Jestではじめる テスト入門

-テスト未経験から中上級者へのガイドライン-

伊藤貴之・椎葉光行 共著
和田卓人 監修

PEAKS

Jestではじめるテスト入門

-テスト未経験から中上級者へのガイドライン-

伊藤貴之・椎葉光行 共著
和田卓人 監修

PEAKS

監修者から

本書は、自動テストを使えるようになりたいと考えている JavaScript プログラマのための入門書です。本書はさまざまな独自性や強み、読みどころを持っていますが、なかでも特筆すべきところを監修者の立場からご紹介しましょう。

まず、網羅性です。テスティングフレームワーク Jest を取り巻く技術に関して、様々な観点から技術やライブラリがカバーされています。日進月歩の技術の世界において、現時点で押さえるべき代表的な技術の説明がひととおり揃っています。

本書はコラムが充実しています。例えば、Jest の開発者 Christoph さんの意見が読めるのは、類書にはない本書ならではの特徴です。作者の言葉とは、つまり一次情報です。技術書における一次情報の重要性は言うまでもありませんね。Christoph さんにインタビューを行った著者（伊藤さん）の行動力に拍手を送りたいと思います。

著者（伊藤さん）自らの失敗に根ざした率直な言葉にも魅力があります。本書の著者も最初から自動テストの重要性を認識していたのではありませんでした。自らの手痛い失敗から自動テストの重要性を学び、本書を通じて初心者のみなさんが同じ落とし穴に落ちないように、経験に基づいた実感のある言葉でガードレールを敷設しています。

著者らは世界的な CI サービス企業 CircleCI 社に勤めていたエンジニアでもあります。つまり、職務を通じて自動テストの活用事例（あるいは、失敗事例）に多く触れており、それらの事例に関わった経験がもたらす説得力があります。また、CircleCI 自身の開発プロセスが垣間見えるコラムも魅力的です。

本書の後半である第 5 章と第 6 章は実践の章です。これらの章は説明の進め方に最大の特徴があります。執筆者はシニアエンジニアの椎葉さんですが、実は第 5、6 章で使用される技術のなかには初めて触るものもありました。そこで椎葉さんは、自らの試行錯誤、その思考過程を包み隠さず文章化していきます。つまり、読者が後半の章から学べるのは、未知を既知にしていくプロセスです。

シニアエンジニアの思考過程をリアルに追体験できるのは、特に初心者のみなさんにとって勉強になるでしょう。出来上がったプロダクトコードやテストコードだけではなく、それらのコードがどう考えられどういう順番で書かれてきたのか、その試行錯誤の過程、設計の取捨選択にプログラミング上達の秘訣が隠れています。それらが包み隠さず書かれている本はなかなか見つかりません。この点も本書の価値と言えるのではないでしょうか。

読者のみなさんが本書から多くのことを読み取り、役立てられることを期待しています。

和田 卓人

はじめに

みなさんはテストを書いていますか？ 私はソフトウェアエンジニアとして約 10 年働いてきましたが、単発で終わるプロジェクトに多く参加していたこともあり、テストに関する経験をあまり得られないまま過ごしてきました。自動テストを使わずに、手動で検証していましたが、あるプロジェクトにて他のエンジニアとテストがないことでトラブルがあり、それがキッカケでテストを書いた方がいいかもしれないと思うようになりました。そんなこともあり、2020 年に本書の前身となる「はじめての Jest 入門」を技術書典 10 のイベントで公開しました。Jest の基本的な機能を中心に解説したのですが、思ったよりも反応があり、テストについて悩んでいる人が沢山いることを知り、Jest を利用したテストの入門書を書くに至りました。

Jest はオールインワンのテストフレームワークです。JavaScript では、以前はテスト環境をセットアップするために、テストランナー、アサーションライブラリ、モックライブラリといくつかのライブラリを組み合わせる必要がありました。しかし、Jest は最初から単体テストを実行するために必要な機能がすべて含まれているため、Jest をインストールするだけでテストを書き始めることができます。

最近では、Node.js にも v18 からテストランナーやアサーションがビルトインの機能として追加されたり、Vitest という新しいテストフレームワークも誕生していたりします。しかし、Jest は既に機能が安定しており、情報も豊富にあり、多くのプロジェクトで利用されている実績があるため、テストを学びたい方にとってはもっとも安全な選択肢の一つと考えます。また、Vitest は Jest と互換性のある API で設計されているため、Jest を学ぶことは他のテストフレームワークを利用する際にも役立ちます。

本書では、実際のプロジェクトへテストを導入する力を身につけることを目標にしています。そのため、Jest についての解説だけではなく、テストを導入するモチベーションからテストを活用するために必要な知識を盛り込んでいます。さらに、共著者の椎葉には新しいプロジェクトでテストを導入する流れを解説してもらっています。椎葉は私とは違い、昔からテストを書いており、その長年の経験を元に、テストを導入する流れを解説しています。テストを導入する流れはプロジェクトや人によって異なりますが、ベストプラクティスの一つとして参考になるはずです。

みなさんがテストを書くための第一歩を踏み出す際に、本書がその一助になれば幸いです。

2023 年 2 月
著者を代表して 伊藤 貴之

謝辞

本書の執筆にあたっては、多くの方のお世話になりました。

クラウドファンディング出資者の皆様に深く感謝します。度重なるスケジュール変更にもかかわらず、皆様が温かく応援し続けてくださったおかげで本書を完成させることができました。

突然の依頼にもかかわらず快くインタビューを引き受けてくださった Christoph Nakazawa さん、心より感謝しています。

大竹 智也さん、irof さん、是村 潤さん、鈴木 成典さん、水上 誠さん、宇佐美 ゆうさんにはレビュアーとしてご協力いただきました。皆様の貴重なアドバイスや的確なご指摘に感謝します。

Marek さん、Zak さん、Andrew さん、Vivian さん、Yohei さん、Eric さん、CircleCI の皆様。CircleCI で学んだたくさんのことが本書の土台になっています。本当にありがとうございます。

筆者の伊藤へテストを書くキッカケをくれた、平松さん、小野寺さん、プロジェクト提案時に沢山のアドバイスをくれた Kim さん、おくつさんに感謝します。皆様の助けがあったからこそ本書を書く機会を頂くことができました。

いつも優しくサポートしてくださった PEAKS の日高さん、高橋さんに心より感謝しています。

そして、執筆に専念できるように支え、励まし続けてくれた家族に感謝します。

対象読者

本書は、テストを導入する理由から解説し、続いて Jest の基礎から CI の導入、そして実践編として新規のアプリケーションにテストを導入する流れを解説しています。既に JavaScript や TypeScript を利用してアプリケーションを開発した経験がある方を想定しているため、コマンドラインの操作方法や JavaScript、TypeScript などの基本的な書き方については解説していません。

次のような JavaScript や TypeScript を利用する開発に携わる読者を想定しています。

- JavaScript もしくは TypeScript を利用したプロジェクトへテストを導入したいと考えている方。
- テストは必要だと思っているが、何から始めればいいのかよくわからない方。
- レガシーなプロジェクトを運用している、もしくはその改善を予定している方。

本書で利用するサンプルコードは説明どおりに実行することで、試すことができるようになっておりますが、ツールのバージョンが変更されたり、利用するサービス（GitHub や CircleCI など）の仕様が変更されると手順が変わる恐れがあります。その際は公式ドキュメントなどを参照して問題を解決して頂く必要があります。本書で扱う技術は一度覚えれば一生使えるものばかりではありません。JavaScript のエコシステムは変化が速く、日々キャッチアップを続ける必要があります。

本書の読み方

本書は次のような構成になっています。第 1 章から第 4 章を伊藤が担当し、第 5 章と第 6 章は椎葉が担当しています。

- 第 1 章では、テストを書く理由やメリット、テストを書くタイミング、必要なテストの種類、そしてテストを導入するために必要な要素について解説しています。テストを書く理由がわからない人や、テストを書きたいと思っているが、テストを書くことを後回しにしてしまう方にこそ読んで頂きたい章になります。
- 第 2 章では、Jest のセットアップから Jest の基本的な機能を 1 つずつ試していきます。Jest の主要な機能を一通り試すので、実際のプロジェクトで何かテストを書くといった際にも、Jest の機能を使いこなし、テストを書くことができるようになります。
- 第 3 章では、既存のプロジェクトにテストを導入する際のアプローチとテストの改善について解説します。
- 第 4 章では、テストを開発フローの一部として利用するために、ブランチモデル、CI、そして静的解析の導入について解説します。
- 第 5 章と第 6 章では、新規プロジェクトの開発において、テストを導入する方法を解説します。第 5 章ではバックエンドの開発、第 6 章ではフロントエンドの開発に焦点を当てています。これまでの第 1 章から第 4 章では、テストを導入するために必要な基本的な要素について解説してきましたが、第 5 章と第 6 章では、実際の開発プロセスでどのようにテストを導入していくかをステップ・バイ・ステップで解説しています。

基本的には、巻頭から順に読み進めることを想定していますが、Jest の機能をピンポイントで知りたい場合は第 2 章を、自動テストを利用した経験があり、Jest や CI について既に知識がある方は、第 5 章の実践的なテスト導入から読み始めても大丈夫です。

開発者 Christoph さんに聞く

Christoph Nakazawa さんは、当時 Facebook, Inc.（現在は Meta Platforms, Inc.）で Jest の開発や普及に従事しており、簡単に紹介すると Jest の生みの親と言えます。本書を書くにあたり、Twitter 上で相談したところ、Jest の成り立ちや Tips、そしてこれからテストを始める方へのアドバイスを教えてくれました。本書では、この Christoph さんに聞いた内容を「**開発者 Christoph さんに聞く**」という見出しのコラムで紹介します。また、Christoph さんは、自身の YouTube チャンネルやブログで Jest や JavaScript のテストフレームワークの開発について解説しています。本書では、Jest そのもののコードベースやアーキテクチャについては触れていないため、興味のある方はぜひ覗いてみてください。

- Twitter: https://twitter.com/cpojer
- JavaScript Testing Framework: https://cpojer.net/posts/building-a-javascript-testing-framework
- YouTube
  - Jest Architecture: https://www.youtube.com/watch?v=3YDiloj8_d0
  - Delightful JavaScript Testing with Jest: https://www.youtube.com/watch?v=cAKYQpTC7MA

サンプルコード

本書で掲載しているコード例は次の URL で公開しています。

- 第 1 章から第 4 章で利用しているサンプルコード
  - https://github.com/jest-book/hello-jest-ts
- 第 5 章から第 6 章で利用しているサンプルコード
  - https://github.com/jest-book/tax-app

想定環境

本書では表 1 の環境で動作確認をしています。

●表 1　想定環境

OS	macOS 13（Ventura）
Node.js	v18.14.0
npm	9.3.1
Jest	29.4.3
TypeScript	4.9.5

ローカル環境で実行する場合には nvm[注1)] もしくは n[注2)] など Node.js のバージョン管理ツールを利用することで、簡単に特定のバージョンの Node.js へ切り替えることができます。

注1)　https://github.com/nvm-sh/nvm
注2)　https://github.com/tj/n

クラウドファンディングと PEAKS

本書は技術書クラウドファンディング・サービスである「PEAKS」のプロジェクトとして開始され、2022 年の 3 月 1 日時点で 709 人もの支援者のサポートによって目標を達成できました。出資者特典である「アーリーアクセス」でいただいたご意見も反映されております。

PEAKS ではこんな本を作りたい！という方を募集しています。次の窓口からご連絡いただければ幸いです。

- https://peaks.cc/requests

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

目次

第1章

なぜテストを書くのか？

最近では CI（Continuous Integration）ツールの普及もあり、以前よりも遥かに自動テストを利用するプロジェクトが増えてきているように感じます。筆者が CircleCI でサポートエンジニアだった際に、お客様のビルドで発生した問題の調査のため、数多くのビルドを確認する機会がありましたが、さまざまな規模のプロジェクトで自動テストが動いていました。当時はウェブ系の大手企業が CircleCI を利用している印象でしたが、最近はスタートアップから大手 IT 企業まで幅広く利用されています。

しかし、自動テストが一切ないプロジェクトも、まだまだあると思います。実際に筆者が数年前に担当していたプロジェクトでは、自動テストが一切なく、すべて手動でテストしていました。過去のプロジェクトにおいて自動テストを書いたこともありましたが、自分が主導していたプロジェクトに率先してテストを導入することはありませんでした。

本書は、自動テストをプロジェクトに導入する第一歩の手助けとなることを目標にしています。後の章では **Jest**（ジェスト）というテストツールの使い方を中心に解説していますが、Jest を使ってテストを書けるようになっただけでは、テストを書くモチベーションが湧かない、書くタイミングがわからないことから、結局テストを書かないことが考えられます。当時の筆者がまさにそうでした。

そのため本書の冒頭に、この「なぜテストを書くのか？」という章を用意しました。「テストはいいぞ！ テスト書け！」と言うことは簡単ですが、それだけを聞いてテストを書くようになる人は少ないと思います。この章では、テストを書くモチベーションやメリット、いつテストを書くのか、どんなテストが必要なのか、どのようにテストを導入すればいいのかを紹介します。この章が、少しでもテストを書くキッカケにつながればと思います。

1.1 自動テストは開発の助けになる

「なぜテストを書くのか？」という問いに一言で答えるなら、「**開発の助けになるから**」と言えます。

XP（エクストリームプログラミング）や TDD（テスト駆動開発）を提唱しているテストの第一人者である Kent Beck の言葉に「僕は、偉大なプログラマなんかじゃない。偉大な習慣を身に付

けた少しましなプログラマなんだ」[注1)]があります。この言葉において、偉大な習慣とはリファクタリングを指しますが、彼は自身の著書である『エクストリームプログラミング』[注2)]にてテストをすることとプログラミングを同等に評価しており、彼の持つ偉大な習慣には自動テストを書くことが含まれています。筆者の経験でも、テストを書くことで自分の書くコードに自信を持つことができました。開発効率を高め、少しでもましなコードを書きたいと思うのであれば、自動テストを書くべきだと考えます。

ここでは、自動テストがなぜ開発の助けになるのかを紹介します。

1.1.1 自動テストは開発スピードを上げる源泉

スタートアップや受託開発の現場では、限られた時間でリリースしなければならない局面が度々訪れます。そのような余裕のない中で、「開発スピードを上げるために、テストコードは書かない！」という判断をしたことはないでしょうか？ 自動テストを書いたことがなければテストコードの書き方から調べる必要がありますし、単純にテストコードを書く時間も必要になります。そのため、一見するとこれはメリットがあるように思えます。恥ずかしながら、筆者も以前のプロジェクトでは同じような言葉を使用したことがありました。はたしてこれは本当でしょうか？

以前はこの質問に対して議論もありましたが、最近では既にある程度の確度の高い答えは出ています。本書の監修者である和田が言う「短期的には得られるが、1 カ月後には逆効果になって、長期的には致命傷になる」です。

もう少し詳しく言うと、「**テストを省くということは保守性を犠牲にするということで、保守性を犠牲にした場合にできる質の低いコードでは中長期的にスピードは得られない**[注3)]」ということです。テストコードがないコードをレガシーコードと呼ぶ[注4)]ことがあるように、テストを省くことで時間を節約しているつもりが、保守性のないコードを生み出し、中長期的には開発のスピードを下げる原因を作っていることになります。

1.1.2 自動テストを書かないことで生まれる問題たち

自動テストを書かないことで生まれる問題は大きく 2 つあると考えています。改善を加えることを困難にさせること、学びの機会を減らすことです。これらの問題に共通する厄介な点としては、すぐには問題が表面化しないことにあります。自動テストがなくてもコードを改善できますし、開発を続けていれば日々学びもあります。経験するプロジェクトによって、自動テストの重要性は異

注1） Martin Fowler 著，児玉公信，友野晶夫，平澤章，梅澤真史訳『リファクタリング：既存のコードを安全に改善する（第 2 版）』（オーム社，2019）p. 48.（原書：Martin Fowler with Kent Beck. Refactoring: Improving the Design of Existing Code. Second Edition. Addison-Wesley. 2018.）

注2） Kent Beck，Cynthia Andres 共著，角征典訳『エクストリームプログラミング』（オーム社，2015）（原書：Kent Beck with Cynthia Andres. Extreme Programming Explained: Embrace Change. Second Edition. Addison-Wesley. 2005.）

注3） 和田卓人「質とスピード」（2022）https://speakerdeck.com/twada/quality-and-speed-2022-spring-edition（2023 年 1 月参照）

注4） マイケル・C. フェザーズ著，平澤章ほか訳，ウルシステムズ株式会社監訳『レガシーコード改善ガイド』（翔泳社，2009）（原書：Michael C. Feathers. Working Effectively with Legacy Code. Prentice Hall PTR. 2005.）

なってくるため、数年ソフトウェアエンジニアをしても、運が悪ければ気づかない問題だと思います。

筆者も以前は自動テストを書かないことについて、なんとなく問題だと思っていましたが、実際に何が問題なのか長い間わかっていませんでした。その経験があるからこそ本章を書いているのですが、本章を読んでもいまいちピンと来ない場合は、ぜひ継続的に開発を続けているシステムで開発する機会を見つけてみてください。継続的に開発を続けているシステムであれば、問題を起こさずに変更を加えなければならない機会が必ず訪れます。そのような状況を経験することで、自動テストがないことで生じる問題についての認識を深めることができるはずです。

1.1.3 コードに改善を加えることが困難になる

自動テストを書かないときに起きる最大の問題は、コードに改善を加えにくくなるということです。これは主に、現在のコードの挙動がわかりにくくなる点と、コードに変更を加える際の不安をコントロールするのが難しくなる点の 2 点に由来します。

現在のコードの挙動がわからない

自動テストがない場合、コードがうまく構造化され、理解しやすい形で実装されていることは稀です。これは最初からそうすることが難しいからです。テストがないということは、コードが洗練されておらず、関数名と実際の処理が一致しない、変数名や関数名が役割を正確に表していない、関数外に依存関係があるなどの問題をうかがわせる兆候だと筆者は考えます。

コードの改善には現在のコードの挙動を理解する必要がありますが、テストがない状態でコードの挙動を確認するには、対象のコード以外にも関連するコードを読んで理解し、実際に動作させ、挙動を確認する必要があります。これには、時間と労力がかかります。さらに、プロジェクトの規模が大きくなればなるほど状況は悪くなり、改善を進めることがより困難になります。

また、コメントや設計ドキュメントだけでも不十分です。コメントや設計ドキュメントは実装の全体像や細部を理解するのを助けてくれますが、現在の実装と一致しているかを機械的に判断できず、簡単に乖離が発生します。

自動テストはコードの振る舞いを直接的に表現でき、関数をどう呼び出すのか、アウトプットは何かなどを教えてくれます。もし振る舞いが変わった際には、テストが失敗するため変化を気づかせてくれるので、ドキュメントとは異なり常に最新の実装の状態を表すことができます。

コードに変更を加える際の不安をコントロールできない

2 つ目の理由として、不安のコントロールが難しくなることが挙げられます。

未知のコードに対して変更を加える場合、誰もがその変更に対して不安を持つことになります。自分の思いつくすべてのケースを手動でテストできたとしても、その考えに抜け漏れがあれば問題

に繋がります。さらに、一度問題を起こせば、不安はより大きくなり、変更を加えることに消極的になります。稼働中のサービスに対して変更を加えるのであればなおさらです。

この不安をコントロールするのに自動テストが役立ちます。自動テストを整備することで、考えるとおりにコードが動作するかを確認でき、動作しなくなったらすぐに気付くことができます。自動テストが十分に整備されていれば影響範囲は自動テストが教えてくれるようになり、一つ一つ変更がうまくいくことを確かめながら作業を進めることができます。

このような小さな成功体験を繰り返すことで、不安を小さくし、少しずつ自信を持ちながらコードに変更を加えることができるようになります。

1.1.4 学びの機会を失う

次に大きな問題は、学習と成長の機会を失うということです。

コードレビューを学びの機会にする

コードレビューをする際に自動テストがない場合、レビュワーはコードが適切に動作するか確認することに多くの時間を費やすことになります。手動でテストすることになるため、変更部分の動作確認の仕方から、必要なデータの準備などを考える必要があり、大変な時間と労力がかかります。動作確認に時間がかかってしまうため、明らかな動作に関する問題以外を追求するのが難しくなります。

自動テストがあれば、コードが動作することを前提にレビューを進められます。レビュワーは主にコードの設計・実装方法・影響範囲などを対象に、より広く踏み込んで確認できます。そのため、レビュワーに動作確認のための負担を強いるのは、レビュイー（レビューを受ける立場）としても大きなデメリットで、他の開発者からアドバイスをもらう機会を失っていることになります。

自動テストを活用することで、コードレビューを単なる凡ミスを見つける機会ではなく、学習の機会となる貴重なチャンスに変えることができます。

視点を変えれば気付くことも変わる

開発している時の視点では、新しい関数を作るのが面倒だという理由や、処理が小さいからといった理由などで、読みにくいコードを書いてしまうことがあります。しかし、テストを書いてみると、関数名が想定される処理と異なることに気づいたり、変数のスコープが広く、どこで変数が利用されているのか分かりづらかったり、関数外に依存関係があり単純にテストできないことに気づいたりします。

このように、テストを書く中で改めてコードの振る舞いや内部の処理について意識するようになり、改善をすることで学びが得られます。単に動くだけのコードを書いているだけでは、学びが少なく、いつまでも同じようなコードを書いてしまいます。

第 2 章

Jest 入門

この章では Jest の基本的な使い方を説明します。まず開発支援ツールとしての Jest の位置付けをざっとおさらいして、そのうえで自動テストを実際に書いて実行してみましょう。

2.1 Jest とは？

Jest[注1)]とは、Meta（旧 Facebook）が開発した OSS の JavaScript のテストフレームワークです。ちなみに、GitHub 上で組織名は今も facebook[注2)]となっています。Jest は Babel, TypeScript, Node, React, Angular, Vue など有名な OSS プロジェクトで利用されており、さらに Meta はもちろんですが、Twitter、The New York Times, Spotify や Airbnb などの有名企業でも利用されています。 また、2022 年 5 月 14 日時点では 5,310,375[注3)]ものパブリックなリポジトリで利用されています。

Jest はオールインワンのテストフレームワークです。テストランナー、アサーション（テストの評価）、モック、コードカバレッジなどの自動テストを実行するために必要な機能が最初から含まれています。そのため Jest があれば他のツールを利用しなくても一通りのテストを実施できます。パフォーマンスに関しても、テストは独立したプロセスで動作し並列でテストを実行できるため、プロジェクトの大小に関わらず Jest を利用できます。テストフレームワークを採用する際にパフォーマンスは重要な要素となりますが、Jest は大規模なプロジェクトにも対応できます。

Column. 開発者 Christoph さんに聞く：Jest の名前やロゴの由来は？

実は Jest には前身となる JST（JavaScript Testing）というテストツールがあったんだ。これは 2011 年の当時 Facebook Chat（現 Messenger）が誕生し、Facebook Chat ではデータストラクチャーを扱っているコードでは DOM を利用してなかった。だから、ブラウザではなく Node.js でテストを行うために JST は開発されていた。現在の Jest とは全く異なり、単純な CLI ツールで Facebook 社内のみで利用されていたんだ。

その頃、Jeff Morrison が JST の開発をしていたんだけど、JST はとても遅く、テストはすべてシリアル（直列）で実行された。でも、彼はプロセスの概念を導入し、並列でテストを実行できるようにして、テストがすごく速くなったんだ。そのタイミングで彼は JST の名前を Jest（JavaScript Test）に

注1） https://jestjs.io/en/（2023 年 2 月参照）
注2） https://github.com/facebook/jest（2023 年 2 月参照）
注3） https://github.com/facebook/jest/network/dependents（2022 年 5 月参照）

変更したんだ。ほかに Jeff's Testing Framework だから Jest という話もあるんだけど、これはジョークだよ。ロゴが靴になっていることについて、実は誰もいい考えが思い浮かばなかったんだ。そんなときに誰かが Jester（ピエロ）に名前が似てると言ったんだ。Josh Duck が Jester（ピエロ）の靴のロゴを作ってきて、彼はエンジニアだったんだけど、他の誰のロゴよりも出来が良かったので採用したんだ[注4)]。

●図 2.1　当時の Jest のロゴ

1 つこのロゴについて気に入っている点があるんだ。3 つドットがあるけど、ドット以外を取り除くと、何かが起こるのを待っているような、進行中のドット（progress dots）みたいじゃない？ これは意図的にデザインしたのではないかと思っているよ。

その後、Orta がウェブサイトをリデザインして、「すべてはピエロのためにあるんだ」と考えたんだ。テストフレームワークは信頼性が高く、真面目でなければならないので、この考え方は結構自虐的でもあるんだよね。でも、ピエロのように考えてみると、この嫌なシリアスさを少し取り除いてくれるような気がしたんだ。 そして、もう少し楽しいものにしてくれると思ったんだ。

Jest のモノレポ（Monorepo：複数のプロジェクトが存在するリポジトリ）の中には、`jest-circus`[注5)]というテストランナーのようなモジュールがあって、すべてのことはこのサーカス（jest-circus）の中で起きているような感じなんだ。ウェブサイトにはトランプのカードも表示されていて、すべてこのテーマ（ピエロ）に沿っているんだ。最初からこのような形にするつもりはなかったんだけど、独立したプロジェクトとしてこのような形になったことは、とてもクールだったと思ってるよ。

2.1.1　再作成された Jest

元々は Facebook の内部でのみ利用されていた Jest でしたが、2013 年に React をオープンソース化（OSS 化）することになり、同じタイミングで Jest も OSS 化されました。これは React のテストで Jest が利用されていたため、テストをすべて書き直すか、Jest を OSS 化するかのどちらかを行う必要があったからです。しかし、OSS として公開してから 1 年ほど改善が行われず、開発者体験は最悪な状態で放置されていました。ユーザからのツイートでは「Jest を使うくらいなら死んだほうがまし」などの辛辣な意見もありました。

2015 年から Christoph さんは Jest のプロジェクトに参加しましたが、当時のマネージャーだっ

注4）　https://github.com/facebook/jest/blob/88a94d5d1bc1f387317a3068bf510ab992c5dc64/website/src/jest/img/jest.svg
注5）　https://github.com/facebook/jest/tree/main/packages/jest-circus

た Tom から「Jest を修正してくれ！」と言われても、即座に「No！ このプロジェクトはひどいから手を出したくない」と断ったほど当時の Jest には問題がありました。

しかし、Christoph さんは調査することで、あらためて Jest には良い部分があることを知り、このプロジェクトに取り組むことにしました。しかし、既存のコードには多くの問題があったため、すべてのコードを書き直し Jest を再作成しました。これが現在ある Jest の原型になっています。再作成された Jest は、テストの実行が大幅に高速化され、Facebook のプロジェクトにあった約 1 万個のテストの実行時間を 1 時間から 3 分へ短縮することができ、さらに多くの細かい改善も加えられました。

その中でも大きな改善として、セットアップが簡単になりました。これにより、誰もがすぐに Jest を使うことができるようになりました。これは当たり前の改善のように聞こえますが、当時の Christoph さんには Jest のセットアップが難しいという認識はありませんでした。これは、Christoph さんが全てのプロジェクトをセットアップしていたため、Facebook 内部では誰もセットアップについて考える必要がなく、なかなか気付くことができませんでした。

同じ時期に `create-react-app` がリリースされ、コミュニティのメンバーによって `create-react-app` へ Jest を導入する提案がされました。当時はまだ以前のバージョンの Jest の先入観がある方が多く、否定的な意見が多く出されました。例えば、Facebook 以外で Jest を使う人はほとんどいないという意見や、CRA（Create React App）には Mocha のほうがおすすめだといった意見がありました[注6) 注7)]。

しかし、Christoph さんが投稿した現在の Jest についての説明以降[注8)]、反応は大きく変わり、Jest の導入が決まりました。この `create-react-app` への導入をきっかけに Jest は OSS として多くのユーザに利用され始めるようになりました。

2.1.2 Jest の今後

実はコアメンバーであった Christoph さんが Facebook を離れてから、Facebook（現 Meta）には専任で Jest の開発を進める人はいません。Christoph さんは、「この 4 年間、Facebook でフルタイムで Jest に関して働いた人はいない」というのが最もよく知られている秘密の 1 つと教えてくれました。しかし、現在もコミュニティのメンバーによって引き続き開発が進められています。実際に最近では 2022 年の 8 月 25 日に `Jest 29`[注9)] がリリースされていることからも、継続的に改善が進められていることがわかります。

さらに、2022 年 5 月 11 日から Jest の所属を Meta から `OpenJS Foundation`[注10)] に移管しま

注6） Jest への意見その 1https://github.com/facebook/create-react-app/pull/250#issuecomment-236980910（2023 年 3 月参照）
注7） Jest への意見その 2https://github.com/facebook/create-react-app/pull/250#issuecomment-236985503（2023 年 3 月参照）
注8） 再作成された Jest の説明 https://github.com/facebook/create-react-app/pull/250#issuecomment-237098619（2023 年 2 月参照）
注9） https://jestjs.io/ja/blog/2022/08/25/jest-29（2023 年 2 月参照）
注10） OpenJS Foundation は、プロジェクトをホストし維持するための中立的な組織であり、JavaScript エコシステムと Web 技術の健全な成長を支援しています。https://openjsf.org/

した[注11) 注12)]。Christoph さんによると「これは遅すぎた変化であり、すでにコミュニティによって主導されている。プロジェクトの所有の異動によって何かが大きく変わるというよりは、この変化は現在の実態を反映させるためのものであり、Jest というプロジェクトがより持続的に改善できるようにするための変更である」と話します。

2.2 セットアップとはじめてのテスト

TypeScript と Jest のセットアップを行い、Jest を利用してテストを書いていきます。

2.2.1 プロジェクトのセットアップ

はじめに `hello-jest-ts` という名前でプロジェクトのディレクトリを作成します。その後、npm（Node Package Manager）を利用し必要となるパッケージをインストールし、TypeScript と Jest の初期化を行います。

> **■ TypeScript とは？**
>
> TypeScript はマイクロソフトによって開発された JavaScript に型を導入したプログラミング言語です。型があることでコンパイル時にさまざまな問題を発見できます。TypeScript は JavaScript の機能を拡張して作られたスーパーセット（上位互換）のプログラミング言語であるため、TypeScript はコンパイルすると JavaScript へ変換されます。JavaScript がベースとなっている言語のため、JavaScript で利用できる文法はすべて TypeScript でも利用できます。型を導入するのは大変では？と思われるかもしれませんが、最初からすべての型を定義する必要はありません。TypeScript には型推論の機能があり、型を明示的に定義していなくても、型を推論し、問題があればエラーでお知らせしてくれます。また、TypeScript は、柔軟な型アノテーションとコンパイラの強力なコメント処理の組み合わせで、強固な型をすぐには定義できない場合でも JavaScript を出力できるため、既存のコードへ少しずつ型を導入できます。
> 型があることで、関数のインタフェースを明確にし、テストとは異なる視点で問題を見つけることができます。さらに、コンパイルは実行前に行うので、コードを書いている時点で問題を見つけることができます。

プロジェクトの初期化

`npm init` で `package.json` を生成します。npm では `package.json` と `package-lock.json` を利用してパッケージを管理します。`package.json` が定義ファイルで `package-lock.json` が状態を管理するファイルになります。 `npm init` を実行する際に`-y` オプションをつけることで、

注11） https://jestjs.io/blog/2022/05/11/jest-joins-openjs
注12） https://engineering.fb.com/2022/05/11/open-source/jest-openjs-foundation/

すべての項目にデフォルト値を設定します。

● **プロジェクトの作成と初期化**

```
$ mkdir hello-jest-ts
$ cd hello-jest-ts
$ npm init -y
```

type に module を設定します。これはプロジェクトが ECMAScript モジュール（ESM）として解釈されるために必要な設定になります。今回のサンプルコードではブラウザで利用するコードもテストの対象にするため ESM を有効化します。

● **package.json に type を追加**

```
{
  "name": "hello-jest-ts",
  "type": "module" // ESMであることを明示
}
```

■ ECMAScript モジュールとは？

ECMAScript モジュール（ESM）は、ES2015 で導入された JavaScript ファイルをモジュール化する言語標準の機能です。JavaScript にはモジュールシステムに関する規格として ESM と CommonJS があります。ESM はブラウザからも Node.js からも利用でき、CommonJS は Node.js からのみ利用できます。ESM と CommonJS の違いとして、ESM は JavaScript の厳格モード（Strict mode）が適用される、Top-level await（async 関数の外で await が利用できる）に対応しているなどがあります[注13)]。また、モジュールのインポートとエクスポートの方法も異なり、ESM では `export` と `import` を利用し、CommonJS では `require` と `module.exports` を利用します。

● **モジュールのインポートとエクスポート: src/chapter2/modules.ts**

```
// ESM
import dns from 'node:dns' // モジュールのインポート
export const foo = () => console.log(dns.getServers()) // モジュールのエクスポート

// CommonJS
const dns = require('node:dns') // モジュールのインポート
const bar = () => console.log(dns.getServers()) // モジュールのエクスポート
```

Jest、TypeScript のインストール

続いて、Jest と TypeScript に必要なパッケージをインストールしていきます。Jest は JavaScript のテストフレームワークなので、デフォルトの設定では TypeScript で書かれたテストは解釈できません。そのため、Jest を実行するタイミングで TypeScript を JavaScript

注13） https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules#top_level_await（2023 年 2 月参照）

へコンパイル（トランスパイル）してから実行する必要があります。テストを実行する際にTypeScript で書かれたコードを適宜コンパイルするため **ts-jest**[注14] を利用します。ts-jest は Jest の transformer[注15] と呼ばれるモジュールで、TypeScript を適宜 JavaScript へコンパイルし実行できます。

他の方法として、Jest の公式ドキュメントで紹介している Babel を利用しても TypeScript から JavaScript へコンパイルすることができます。しかし、Babel はコードの変換のみを行うため、静的型検査はできません。ts-jest は tsc コマンド（TypeScript 公式のコマンド）と同様に **TypeScript compiler API** を利用して TypeScript や JavaScript のファイルをコンパイルしているため、コードの変換だけでなく静的型検査もできます。Babel と tsc を組み合わせることでも同様のことができますが、構成をシンプルにするために今回は ts-jest を利用します。

また、Babel の代替として Rust 製の **SWC**（Speedy Web Compiler）があります。SWC の公式サイトでは、SWC はシングルスレッドで Babel よりも 20 倍速く、4 コア利用した場合では 70 倍速いとコンパイルが高速であることが強調されています[注16]。テストにかかる時間を短縮したい場合は SWC の利用を検討してみてもいいかもしれません。

ただし、ts-jest の公式サイトでも、v28.0.0 から徐々にコンパイラを esbuild や swc へ移行すると注記されています[注17]。そのため ts-jest を利用し続けても、今後処理速度が改善される可能性があります。

今回インストールするパッケージは `jest`、`typescript`、`ts-jest`、`ts-node`、`@types/jest`、`@types/node` になります。本書を進める際には同じパッケージを利用して頂くことでパッケージのバージョンが異なることによる挙動の違いなどの問題を回避できます。実際のプロジェクトで利用する際にはパッケージのバージョンは最新のものに置き換えてください。

● **Jest と TypeScript のセットアップに必要なパッケージをインストール**

```
$ npm install --save-dev --save-exact jest@29.4.3 typescript@4.9.5 ts-jest@29.0.5
ts-node@10.9.1 @types/jest@29.4.0 @types/node@ts4.9
```

2.2.2 TypeScript のセットアップ

`tsc --init` コマンドを利用して TypeScript の設定ファイルである `tsconfig.json` を作成します。

● **TypeScript のセットアップ**

```
$ npx tsc --init --target es2020 --module esnext --outDir './dist' --moduleResolution node
--noImplicitAny false --noImplicitThis false --allowJs
```

注14） https://www.npmjs.com/package/ts-jest（2023 年 2 月参照）
注15） https://jestjs.io/docs/code-transformation（2023 年 2 月参照）
注16） https://swc.rs/（2023 年 2 月参照）
注17） https://kulshekhar.github.io/ts-jest/docs/getting-started/presets/（2023 年 2 月参照）

このセットアップで tsconfig.json に設定されるオプションは次のようになります。

●表 2.1　TypeScript のセットアップで設定されるオプション

オプション名	説明
target	出力される JavaScript の言語バージョンを設定するオプションです。デフォルトでは es2016 が設定されますが、今回利用する Node.js のバージョンはさらに新しい構文に対応しているため、ts2020 を指定します。
module	モジュールの種類を指定するオプションです。デフォルトでは commonjs ですが、ESM を有効化するため esnext を指定します。
moduleResolution	モジュール拡張子から import 対象のファイルを検索する方法を指定するオプションです。CommonJS のモジュールを利用する場合も考慮し、node を指定します。
outDir	コンパイルしたファイルの出力先を指定するオプションです。サンプルコードでは./dist を指定します。
allowJs	JavaScript ファイルの利用を許可するオプションです。次に紹介する ts-jest の設定で警告を回避するため、true を指定します。
esModuleInterop	ESM から CommonJS モジュールのインポートをサポートするオプションです。CommonJS モジュールに自動で default を付与し、ESM からインポート可能にします。
forceConsistentCasingInFileNames	import 時に文字列の大文字と小文字を区別するオプションです。
strict	静的型検査を強制するオプションです。
skipLibCheck	.d.ts ファイルの静的型検査をスキップするオプションです。
noImplicitAny	any 型を持つ式と宣言に対するエラー報告を有効にするオプションです。サンプルコードでは any 型を利用する場合があるため、false を指定します。
noImplicitThis	this に any 型が与えられた場合にエラー報告を有効にするオプションです。サンプルコードでは any 型を利用する場合があるため、false を指定します。

コンパイル対象のファイルを指定するため include オプションを tsconfig.json に追加します。このプロジェクトでは src ディレクトリ配下にすべてのコードを追加していきます。

● コンパイル対象のファイルを指定: tsconfig.json

```
{
  "compilerOptions": {}, // 上記のステップで設定されたオプション
  "include": [
```

第 3 章

既存コードへのアプローチとテストの改善

既存のコードベースに対して、テストをうまく導入することは容易ではありません。特に最初は、「テストを導入したい」という思いに囚われ、空回りすることがあります。また、既存のプロジェクトにテストコードがあったとしても、それが適切に仕様を表現したり、実装を適切にテストできたりしていない場合があります。最初から完璧なテストを書くのは難しく、コードベースと同じようにテストコードも改善が必要です。この章では、既存のコードに対してテストを導入する際のアプローチや、既存のテストの改善について解説します。

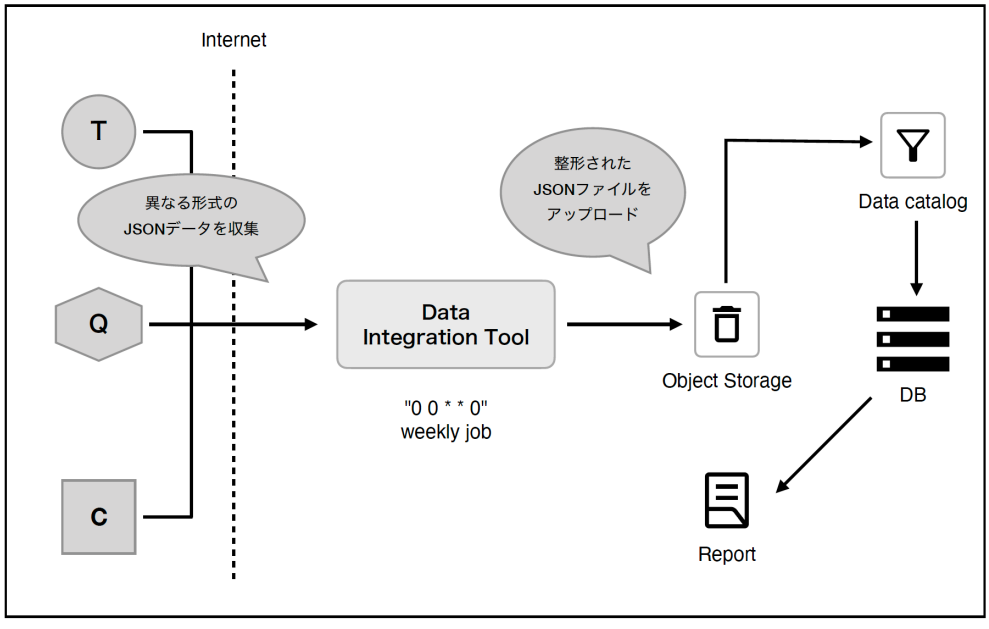
3.1 既存コードへのアプローチ

既存コードへテストを導入した際に、無闇にテストを追加しても安全にコードを変更できない、うまくテストが活用されない場合があります。これはアプローチに問題がある場合があると考えます。このアプローチの失敗例を知ることで、同じような過ちを繰り返さないための気づきを得ることができるので、この章では筆者の体験したアプローチの失敗例について紹介します。ここでは「リファクタリング」や「レガシーコード改善」などの既存のコードを安全に改善するための手法についての解説はせずに、失敗例の振り返りを中心に理想的なアプローチを考えていきます。失敗例ではどうしてそういったアプローチをとってしまったのか、どうすれば意味のあるテストが導入できたのか、安全に変更を加えるにはどうすればよかったのかなどを紹介します。

3.1.1 無駄なテストの導入と無謀な変更の適用

ここでは筆者が当時担当した既存のプロジェクトへのテスト導入の失敗を振り返ります。

筆者がテストを書くモチベーションが上がりつつあった時期に、新規で開発したデータ収集のツールに変更を加える機会がありました。これは 1 年ほど前に自分で作ったツールの改修案件でした。ドキュメントやテストはありませんでしたが、コードを見れば中身が思い出せるくらいの感覚でした。既にツールは運用されており、実際に情報を取得し、DB に書き込む操作が定期的に実行されている状態で、変更は安全に加える必要がありました。



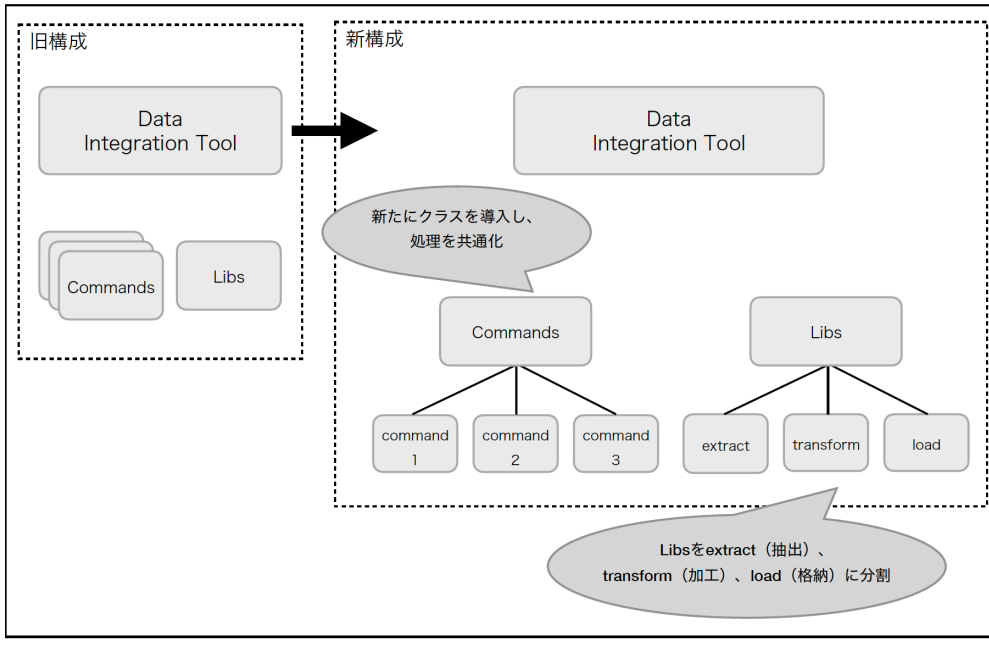
●図 3.1　ツールの構成

はじめに単体テストを導入したいと考えました。テストの効果を知り始めた頃だったので、まずこのプロジェクトにはテストが一切ないことから、テストを追加するのが最優先だと考えました。そのため関数毎に細かく単体テストを書きました。単体テストは書きやすく、実行しやすかったので、ひたすら既存のコードに対して単体テストを追加しました。外部 API や SDK はすべてモック化し、インターネットがない環境でもテストを実行できるようにしました。

すべてのファイルにテストを追加した後、達成感でいっぱいでした。「これはいいぞ！ これでやっと変更を加えていける」と思いました。

次にツールの中身の構成を変更することにしました。テストを導入して気づいたのですが、コードには冗長な処理があり、共通化できそうだと考えました。また、今までは単純にコマンド毎にスクリプトを作成し、外部から情報を取得する、取得した情報を加工する、DB に登録するなどの一連の処理を書いており、処理の流れが把握しづらいとも思いました。

なので ETL（Extract：データの取得、Transform：データの変換、Load：データの保存）の概念を導入して、散らばっている処理を分類しようと思いつきました。クラスを新たに用意し、初期化処理や値を次のステップへ渡す処理を共通化したり、ステップの各処理でどのような処理を行っているのかメタ情報を設定したりするなど、簡単にわかりやすいコマンドを作成できるように改善を加えました。



●図 3.2　構成を変更

「最高の改善だな」と考えていましたが、この変更の中で次のような問題は起こりました。ちゃんと段階を踏めば構成の変更自体は問題なかったのですが、アプローチに問題がありました。

■ 発生した問題たち

- 単体テストは通っているのに、動かない
- 変更が大きく、更新するのに不安があった
- 仕様が曖昧に

単体テストは通っているのに、動かない

結構な時間と労力を掛けて導入した単体テストでしたが、ETL の概念を導入して関数を移動させたり、インタフェースを修正していたりする中で、いくつかバグを埋め込んでいました。そのため単体テストは通るのに、実際に動作させると動かないという問題が発生しました。

最初に導入した単体テストでは問題を見つけることができていませんでした。ここまで、実は各コマンドの一連の流れのテスト、つまりインテグレーションテストは存在せず、自動テストで以前と同じ挙動なのかは確認できていませんでした。

なんでインテグレーションテストがなかったのか。それはメインの処理で呼び出している関数群はすでに単体テストで実行されており、処理としては各関数を呼び出しているだけなので、分岐が

ないように見えていました。そのため、「これはテストしなくてもいいのではないか？」と考えており、インテグレーションテストは追加していませんでした。

変更が大きく、更新するのに不安があった

途中から構成を変更しましたが、その際に互換性は考えずに既存のファイルを修正したり、削除したりしていました。そのため、新しい構成で１つのスクリプトから段階的にリリースすることは難しく、すべてを書き換えてからリリースしました。いわゆるビッグバンリリースです。全部書き換えてからの切り替えなので変更量は大きく、ちゃんと動作するかについて不安がありました。

仕様が曖昧に

テストを書き、コードを改善する中で、既存の仕様で不要だと思う点や変えた方がいい点が見えてきました。「細かい変更だし、改善の一つでしょ」と軽く考えていた筆者は簡単に変更を加えていました。この小さい変更は後々、コマンドの挙動にも影響を与えるものでした。

今回大きな構成の変更を加え、既存の仕様をテストしてない状況でさらに挙動が変わる変更まで加えていたので、開発中は仕様が曖昧な状況でした。一人プロジェクトだとこういった思い切った変更は往々にしてあると思いますが、安全にリリースするという観点で考えるとあまりにも無茶なやり方でした。

Column.　動くのがゴールかスタートか

受託開発など単発で終わるようなプロジェクトばかりに参加していたせいか、システム開発では「早く動くものを作るのが正義、まずはリリースできればいい」といった思いがありました。しかし、自社プロダクトをチームで開発した際に「あれ？ 何か自分がやっている事はどうやら他のエンジニアと違うな」と違和感を抱きました。チーム開発をしている際に、すでに動くものを開発した後で、他のエンジニアが引き続き何やら熱心に作業しているのですが、最初は何をやっているのかわかりませんでした。逆になぜそんなに時間をかけているんだろうとすら思っていました。最初のうちはただ傍観していたのですが、徐々に何をしているか気づきました。彼らは最低限動くものを完成させた後に既存のコードにテストを追加したり、中身の構造を変えたりして、よりシンプルに変更しやすいようにと改善を加えていたのでした。なのでそのエンジニアが書いたコードは最初は自分と同じような雑多な感じで書いていると思いましたが、徐々に構造化され、一見複雑に見えるけど構造を理解すれば、よりわかりやすく、変更しやすいように作られていました。そこで「ゴールが違う」とぼんやり思いました。筆者はシステムが動くのはゴールでしたが、他のエンジニアはシステムが動くのはスタートだと感じました。

この感覚に至るには、システムの運用経験が必要だと思います。受託開発などの単発で終わるようなプロジェクトに参加しがちな方は、1 年間でも同じシステムの運用経験をすることで、こうした継続的に安定したサービスを提供しているエンジニアの感覚に触れる、もしくは自分でその考えに行き着くことができるかもしれません。

第 4 章

テストを開発フローの一部へ

テストを導入しただけでは、テストを開発のワークフローに載せたとは言い切れません。この状態ではテストはありますが、いつ実行すればいいのか、テストを活用して問題を見つけられるかは開発者に委ねられています。開発者によってテストを活用する方もいれば、活用しない方もいる状態です。

せっかく導入したテストを最大限活用するためにも、CI（Continuous Integration：継続的インテグレーション）を導入します。CI を活用してテストを自動実行する環境を整えることで、開発フローにテストを組み込むことができます。テストや静的解析を CI 上で実行し、すべてのチェックが成功したらコードレビュー、マージという流れで開発をスムーズに進めることができます。

この CI を導入するためには、先にブランチモデルが確立されている必要があります。既にブランチモデルが導入されていれば今まで手動でやっていたことを CI 上で自動化するだけになりますが、まだブランチモデルが曖昧な場合、ブランチモデルを決め、どのブランチと環境が一致しているのか、コードレビュー時や、デプロイ前にどのようなチェックを行うのか明確にする必要があります。

さらに、静的解析ツールを導入することで、開発フローをより洗練させることができます。静的解析とはテストとは異なる手法でコードに関する問題を検出する手法です。「1.3.2 Testing Trophy」でも一番下の土台となる部分として紹介されていました。静的解析はコードを実行せずに解析するため高速に動作します。また、開発フローに静的解析を組み込むことで、より早い段階で問題に気付くことができます。

この章では、ブランチモデルの導入から始め、次に CI の導入、最後に静的解析の導入という流れで進めていきます。

4.1 ブランチモデルの導入

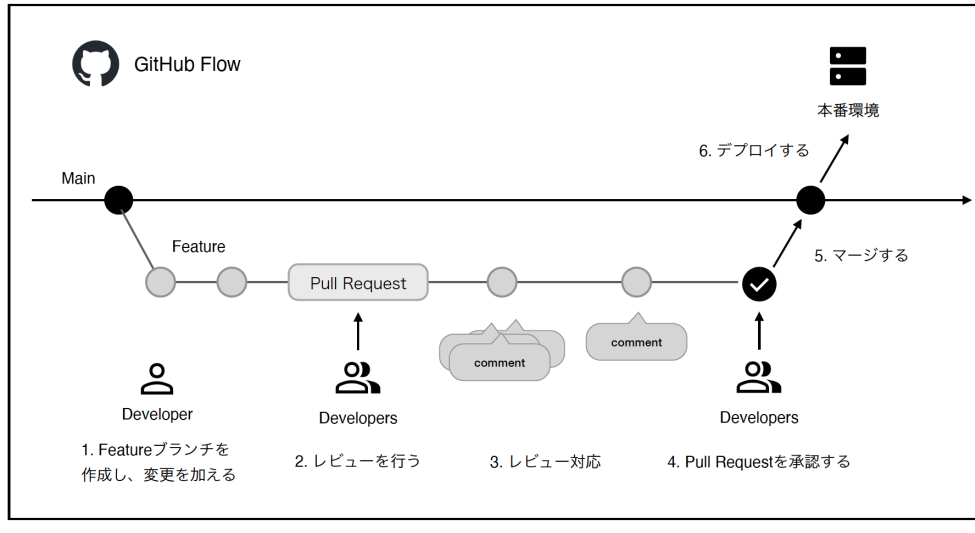
1 人で開発する場合は、ブランチモデルについてあまり気にする必要はなく、main ブランチにコミットを追加していくだけでも開発を進めることができますが、チーム開発ではそれだと上手くいかないことがあります。ブランチモデルを導入することで、複数のメンバーで開発する場合でも、変更が他のメンバーに与える影響を最小限に抑えたり、環境とブランチを一致させたり、変更を追加しやすくしたりできます。

どのブランチモデルを採用するかは、チームサイズやプロダクトの特性に応じて検討する必要があります。そのため、ここでは代表的な 2 つの軽量ブランチモデルである GitHub Flow とトラン

クベース開発の 2 つを紹介します。

4.1.1 GitHub Flow

GitHub Flow は、GitHub の機能である Pull Request の利用を前提とするブランチモデルです。ブランチは main と feature（作業用ブランチ）の 2 種類しか存在しません。main ブランチはベースとなるブランチで、feature ブランチは main ブランチに変更を加える際に利用するブランチです。feature（機能）ブランチという名前ですが、機能に関する変更に限らず、バグの修正・機能追加・変更・削除など、どのような変更であっても feature ブランチと呼びます。ブランチ名は変更内容によって好きにつけてよく、ブランチ名を feature としたり、ブランチ名の先頭に feature-という接頭辞をつけたりする必要はありません。



●図 4.1　GitHub Flow の流れ

Pull Request を利用することを前提としており、何か変更を加えたいときには feature ブランチを作成し、Pull Request を送り、レビューをしてもらう必要があります。Pull Request を利用することで非同期でコードレビューができ、またコード単位でコメントを付けることや、コミット単位で実行された CI のビルド結果をあわせて確認することもできます。すべてのフィードバックに対応し、他の開発者から承認をもらうことで、Pull Request を main ブランチへマージします。main ブランチにマージした後は、速やかに最新のコードを本番環境にデプロイします。デプロイに関する厳密なルールはありませんが、main ブランチと本番環境を一致させるため、main ブランチは常にデプロイ可能な状態にすることが望ましいでしょう。

GitHub Flow を利用したデプロイまでの流れ

GitHub Flow において開発者は次のステップで変更を加えデプロイします。

1. main ブランチから feature ブランチを作成し変更を加える
2. Pull Request を作成し、他の開発者からコードレビューを受ける
3. コードレビューに対応して修正を加える（2、3 は何度か繰り返す）
4. 他の開発者から Pull Request の承認をもらう
5. main ブランチへ Pull Request をマージする
6. main ブランチのコードをデプロイする

GitHub Flow はシンプルな流れで進めることができるため、多くのチームやプロジェクトで採用されています。main と feature という 2 つのブランチしかないため、変更を取り込みやすく、また Pull Request を利用することでレビューを徹底できます。そのため、変更を取り込みやすくしつつも内容を精査できるので、コードの品質を保つことができます。

ただし、GitHub Flow を導入するには、コードレビューを実施する文化や体制が整備されていることや、デプロイが自動化もしくは半自動化されていることなど、いくつかのハードルがあります。特に、チームの人数が少ない場合はコードレビューが大きな負担になる可能性があります。変更されたコードが動作することをテストで確認できていたとしても、コードレビューでは、コードが動作していること以外にもさまざまな観点でレビューする必要があります。Pull Request に含まれる変更が大きい場合や、変更の背景が説明にない場合、対応のしかたに問題がある場合など、Pull Request 自体に問題があると、レビュアーに大きな負担がかかります。そのような状況では、コードレビューは形骸化してしまい、ただ承認をもらうだけの儀式になってしまうことがあります。

Column. 軽量なブランチモデルでは検証はどうするのか？

プロダクトによっては、変更をいきなり本番環境にデプロイすることが難しい場合があり、事前に検証環境で確認したい場合があります。GitHub Flow やトランクベース開発などの軽量なブランチモデルでは、本番環境は main ブランチに紐付けられていますが、検証環境に紐付けられたブランチはありません。そのため、事前に機能を検証する場合には、次のような検証環境を用意するための工夫が必要になります。

- Pull Request 毎に検証環境を作成する。
- 検証環境に対応するブランチを作成する。例えば、staging-などの接頭辞をつけたブランチを作成し、staging-という接頭辞があった場合は検証環境にデプロイする。

しかし、検証環境を作成し、適切に運用するには、多くの場合コストがかかります。また、本番環境と完全に同じ環境を用意することは困難であり、さらに、大量のトラフィックを受ける本番環境でしか再現しない問題もあります。そのため、フィーチャーフラグやカナリアリリースといった方法を用いて、検証環境を使用せずに本番環境で機能を検証できます。

■ フィーチャーフラグとは？

フィーチャーフラグ（機能フラグやフィーチャートグルとも呼ばれる）とは、コードを変更せずに、フラグを利用してシステムの振る舞いを変更できるプラクティスです。新しいコードをリリースしても、フィーチャーフラグを有効化しなければ、システムの動作は変わりません。フラグを切り替えたタイミングで初めてシステムの動作を変えることができます。
フィーチャーフラグを管理するためには、単純にデータベースで管理する方法や、LaunchDarkly[注1)] や Optimizely[注2)] などのサービスを利用する方法があります。LaunchDarkly などのサービスを利用することで、特定の属性を持つユーザーに対して機能を開放するルールを簡単に作成し、ルールにマッチした場合にのみ機能を有効化できます。
CircleCI でも、単純にデータベースを利用したり、Optimizely を利用したりして、フィーチャーフラグを管理していました。新しい機能をリリースする際には、数 % のユーザーで機能を試し、問題がないことを確認してから徐々にユーザーを増やすことで、問題が起きた場合でも影響を最小限に抑えるようにしていました。

■ カナリアリリースとは？

カナリアリリースとは、新しいバージョンのアプリケーションを本番環境へリリースする際に、まず一部のユーザのみに公開して問題がないことを検証してから全体に公開する手法です。それにより、新バージョンに問題があった場合でもユーザへの影響を最小限に抑えられます。

軽量なブランチモデルでは、main ブランチに常にデプロイ可能な変更を追加していきます。しかしながら、変更によっては検証が必要な場合や、外部に公開するタイミングを調整したい場合があります。このような場合には、公開できない変更を main ブランチにマージできません。そのため、変更を含むブランチは時間が経つにつれて main ブランチとの差分が大きくなり、マージがより難しくなり、軽量なブランチモデルのメリットであった変更を追加しやすいという効果が半減してしまいます。

そこで、フィーチャーフラグやカナリアリリースなどを使用することで、変更を公開するタイミングと変更を取り込むタイミングを切り離すことができます。これらの手法は、本番環境での検証を前提としながら、公開する範囲を狭くし、徐々に広げていくことで、リスクを抑えながら検証できます。つまり、フィーチャーフラグやカナリアリリースは、軽量なブランチモデルのメリットを生かしつつ、変更を公開するリスクを最小限に抑えるための有用な手法と言えます。

4.1.2 トランクベース開発

GitHub Flow に似た軽量なブランチモデルの中にトランクベース開発があります。トランク（Trunk）とは木の幹という意味で、この幹というのは Git でいう main ブランチを表します。もともと Subversion という Git とは異なるバージョン管理システムでは Git の main ブランチと同

注1） https://launchdarkly.com/
注2） https://www.optimizely.com/

第5章

バックエンド開発と自動テスト

実際に自動テストを書きながら開発を進めようとすると、次から次へと頭の中に疑問が浮かんできます。

- いつテストを書けばいいんだろう
- どんなテストを書けばいいんだろう
- 何をテストするべきなんだろう
- （その他諸々の気になること……）

こんなときは、他の人の進め方をちょっとのぞいてみたいですよね。少しだけでもヒントが見つかれば、それを足がかりにして次の一歩を踏み出せます。そこで本書では実践例として、筆者（椎葉）の開発の進め方を紹介します。サンプルアプリケーションの開発を通して、自動テストを取り入れた開発について学んでいきましょう。

■ 正解というわけではないよ

これから紹介するのは、あくまでも筆者の進め方です。これが正解というわけではありません。何が適切かはプロダクトの性質やチームのスキル、組織の文化などの環境によって変わります。また、いろいろな考え方や好みもあります。筆者も今後勉強を続けていく中で考え方が変わるかもしれません。ここで最初の足がかりとしてひとつの進め方を学んだら、みなさんの状況や好みに合った進め方を見つけ出してください。

5.1 退職金の所得税計算アプリケーション

北新地のとある串カツ屋さんで紅生姜の串カツを食べながら「どんなサンプルアプリケーションにしようか」と著者間で話し合った結果「退職金の所得税を計算するウェブアプリケーションを作ろう！」と決まりました。退職金にかかる所得税を確認したいときに便利なアプリケーションです[注1)]。

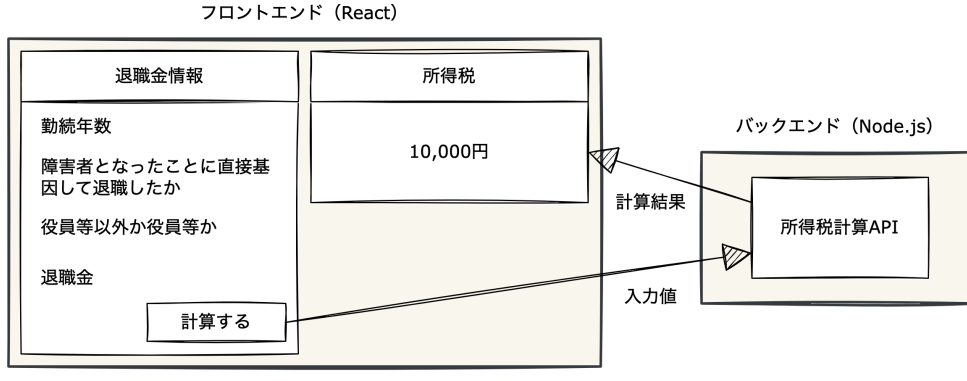
注1）　なぜ退職金の税金計算に興味を持ったかは……内緒です。

5.1.1　アプリケーション概要

次の項目を入力すると所得税を計算するシンプルなウェブアプリケーションです。

- 勤続年数
- 障害者となったことに直接基因して退職したか
- 役員等以外か役員等か
- 退職金

フロントエンドには React を、バックエンドには Node.js を使用します。



●図 5.1　退職金の所得税計算アプリケーション（イメージ）

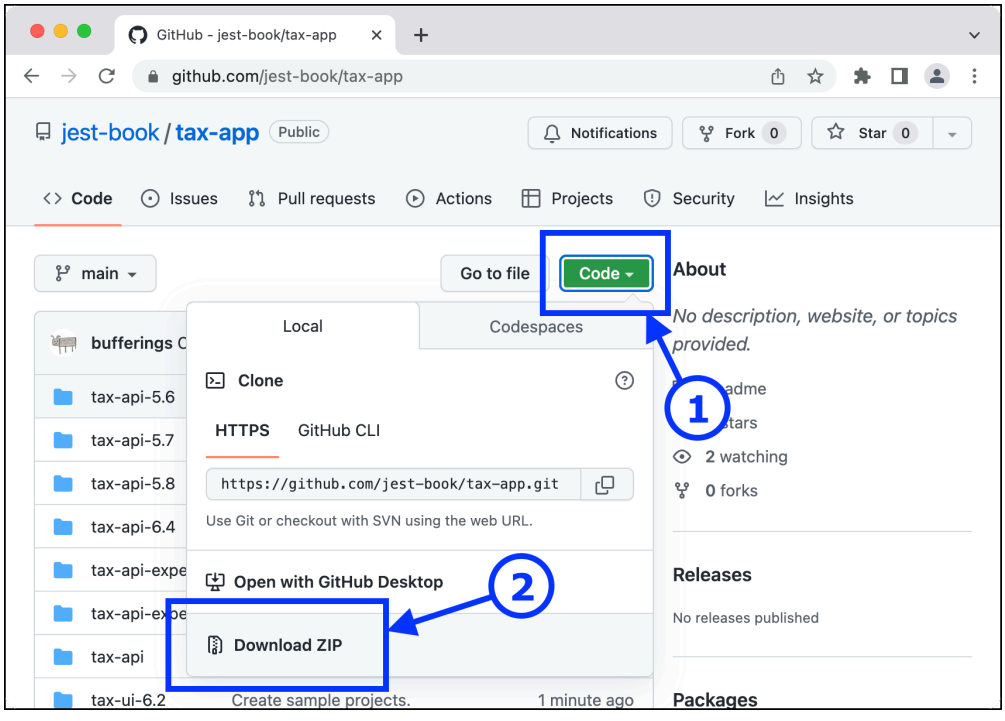
所得税の計算には、国税庁ホームページの「退職金と税」のページを参考にして、令和 4 年分の計算方法を使用します[注2)]。細かい計算は後ほど見ていきますが、退職所得控除を設ける、他の所得と分離して課税されるなど、税負担が通常よりも軽くなるよう配慮されています。退職金には所得税に加えて、住民税もかかりますが今回は所得税のみを取り扱います。

本章では自動テストを片手にバックエンドの開発を進めます。次章ではフロントエンド開発に取り組みます。もしよろしければ、みなさんも一緒にコードを書きながら読み進めてみてください。

5.1.2　サンプルコードのダウンロード

本章と次章で使用するサンプルコードは https://github.com/jest-book/tax-app にあります。筆者と一緒に手元で確認しながら読み進める方は、リポジトリの Zip ファイルをダウンロードして解凍しておいてください。フォークやクローンをしていただいても構いません。

注2）　退職金と税｜国税庁 https://www.nta.go.jp/publication/pamph/koho/kurashi/html/02_3.htm（2023 年 2 月参照）



●図 5.2　サンプルコードのダウンロード

サンプルコードのフォルダ構成を表 5.1 に記載しています。名前の最後に番号が付いているフォルダは参照用です。番号に対応した節を読み終えた状態になっていますので、コードを確認したいときにご参照ください。番号がついていないフォルダは、空のプロジェクトです。初期設定が終わった状態になっています。

●表 5.1　サンプルコードのフォルダ構成

フォルダ	用途
tax-api-experiment	最初に使用する空の実験用プロジェクト
tax-api-experiment-5.3	（参照用）5.3 節を読み終わった状態の実験用プロジェクト
tax-api	バックエンドの本実装に使用する空のプロジェクト
tax-api-n.m	（参照用）n.m 節を読み終わった状態のバックエンド用プロジェクト
tax-ui	フロントエンドの本実装に使用する空のプロジェクト
tax-ui-n.m	（参照用）n.m 節を読み終わった状態のフロントエンド用プロジェクト

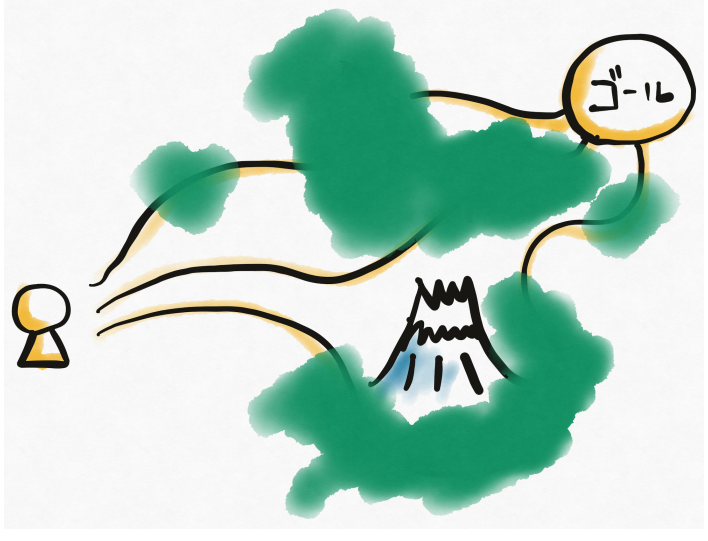
エディタには Visual Studio Code を使用しますが、お好みのエディタを使用していただいて問題ありません。ファイルの保存時に ESLint と Prettier が適用されるように設定しておくことをお勧めします。

5.2 退職金の所得税計算 API

それでは、退職金の所得税計算 API を作りましょう。Node.js と Express[注3)] を使用します。Express は Node.js 用のウェブアプリケーションフレームワークです。

実は筆者は、随分久しぶりに Node.js のアプリケーションを書くので、頭の中はまっさらな状態です。所得税の計算に対する実装はなんとなく想像できるため、特に不安はありません。しかし、Express で API をどのように作るかについては、全く覚えていません。

さて、どこから始めましょうか。まず自動テストを書いて考えましょうか。仕事で取り組む場合は、見積もりや設計から始めるかもしれませんね。このような状況で筆者がよく実践するのは、そのどちらでもなく「ひとまずゴールまで行ってみる」です。実現したいことに対して見えていない部分や不安に思う部分を解決しながら、素早くゴールまで行ってしまいます。



●図 5.3　ひとまずゴールまで行ってみる

このステップでは全体像の把握と素早さを最優先とします。そのため、できるだけ簡単な実装で動作を確認します。ときには自動テストも使用しますが、常に使用するわけではありません。ひとまずゴールまで行ってみて頭の中に道を描けたら、もう一度スタート地点に戻ります。そして、今度は自動テストを書きながら一歩ずつ丁寧に実装を進めます。このときには全体像が見えているので自信を持って進められます。

前置きはこれくらいにして、実際にひとまずゴールまで行ってみましょう。

注3)　https://expressjs.com/

■ **不安がないならスキップしても構わないよ**

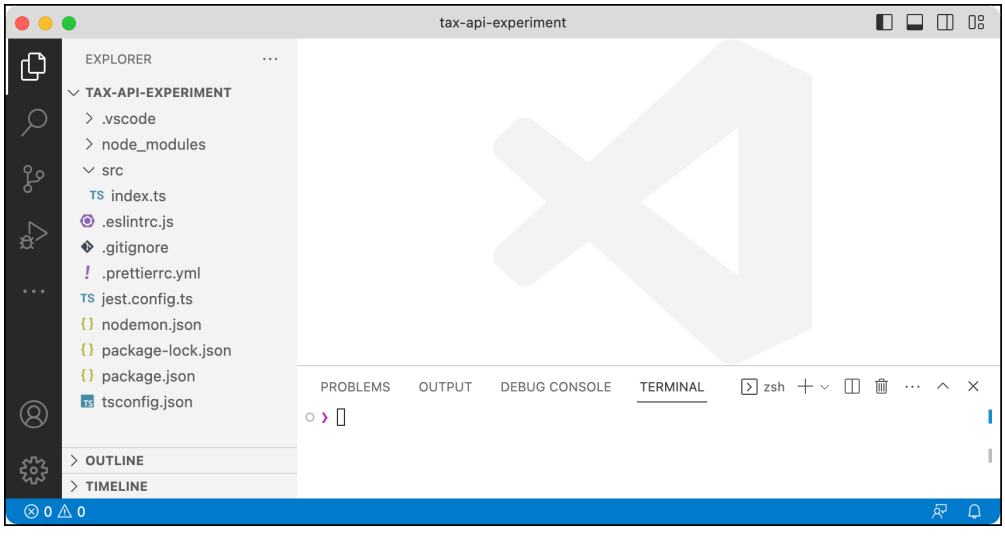
今の筆者のように見えていない部分がある場合は、ひとまずゴールまで行ってみることをお勧めします。しかし、読者のみなさんが既にその技術に詳しい、または同じようなプロダクトを過去に実装した経験があるなどの理由で不安がない場合は、このステップは飛ばしても構いません。

5.3 ひとまずゴールまで行ってみる

実験用のプロジェクトを準備します。サンプルコードの `tax-api-experiment` フォルダに移動し、次のコマンドを実行してライブラリをインストールしてください。

```
$ npm install
```

この実験用プロジェクトは本実装には利用しないので、気楽にいろいろなコードを書きましょう。



●図 5.4　実験用プロジェクト

プロジェクトの準備ができたので、確認したいことを TODO リストに書き出します。「退職金の所得税計算 API を作る」というゴールに対して、筆者が確認しておきたいのは次の 3 点です。

■ TODO リスト

- ■ リクエストを受けつける
- ■ JSON を返す
- ■ POST で JSON ボディを受け取る

「リクエストを受けつける」から確認します。

5.3.1 リクエストを受けつける

Express のドキュメントに目を通して簡単な実装を考えます。TypeScript を使用するため多少読み替えが必要ですが、次のように書けば HTTP のリクエストを受けつけられそうです。

● src/index.ts

```
import express from 'express'

const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

起動してみます。

```
$ npm run dev

> tax-api-experiment@0.1.0 dev
> nodemon src/index.ts

[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): src/**/*
[nodemon] watching extensions: js,ts,json
[nodemon] starting `ts-node src/index.ts`
Example app listening on port 3000
```

動くでしょうか。curl コマンドで確認します。

```
$ curl localhost:3000
Hello World!
```

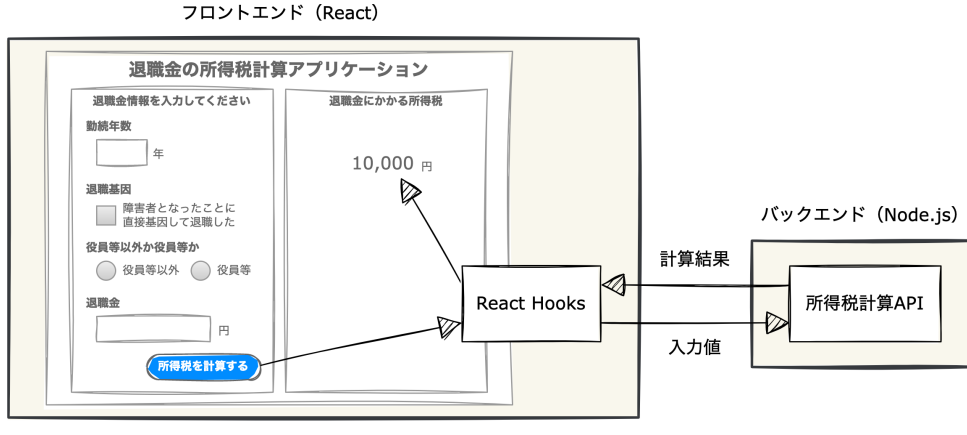
第 6 章

フロントエンド開発と自動テスト

フロントエンドアプリケーションの開発に入りましょう。フロントエンドの開発でも自動テストを使用しますが、その使い方はバックエンドとは少し異なります。本章ではフロントエンドのアプリケーション開発を通して、その違いを見ていきます。

さっそくですが、バックエンドで最初に取り組んだ「ひとまずゴールまで行ってみる」の説明はスキップします。フロントエンドでも実施するのですが、やることはバックエンドと同じで全体像の素早い確認です。みなさんは、もうやり方を知っているので大丈夫ですよね。ひとまずゴールまで行ってみて、少し立ち止まって考えた後の状態から始めます。それでは、違いを楽しみながら進めましょう。

6.1 退職金の所得税計算ページ



●図 6.1　フロントエンドアプリケーション概要

所得税計算 API の呼び出しには React Hooks を使用します。それによって、API アクセスのロジックを画面描画のロジックから切り離します。1 ページだけのアプリケーションなので、フォルダ構造は特に考えずに、`src` フォルダ直下へすべて置くことにします。TODO リストは、次のとおりです。

■ TODO リスト

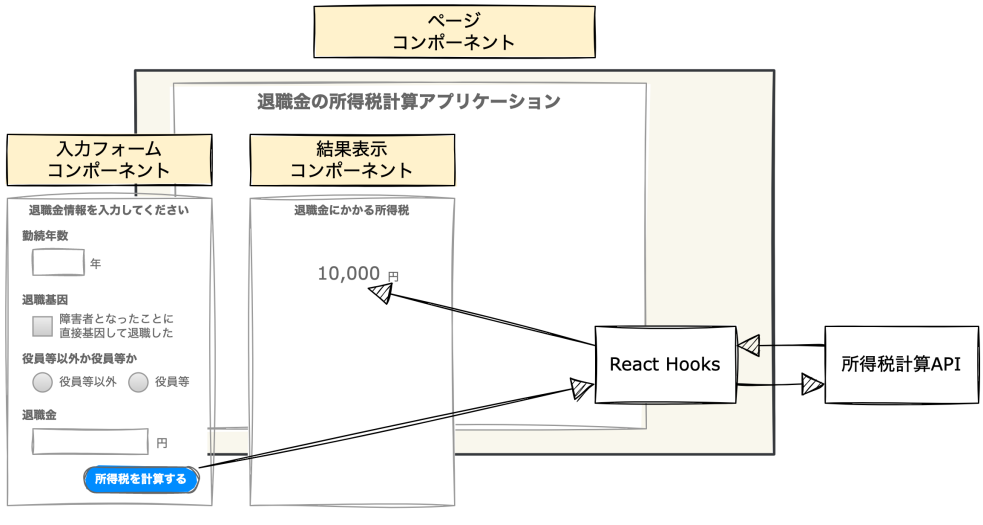
- ■ 見た目を実装
- ■ 振る舞いを実装

見た目と振る舞いの 2 つに大きく分けました。どちらから実装してもいいのですが、せっかくのフロントエンドなので見た目から実装しましょう。

プロジェクトを準備します。サンプルコードの tax-ui フォルダに移動し、次のコマンドを実行してライブラリをインストールしてください。

```
$ npm install
```

6.2 見た目を実装



●図 6.2　画面の構成

ページコンポーネント内に入力フォーム用のコンポーネントと、結果表示用のコンポーネントを配置します。入力フォームコンポーネントのサブミットボタンを押すと、ページコンポーネントがフックを使用して API へアクセスし、その結果を結果表示コンポーネントに渡します。見た目の実装として、次の 3 つのコンポーネントを作成します。

■ TODO リスト

- ■ 見た目を実装
  - **入力フォームコンポーネント**
  - **結果表示コンポーネント**
  - **ページコンポーネント**
- ■ 振る舞いを実装

6.2.1 何をガイドロープにするか

実装を始める前に、自動テストについて考えます。筆者は、これから作成する React のコンポーネントが想定どおりの見た目であることを確認しながら進みたいと考えています。関数コンポーネントを作成するので、自動テストによって、その関数が特定の入力に対して期待どおりの値を返すことをテストしたらいいのでしょうか。

確かに、関数としての動きはその自動テストで確認できるでしょう。しかし、それでは筆者の不安は拭えません。結局その自動テストでは、関数コンポーネントがどのように描画されるかを確認できないからです。バックエンドの開発は自動テストによって安心して進められましたが、見た目の実装は、やはり目で見て確認をしながら進めたいなと思います。

では、どうやってコンポーネントの見た目を確認しましょうか。実際にアプリケーションを起動して確認してもいいのですが、もっと便利な方法があります。それが「2.7.4 Storybook の活用」で紹介した Storybook です。Storybook を使うと、コンポーネントをアプリケーションに組み込まなくても見た目の確認ができます。また、後ほど説明するビジュアルリグレッションテストにも利用できます。

見た目の実装は、自動テストの代わりに Storybook をガイドロープにして進めます。

6.2.2 空のコンポーネントと Story を作成

入力フォームコンポーネントを作成しましょう。入力フォームの見た目だけの実装です。ボタンを押した場合の動作などの「振る舞い」は、ここでは実装しません。まずは Storybook を起動します。

```
$ npm run storybook
```

起動するとブラウザが開き、Storybook のページが表示されます。もし、ページが自動で開かない場合は http://localhost:6006 を直接開いてください。まだ何も Story を作っていないので、Story が存在しないことを示すエラー画面が表示されます。

●図 6.3　Storybook を起動

空の入力フォームコンポーネントを作成します。ファイル名は InputForm.tsx にしました。空とは言っても、何かを表示して確認したいので、Chakra UI[注1)]のボタンを置いてみます。今回はコンポーネントライブラリとして Chakra UI を使用します。Chakra UI は React 用のシンプルで使いやすいコンポーネントライブラリです。

● src/InputForm.tsx

```
import { Button } from '@chakra-ui/react'

export const InputForm = () => <Button colorScheme="blue">Hello</Button>
```

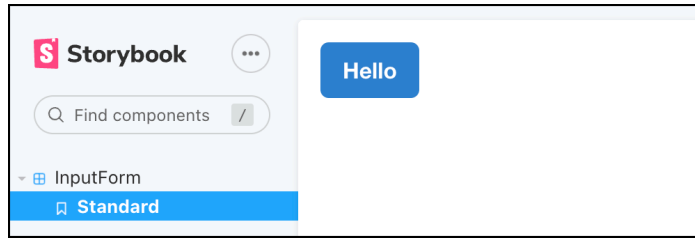
InputForm.stories.tsx という名前でファイルを作成して、入力フォームコンポーネント用の Story を用意します。Story は CSF 3.0 のスタイルで書きます。

● src/InputForm.stories.tsx

```
import { ComponentMeta, ComponentStoryObj } from '@storybook/react'

import { InputForm } from './InputForm'

export default {
  component: InputForm,
} as ComponentMeta<typeof InputForm>

export const Standard: ComponentStoryObj<typeof InputForm> = {}
```

入力フォームコンポーネントの表示を確認するために、Standard という名前の Story を作成しました。自動で変更が反映されるので、Storybook をリスタートする必要はありません。ページをリロードすると、新しく作成した Story がサイドバーに表示されます。InputForm/Standard を開いてください。

注1)　https://chakra-ui.com/



●図 6.4　空の入力フォームコンポーネント

ボタンが表示されました。Chakra UI のテーマも適用されています。これで、コンポーネントの見た目を確認しながら開発を進める準備ができました。

6.2.3　Storybook で確認しながら実装

入力フォームコンポーネントのコードを更新すると自動的に Storybook も更新されるので、見た目を確認しながら実装していきましょう。Storybook にエラーが表示された場合は、ページをリロードしてみてください。「ああでもない、こうでもない」と試行錯誤して、このような見た目になりました。

●図 6.5　入力フォームコンポーネント

筆者は今回はじめて Chakra UI を利用したのですが、とても使いやすいですね。CSS を書かず

に、コンポーネントのプロパティだけで見た目を設定できて便利です。コードはこのようになりました。

● **src/InputForm.tsx**

```
import {
  Button,
  Card,
  CardBody,
  CardHeader,
  CardProps,
  Center,
  Checkbox,
  FormControl,
  FormHelperText,
  FormLabel,
  Heading,
  HStack,
  Input,
  InputGroup,
  InputRightAddon,
  Radio,
  RadioGroup,
  Spacer,
  Stack,
  VStack,
} from '@chakra-ui/react'

export const InputForm = ({ ...props }: CardProps) => (
  <Card w="400px" {...props}>
    <CardHeader>
      <Center>
        <Heading as="h3" size="md">
          退職金情報を入力してください
        </Heading>
      </Center>
    </CardHeader>
    <CardBody>
      <form>
        <VStack spacing={5}>
          <FormControl>
            <FormLabel fontWeight="bold">勤続年数</FormLabel>
            <HStack>
              <InputGroup w="120px">
                <Input type="number" defaultValue="10" />
                <InputRightAddon>年</InputRightAddon>
              </InputGroup>
              <FormHelperText>1年未満の端数は切り上げ</FormHelperText>
            </HStack>
            <Spacer />
          </FormControl>
          <FormControl>
            <FormLabel fontWeight="bold">退職基因</FormLabel>
            <Checkbox>障害者となったことに直接基因して退職した</Checkbox>
```

```
          </FormControl>
          <FormControl>
            <FormLabel fontWeight="bold">役員等以外か役員等か</FormLabel>
            <RadioGroup defaultValue="0">
              <Stack direction="row">
                <Radio value="0">役員等以外</Radio>
                <Radio value="1">役員等</Radio>
              </Stack>
            </RadioGroup>
          </FormControl>
          <FormControl>
            <FormLabel fontWeight="bold">退職金</FormLabel>
            <InputGroup w="200px">
              <Input type="number" defaultValue="5000000" />
              <InputRightAddon>円</InputRightAddon>
            </InputGroup>
          </FormControl>

          <Button colorScheme="blue" alignSelf="flex-end" type="submit">
            所得税を計算する
          </Button>
        </VStack>
      </form>
    </CardBody>
  </Card>
)
```

これで、入力フォームコンポーネントが作成できました。Storybook で確認しながらテンポよく進められたので、思ったとおりの見た目のコンポーネントを快適に実装できました。

■ TODO リスト

- ■ 見た目を実装
  - **[OK]** 入力フォームコンポーネント
  - 結果表示コンポーネント
  - ページコンポーネント
- ■ 振る舞いを実装

■ 実際のプロジェクトでは

筆者はデザインのスキルが全然ありませんので、UI のデザインについては大目に見てください。実際のプロジェクトでは、今回のようにコンポーネントライブラリを直接利用したり、いきなりコンポーネントを配置したりすることは、あまりないのではないでしょうか。CircleCI では、デザイナーが Figma（https://www.figma.com/）で用意したデザインを元に、社内のデザインシステムからコンポーネントを選択して細かな調整をしていました。

おわりに

本を読むことがもともと好きで、いつか自分でも本を書いてみたいと思っていました。しかし、実際に書いてみると、いろいろと勝手が違うことがわかりました。これまで読んでいた本で読みにくい、分かりづらいと思うことは何度もありました。しかし、自分で文章を書いてみることで、読みやすく、違和感のない文章を書くことがいかに難しいことなのかわかりました。また、読みやすさを重視して書こうと思うと、ついつい無難な表現になってしまい、何も記憶に残らない文章になってしまうのではないかと、読みやすい文章も書けないのに無駄な心配をすることもありました。

私の好きなドラマに、ピース又吉さんの小説をドラマ化した「火花」があります。Netflix 限定のドラマです。そこで林遣都さん演じる芸人の徳永は「ストレートにいうと伝わらないことがあるから、敢えて思っていることの反対のことをいうことで思いを伝える」という漫才をするシーンがあるのですが、ややこしいけど、納得してしまったことがありました。技術書の中には 1 回読んだだけではわからず、何回か読んで、ぼんやりとわかってくることが実際にあります。そういった表現の方が後から思い出すことがあるなと思いました。そうすると、実は技術書の著者の方々は敢えて、そういった分かりづらい言葉を使って表現していたのではないかと思うのでした。

今回本を書くことで、改めて本を読むことの大切さや、本を読むことがいかにコストパフォーマンスに優れたことなのかを実感しました。本書を書くにあたり、構成を考えたり、検証したり、誤った情報を出さないように文献を確認したり、表現を改善したりと、膨大な作業が必要でした。そのため、今は読書したいという気持ちが高まっています。私の本棚には積読している本がたくさんあるので、読み進めたいと思っています。執筆が終わった今、時間がたっぷりあるのです。

余談ですが、去年の 12 月にレイオフがあり、CircleCI から離れることになりました。CircleCI の事業縮小に伴い、日本にあったエンジニアリングチームがすべて解体されました。第 5 章のサンプルコードの題材として、退職金の所得税計算を利用しましたが、これは私がレイオフになったことがキッカケでした。実際にスクリプトを利用して税金を計算しました。退職金をもらうことができたので、レイオフに関してはあまりネガティブな感想はなく、むしろ、レイオフがあったおかげで執筆の時間をなんとか確保できました。本書では退職金の具体的な数字は掲載されていませんが、実際にお会いする機会がありましたら聞いてみてください。

最後にとりとめもない話をしてしまいましたが、また、みなさんとお会いできるのを楽しみにしております。

伊藤 貴之

権利表記

本書の内容についてのお問い合わせは、はじめにをご覧ください。

索引

監修者

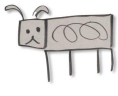
和田 卓人 @t_wada

プログラマ、テスト駆動開発者。タワーズ・クエスト株式会社取締役社長。学生時代にソフトウェア工学を学び、オブジェクト指向分析/設計に傾倒。執筆活動や講演、ハンズオンイベントなどを通じてテスト駆動開発の普及に努めている。『プログラマが知るべき97のこと』（オライリージャパン、2010）監修。『SQLアンチパターン』（オライリージャパン、2013）監訳。『テスト駆動開発』（オーム社、2017）翻訳。『事業をエンジニアリングする技術者たち』（ラムダノート、2022）編。

著者

伊藤 貴之 @ganezasan

大手のSIerやシード期のベンチャーで新規プロダクトの開発に携わる。2014年に地元の岩手で起業し、クラウドソーシング上でスマートフォン向けのアプリ開発などを行い、「Lancer of the Year 2015」にて「地域を元気にするランサー賞」を受賞。2018年にCircleCIに入社し、テクニカルサポートとして、また2021年から翌2022年12月までインフラエンジニアとして勤務。CI/CDちょっとできます。



椎葉 光行 @bufferings

大学在学中にアルバイトでプログラミングを覚え、卒業後はベンチャーで開発に携わる。2010年より楽天グループ株式会社に勤務。Javaのウェブアプリケーション開発に取り組む中で、テスト駆動開発に興味を持ちTDDBC大阪を開催。アジャイル開発や組織改善にも取り組み、スクラムフェス大阪2021ではキーノートを務める。2021年10月から翌2022年12月までCircleCIでシニアフルスタックエンジニアとして楽しくコードを書いて過ごした。休みの日は家族でぼーっと過ごすのが好き。

編集者

森田 尚

Jestではじめるテスト入門

2023年3月1日　電子書籍 初版第1刷 発行
2023年3月20日　電子書籍 v1.1.0 発行
2023年3月24日　電子書籍 v1.1.1 発行

著　　者： 伊藤貴之・椎葉光行
監 修 者： 和田卓人
編　　集： 森田尚

問い合わせ： PEAKS出版
東京都台東区北上野2丁目8番4号
https://peaks.cc/