PROYECTO INTEGRADOR
REDES-S.OPERATIVOS

Avance II

PIRO-maniacs

Ariel Arévalo Alvarado B50562
Antonio Badilla Olivas B80874
Geancarlo Rivera Hernández C06516
Jean Paul Chacón González C11993

Logger

```
high_resolution_cloc ::time_point Logger::start{
k high_resolution_cloc ::now()
};k
void Logger::print(const string &message) {
  cout << "[" << duration() << " ms]" << "[INFO]: "
<< message << endl;
}
void Logger::info(const string &message) {
  cout << "[" << duration() << " ms]" << "[INFO]: "
<< message << endl;
}
void Logger::error(const string &message) {
  cout << "[" << duration() << " ms]" << "[ERROR]: "
<< message << endl;
}
void Logger::error(const string &message, const
exception &e) {
  error(message);
  print_exception(e);
}
u_int64_t Logger::duration() {
  return duration_cast<milliseconds>(
high_resolution_cloc ::now() - start).count();
}
```

```
void Logger::print_exception(const exception &e, int
level) {
  cout << "Caused by:   << e.what() << endl;
  try {   "
    rethrow_if_nested(e);
  }
  catch (const exception &ne) {
    print_exception(ne, level + 1);
  }
}
```

Socket

```
void IPv4SslSocket::listen(int queueSize) const {
  if (-1 == Sys::listen(socketFD, queueSize)) {
    throw runtime_error(appendErr("IPv4SslSocket::listen: "));
  }
}

shared_ptr<IPv4SslSocket> IPv4SslSocket::accept() const {
  int newFD{Sys::accept(socketFD, nullptr, nullptr)};
  if (-1 == newFD) {
    throw runtime_error(appendErr("IPv4SslSocket::accept: "));
  }

  return std::make_shared<IPv4SslSocket>(newFD, this);
}
```

Socket

```
void IPv4SslSocket::bind(int port) const {
  struct sockaddr_in host4{
    AF_INET,
    htons(port),
    INADDR_ANY
  };

  if (-1 == Sys::bind(socketFD, reinterpret_cast<sockaddr *>(&host4),
    sizeof(host4))) {
    throw runtime_error(appendErr("IPv4SslSocket::bind: "));
  }
}
```

FigureHttpServer

```
void FigureHttpsServer::start() {
  listener.bind(PORT);

  listener.listen(QUEUE);

  for (int i{0}; i < NUM_WORKERS; ++i) {
    this->workers.emplace_back(&FigureHttpsServer::handleRequests, this);
  }

  Logger::info("Listener certificates: \n" + listener.getCerts());

  Logger::info("Listening.");

  while (true) {
    try {
      auto client{listener.accept()};
      Logger::info("Accepted connection with socket: " + to_string(client->getSocketFD()));
      this->clientQueue.enqueue(client);
    } catch (exception &e) {
      Logger::error("Listener error: ", e);
    }
  }
}
```

```cpp
void FigureHttpsServer::handleRequests() {
  while (true) {
    auto client{clientQueue.dequeue()};
    if (!client) {
      break;
    } else {
      try {
        client->sslAccept();
        string request{client->sslRead()};
        map<string, map<string, string>> parsedRequest{parseHttpRequest(request)};
        string url{parsedRequest["Request-Line"]["URL"]};
        if (!validateUrlFormat(url)) {
          sendHttpsResponse(client, 404, "");
        }
        string body{figureController.getFigureByName(getLastPath(url))};
        map<string, string> headers{parsedRequest["Headers"]};
        if (body.empty()) {
          sendHttpsResponse(client, 404, body);
        } else {
          sendHttpsResponse(client, 200, body);
        }
      } catch (exception &e) {
        Logger::error("Client error: ", e);
        sendHttpResponse(client, 500, "");
      }
      Logger::info("Handled connection with socket: " + to_string(client->getSocketFD()));
    }
  }
}
```

```
string FigureHttpsServer::generateHttpResponse(int statusCode, const string &body) {
  string statusMessage;
  switch (statusCode) {
    case 200: statusMessage = "OK";
      break;
    case 404: statusMessage = "Not Found";
      break;
    case 500: statusMessage = "Internal Error";
      break;
    default:throw invalid_argument("Unknown HTTP status code");
  }

  string response =
      "HTTP/1.1 " + to_string(statusCode) + " " + statusMessage + "\r\n";
  response += "Content-Type: text/html\r\n";
  response += "Content-Length: " + to_string(body.size()) + "\r\n";
  response += "Connection: close\r\n";
  response += "\r\n" + body;

  return response;
}
```

```cpp
map<string, map<string, string>> FigureHttpsServer::parseHttpRequest(
    const string &request) {
  istringstream requestStream(request);
  string method, url, version;
  requestStream >> method >> url >> version;

  map<string, map<string, string>> result{
      {"Request-Line", {{"Method", method}, {"URL", url}, {"Version", version}}},
      {"Headers", parseHeaders(requestStream)}};

  return result;
}
```

```cpp
map<string, string> FigureHttpsServer::parseHeaders(istringstream &stream) {
  map<string, string> headers;
  string line;
  while (getline(stream, line, '\n')) { // Specify '\n' as the delimiter
    // Check if line is not just '\r'
    if (line.size() > 1) {
      // Remove the trailing '\r' if it exists
      if (line.back() == '\r') {
        line.pop_back();
      }
      auto colonPos = line.find(':');
      if (colonPos != string::npos) {
        string name = line.substr(0, colonPos);
        string value = line.substr(colonPos + 2);  // Skip the colon and the space after it
        headers[name] = value;
      }
    }
  }
  return headers;
}
```

```
void FigureHttpsServer::sendHttpsResponse(const std::shared_ptr<IPv4SslSocket> &client,
  int statusCode, const std::string &body) {
  string response =
    FigureHttpsServer::generateHttpResponse(statusCode, body);
  client->sslWrite(response);
}
```

```
void FigureHttpsServer::stop() {
  for (auto &worker : workers) {
    clientQueue.enqueue(nullptr);
  }

  for (auto &handler : workers) {
    handler.join();
  }
}
```

FigureHtmlRepository

```
string FigureHtmlRepository::findByName(const string& name) const {
  string filenameStr{getResourcePath().c_str()};
  filenameStr += "/" + name + ".html";
  struct stat sb;
  if (!std::filesystem::exists(filenameStr)) {
    throw std::invalid_argument("Resource file not found");
  }
  std::ifstream figureFile(filenameStr);
  std::stringstream buffer;
  buffer << figureFile.rdbuf();
  figureFile.close();
  return buffer.str();
}
```

```
std::filesystem::path FigureHtmlRepository::getResourcePath() const {
  std::filesystem::path currentFilePath(__FILE__);
  std::filesystem::path rootPath = currentFilePath.parent_path();
  std::filesystem::path resRootPath = (rootPath.parent_path()).parent_path();
  resRootPath = resRootPath.parent_path();
  std::filesystem::path resourcePath = resRootPath / "res";
  return resourcePath;
}
```

FigureController

```
string FigureController::getFigureByName(const string &name) const {
  try {
    string figureHtml{figureRepository.findByName(name)};
    return figureHtml;
  }
  catch(invalid_argument &ia) {
    return "";
  }
}
```

LegoServer

```
int main(int argc, char *argv[]) {
    Logger::initialize();
    if(argc < 2) {
      Logger::error("Missing certificate path.");
      exit(1);
    }
    string certPath{argv[1]};
    try {
        FigureHttpsServer server{certPath};
        signalHandle();
        server.start();
    } catch (exception const &e) {
        Logger::error("Server has crashed.", e);
        exit(1);
    }
}
void signalAction(int signum) {
    Logger::info("Exiting.");
    exit(signum);
}
void signalHandle() {
    signal(SIGINT, signalAction);
    signal(SIGTERM, signalAction);
}
```

Gracias!