Asynqro

how to stop worrying about the futures

Dennis Kormalev

Toronto, 2019

```cpp
std::future<int> doSomethingAsync(int x) {
  return std::async(std::launch::async, [x]() {
    //...
    return x * 2;
  });
}
```

```cpp
std::future<int> doSomethingAsync(int x) {
  return std::async(std::launch::async, [x]() {
    //...
    return x * 2;
  });
}
```

```cpp
void caller() {
  doSomethingAsync(42);
  //Oops...
}
```

```
std::future<int> doSomethingAsync(int x) {
  return std::async(std::launch::async, [x]() {
    //...
    return x * 2;
  });
}
```

```
void caller() {
  doSomethingAsync(42);
  //Oops...
}
```

```
void errorCheckCaller() {
  try {
    doSomethingAsync(42).get();
  } catch (...) {
    //Meh...
  }
}
```

```
std::future<int> doSomethingAsync(int x) {
  return std::async(std::launch::async, [x]() {
    //...
    return x * 2;
  });
}
```

```
void caller() {
  doSomethingAsync(42);
  //Oops...
}
```

```
std::future<bool> transformingCaller() {
  return std::async(std::launch::async, []() {
    std::future<int> f = doSomethingAsync(42);
    return f.get() % 2;
  });
  //Meh...
}
```

```
void errorCheckCaller() {
  try {
    doSomethingAsync(42).get();
  } catch (...) {
    //Meh...
  }
}
```

```
std::future<int> doSomethingAsync(int x) {
  return std::async(std::launch::async, [x]() {
    //...
    return x * 2;
  });
}
```

```
void caller() {
  doSomethingAsync(42);
  //Oops...
}
```

```
std::future<bool> transformingCaller() {
  return std::async(std::launch::async, []() {
    std::future<int> f = doSomethingAsync(42);
    return f.get() % 2;
  });
  //Meh...
}
```

```
void errorCheckCaller() {
  try {
    doSomethingAsync(42).get();
  } catch (...) {
    //Meh...
  }
}
```

```
void massiveCaller() {
  std::vector<std::future<int>> futures;
  for (int i = 0; i < 10000; ++i)
    futures.push_back(doSomethingAsync(i));
  for (const auto &f : futures)
    f.get();
  //Oops...
}
```

```
template<typename T> using Future = asynqro::Future<T, std::string>;
Future<int> doSomethingBetterAsync(int x) {
  return asynqro::tasks::run([x]() {
    //...
    return x * 2;
  });
}
```

```
template<typename T> using Future = asynqro::Future<T, std::string>;
Future<int> doSomethingBetterAsync(int x) {
  return asynqro::tasks::run([x]() {
    //...
    return x * 2;
  });
}
```

```
void betterCaller() {
  doSomethingBetterAsync(42);
}
```

```
template<typename T> using Future = asynqro::Future<T, std::string>;
Future<int> doSomethingBetterAsync(int x) {
  return asynqro::tasks::run([x]() {
    //...
    return x * 2;
  });
}
```

```
void betterCaller() {
  doSomethingBetterAsync(42);
}
```

```
void errorCheckBetterCaller() {
  doSomethingBetterAsync(42)
    .onFailure([](const auto &f) {
     //...
    });
}
```

```
template<typename T> using Future = asynqro::Future<T, std::string>;
Future<int> doSomethingBetterAsync(int x) {
  return asynqro::tasks::run([x]() {
    //...
    return x * 2;
  });
}
```

```
void betterCaller() {
  doSomethingBetterAsync(42);
}
```

```
Future<bool> transformingBetterCaller() {
  return doSomethingBetterAsync(42)
    .map([](int x) {
      return x % 2;
    });
}
```

```
void errorCheckBetterCaller() {
  doSomethingBetterAsync(42)
    .onFailure([](const auto &f) {
      //...
    });
}
```

```
template<typename T> using Future = asynqro::Future<T, std::string>;
Future<int> doSomethingBetterAsync(int x) {
  return asynqro::tasks::run([x]() {
    //...
    return x * 2;
  });
}
```

```
void betterCaller() {
  doSomethingBetterAsync(42);
}
```

```
Future<bool> transformingBetterCaller() {
  return doSomethingBetterAsync(42)
    .map([](int x) {
      return x % 2;
    });
}
```

```
void errorCheckBetterCaller() {
  doSomethingBetterAsync(42)
    .onFailure([](const auto &f) {
      //...
    });
}
```

```
void massiveBetterCaller() {
  std::vector<Future<int>> futures;
  for (int i = 0; i < 10000; ++i)
    futures.push_back(doSomethingBetterAsync(i));
  Future<int>::sequence(futures).wait();
}
```

```
Future<bool, Failure> Worker::fetchData(QString username, QString password) {
  return api->authenticate(username, password)
    .recoverWith([this, username, password](const Failure &f) -> Future<User, Failure> {
      if (!f.isNetworkRelated())
        return WithFailure(f);
      return system->enableProxy() >> [this](){ return api->authenticate(username, password); };
    })
    .flatMap([this](const User &userInfo) {
      auto fullUserData = api->fetchFullUserData()
        .onSuccess([this](const FullUserData &data) { emit userInfoFetched(data); });

      auto taken = api->fetchTakenBooks()
        .flatMap([this](const QVector<QString> &bookIds) {
          auto books = traverse::map([this](const auto &id) { return api->fetchBook(id); });
          return Future<Book>::sequence(std::move(books));
        })
        .map([this](const QVector<Book> &books) { return Book::qmled(books); })
        .onSuccess([this](const QVariantList &books) { emit loanedBooksFetched(books); });

      auto suggestions = api->fetchSuggestions()
        .innerFilter([userInfo](const Book &book) { return book->ageRate < userInfo.age; })
        .map([](const QVector<Book> &books) { return Book::qmled(books); })
        .onSuccess([this](const QVariantList &books) { emit suggestionsFetched(books); });

      return fullUserData + taken + suggestions;
    })
    .andThenValue(true);
}
```

```
Future<bool, Failure> Worker::fetchData(QString username, QString password) {
  return api->authenticate(username, password)
    .recoverWith([this, username, password](const Failure &f) -> Future<User, Failure> {
      if (!f.isNetworkRelated())
        return WithFailure(f);
      return system->enableProxy() >> [this](){ return api->authenticate(username, password); };
    })
    .flatMap([this](const User &userInfo) {
      auto fullUserData = api->fetchFullUserData()
        .onSuccess([this](const FullUserData &data) { emit userInfoFetched(data); });

      auto taken = api->fetchTakenBooks()
        .flatMap([this](const QVector<QString> &bookIds) {
          auto books = traverse::map([this](const auto &id) { return api->fetchBook(id); });
          return Future<Book>::sequence(std::move(books));
        })
        .map([this](const QVector<Book> &books) { return Book::qmled(books); })
        .onSuccess([this](const QVariantList &books) { emit loanedBooksFetched(books); });

      auto suggestions = api->fetchSuggestions()
        .innerFilter([userInfo](const Book &book) { return book->ageRate < userInfo.age; })
        .map([](const QVector<Book> &books) { return Book::qmled(books); })
        .onSuccess([this](const QVariantList &books) { emit suggestionsFetched(books); });

      return fullUserData + taken + suggestions;
    })
    .andThenValue(true);
}
```

```
std::optional<UserInfo> Worker::fetchUserInfo(std::shared_ptr<std::promise<bool>> p, QString username, QString password) {
  try {
    return api->authenticate(username, password).get();
  } catch (const NetworkException &e) {
    try {
      system->enableProxy().get();
      return api->authenticate(username, password).get();
    } catch (const std::exception &e) {
      p->set_exception(std::make_exception_ptr(e));
      return std::nullopt;
    } catch (...) {
      p->set_exception(std::make_exception_ptr(std::runtime_error("Error!")));
      return std::nullopt;
    }
  } catch (const std::exception &e) {
    p->set_exception(std::make_exception_ptr(e));
    return std::nullopt;
  } catch (...) {
    p->set_exception(std::make_exception_ptr(std::runtime_error("Error!")));
    return std::nullopt;
  }
  return userInfo;
}

std::future<void> Worker::fetchTakenBooks() {
  return std::async(std::launch::async, [this]() {
    auto taken = api->fetchTakenBooks().get();
    std::vector<std::future<Book>> takenBooksFutures;
    takenBooksFutures.reserve(taken.size());
    for (const auto &bookId : *taken)
      takenBooksFutures << api->fetchBook(bookId);
    QVector<QString> taken;
    taken = api->fetchTakenBooks().get();
    QVector<Book> takenBooks;
    takenBooks.reserve(taken.size());
    for (std::future<Book> &&future : takenBooksFutures)
      takenBooks << future.get();
    emit suggestionsFetched(Book::qmled(takenBooks));
  });
}

std::future<void> Worker::fetchSuggestions() {
  return std::async(std::launch::async, [this]() {
    suggestions = api->fetchSuggestions().get();
    auto ageFilter = [userInfo = *maybeUser] (const Book &book) {
      return book->ageRate < userInfo.age;
    };
    suggestions.erase(std::remove_if(suggestions.begin(), suggestions.end(), ageFilter),
                      suggestions.end());
    emit suggestionsFetched(Book::qmled(suggestions));
  });
}

std::future<bool> Worker::fetchData(QString username, QString password) {
  auto p = std::make_shared<std::promise<bool>>();
  std::async(std::launch::async, [this, p, username, password]() {
    auto maybeUser = fetchUserInfo(p, username, password);
    if (!maybeUser)
      return;
    auto fullUserDataFuture = api->fetchFullUserData();
    auto takenFuture = fetchTakenBooks();
    auto suggestionsFuture = fetchSuggestions();
    try {
      fullUserData = fullUserDataFuture.get();
      emit userInfoFetched(fullUserData);
      takenFuture.get();
      suggestionsFuture.get();
    } catch (const std::exception &e) {
      p->set_exception(std::make_exception_ptr(e));
      return;
    } catch (...) {
      p->set_exception(std::make_exception_ptr(std::runtime_error("Error!")));
      return;
    }
    p->set_value(true);
  });
  return p->get_future();
}
```

```cpp
Future<bool, Failure> Worker::fetchData(QString username, QString password) {
  return api->authenticate(username, password)
    .recoverWith([this, username, password](const Failure &f) -> Future<User, Failure> {
      if (!f.isNetworkRelated())
        return WithFailure(f);
      return system->enableProxy() >> [this](){ return api->authenticate(username, password); };
    })
    .flatMap([this](const User &userInfo) {
      auto fullUserData = api->fetchFullUserData()
        .onSuccess([this](const FullUserData &data) { emit userInfoFetched(data); });

      auto taken = api->fetchTakenBooks()
        .flatMap([this](const QVector<QString> &bookIds) {
          auto books = traverse::map([this](const auto &id) { return api->fetchBook(id); });
          return Future<Book>::sequence(std::move(books));
        })
        .map([this](const QVector<Book> &books) { return Book::qmled(books); })
        .onSuccess([this](const QVariantList &books) { emit loanedBooksFetched(books); });

      auto suggestions = api->fetchSuggestions()
        .innerFilter([userInfo](const Book &book) { return book->ageRate < userInfo.age; })
        .map([](const QVector<Book> &books) { return Book::qmled(books); })
        .onSuccess([this](const QVariantList &books) { emit suggestionsFetched(books); });

      return fullUserData + taken + suggestions;
    })
    .andThenValue(true);
}
```

28 lines
No extra threads
No thread flooding
No room for errors

78 lines
Extra waiting threads
Possible thread flooding
Plenty of room for errors

```cpp
std::optional<UserInfo> Worker::fetchUserInfo(std::shared_ptr<std::promise<bool>> p, QString username, QString password) {
  try {
    return api->authenticate(username, password).get();
  } catch (const NetworkException &e) {
    try {
      system->enableProxy().get();
      return api->authenticate(username, password).get();
    } catch (const std::exception &e) {
      p->set_exception(std::make_exception_ptr(e));
      return std::nullopt;
    } catch (...) {
      p->set_exception(std::make_exception_ptr(std::runtime_error("Error!")));
      return std::nullopt;
    }
  } catch (const std::exception &e) {
    p->set_exception(std::make_exception_ptr(e));
    return std::nullopt;
  } catch (...) {
    p->set_exception(std::make_exception_ptr(std::runtime_error("Error!")));
    return std::nullopt;
  }
  return userInfo;
}

std::future<void> Worker::fetchTakenBooks() {
  return std::async(std::launch::async, [this]() {
    auto taken = api->fetchTakenBooks().get();
    std::vector<std::future<Book>> takenBooksFutures;
    takenBooksFutures.reserve(taken.size());
    for (const auto &bookId : *taken)
      takenBooksFutures << api->fetchBook(bookId);
    QVector<QString> taken;
    taken = api->fetchTakenBooks().get();
    QVector<Book> takenBooks;
    takenBooks.reserve(taken.size());
    for (std::future<Book> &&future : takenBooksFutures)
      takenBooks << future.get();
    emit suggestionsFetched(Book::qmled(takenBooks));
  });
}

std::future<void> Worker::fetchSuggestions() {
  return std::async(std::launch::async, [this]() {
    suggestions = api->fetchSuggestions().get();
    auto ageFilter = [userInfo = *maybeUser] (const Book &book) {
      return book->ageRate < userInfo.age;
    };
    suggestions.erase(std::remove_if(suggestions.begin(), suggestions.end(), ageFilter),
                      suggestions.end());
    emit suggestionsFetched(Book::qmled(suggestions));
  });
}

std::future<bool> Worker::fetchData(QString username, QString password) {
  auto p = std::make_shared<std::promise<bool>>();
  std::async(std::launch::async, [this, p, username, password]() {
    auto maybeUser = fetchUserInfo(p, username, password);
    if (!maybeUser)
      return;
    auto fullUserDataFuture = api->fetchFullUserData();
    auto takenFuture = fetchTakenBooks();
    auto suggestionsFuture = fetchSuggestions();
    try {
      fullUserData = fullUserDataFuture.get();
      emit userInfoFetched(fullUserData);
      takenFuture.get();
      suggestionsFuture.get();
    } catch (const std::exception &e) {
      p->set_exception(std::make_exception_ptr(e));
      return;
    } catch (...) {
      p->set_exception(std::make_exception_ptr(std::runtime_error("Error!")));
      return;
    }
    p->set_value(true);
  });
  return p->get_future();
}
```

https://github.com/dkormalev/asynqro