Refactoring QLever’s ‘Server‘

How to Split a Long-Lived Monolithic Class in a Live Codebase

Tomáš Dámek Bachelor Thesis Presentation 22.09.2026

The Demo Setup

- Toy data: one triple — :student1 :name "Alex"
- One pending change: — :student1 :hasGrade "F"
- After exam review one point saves the student from failing
  - fix it in one request: clear the one pending value, insert the right one

- Question: does this leave a clean state — F gone, E in?

What did the demo just expose?

```
593   } else if (auto cmd = checkParameter("cmd", "load-materialized-view")) {
594     requireValidAccessToken("load-materialized-view");
595     logCommand(cmd, "explicitly load materialized view");
596
597     // Extract materialized view name parameter.
598     auto name = ad_utility::url_parser::getParameterCheckAtMostOnce(
599         parameters, "view-name");
600     AD_CONTRACT_CHECK(name.has_value());
601
602     auto qec = qlever().createQueryExecutionContext(indexAndViews);
603     indexAndViews->materializedViewsManager_.loadView(name.value(), qec.get());
604
605     // Construct simple response JSON.
606     nlohmann::json json{{"materialized-view-loaded", name.value()}};
607     response = createJsonResponse(json, request);
608
609     // Prevent regular query processing by removing the query from the request.
610     parsedHttpRequest.operation_ = None{};
611   } else if (auto cmd = checkParameter("cmd", "delete-materialized-view")) {
612     requireValidAccessToken("delete-materialized-view");
613     logCommand(cmd, "delete materialized view");
614
615     // Extract materialized view name parameter.
616     auto name = ad_utility::url_parser::getParameterCheckAtMostOnce(
617         parameters, "view-name");
618     AD_CONTRACT_CHECK(name.has_value(),
619                       "Deleting a materialized view requires a name to be set "
620                       "via the 'view-name' parameter");
621
622     // Snapshot again instead of using `indexAndViews` from the beginning of
623     // this function (see `clear-delta-triples` above for the same pattern), so
624     // that we delete the view from the index that is currently being served and
625     // not from a stale one that a concurrent rebuild has swapped out in the
626     // meantime. Deleting from a stale manager is not unsafe (the rebuild called
627     // `MaterializedViewsManager::retireOnDiskFiles` on it, which makes
628     // `deleteView` throw), it would just needlessly fail.
629     indexAndViewsSnapshot()->materializedViewsManager_.deleteView(name.value());
630
631     // Construct simple response JSON.
632     nlohmann::json json{{"materialized-view-deleted", name.value()}};
633     response = createJsonResponse(json, request);
634
635     // Prevent regular query processing by removing the query from the request.
636     parsedHttpRequest.operation_ = None{};
637   }
```

no trailing "else" — an unrecognized cmd= value falls through silently

```
    Awaitable<void> Server::process(RequestT& request, ResponseT&& send) {
  using namespace ad_utility::httpUtils;
  using namespace responseJson;
  // Acquire the current index and the materialized views manager exactly once
  // for the whole request, under a single read lock. This way a concurrent
  // rebuild that swaps both in cannot make different helpers observe a
  // mismatched (index, manager) pair.
  auto indexAndViews = indexAndViewsSnapshot();
  auto& index = indexAndViews->index_;

  // Log some basic information about the request. Start with an empty line so
  // that in a low-traffic scenario (or when the query processing is very fast),
  // we have one visual block per request in the log.
  std::string_view contentType = request.base()[http::field::content_type];
  AD_LOG_INFO << std::endl;
  AD_LOG_INFO << "Request received via " << request.method()
              << (contentType.empty()
                      ? absl::StrCat(", no content type specified")
                      : absl::StrCat(", content type \"", contentType, "\""))
              << std::endl;

  // Start timing.
  ad_utility::Timer requestTimer{ad_utility::Timer::Started};

  // Parse the path and the URL parameters from the given request. Works for GET
  // requests as well as the two kinds of POST requests allowed by the SPARQL
  // standard, see method `getUrlPathAndParameters`.
  auto parsedHttpRequest = SparqlProtocol::parseHttpRequest(request);
  const auto& parameters = parsedHttpRequest.parameters_;

  // We always want to call `Server::checkParameter` with the same first
  // parameter.
  auto checkParameter = absl::bind_front(
      &ad_utility::url_parser::checkParameter, std::cref(parameters));

  // Check the access token. If an access token is provided and the check fails,
  // throw an exception and do not process any part of the query (even if the
  // processing had been allowed without access token).
  bool accessTokenOk = checkAccessToken(parsedHttpRequest.accessToken_);
  auto requireValidAccessToken =
      [&accessTokenOk](const std::string& actionName) {
        if (!accessTokenOk) {
          throw HttpError(http::status::forbidden,
                          absl::StrCat(actionName,
                                       " requires a valid access token but no "
                                       "access token was provided"));
        }
      };

  // Process all URL parameters known to QLever. If there is more than one,
  // QLever processes all of them, but only returns the result from the last
  // one. In particular, if there is a "query" parameter, it will be processed
  // last and its result returned.
  //
  // Some parameters require that "access-token" is set correctly. If not, that
  // parameter is ignored.
  std::optional<http::response<streamable_body>> response;

  // Execute commands (URL parameter with key "cmd").
  auto logCommand = [](const std::optional<std::string_view>& cmd,
                       std::string_view actionMsg) {
    AD_LOG_INFO << "Processing command \"" << cmd.value() << "\"" << ": "
                << actionMsg << std::endl;
  };
  if (auto cmd = checkParameter("cmd", "stats")) {
    logCommand(cmd, "get index statistics");
    response = createJsonResponse(composeIndexStats(index), request);
  } else if (auto cmd = checkParameter("cmd", "cache-stats")) {
    logCommand(cmd, "get cache statistics");
    response = createJsonResponse(
        composeCacheStats(cache(), namedResultCache()));
  } else if (auto cmd = checkParameter("cmd", "clear-cache")) {
    logCommand(cmd, "clear the cache (unpinned elements only)");
    cache().clearUnpinnedOnly();
    response = createJsonResponse(
        composeCacheStats(cache(), namedResultCache()), request);
  } else if (auto cmd = checkParameter("cmd", "clear-cache-complete")) {
    requireValidAccessToken("clear-cache-complete");
    logCommand(cmd, "clear cache completely (including unpinned elements)");
    cache().clearAll();
    response = createJsonResponse(
        composeCacheStats(cache(), namedResultCache()), request);
  } else if (auto cmd = checkParameter("cmd", "clear-named-cache")) {
    requireValidAccessToken("clear-named-cache");
    logCommand(cmd, "clear the cache for named results");
    namedResultCache().clear();
    response = createJsonResponse(
        composeCacheStats(cache(), namedResultCache()), request);
  } else if (auto cmd = checkParameter("cmd", "clear-delta-triples")) {
    requireValidAccessToken("clear-delta-triples");
    logCommand(cmd, "clear delta triples");
    // The function requires a SharedCancellationHandle, but the operation is
    // not cancellable.
    auto handle = std::make_shared<ad_utility::CancellationHandle<>>();
    // We don't directly `co_await` because of lifetime issues (bugs) in the
    // Conan setup.
    auto coroutine = computeInNewThread(
        updateThreadPool_,
        [this] {
          // Snapshot here, on the (single-threaded) `updateThreadPool_`, so we
          // modify the currently active index and not a stale one that a
          // concurrent rebuild may have swapped out (whose changes would be
          // lost).
          auto snapshot = indexAndViewsSnapshot();
          return snapshot->index_.deltaTriplesManager()
              .modify<DeltaTriplesCount>([](auto& deltaTriples) {
                deltaTriples.clear();
                return deltaTriples.getCounts();
              });
        },
        handle);
    auto countAfterClear = co_await std::move(coroutine);
    response = createJsonResponse(json(countAfterClear), request);
  } else if (auto cmd = checkParameter("cmd", "vacuum-delta-triples")) {
    requireValidAccessToken("vacuum-delta-triples");
    logCommand(cmd, "vacuum (remove redundant) delta triples");

    auto handle = std::make_shared<ad_utility::CancellationHandle<>>();
    std::optional<TimeLimit> timeLimit =
        co_await verifyUserSubmittedQueryTimeout(
            checkParameter("timeout", std::nullopt), accessTokenOk, request,
            send);
    if (!timeLimit.has_value()) {
      // If the optional is empty, this indicates an error response has been
      // sent to the client already. We can stop here.
      co_return;
    }
    auto cancelTimeoutOnDestruction =
        cancelAfterDeadline(handle, timeLimit.value());

    auto coroutine = computeInNewThread(
        updateThreadPool_,
        [this, handle] {
          // Snapshot on the update thread (see `clear-delta-triples` above).
          auto snapshot = indexAndViewsSnapshot();
          return snapshot->index_.deltaTriplesManager().modify<nlohmann::json>(
              [handle](auto& deltaTriples) {
                return deltaTriples.vacuum(handle);
              });
        },
        handle);
    auto vacuumStats = co_await std::move(coroutine);
    response = createJsonResponse(vacuumStats, request);
  } else if (auto cmd = checkParameter("cmd", "get-settings")) {
    logCommand(cmd, "get server settings");
    response = createJsonResponse(
        json(globalRuntimeParameters.rlock()->toMap()), request);
  } else if (auto cmd = checkParameter("cmd", "get-index-id")) {
    logCommand(cmd, "get index ID");
    response =
        createOkResponse(index.getIndexId(), request, MediaType::textPlain);
  } else if (auto cmd = checkParameter("cmd", "dump-active-queries")) {
    requireValidAccessToken("dump-active-queries");
    logCommand(cmd, "dump active queries");
    auto json = nlohmann::json::object();
    for (auto& [key, value] : queryRegistry_.getActiveQueries()) {
      json[nlohmann::json(key)] = std::move(value);
    }
    response = createJsonResponse(json, request);
  } else if (auto cmd = checkParameter("cmd", "rebuild-index")) {
    requireValidAccessToken("rebuild-index");
    logCommand(cmd, "rebuilding index");
    auto config = co_await rebuildIndexUnlessInProgress(
        checkParameter("rebuild-tmp-dir", std::nullopt),
        checkParameter("rebuild-previous-index-dir", std::nullopt));
    if (config.has_value()) {
      response = createJsonResponse(config->successResponseAsJson(), request);
    } else {
      response = createHttpResponseFromString(
          "Another rebuild is currently in progress!",
          http::status::too_many_requests, request, MediaType::textPlain);
    }
  } else if (auto cmd = checkParameter("cmd", "write-materialized-view")) {
    requireValidAccessToken("write-materialized-view");
    logCommand(cmd, "write materialized view");

    // Extract name parameter for materialized view.
    auto name = ad_utility::url_parser::getParameterCheckAtMostOnce(
        parameters, "view-name");
    AD_CONTRACT_CHECK(name.has_value(),
                      "Writing a materialized view requires a name to be set "
                      "via the 'view-name' parameter");
    AD_CONTRACT_CHECK(name.value() != "",
                      "The name for the view may not be empty");

    // Extract query body.
    auto query = std::visit(
        [](const auto& op) -> Query {
          using T = std::decay_t<decltype(op)>;
          if constexpr (std::is_same_v<T, Query>) {
            return op;
          } else {
        static_assert(
            ad_utility::SameAsAny<T, Update, GraphStoreOperation, None>);
        throw std::runtime_error(
            "Action 'write-materialized-view' requires a 'SELECT' query.");
      }
    },
    parsedHttpRequest.operation_);

  // Extract time limit.
  auto timeLimit = co_await verifyUserSubmittedQueryTimeout(
      checkParameter("timeout", std::nullopt), accessTokenOk, request, send);
  AD_CONTRACT_CHECK(timeLimit.has_value(), "Missing timeout");

  // Call `QLever::writeMaterializedView` with the extracted parameters. This
  // assumes that the access token has already been checked. Note that storing
  // the coroutine in a variable first and then awaiting it is required due to
  // lifetime issues on certain compilers.
  auto cancellationHandle =
      std::make_shared<ad_utility::CancellationHandle<>>();
  auto coroutine = computeInNewThread(
      queryThreadPool_,
      [name, query, requestTimer, cancellationHandle, timeLimit,
       this]() mutable {
        qlever().writeMaterializedView(
            name.value(), std::move(query.query_), query.datasetClauses_,
            std::move(cancellationHandle), timeLimit.value(), requestTimer);
      },
      cancellationHandle);
  co_await std::move(coroutine);

  // Construct simple response JSON.
  nlohmann::json json{{"materialized-view-written", name.value()}};
  response = createJsonResponse(json, request);

  // Prevent regular query processing by removing the query from the request.
  parsedHttpRequest.operation_ = None{};
} else if (auto cmd = checkParameter("cmd", "load-materialized-view")) {
  requireValidAccessToken("load-materialized-view");
  logCommand(cmd, "explicitly load materialized view");

  // Extract materialized view name parameter.
  auto name = ad_utility::url_parser::getParameterCheckAtMostOnce(
      parameters, "view-name");
  AD_CONTRACT_CHECK(name.has_value());

  auto qec = qlever().createQueryExecutionContext(indexAndViews);
  indexAndViews->materializedViewsManager_.loadView(name.value(), qec.get());

  // Construct simple response JSON.
  nlohmann::json json{{"materialized-view-loaded", name.value()}};
  response = createJsonResponse(json, request);

  // Prevent regular query processing by removing the query from the request.
  parsedHttpRequest.operation_ = None{};
} else if (auto cmd = checkParameter("cmd", "delete-materialized-view")) {
  requireValidAccessToken("delete-materialized-view");
  logCommand(cmd, "delete materialized view");

  // Extract materialized view name parameter.
  auto name = ad_utility::url_parser::getParameterCheckAtMostOnce(
      parameters, "view-name");
  AD_CONTRACT_CHECK(name.has_value(),
                    "Deleting a materialized view requires a name to be set "
                    "via the 'view-name' parameter");

  // Snapshot again instead of using `indexAndViews` from the beginning of
  // this function (see `clear-delta-triples` above for the same pattern), so
  // that we delete the view from the index that is currently being served and
  // not from a stale one that a concurrent rebuild has swapped out in the
  // meantime. Deleting from a stale manager is not unsafe (the rebuild called
  // `MaterializedViewsManager::retireOnDiskFiles` on it, which makes
  // `deleteView` throw), it would just needlessly fail.
  indexAndViewsSnapshot()->materializedViewsManager_.deleteView(name.value());

  // Construct simple response JSON.
  nlohmann::json json{{"materialized-view-deleted", name.value()}};
  response = createJsonResponse(json, request);

  // Prevent regular query processing by removing the query from the request.
  parsedHttpRequest.operation_ = None{};
}

// Ping with or without message.
if (parsedHttpRequest.path_ == "/ping") {
  if (auto msg = checkParameter("msg", std::nullopt)) {
    AD_LOG_INFO << "Alive check with message \"" << msg.value() << "\""
                << std::endl;
  } else {
    AD_LOG_INFO << "Alive check without message" << std::endl;
  }
  response = createOkResponse("This QLever server is up and running\n",
                              request, MediaType::textPlain);
}

// Prometheus metrics scrape endpoint.
if (parsedHttpRequest.path_ == "/metrics") {
  requireValidAccessToken("metrics");
  if (!metricsReader_) {
    response = createNotFoundResponse(
        "Metrics not enabled (use --enable-metrics)", request);
  } else {
    response = createOkResponse(metricsReader_->getMetricsText(), request,
                                MediaType::textPlain);
  }
}

// Set description of KB index.
if (auto description = checkParameter("index-description", std::nullopt)) {
  requireValidAccessToken("index-description");
  AD_LOG_INFO << "Setting index description to: \"" << description.value()
              << "\"" << std::endl;
  index.setKbName(std::string{description.value()});
  response = createJsonResponse(composeIndexStats(index), request);
}

// Set description of text index.
if (auto description = checkParameter("text-description", std::nullopt)) {
  requireValidAccessToken("text-description");
  AD_LOG_INFO << "Setting text description to: \"" << description.value()
              << "\"" << std::endl;
  index.setTextName(std::string{description.value()});
  response = createJsonResponse(composeIndexStats(index), request);
}

// Set one or several of the runtime parameters.
for (auto key : globalRuntimeParameters.rlock()->getKeys()) {
  if (auto value = checkParameter(key, std::nullopt)) {
    requireValidAccessToken("setting runtime parameters");
    AD_LOG_INFO << "Setting runtime parameter \"" << key << "\""
                << " to value \"" << value.value() << "\"" << std::endl;
    globalRuntimeParameters.wlock()->setFromString(
        key, std::string{value.value()});
    response = createJsonResponse(
        json(globalRuntimeParameters.rlock()->toMap()), request);
  }
}

// Store the QueryExecutionTree outside the lambda, s.t. we have access in
// case of errors to create an informative error message that includes the
// runtime information.
std::optional<PlannedQuery> plannedQuery;
auto visitOperation =
    [&checkParameter, &accessTokenOk, &request, &send, &parameters,
     &requestTimer, &plannedQuery, &indexAndViews, this](
        std::vector<ParsedQuery> operations, std::string operationName,
        const std::string& operationString,
        std::function<bool(const ParsedQuery&)> expectedOperation,
        const std::string msg, SharedTimeTracer tracer) -> Awaitable<void> {
  auto timeLimit = co_await verifyUserSubmittedQueryTimeout(
      checkParameter("timeout", std::nullopt), accessTokenOk, request, send);
  if (!timeLimit.has_value()) {
    // If the optional is empty, this indicates an error response has been
    // sent to the client already. We can stop here.
    co_return;
  }
  // Empty when the header is absent.
  std::string_view clientIp = request.base()["X-Real-IP"];
  ad_utility::websocket::MessageSender messageSender =
      createMessageSender(queryHub_, request, operationString, clientIp);
  // Grab the shared handle before `messageSender` is moved below.
  using enum ad_utility::websocket::QueryStatus;
  auto queryStatus = messageSender.sharedStatus();
  // Outside the `try ... qecPtr` owns the id whose destructor writes the
  // `end` event, so the status must be set before it unwinds.
  // Workaround for a GCC 15/16 bug: the hidden object of a by-value
  // structured binding is not destroyed when the coroutine frame is
  // destroyed while suspended (gcc.gnu.org bug 124584).
  auto preparedOp = prepareOperation(
      operationName, operationString, std::move(messageSender), parameters,
      timeLimit.value(), accessTokenOk, clientIp);
  auto& [makeQec, cancellationHandle, cancelTimeoutOnDestruction] =
      preparedOp;
  try {
    if (!ql::ranges::all_of(operations, expectedOperation)) {
      throw std::runtime_error(absl::StrCat(
          msg, ad_utility::truncateOperationString(operationString)));
    }
    if (ql::ranges::all_of(operations, &ParsedQuery::hasUpdateClause)) {
      metrics_->startedSparqlOperations_->Add(1, {OperationType::update});
      co_await processUpdate(std::move(makeQec), std::move(operations),
                             requestTimer, tracer, cancellationHandle,
                             std::move(request), send, timeLimit.value(),
                             plannedQuery);
    } else {
      AD_CORRECTNESS_CHECK(operations.size() == 1);
      ParsedQuery query = std::move(operations[0]);
      AD_CORRECTNESS_CHECK(query.hasSelectClause() || query.hasAskClause() ||
                           query.hasConstructClause());
      metrics_->startedSparqlOperations_->Add(1, {OperationType::query});
      // Queries run against a consistent snapshot taken at the start of the
      // request, so build the execution context from that snapshot here.
      auto qecPtr = makeQec(indexAndViews);
      co_await processQuery(parameters, std::move(query), requestTimer,
                            cancellationHandle, *qecPtr, std::move(request),
                            send, timeLimit.value(), plannedQuery);
    }
    queryStatus->store(OK);
    co_return;
  } catch (const ad_utility::CancellationException& e) {
    queryStatus->store(e.state() == ad_utility::CancellationState::TIMEOUT
                           ? TIMEOUT
                           : CANCELLED);
    throw;
  }
};
auto visitQuery = [&index, &visitOperation](Query query) -> Awaitable<void> {
  // We need to copy the query string because `visitOperation` below also
  // needs it.
  auto parsedQuery = SparqlParser::parseQuery(
      &index.encodedIriManager(), query.query_, query.datasetClauses_);
  auto dummy = std::make_shared<ad_utility::timer::TimeTracer>("dummy");
  return visitOperation(
      {std::move(parsedQuery)}, "SPARQL query", std::move(query.query_),
      std::not_fn(&ParsedQuery::hasUpdateClause),
      "SPARQL QUERY was requested via the HTTP request, but the "
      "following update was sent instead of an query: ",
      dummy);
};
auto visitUpdate = [&index, &visitOperation, &requireValidAccessToken](
                       Update update) -> Awaitable<void> {
  requireValidAccessToken("SPARQL Update");
  // We need to copy the update string because `visitOperation` below also
  // needs it.
  auto tracer = std::make_shared<ad_utility::timer::TimeTracer>("update");
  tracer->beginTrace("parsing");
  auto parsedUpdates = SparqlParser::parseUpdate(
      index.getBlankNodeManager(), &index.encodedIriManager(), update.update_,
      update.datasetClauses_);
  tracer->endTrace("parsing");
  return visitOperation(
      std::move(parsedUpdates), "SPARQL update", std::move(update.update_),
      &ParsedQuery::hasUpdateClause,
      "SPARQL UPDATE was requested via the HTTP request, but the "
      "following query was sent instead of an update: ",
      tracer);
};
auto visitGraphStore =
    [&request, &visitOperation, &requireValidAccessToken,
     &index](GraphStoreOperation operation) -> Awaitable<void> {
  auto tracer = std::make_shared<ad_utility::timer::TimeTracer>("update");
  tracer->beginTrace("parsing");
  std::vector<GraphStoreOperation> parsedOperations =
      GraphStoreProtocol::transformGraphStoreProtocol(std::move(operation),
                                                      request, index);
  tracer->endTrace("parsing");

  if (ql::ranges::any_of(parsedOperations, &ParsedQuery::hasUpdateClause)) {
    AD_CORRECTNESS_CHECK(
        ql::ranges::all_of(parsedOperations, &ParsedQuery::hasUpdateClause));
    requireValidAccessToken("Update from Graph Store Protocol");
  }

  // Don't check for the `ParsedQuery`s actual type (Query or Update) here
  // because graph store operations can result in both.
  auto trueFunc = [](const ParsedQuery&) { return true; };
  std::string operationString = parsedOperations[0]._originalString;
  return visitOperation(
      std::move(parsedOperations),
      absl::StrCat("Graph Store (", std::string_view{request.method_string()},
                   ")"),
      std::move(operationString), trueFunc, "Unused dummy message", tracer);
};
auto visitNone = [&response, &send, &request](None) -> Awaitable<void> {
  // If there was no "query", but any of the URL parameters processed before
  // produced a `response`, send that now. Note that if multiple URL
  // parameters were processed, only the `response` from the last one is sent.
  if (response.has_value()) {
    return send(std::move(response.value()));
  }

  // At this point, if there is a "?" in the query string, it means that there
  // are URL parameters which QLever does not know or did not process.
  if (request.target().find("?") != std::string::npos) {
    return send(createBadRequestResponse("Unknown query parameters",
                                         std::move(request)));
  }
  // No path matched up until this point, so return 404 to indicate the client
  // made an error and the server will not serve anything else.
  return send(createNotFoundResponse("Unknown path", std::move(request)));
};

co_return co_await processOperation(
    std::move(parsedHttpRequest.operation_),
    ad_utility::OverloadCallOperator{visitQuery, visitUpdate, visitGraphStore,
                                     visitNone},
    std::move(request), send, plannedQuery);
}
```

Why is this problematic?

The code smells story

- The function is too long:
  - NLOC (Number Lines Of Code) = 357
- The function flow is (for humans) hard to follow:
  - CCN (Cyclomatic Complexity Number) = 39
  - Cognitive Complexity = 104

```
void flatIfs(int* x, int& y) {
  if (x[0] > 0) { y += 0; }
  if (x[1] > 0) { y += 1; }
  if (x[2] > 0) { y += 2; }
  if (x[3] > 0) { y += 3; }
  if (x[4] > 0) { y += 4; }
  if (x[5] > 0) { y += 5; }
  if (x[6] > 0) { y += 6; }
  if (x[7] > 0) { y += 7; }
  if (x[8] > 0) { y += 8; }
  if (x[9] > 0) { y += 9; }
  if (x[10] > 0) { y += 10; }
  if (x[11] > 0) { y += 11; }
  if (x[12] > 0) { y += 12; }
  if (x[13] > 0) { y += 13; }
  if (x[14] > 0) { y += 14; }
  if (x[15] > 0) { y += 15; }
  if (x[16] > 0) { y += 16; }
  if (x[17] > 0) { y += 17; }
  if (x[18] > 0) { y += 18; }
  if (x[19] > 0) { y += 19; }
  if (x[20] > 0) { y += 20; }
  if (x[21] > 0) { y += 21; }
  if (x[22] > 0) { y += 22; }
  if (x[23] > 0) { y += 23; }
  if (x[24] > 0) { y += 24; }
  if (x[25] > 0) { y += 25; }
  if (x[26] > 0) { y += 26; }
  if (x[27] > 0) { y += 27; }
  if (x[28] > 0) { y += 28; }
  if (x[29] > 0) { y += 29; }
  if (x[30] > 0) { y += 30; }
  if (x[31] > 0) { y += 31; }
  if (x[32] > 0) { y += 32; }
  if (x[33] > 0) { y += 33; }
  if (x[34] > 0) { y += 34; }
  if (x[35] > 0) { y += 35; }
  if (x[36] > 0) { y += 36; }
  if (x[37] > 0) { y += 37; }
}
```

"I have two very minor requests, other than that this now looks **nice** and **clean** :)"

Johannes Kalmbach PR #2965
(opened by Tomáš Dámek)

Refactor. Review. Repeat.

- Fowler's Refactoring methodology:
  - preserve behavior, one step at a time
- Plan:
  - Find code smells → Refactor → Review & Test → deploy
- Methods:
  - Extract/Move function, Replace Error code with Exception

Refactor. Review. Repeat.

- Approach:
  - Decompose long functions into small, testable units
  - Move HTTP-agnostic engine logic into `Qlever` engine
  - Extract out of `Server` independent, testable helper units
  - Resolving code smells uncovered along the way

“An initial round of suggestions, but I like that you make more of the setup code **testable** and **modular.**”

Johannes Kalmbach PR #3189
(opened by Tomáš Dámek)

The Process Function Refactor

- Central entry point for every incoming HTTP request

- Decides what's being asked:
  - SPARQL query/update, a custom `cmd=` command, or a system endpoint (`/ping`, `/metrics`)

- Before refactoring:
  - Hard to read, inlined logic instead of named functions, duplicated code

Duplicated logic

(token checks, logging)

BEFORE

requireValidAccessToken(...) + logCommand(...) inline in every branch

```
} else if (auto cmd = checkParameter("cmd", "clear-cache-complete")) {
  requireValidAccessToken("clear-cache-complete");
  logCommand(cmd, "clear cache completely (including unpinned elements)");
  cache().clearAll();
  response = createJsonResponse(
      composeCacheStats(cache(), namedResultCache()), request);
} else if (auto cmd = checkParameter("cmd", "clear-named-cache")) {
  requireValidAccessToken("clear-named-cache");
  logCommand(cmd, "clear the cache for named results");
  namedResultCache().clear();
  response = createJsonResponse(
      composeCacheStats(cache(), namedResultCache()), request);
// ... 4 branches elided (clear-delta-triples, vacuum-delta-triples, ...)
} else if (auto cmd = checkParameter("cmd", "dump-active-queries")) {
  requireValidAccessToken("dump-active-queries");
  logCommand(cmd, "dump active queries");
  auto json = nlohmann::json::object();
  // ...
```

AFTER

one metadata table + one checkAndLogCommand() helper

```
struct CommandMeta {
  std::string_view name_;
  std::string_view description_;
  bool requiresAuth_;
  bool supportsOperation_ = false;
};

constexpr std::array commands = {
    // ... stats, cache-stats, clear-cache elided
    CommandMeta{"clear-cache-complete",
                "clear cache completely (including unpinned elements)", true},
    CommandMeta{"clear-named-cache", "clear the cache for named results", true},
    // ... 4 more elided (clear-delta-triples, vacuum-delta-triples, ...)
    CommandMeta{"dump-active-queries", "dump active queries", true},
    // ... 4 more
};

void checkAndLogCommand(std::string_view cmd, bool accessTokenOk,
                        const sparqlOperation::Operation& operation) {
  auto it = ql::ranges::find(commands, cmd, &CommandMeta::name_);
  AD_CORRECTNESS_CHECK(it != commands.end());
  if (it->requiresAuth_) {
    requireValidAccessToken(accessTokenOk, it->name_);
  }
  if (!it->supportsOperation_) {
    requireNoOperation(operation, it->name_);
  }
  AD_LOG_INFO << "Processing command \"" << it->name_
              << "\": " << it->description_ << std::endl;
}
```

Inlined dispatch code

Solution: extract function

BEFORE

24 lines

```
} else if (auto cmd = checkParameter("cmd", "clear-delta-triples")) {
  requireValidAccessToken("clear-delta-triples");
  logCommand(cmd, "clear delta triples");
  // The function requires a SharedCancellationHandle, but the operation is
  // not cancellable.
  auto handle = std::make_shared<ad_utility::CancellationHandle<>>();
  // We don't directly `co_await` because of lifetime issues (bugs) in the
  // Conan setup.
  auto coroutine = computeInNewThread(
      updateThreadPool_,
      [this] {
        // Snapshot here, on the (single-threaded) `updateThreadPool_`, so we
        // modify the currently active index and not a stale one that a
        // concurrent rebuild may have swapped out (whose changes would be
        // lost).
        auto snapshot = indexAndViewsSnapshot();
        return snapshot->index_.deltaTriplesManager()
            .modify<DeltaTriplesCount>([](auto& deltaTriples) {
              deltaTriples.clear();
              return deltaTriples.getCounts();
            });
      },
      handle);
  auto countAfterClear = co_await std::move(coroutine);
```

AFTER

3 lines

```
} else if (commandIs("clear-delta-triples")) {
  auto countAfterClear = co_await processClearDeltaTriples();
  response = jsonResponse(json(countAfterClear));
```

Extraction of cmd dispatch

BEFORE

57 lines

```
if (commandIs("stats")) {
  response = jsonResponse(composeIndexStats(index));
} else if (commandIs("cache-stats")) {
  response = jsonResponse(cacheStats());
} else if (commandIs("clear-cache")) {
  cache().clearUnpinnedOnly();
  response = jsonResponse(cacheStats());
} else if (commandIs("clear-cache-complete")) {
  cache().clearAll();
  response = jsonResponse(cacheStats());
} else if (commandIs("clear-named-cache")) {
  namedResultCache().clear();
  response = jsonResponse(cacheStats());
} else if (commandIs("clear-delta-triples")) {
  auto countAfterClear = co_await processClearDeltaTriples();
  response = jsonResponse(json(countAfterClear));
} else if (commandIs("vacuum-delta-triples")) {
  auto vacuumStats = co_await processVacuumDeltaTriples(
      checkParameter("timeout", std::nullopt), accessTokenOk);
  response = jsonResponse(vacuumStats);
} else if (commandIs("get-settings")) {
  response = jsonResponse(json(globalRuntimeParameters.rlock()->toMap()));
} else if (commandIs("get-index-id")) {
  response =
      createOkResponse(index.getIndexId(), request, MediaType::textPlain);
} else if (commandIs("dump-active-queries")) {
  auto json = nlohmann::json::object();
  for (auto& [key, value] : queryRegistry_.getActiveQueries()) {
    json[nlohmann::json(key)] = std::move(value);
  }
  response = jsonResponse(json);
} else if (commandIs("rebuild-index")) {
  response = co_await processRebuildIndex(parameters, request);
} else if (commandIs("write-materialized-view")) {
  auto materializedViewStats = co_await processWriteMaterializedView(
      parameters, parsedHttpRequest.operation_, accessTokenOk, requestTimer);
  response = jsonResponse(materializedViewStats);
  // Prevent regular query processing by removing the query from the
  // request.
  parsedHttpRequest.operation_ = None{};
} else if (commandIs("load-materialized-view")) {
  response =
      jsonResponse(processLoadMaterializedView(parameters, indexAndViews));
  // Prevent regular query processing by removing the query from the
  // request.
  parsedHttpRequest.operation_ = None{};
} else if (commandIs("delete-materialized-view")) {
  response = jsonResponse(processDeleteMaterializedView(parameters));
  // Prevent regular query processing by removing the query from the
  // request.
  parsedHttpRequest.operation_ = None{};
} else if (commandIs("unload-materialized-view")) {
  response = jsonResponse(processUnloadMaterializedView(parameters));
  // Prevent regular query processing by removing the query from the
  // request.
  parsedHttpRequest.operation_ = None{};
}
```

AFTER

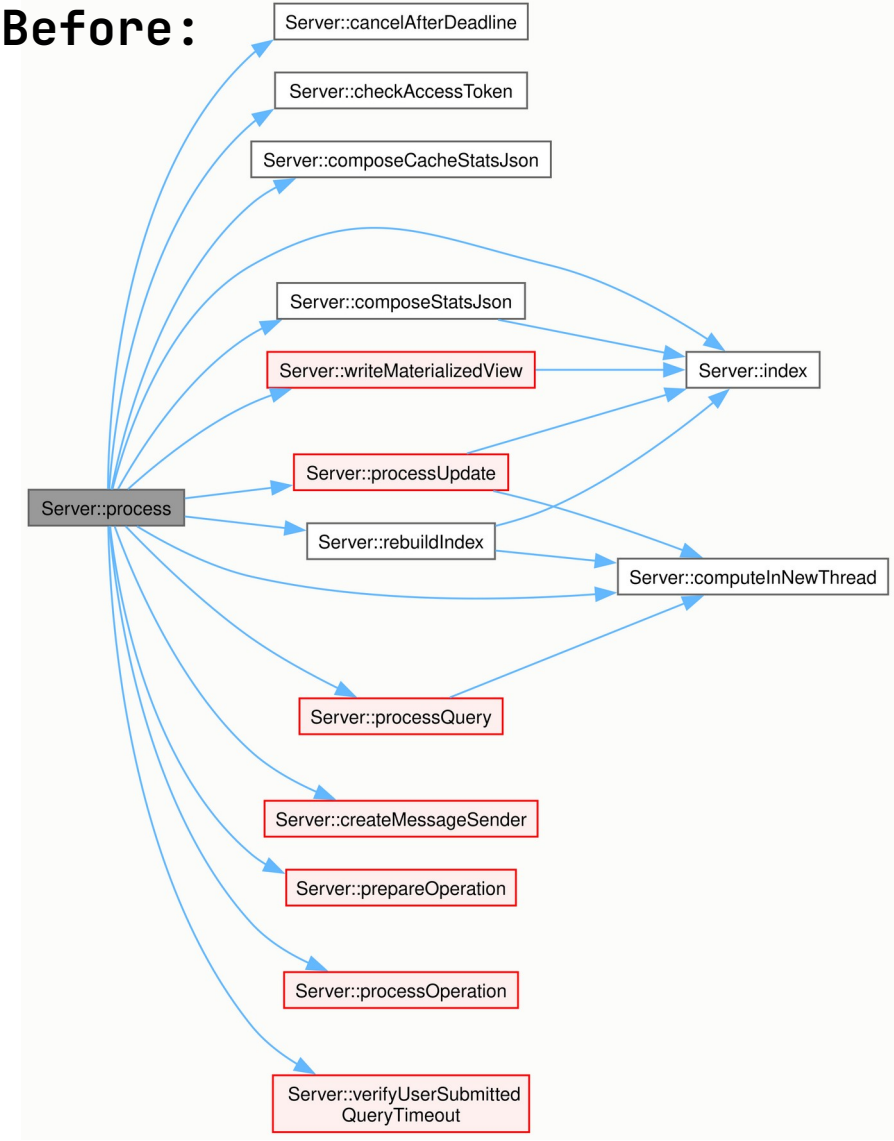
4 lines

```
auto commandResult = co_await processCommands(
    indexAndViews, parameters, parsedHttpRequest.operation_, accessTokenOk,
    requestTimer, request);
std::optional<ResponseT> response = std::move(commandResult.response_);
```

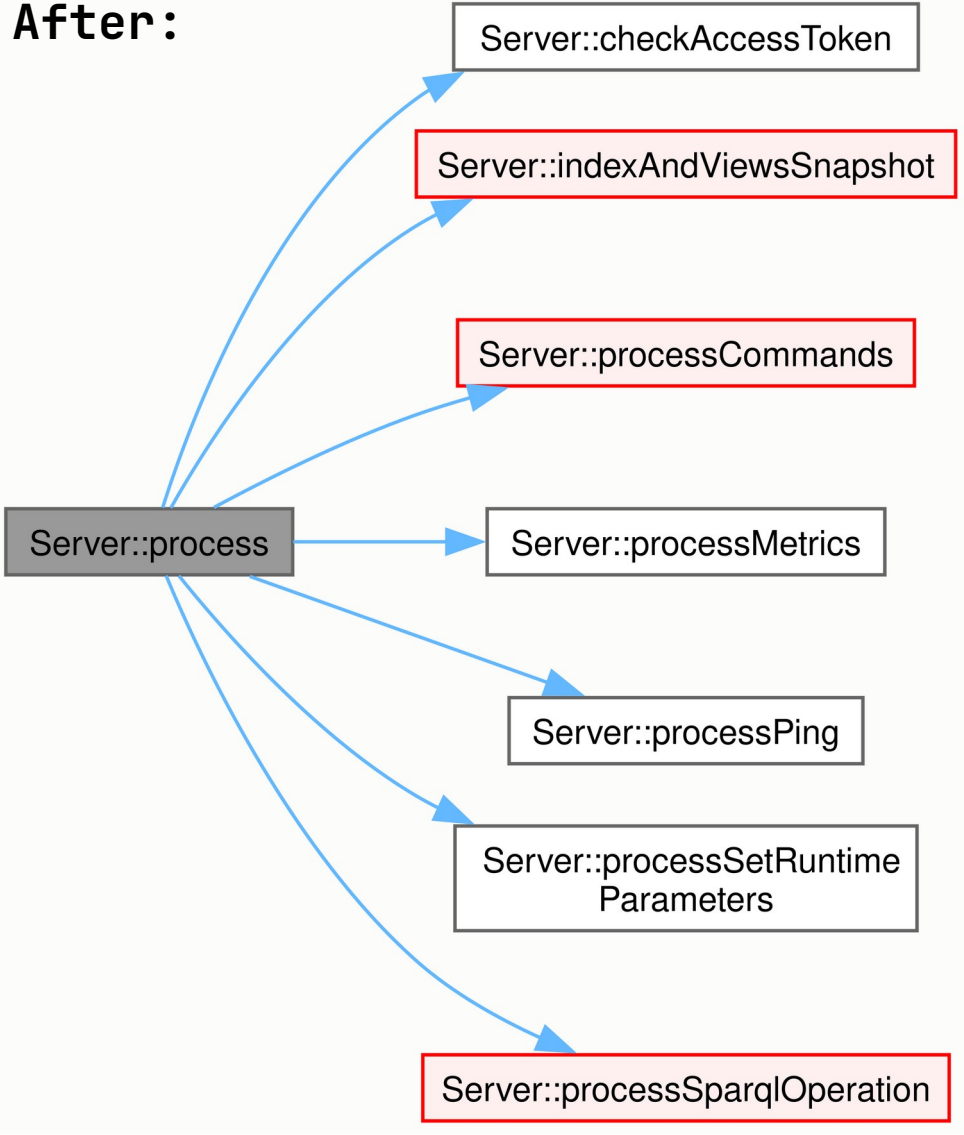
Server::process() shrinking across the refactor

Baseline (93dfb43d) — 357 NLOC

Before:



After:



“@Damekt Thanks, looks good now + will merge”

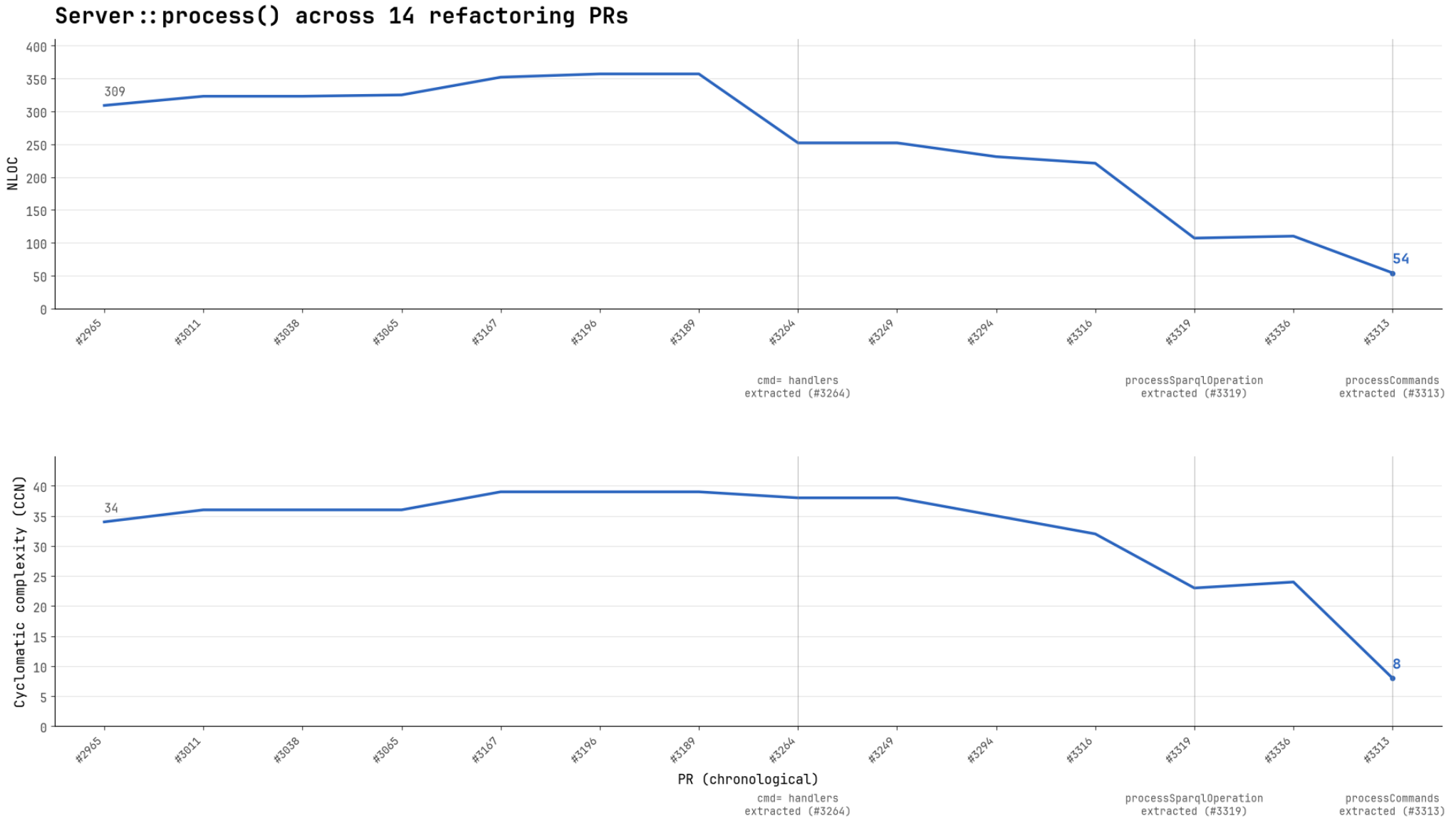
Disclaimer: Illustrative review comment, not from the PR changes before.

Prof. Hannah Bast PR #3065 (opened by Tomáš Dámek)

How did we do?

- Measured with `lizard` (NLOC/CCN) and `clang-tidy` (cognitive complexity), before vs. after commit

Metric:	run()	process()
NLOC (Number Lines Of Code)	59 → 12	357 → 54
CCN (Cyclomatic Complexity Number)	6 → 2	39 → 8
Cognitive Complexity	**28 → 2**	**104 → 12**


Server::process() across 14 refactoring PRs
NLOC
400
350
300
250
200
150
100
50
0
309
54
#2965
#3011
#3038
#3065
#3167
#3196
#3189
#3264
#3249
#3294
#3316
#3319
#3336
#3313
cmd= handlers
extracted (#3264)
processSparqlOperation
extracted (#3319)
processCommands
extracted (#3313)
Cyclomatic complexity (CCN)
40
35
30
25
20
15
10
5
0
34
8
PR (chronological)

Testability

- New tests:
  - ServerTest.handleHttpRequest
  - ServerTest.makeWebSocketSessionSupplier

BEFORE

88 lines, all buried inside run() — needs a real bound socket to test any of it

```
void Server::run() {
  using namespace ad_utility::httpUtils;

  // Function that handles a request asynchronously, will be passed as argument
  // to `HttpServer` below.
  auto httpSessionHandler =
      [this](auto request, auto&& send) -> boost::asio::awaitable<void> {
    // Version of send with maximally permissive CORS header (which allows the
    // client that receives the response to do with it what it wants).
    // NOTE: For POST and GET requests, the "allow origin" header is sufficient,
    // while the "allow headers" header is needed only for OPTIONS request. The
    // "allow methods" header is purely informational. To avoid two similar
    // lambdas here, we send the same headers for GET, POST, and OPTIONS.
    auto sendWithAccessControlHeaders =
        [&send](auto response) -> boost::asio::awaitable<void> {
      response.set(http::field::access_control_allow_origin, "*");
      response.set(http::field::access_control_allow_headers, "*");
      response.set(http::field::access_control_allow_methods,
                   "GET, POST, OPTIONS");
      co_return co_await send(std::move(response));
    };
    // Reply to OPTIONS requests immediately by allowing everything.
    // NOTE: Handling OPTIONS requests is necessary because some POST queries
    // (in particular, from the QLever UI) are preceded by an OPTIONS request (a
    // so-called "preflight" request, which asks permission for the POST query).
    if (request.method() == http::verb::options) {
      AD_LOG_INFO << std::endl;
      AD_LOG_INFO << "Request received via " << request.method()
                  << ", allowing everything" << std::endl;
      co_return co_await sendWithAccessControlHeaders(
          createOkResponse("", request, MediaType::textPlain));
    }
    // Process the request using the `process` method and if it throws an
    // exception, log the error message and send a HTTP/1.1 400 Bad Request
    // response with that message. Note that the C++ standard forbids co_await
    // in the catch block, hence the workaround with the `exceptionErrorMsg`.
    std::optional<std::string> exceptionErrorMsg;
    std::optional<boost::beast::http::status> httpResponseStatus;
    try {
      co_await process(request, sendWithAccessControlHeaders);
    } catch (const HttpError& e) {
      httpResponseStatus = e.status();
      exceptionErrorMsg = e.what();
      metrics_->httpErrors_->Add(1, {HttpErrorType::http});
    } catch (const std::exception& e) {
      exceptionErrorMsg = e.what();
      metrics_->httpErrors_->Add(1, {HttpErrorType::internal});
    }
    if (exceptionErrorMsg.has_value()) {
      AD_LOG_ERROR << exceptionErrorMsg.value() << std::endl;
      auto status =
          httpResponseStatus.value_or(boost::beast::http::status::bad_request);
      auto response = createHttpResponseFromString(
          exceptionErrorMsg.value(), status, request, MediaType::textPlain);
      co_return co_await sendWithAccessControlHeaders(std::move(response));
    }
  };

  auto webSocketSessionSupplier = [this](net::any_io_executor& ioExecutor) {
    // This must only be called once
    AD_CONTRACT_CHECK(queryHub_.expired());
    auto queryHub =
        std::make_shared<ad_utility::websocket::QueryHub>(ioExecutor);
    // Make sure the `queryHub` does not outlive the ioContext it has a
    // reference to, by only storing a `weak_ptr` in the `queryHub_`. Note: This
    // `weak_ptr` may only be converted back to a `shared_ptr` inside a task
    // running on the `io_context`.
    queryHub_ = queryHub;
    return [this, queryHub = std::move(queryHub)](
               const http::request<http::string_body>& request,
               tcp::socket socket) {
      return ad_utility::websocket::WebSocketSession::handleSession(
          *queryHub, queryRegistry_, request, std::move(socket));
    };
  };

  // First set up the HTTP server, so that it binds to the socket, and
  // the "socket already in use" error appears quickly.
  auto httpServer = HttpServer{port_, "0.0.0.0", static_cast<int>(numThreads_),
                               std::move(httpSessionHandler),
                               std::move(webSocketSessionSupplier)};

  AD_LOG_INFO << "The server is ready, listening for requests on port "
              << std::to_string(httpServer.getPort()) << " ..." << std::endl;

  // Start listening for connections on the server.
  httpServer.run();
}
```

AFTER

19 lines — two named, independently testable functions

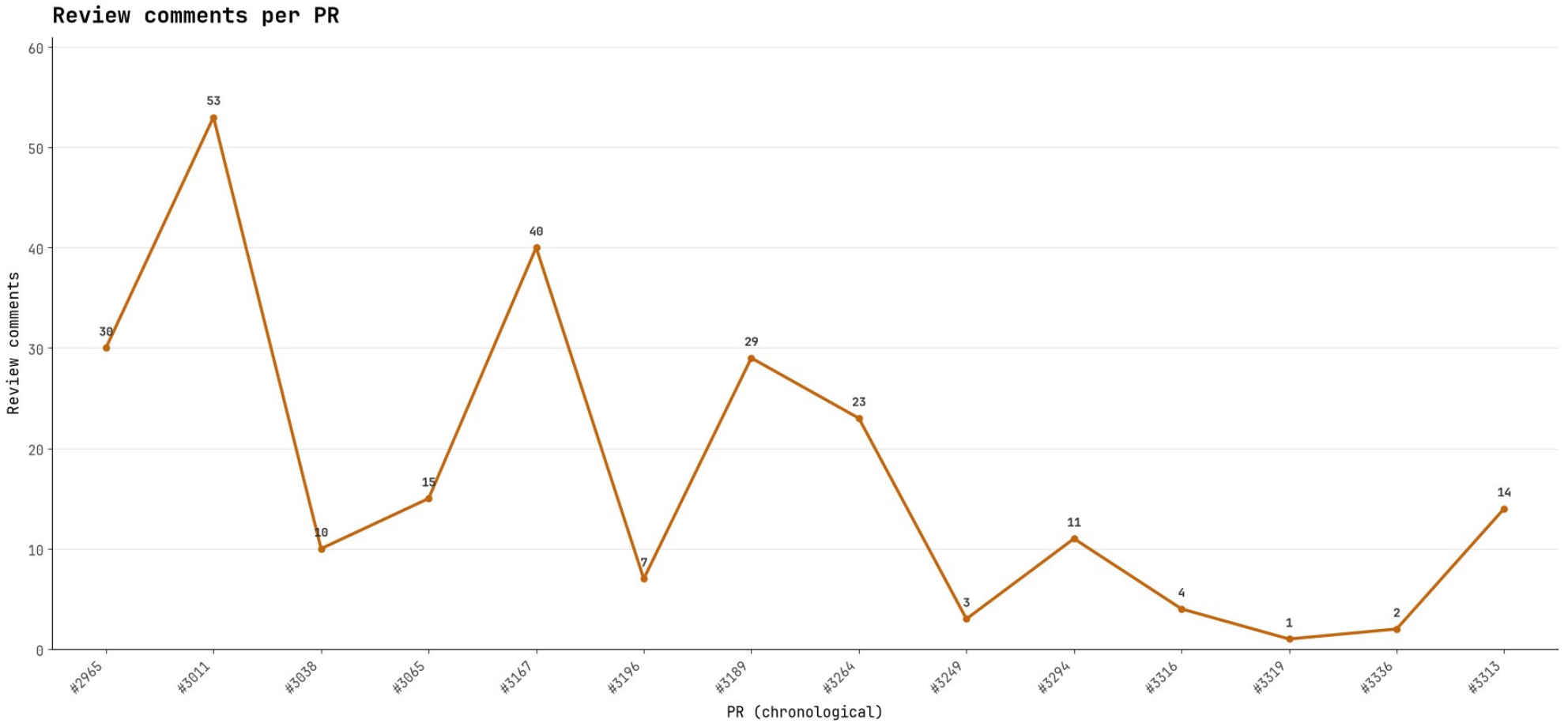
```
void Server::run() {
  auto httpSessionHandler = [this](auto request, auto&& send) {
    return handleHttpRequest(std::move(request), AD_FWD(send));
  };

  // `HttpServer`'s constructor binds the socket synchronously; keep this as
  // the first statement in `run()` so a port already in use fails fast,
  // before any other startup work.
  auto httpServer =
      HttpServer{port_, "0.0.0.0", static_cast<int>(numThreads_),
                 std::move(httpSessionHandler),
                 absl::bind_front(&Server::makeWebSocketSessionSupplier, this)};

  AD_LOG_INFO << "The server is ready, listening for requests on port "
              << std::to_string(httpServer.getPort()) << " ..." << std::endl;

  // Start listening for connections on the server.
  httpServer.run();
}
```

Review friction dropped as the code did



Thank you for your attention.

Refactoring QLever’s ‘Server‘

How to Split a Long-Lived Monolithic Class in a Live Codebase

Sources

- Academic sources
  - Fowler, M. & Beck, K. — Refactoring: Improving the Design of Existing Code (2nd ed.), Addison-Wesley, 2019
  - McCabe, T. J. — A Complexity Measure, IEEE Transactions on Software Engineering, SE-2(4), 308–320, 1976
  - Campbell, G. A. — Cognitive Complexity: A New Way of Measuring Understandability, SonarSource white paper, 2023
  - Fowler, M. — Replace Error Code with Exception, refactoring.com catalog (web only, not in the printed 2nd edition)
- Analysis tools
  - lizard — NLOC / cyclomatic complexity (CCN) analyzer
  - clang-tidy — cognitive complexity check (readability-function-cognitive-complexity)
  - Doxygen + Graphviz — call-graph extraction