Server Design

The server was implemented as a Java servlet deployed by a `Tomcat` server. The servlet logic is contained in the `SwipeServlet` class of the `a2_server` project's `twinder` package. The `SwipeServlet` class extends the `HttpServlet` class and therefore overrides the `doGet()` and `doPost()` methods. Only the `doPost()` method is relevant to this assignement. When it is called, the servlet acts as a publisher to the `RabbitMQ` broker running on a separate Ubuntu EC2 instance.

The `SwipeServlet` has an attribute `public RmqConnectionHandler collectionHandler` which is a wrapper for the servlet's connection to the `RabbitMQ` broker. The `RmqConnectionHandler` is part of the `rmq` package and presents a channel pool api to the `SwipeServlet` via its `borrowChannel()` and `returnChannel()` methods. The channel pool itself is implemented using a `BlockingQueue<Channel>`. An `RmqConnectionHandler` instance is created by calling the static method `RmqConnectionHandler.createConnectionHandler()` with the following arguments: 1) a `Connection` instance; 2) an `int` for the number of channels that will be established on the `Connection`.

When the `init()` method of the `SwipeServlet` is called, it creates and assigns the `RmqConnectionHandler` to its `connectionHandler` instance variable. It then uses the `connectionHandler`'s `declareExchange()` method to declare the durable `fanout` exchange on the `RabbitMQ` broker that it will publish messages to. Declaring the exchange each time the servlet is initiated ensures that if the exchange happened to be removed by the broker then it will be recreated before any messages are published to it.

I chose to initialize the `SwipeServlet`'s `connectionHandler` with a channel pool of 100 channels. This is because during a1 I found that the optimal threadcount for a single servlet is around 100 threads. Therefore, having a 100 channel threadpool would follow the one-channel-per-thread design that is optimal for `RabbitMQ` broker-based systems.

When the `Tomcat` server receives an HTTP post request from the client, the `SwipeServlet`'s `doPost()` method is called. It first validates the given `HttpServletRequest` object's url path. If the request's url path was null, empty, missing necessary parameters, or had invalid parameters then the response code is set to `HttpServletResponse.SC_NOT_FOUND` (i.e. HTTP 404), and the client is informed that the path is invalid and the function returns.

After validating the path, a `Channel` to the `RabbitMQ` broker is borrowed by calling the `connectionHandler.borrowChannel()` method. Then the request's json body is read into a `String` body and is parsed into a `PostRequestJson` object using the `Gson` api. The

`PostRequestJson` object is defined as a static nested class within the `SwipeServlet` and is used as a wrapper for easy validation of the request's json body. The `PostRequestJson` class has two `int` fields and one `String` field for the `swiper`, `swipee`, and `comment` fields respectively. If any of the request's json fields are invalid the response status is set to `HttpServletResponse.SC_BAD_REQUEST` (i.e. HTTP 400) and and the client is informed that there was an issue with their json payload.

Once the json body is validated, it is determined whether the swipe was a like (i.e. if the request's url path ends with `/right` in which case `liked=true`) or a dislike (i.e. the url path ends with `/left` in which case `like=false`). Then the `swiper`, `swipee` and `liked` values are concatenated into a `String` message (formatted as a json) which is then published to the exchange using the borrowed `Channel`'s `basicPublish()` method. If there was no issue in publishing that message, then the response code is set to `HttpServletResponse.SC_OK` (i.e. HTTP 200) and the client is informed that the write was successful.

After the publisher logic is complete the borrowed `Channel` is returned to the pool by calling the `connectionHandler.returnChannel()` method and passing the borrowed channel to it. This allows the shared `Channel` resources to then be utilized by another thread.