAllStates() {
        // Return a copy to prevent external modification
        return new HashMap<>(states);
    }

    public Map<String, String> getAllPowers() {
        // Return a copy to prevent external modification
        return new HashMap<>(powers);
    }

    public void togglePlugState(String plug) {
        String currentState = getPlugState(plug);
        String newState = "on".equals(currentState) ? "off" : "on";
        setPlugState(plug, newState);
    }
}
```

PlugsResource.java

```
package ece448.iot_hub;

import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
```

```
import java.util.Map;
import java.util.TreeMap;

import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class PlugsResource {
  private final MQTTController mqttController;
  private static final Logger logger = LoggerFactory.getLogger(PlugsResource.class);

  @Autowired
  public PlugsResource(MQTTController mqttController) {
    this.mqttController = mqttController;
  }

  synchronized public void pubAction(String plug, String action) throws Exception {
```

```
    logger.info("Publishing action for {}: {}", plug, action);
    mqttController.publishAction(plug, action);
}

synchronized public String getPlugsState(String plug) throws Exception {
    String state = mqttController.getState(plug);
    return state;
}

synchronized public String getPlugsPower(String plug) throws Exception {
    return mqttController.getPower(plug);
}

synchronized public Object getAllThePlugs() throws Exception {
    List<Map<String, Object>> ter = new ArrayList<>();
    for (String plug : mqttController.getStates().keySet()) {
        Map<String, Object> plugInfo = new HashMap<>();
        plugInfo.put("name", plug);
        plugInfo.put("state", mqttController.getState(plug));
        plugInfo.put("power", mqttController.getPower(plug));
        ter.add(plugInfo);
    }
    return ter;
}

@GetMapping("/api/plugs")
```

```
public List<Map<String, Object>> getPlugs() {
    List<Map<String, Object>> plugs = new ArrayList<>();
    for (String plugName : mqttController.getStates().keySet()) {
        plugs.add(makePlug(plugName));
    }
    return plugs;
}

@GetMapping("/api/plugs/{plugName}")
public ResponseEntity<Object> getPlugDetails(
        @PathVariable("plugName") String plugName,
        @RequestParam(value = "action", required = false) String action) {

    // Check if the plug exists
    if (!mqttController.getStates().containsKey(plugName)) {
        return ResponseEntity.notFound().build();
    }

    // Handle action if provided
    if (action != null) {
        if (!isValidAction(action)) {
            return ResponseEntity.badRequest().body("Invalid action");
        }
        mqttController.publishAction(plugName, action);
        logger.info("Published action {} for plug {}", action, plugName);
    }
```

```
    return ResponseEntity.ok(makePlug(plugName));
}

@GetMapping("/{plugName}")
public ResponseEntity<Object> getPlug(
        @PathVariable("plugName") String plugName,
        @RequestParam(value = "action", required = false) String action) {

    // Check if the plug exists
    if (!mqttController.getStates().containsKey(plugName)) {
        return ResponseEntity.notFound().build();
    }

    // Handle action if provided
    if (action != null) {
        if (!isValidAction(action)) {
            return ResponseEntity.badRequest().body("Invalid action");
        }
        mqttController.publishAction(plugName, action);
        logger.info("Published action {} for plug {}", action, plugName);
    }

    // For this endpoint, just return the state
    return ResponseEntity.ok(mqttController.getState(plugName));
}
```

```
private boolean isValidAction(String action) {
   return action.equals("on") || action.equals("off") || action.equals("toggle") ||
         action.equals("oZ") || action.equals("oG") || action.equals("or") ||
         action.equals("oz") || action.equals("og");
}

private HashMap<String, Object> makePlug(String plugName) {
   HashMap<String, Object> ret = new HashMap<>();
   ret.put("name", plugName);
   ret.put("state", mqttController.getState(plugName));
   ret.put("power", mqttController.getPower(plugName));
   return ret;
}

@GetMapping("/{plug}")
public ResponseEntity<String> handleAction(
   @PathVariable("plug") String plug,
   @RequestParam(name = "action", required = false) String action) {

   // Check if the plug exists by checking if it has a state
   if (mqttController.getState(plug) == null) {
      return ResponseEntity.notFound().build();
   }

   if (action == null) {
```

```
        // If no action specified, return current state
        String state = mqttController.getState(plug);
        return ResponseEntity.ok(state);
    }

    try {
        // Publish the action via MQTT and let the controller handle it
        mqttController.publishAction(plug, action);

        // Wait briefly for state to propagate
        Thread.sleep(100);

        String state = mqttController.getState(plug);
        return ResponseEntity.ok(state);
    } catch (Exception e) {
        logger.error("Failed to handle action for plug {}: {}", plug, e.getMessage());
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }
}

@GetMapping("/api/{plug}")
public ResponseEntity<String> handleSimulator(
        @PathVariable("plug") String plug,
        @RequestParam(value = "action", required = false) String action) {

    logger.info("Direct simulator request: plug={}, action={}", plug, action);
```

```
if (action != null) {
    try {
        // First publish the action via MQTT
        mqttController.publishAction(plug, action);

        // Then update the state
        if (action.equals("on")) {
            mqttController.setState(plug, "on");
        } else if (action.equals("off") || action.equals("oG") || action.equals("or")) {
            mqttController.setState(plug, "off");
        } else if (action.equals("toggle")) {
            String currentState = mqttController.getState(plug);
            String newState = "on".equals(currentState) ? "off" : "on";
            mqttController.setState(plug, newState);
        }

        // Wait briefly for state to propagate
        Thread.sleep(100);

        // Return updated state
        return ResponseEntity.ok(mqttController.getState(plug));
    } catch (Exception e) {
        logger.error("Failed to handle simulator action for plug {}: {}", plug, e.getMessage());
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }
}
```

```
    // If no action specified, return current state
    return ResponseEntity.ok(mqttController.getState(plug));
}

@PostMapping(path = "/plugs/{name}/action/{action}")
public ResponseEntity<String> publishAction(
        @PathVariable("name") String name,
        @PathVariable("action") String action) {
    try {
        // Validate action
        if (!Arrays.asList("on", "off", "toggle").contains(action)) {
            return ResponseEntity.badRequest().body("Invalid action: " + action);
        }

        // First publish the action via MQTT
        mqttController.publishAction(name, action);

        // Then update the state
        if (action.equals("on")) {
            mqttController.setState(name, "on");
        } else if (action.equals("off")) {
            mqttController.setState(name, "off");
        } else if (action.equals("toggle")) {
            String currentState = mqttController.getState(name);
            String newState = "on".equals(currentState) ? "off" : "on";
```

```
            mqttController.setState(name, newState);
        }

        // Wait briefly for state to propagate
        Thread.sleep(100);

        return ResponseEntity.ok(mqttController.getState(name));
    } catch (Exception e) {
        logger.error("Error publishing action: {}", e.getMessage(), e);
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
            .body("Error publishing action: " + e.getMessage());
    }
  }
}
```

HubConfig.java

```
package ece448.iot_hub;

import com.fasterxml.jackson.annotation.JsonCreator;
import com.fasterxml.jackson.annotation.JsonProperty;

public class HubConfig {

private final int httpPort;
private final String mqttBroker;
private final String mqttClientId;
private final String mqttTopicPrefix;
```

```
@JsonCreator
public HubConfig(
@JsonProperty(value = "httpPort", required = true) int httpPort,
@JsonProperty(value = "mqttBroker", required = true) String mqttBroker,
@JsonProperty(value = "mqttClientId", required = true) String mqttClientId,
@JsonProperty(value = "mqttTopicPrefix", required = true) String mqttTopicPrefix) {
this.httpPort = httpPort;
this.mqttBroker = mqttBroker;
this.mqttClientId = mqttClientId;
this.mqttTopicPrefix = mqttTopicPrefix;
}

public int getHttpPort() {
return httpPort;
}

public String getMqttBroker() {
return mqttBroker;
}

public String getMqttClientId() {
return mqttClientId;
}

public String getMqttTopicPrefix() {
```

```
return mqttTopicPrefix;
}
}
```

Main.java

```
package ece448.iot_hub;

import java.io.File;
import java.util.HashMap;

import com.fasterxml.jackson.databind.ObjectMapper;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.context.ConfigurableApplicationContext;

public class Main implements AutoCloseable {

public static void main(String[] args) throws Exception {
// load configuration file
String configFile = args.length > 0 ? args[0] : "hubConfig.json";
HubConfig config = mapper.readValue(new File(configFile), HubConfig.class);
logger.info("{}: {}", configFile, mapper.writeValueAsString(config));


try (Main m = new Main(config, args)) {
```

```
        for (;;) {
                Thread.sleep(60000);
                }
        }
}

public Main(HubConfig config, String[] args) throws Exception {
// Spring app
        HashMap<String, Object> props = new HashMap<>();
        props.put("server.port", config.getHttpPort());
        props.put("mqtt.broker", config.getMqttBroker());
        props.put("mqtt.clientId", config.getMqttClientId());
        props.put("mqtt.topicPrefix", config.getMqttTopicPrefix());
        SpringApplication app = new SpringApplication(App.class);
        app.setDefaultProperties(props);
        this.appCtx = app.run(args);
        }

@Override
public void close() throws Exception {
        appCtx.close();
        }

private final ConfigurableApplicationContext appCtx;

private static final ObjectMapper mapper = new ObjectMapper();
```

```
private static final Logger logger = LoggerFactory.getLogger(Main.class);
}
```

iot_sim repository

iot_sim/http_server/JHTTP.java

```
package ece448.iot_sim.http_server;

import java.net.*;
import java.util.concurrent.*;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * Adopted from Java Network Programming 4th.
 * - Allow JHTTP to start in its own thread.
 * - Allow to stop JHTTP for grading.
 */
public class JHTTP {
	private static final Logger logger = LoggerFactory.getLogger(JHTTP.class);
	private final ServerSocket server;
	private final int port;
	private final RequestHandler handler;

	public JHTTP(int port, RequestHandler handler) throws Exception {
		this.server = new ServerSocket();
```

```
            this.port = port;
            this.handler = handler;
        }

        public void start() throws Exception {
            CountDownLatch c = new CountDownLatch(1);
            Thread t = new Thread(() -> loopForever(c));
            t.setDaemon(true);
            t.start();
            if (!c.await(60, TimeUnit.SECONDS))
                throw new Exception("JHTTP start timeout.");
        }

        public void close() throws Exception {
            server.close();
        }

        protected void loopForever(CountDownLatch c) {
            ExecutorService pool = Executors.newFixedThreadPool(50);
            try {
                server.setReuseAddress(true);
                server.bind(new InetSocketAddress(port));
                logger.info("JHTTP: accepting connections on port {}",
server.getLocalPort());
                c.countDown();
                while (true) {
```

```
                        Socket request = server.accept();
                        Runnable r = new RequestProcessor(request, handler);
                        pool.submit(r);
                    }
                }
                catch (SocketException e) {
                    logger.info("JHTTP: disconnnected {}", e.getMessage());
                }
                catch (Throwable th) {
                    logger.error("JHTTP: exit", th);
                    System.exit(-1);
                }
                finally {
                    pool.shutdownNow();
                }
            }
}
```

iot_sim/http_server/RequestHandler.java

```
package ece448.iot_sim.http_server;

import java.util.Map;

/**
 * Return a string upon a GET request.
 */
public interface RequestHandler {
```

```
    public String handleGet(String path, Map<String, String> params);
}
```

iot_sim/http_server/RequestProcessor.java

```
package ece448.iot_sim.http_server;

import java.io.*;
import java.net.*;
import java.util.*;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * Adopted from Java Network Programming 4th.
 * - Delegate to RequestHandler instead of returning files.
 */
public class RequestProcessor implements Runnable {

        private final static Logger logger =
LoggerFactory.getLogger(RequestProcessor.class);
        private final Socket connection;
        private final RequestHandler handler;

        public RequestProcessor(Socket connection, RequestHandler handler) {
                this.connection = connection;
                this.handler = handler;
```

```
	}

	@Override
	public void run() {
		try {
			OutputStream raw = new
BufferedOutputStream(connection.getOutputStream());
			Writer out = new OutputStreamWriter(raw);
			Reader in = new InputStreamReader(new
BufferedInputStream(connection.getInputStream()), "US-ASCII");
			StringBuilder requestLine = new StringBuilder();
			while (true) {
				int c = in.read();
				if (c == -1)
					return;
				if (c == '\r' || c == '\n')
					break;
				requestLine.append((char) c);
			}

			String get = requestLine.toString();

			logger.info("JHTTP: {} {}", connection.getRemoteSocketAddress(), get);

			String[] tokens = get.split("\\s+");
			String method = tokens[0];
			String version = (tokens.length > 2) ? tokens[2] : "";
```

```
                if (method.equals("GET")) {
                        String[] fields = tokens[1].split("\\?");
                        String path = fields[0];
                        HashMap<String, String> params = new HashMap<>();
                        if (fields.length > 1) {
                                for (String pair : fields[1].split("\\&")) {
                                        String[] kv = pair.split("=");
                                        params.put(kv[0], kv[1]);
                                }
                        }

                        String rsp = handler.handleGet(path, params);
                        if (rsp != null) {
                                byte[] theData = rsp.getBytes("UTF-8");
                                if (version.startsWith("HTTP/")) { // send a MIME header
                                        sendHeader(out, "HTTP/1.0 200 OK", "text/html",
theData.length);
                                }

                                // send data; it may be an image or other binary data
                                // so use the underlying output stream
                                // instead of the writer
                                raw.write(theData);
                                raw.flush();
                        } else { // can't find the file
                                String body = new
StringBuilder("<HTML>\r\n").append("<HEAD><TITLE>File Not Found</TITLE>\r\n")
```

```
        .append("</HEAD>\r\n").append("<BODY>")
                                                .append("<H1>HTTP Error 404: File Not
Found</H1>\r\n").append("</BODY></HTML>\r\n")
                                                .toString();
                                        if (version.startsWith("HTTP/")) { // send a MIME header
                                                sendHeader(out, "HTTP/1.0 404 File Not Found",
"text/html; charset=utf-8", body.length());
                                        }
                                        out.write(body);
                                        out.flush();
                                }
                        } else { // method does not equal "GET"
                                String body = new
StringBuilder("<HTML>\r\n").append("<HEAD><TITLE>Not Implemented</TITLE>\r\n")

        .append("</HEAD>\r\n").append("<BODY>").append("<H1>HTTP Error 501: Not
Implemented</H1>\r\n")
                                                .append("</BODY></HTML>\r\n").toString();
                                if (version.startsWith("HTTP/")) { // send a MIME header
                                        sendHeader(out, "HTTP/1.0 501 Not Implemented",
"text/html; charset=utf-8", body.length());
                                }
                                out.write(body);
                                out.flush();
                        }
                } catch (SocketException ex) {
```

```
                    logger.warn("JHTTP: {} disconnected",
connection.getRemoteSocketAddress());
                } catch (Throwable ex) {
                    logger.warn("JHTTP: {} disconnected",
connection.getRemoteSocketAddress(), ex);
                } finally {
                    try {
                        connection.close();
                    } catch (IOException ex) {
                    }
                }
            }

            private void sendHeader(Writer out, String responseCode, String contentType, int
length) throws IOException {
                out.write(responseCode + "\r\n");
                Date now = new Date();
                out.write("Date: " + now + "\r\n");
                out.write("Server: JHTTP2\r\n");
                out.write("Content-length: " + length + "\r\n");
                out.write("Content-type: " + contentType + "\r\n\r\n");
                out.flush();
            }
}
```

HttpCommands.java

```
package ece448.iot_sim;
```

```
import java.util.List;
import java.util.Map;
import java.util.TreeMap;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import ece448.iot_sim.http_server.RequestHandler;

public class HTTPCommands implements RequestHandler {

	// Use a map so we can search plugs by name.
	private final TreeMap<String, PlugSim> plugs = new TreeMap<>();

	public HTTPCommands(List<PlugSim> plugs) {
		for (PlugSim plug: plugs)
		{
			this.plugs.put(plug.getName(), plug);
		}
	}

	@Override
	public String handleGet(String path, Map<String, String> params) {
		// list all: /
		// do switch: /plugName?action=on|off|toggle
		// just report: /plugName
```

```
logger.info("HTTPCmd {}: {}", path, params);

if (path.equals("/"))
{
        return listPlugs();
}

PlugSim plug = plugs.get(path.substring(1));
if (plug == null)
        return null; // no such plug

String action = params.get("action");
if (action == null)
        return report(plug);

// P2: add your code here, modify the next line if necessary
if (action.equals("on")) {
        plug.switchOn();
        return report(plug);
}

else if (action.equals("off")) {
        plug.switchOff();
        return report(plug);
}
```

```
        else if (action.equals("toggle")) {
                plug.toggle();
                return report(plug);
        }

        else {
                return report(plug);
        }
}

protected String listPlugs() {
        StringBuilder sb = new StringBuilder();

        sb.append("<html><body>");
        for (String plugName: plugs.keySet())
        {
                sb.append(String.format("<p><a href='/%s'>%s</a></p>",
                        plugName, plugName));
        }
        sb.append("</body></html>");

        return sb.toString();
}

protected String report(PlugSim plug) {
```

```
                String name = plug.getName();
                return String.format("<html><body>"
                        +"<p>Plug %s is %s.</p>"
                        +"<p>Power reading is %.3f.</p>"
                        +"<p><a href='/%s?action=on'>Switch On</a></p>"
                        +"<p><a href='/%s?action=off'>Switch Off</a></p>"
                        +"<p><a href='/%s?action=toggle'>Toggle</a></p>"
                        +"</body></html>",
                        name,
                        plug.isOn()? "on": "off",
                        plug.getPower(), name, name, name);
        }

        private static final Logger logger = LoggerFactory.getLogger(HTTPCommands.class);
}
```

Main.java

```
package ece448.iot_sim;

import java.io.File;
import java.util.ArrayList;

import com.fasterxml.jackson.databind.ObjectMapper;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import ece448.iot_sim.http_server.JHTTP;

import org.eclipse.paho.client.mqttv3.IMqttClient;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;

public class Main implements AutoCloseable {
	@Autowired
	private final MqttClient mqttClient;
	private final JHTTP http;
	public static void main(String[] args) throws Exception {
		// load configuration file
		String configFile = args.length > 0 ? args[0] : "simConfig.json";
		SimConfig config = mapper.readValue(new File(configFile), SimConfig.class);
		logger.info("{}: {}", configFile, mapper.writeValueAsString(config));

		try (Main m = new Main(config))
		{
			// loop forever
			for (;;)
			{
```

```
                    Thread.sleep(60000);
                }
            }
        }

        public Main(SimConfig config) throws Exception {
            // create plugs
            ArrayList<PlugSim> plugs = new ArrayList<>();
            for (String plugName: config.getPlugNames()) {
                plugs.add(new PlugSim(plugName));
            }

            // start power measurements
            MeasurePower measurePower = new MeasurePower(plugs);
            measurePower.start();

            // start HTTP commands
            this.http = new JHTTP(config.getHttpPort(), new HTTPCommands(plugs));
            this.http.start();

            //MQTT setup
            mqttClient = new MqttClient(config.getMqttBroker(),
config.getMqttClientId());
            try {
                MqttConnectOptions options = new MqttConnectOptions();
                options.setCleanSession(true);
```

```
                options.setAutomaticReconnect(true);
                options.setConnectionTimeout(10);
                options.setKeepAliveInterval(60);
                mqttClient.connect(options);
                logger.info("MQTT Connected to broker: {}", config.getMqttBroker());
            } catch (MqttException e) {
                logger.error("Failed to connect to MQTT broker: {}", e.getMessage());
                throw e;
            }

            MqttCommands mqttCmd = new MqttCommands(plugs,
config.getMqttTopicPrefix());
            mqttClient.setCallback(new MqttCallback() {
                @Override
                public void connectionLost(Throwable cause) {
                    logger.warn("Connection Lost: {}", cause.getMessage());
                    // Attempt to reconnect
                    while (!mqttClient.isConnected()) {
                        try {
                            logger.info("Attempting to reconnect to MQTT
broker...");
                            mqttClient.reconnect();
                            logger.info("Successfully reconnected to MQTT
broker");
                            break;
                        } catch (MqttException e) {
```

```
                                        logger.error("Failed to reconnect: {}",
e.getMessage());
                                        try {
                                                Thread.sleep(5000); // Wait 5 seconds
before retrying
                                        } catch (InterruptedException ie) {
                                                Thread.currentThread().interrupt();
                                                break;
                                        }
                                }
                        }
                        @Override
                        public void messageArrived(String topic, MqttMessage message)
throws Exception {
                                logger.info("Recieved MQTT Message on topic : " + topic);
                                mqttCmd.handleMessage(topic, message);
                        }
                        @Override
                        public void deliveryComplete(IMqttDeliveryToken token) {
                                logger.info("Delivery complete for token: " + token);
                        }
                });

                mqttClient.subscribe(mqttCmd.getTopic(), 0);

                //Publishing the updates
```

```
            //MqttUpdates mqttUpd = new MqttUpdates(config.getMqttTopicPrefix(),
mqttClient);


            for (PlugSim plug : plugs) {
                plug.addObserver((name, key, value) -> {
                    try {
                        MqttUpdates mqttUpd = new
MqttUpdates(config.getMqttTopicPrefix(), mqttClient);
                        String topic = mqttUpd.getTopic(name, key);
                        MqttMessage message = mqttUpd.getMessage(value);
                        if (mqttClient.isConnected()) {
                            mqttClient.publish(topic, message);
                        } else {
                            logger.warn("MQTT client not connected,
skipping publish for {} {} {}", name, key, value);
                        }
                    } catch (Exception e) {
                        logger.error("Failed to publish {} {} {}", name, key, value,
e);
                    }
                });
            }
        }

        @Override
        public void close() throws Exception {
            http.close();
```

```
                if (mqttClient != null && mqttClient.isConnected()) {
                        mqttClient.disconnect();
                        mqttClient.close();
                }
        }

        private static final ObjectMapper mapper = new ObjectMapper();
        private static final Logger logger = LoggerFactory.getLogger(Main.class);
}
```

MeasurePower.java

```
package ece448.iot_sim;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * Take power measurements every 1 second.
 */
public class MeasurePower {

        private final List<PlugSim> plugs;

        public MeasurePower(List<PlugSim> plugs) {
                this.plugs = plugs;
```

```
}

public void start() {
	Thread t = new Thread(() -> {
		try
		{
			for (;;)
			{
				measureOnce();
			}
		}
		catch (Throwable th)
		{
			logger.error("Power: exit {}", th.getMessage(), th);
			System.exit(-1);
		}
	});

	// make sure this thread won't block JVM to exit
	t.setDaemon(true);

	// start measuring
	t.start();
}

/**
```

```
	 * Measure and wait 1s.
	 */
	protected void measureOnce() {
		try
		{
			for (PlugSim plug: plugs)
			{
				plug.measurePower();
			}

			Thread.sleep(1000);
		}
		catch (InterruptedException e)
		{
		}
	}

	private static final Logger logger = LoggerFactory.getLogger(MeasurePower.class);
}
```

MqttCommands.java

```
package ece448.iot_sim;

import java.util.List;
import java.util.TreeMap;

import org.eclipse.paho.client.mqttv3.MqttMessage;
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MqttCommands {
   protected final TreeMap<String, PlugSim> plugs;
   private final String topicPrefix;
   private static final Logger logger = LoggerFactory.getLogger(MqttCommands.class);

   public MqttCommands(List<PlugSim> plugs, String topicPrefix) {
      this.plugs = new TreeMap<>();
      for (PlugSim plug : plugs) {
         this.plugs.put(plug.getName(), plug);
      }
      this.topicPrefix = topicPrefix;
   }

   public String getTopic() {
      return topicPrefix + "/action/#";
   }

   // Handling incoming MQTT messages
   public void handleMessage(String topic, MqttMessage message) {
      try {
         String[] parts = topic.split("/");
         if (parts.length < 2) {
            logger.warn("Invalid topic format: {}", topic);
```

```
            return;
        }

        String plugName = parts[parts.length-2];
    String action = parts[parts.length-1];

    PlugSim plug = plugs.get(plugName);
    if (plug != null) {
        switch (action) {
            case "on":
                plug.switchOn();
                break;
            case "off":
                plug.switchOff();
                break;
            case "toggle":
                plug.toggle();
                break;
            default:
                logger.warn("Unknown action: {}", action);
        }
    }
} catch (Exception e) {
    logger.error("Error handling MQTT message: {}", e.getMessage(), e);
    }
}
```

```
    public void addPlug(PlugSim plug) {
        plugs.put(plug.getName(), plug);
    }
}
```

MqttUpdates.java

```
package ece448.iot_sim;

import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MqttUpdates {
    private final String topicPrefix;
    private final MqttClient mqttClient;
    private static final Logger logger = LoggerFactory.getLogger(MqttUpdates.class);

    public MqttUpdates(String topicPrefix, MqttClient mqttClient) {
        this.topicPrefix = topicPrefix;
        this.mqttClient = mqttClient;
    }

    // Generating topic for given plug and key
    public String getTopic(String name, String key) {
        return topicPrefix + "/update/" + name + "/" + key;
    }
```

```
    // Generating MQTT message for given value
    public MqttMessage getMessage(String value) {
        MqttMessage msg = new MqttMessage(value.getBytes());
        msg.setRetained(true);
        return msg;
    }

    // Publishing update to the MQTT broker
    public void publishUpdate(String name, String key, String value) {
        try {
            String topic = getTopic(name, key);
            MqttMessage msg = getMessage(value);
            mqttClient.publish(topic, msg);
            logger.info("Published update: {} -> {}", topic, value);
        } catch (Exception e) {
            logger.error("Failed to publish update for {} {} {}", name, key, value, e);
        }
    }
}
```

PlugSim.java

```
package ece448.iot_sim;

import java.util.List;

import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;
import java.util.List;
import java.util.ArrayList;

/**
 * Simulate a smart plug with power monitoring.
 */
public class PlugSim {

	private final String name;
	private boolean on = false;
	private double power = 0; // in watts

	public PlugSim(String name) {
		this.name = name;
	}

	/**
	 * No need to synchronize if read a final field.
	 */
	public String getName() {
		return name;
	}

	public static interface Observer {
		void update (String name, String key, String value);
```

```
}

private final List<Observer> observers = new ArrayList<>();

public void addObserver(Observer observer) {
        observers.add(observer);
        observer.update(name, "state", on ? "on" : "off" );
        observer.update(name, "power", String.format("%.3f", power));
}
/**
 * Switch the plug on.
 */
synchronized public void switchOn() {
        // P1: add your code here
        on = true;
        measurePower();
        notifyObservers("state", "on");
}

/**
 * Switch the plug off.
 */
synchronized public void switchOff() {
        // P1: add your code here
        on = false;
        notifyObservers("state", "off");
```

```
}

/**
 * Toggle the plug.
 */
synchronized public void toggle() {
	// P1: add your code here
	on = !on;
	notifyObservers("state", on ? "on" : "off");
	if(on) {
		measurePower();
		notifyObservers("power", String.format("%.3f", power));
	}
}

/**
 * Measure power.
 */
synchronized public void measurePower() {
	if (!on) {
		updatePower(0);
		return;
	}

	// a trick to help testing
	if (name.indexOf(".") != -1)
```

```
        {
                updatePower(Integer.parseInt(name.split("\\.")[1]));
        }
        // do some random walk
        else if (power < 100)
        {
                updatePower(power + Math.random() * 100);
        }
        else if (power > 300)
        {
                updatePower(power - Math.random() * 100);
        }
        else
        {
                updatePower(power + Math.random() * 40 - 20);
        }
        notifyObservers("power", String.format("%.3f", power));
}

private void notifyObservers(String key, String value) {
        for (Observer observer : observers) {
                observer.update(name, key, value);
        }
}

protected void updatePower(double p) {
```

```
            power = p;
            logger.debug("Plug {}: power {}", name, power);
        }

        /**
         * Getter: current state
         */
        synchronized public boolean isOn() {
            return on;
        }

        /**
         * Getter: last power reading
         */
        synchronized public double getPower() {
            return power;
        }

        private static final Logger logger = LoggerFactory.getLogger(PlugSim.class);


}
```

SimConfig.java

```
package ece448.iot_sim;

import java.util.List;
```

```
import com.fasterxml.jackson.annotation.JsonCreator;
import com.fasterxml.jackson.annotation.JsonProperty;

public class SimConfig {

	private final int httpPort;
	private final List<String> plugNames;
	private final String mqttBroker;
	private final String mqttClientId;
	private final String mqttTopicPrefix;

	@JsonCreator
	public SimConfig(
		@JsonProperty(value = "httpPort", required = true) int httpPort,
		@JsonProperty(value = "plugNames", required = true) List<String>
plugNames,
		@JsonProperty(value = "mqttBroker", required = true) String mqttBroker,
		@JsonProperty(value = "mqttClientId", required = true) String mqttClientId,
		@JsonProperty(value = "mqttTopicPrefix", required = true) String
mqttTopicPrefix) {
		this.httpPort = httpPort;
		this.plugNames = plugNames;
		this.mqttBroker = mqttBroker;
		this.mqttClientId = mqttClientId;
		this.mqttTopicPrefix = mqttTopicPrefix;
	}
```

```
	public int getHttpPort() {
		return httpPort;
	}

	public List<String> getPlugNames() {
		return plugNames;
	}

	public String getMqttBroker() {
		return mqttBroker;
	}

	public String getMqttClientId() {
		return mqttClientId;
	}

	public String getMqttTopicPrefix() {
		return mqttTopicPrefix;
	}
}
```