12 Project: Build an expense tracker with Remix

This chapter covers

- Defining the desired goal for a project
- Figuring out how to use Remix to get to that goal

Congratulations. You've learned all I have to teach you. From here on out, you're ready to dive deeper and expand your knowledge at an impressive pace.

You're probably still early in your React developer career. The state of modern web development is such that you'll experience continual new development, better libraries, new tools, changing paradigms, and general chaos. You have to keep adapting; you can't rest on your laurels.

In these last three chapters, I'm going to give you some tougher challenges, all of which are going to involve a lot of self-study. You need to be good at reading online documentation and finding out how different bits go together. You'll not only be using tools in ways I haven't described in this book; you may use new tools that you haven't even heard about.

That experience is what your career is going to look like from now on. You'll rarely join a new project and instantly be familiar with every tool used in the stack. You'll almost always have something new to learn (or maybe even something old), so you might as well start getting used to this situation. Over the years, you might use hundreds of frontend libraries across dozens of frameworks, mastering the skill of learning new things quickly so that you can use them in a new setting.

In this first project, we're going to use Remix to create an expense tracker. The first section covers all the details about how the project is going to work and what it's going to look like. Then I'll explain what you will have to do. I've created three challenges in this project:

- *Starting from scratch* —First, if you want a big challenge, you can start from scratch with a minimal application that contains only basic user management. You can log in and sign up, and that's it. You have to build the rest.
- *Taking the backend challenge* —If you're a little less ambitious, or if you feel that you've created more than enough simple React components, you can try the backend challenge. In this challenge, you start with the basic framework of the entire application, and you get all the visual frontend components premade. You "only" have to extend the database and hook up all the forms, data loaders, and action handlers required to convert the application from dummy data to real, database-persisted data.
- *Taking the frontend challenge* —Finally, if you're a bit insecure about the whole backend thing, you can stick with the frontend challenge. You get a backend-only application with all the pages, data managers, loaders, and actions set up for you. But all the pages are missing the visual components that will make them look nice, so you have to create React components to display beautiful forms, visually pleasing UI components, and complex page layouts.

Oh, there's one other thing I haven't mentioned: when you use Remix, you almost always use TypeScript. Therefore, all these challenges have to be completed in TypeScript with all the types and interfaces required. Luckily, you get a lot for free because Prisma autogenerates a lot of relevant types based on the schema and because Remix itself is so well typed. See, there's your first curveball. Let's get to it!

Note The source code for the examples in this chapter is available at **https://reactlikea.pro/ch12**.

12.1 Creating the expense tracker

Figuring out how much you have left to spend on delicious cookies after all your regular expenses have been paid is a struggle for many people. In this project, you'll build a small simple expense tracker, in which you can enter your monthly income and all your regular expenses grouped by category. The application displays several pages before you log in, as you see in figure 12.1.

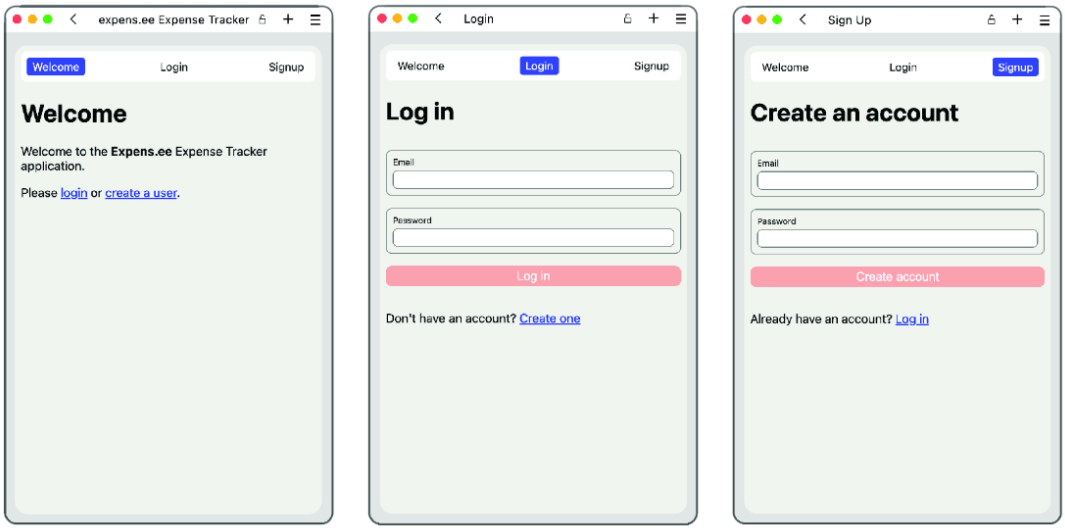

Figure 12.1 The three pages before you log in are a simple welcome page and the login and signup forms.

When you're logged in, you see a different set of pages (figure 12.2).

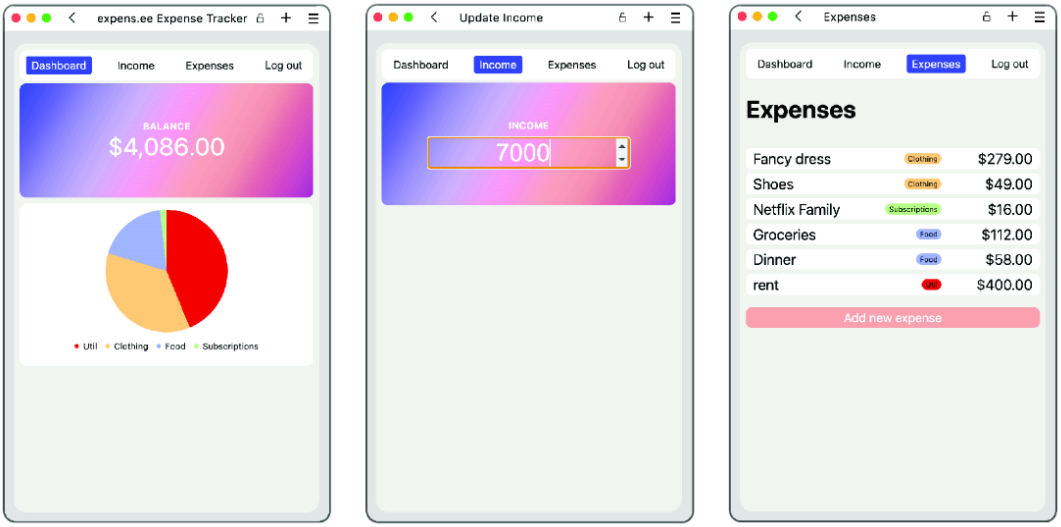

Figure 12.2 When you're logged in, you can view your dashboard, edit your monthly income, or see the list of expenses, to which you can add new expenses. This figure may not look like much in grayscale on a printed page, but believe me, it looks marvelous in full color!

The form for adding an expense has two different states, depending on whether you want to add a new category or pick an existing one (figure 12.3).

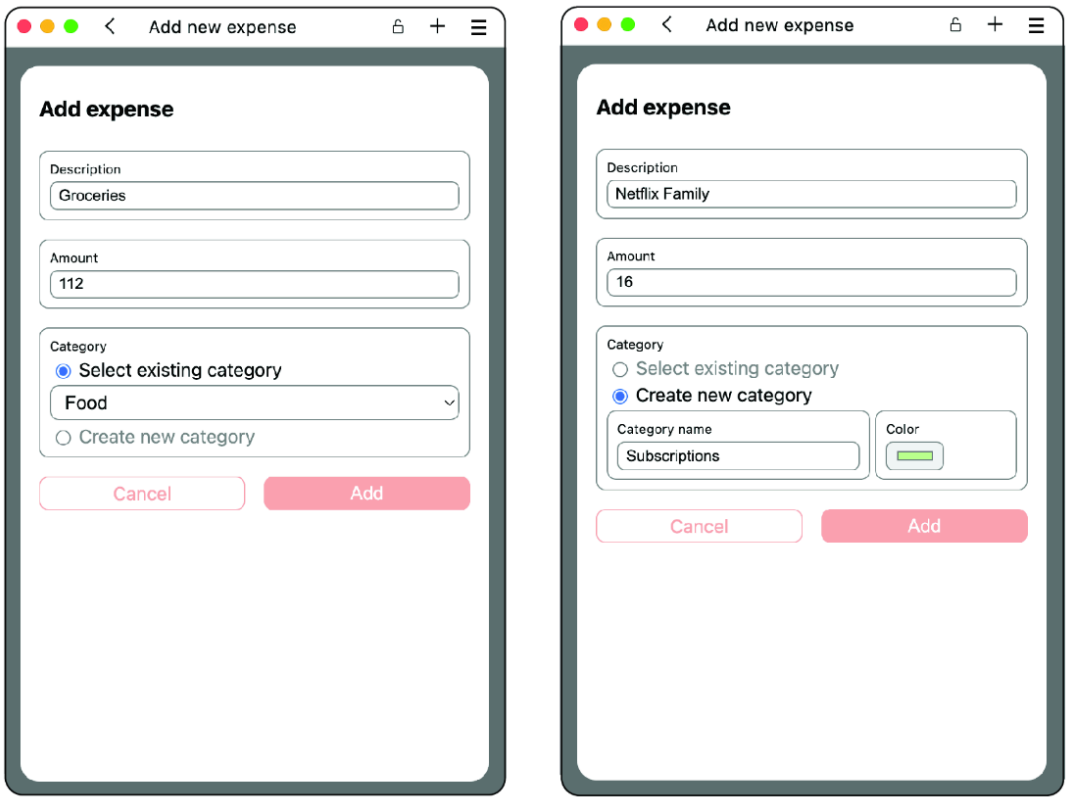

Figure 12.3 When you're adding a new expense, you can pick an existing category or add a new one with a specific color.

When you use the application yourself, you may notice that the URL changes as you navigate between pages, which is possible through the built-in routing system in Remix (discussed in chapter 11). Routing is actually central to the way Remix defines an application.

But that's enough talking and presenting screenshots. Why don't you try this application right now? Go to **https://www.expens.ee** and check it out. At that domain, you can try the exact expense-tracker application that you're going to build in this chapter. I've set up a database, deployed it to a real host, and pointed a domain at it. That's all it takes for a Remix project to go live!

12.1.1 Choosing your own adventure

As I mentioned in the start of this chapter, you get to choose your challenge based on complexity level. There are three levels, with the first being the hardest:

1. Build the entire application from scratch.
2. Start with all the frontend bits premade and build only the backend.
3. Start with all the backend bits premade and build only the frontend.

The rest of this chapter consists of a bunch of hints and a lot of links to show you how to go about completing this project. Please read on to see how to tackle the adventure of your choice.

Whichever challenge you choose, you should follow up by reading section 12.5, which rounds off the chapter and the project in general.

LEVEL 1: BUILD THE WHOLE APPLICATION

If you choose level 1, please open the project located in the `skeleton` folder.

EXAMPLE: SKELETON

This example is in the `skeleton` folder. Note that this example is non-standard and doesn't use the regular Vite setup, but a custom Remix setup. Before you run this example locally, make sure to run the setup (only once for this example):

```
$ npm run setup -w ch12/skeleton
```

Then you can run the example by running this command:

```
$ npm run dev -w ch12/skeleton
```

Alternatively, you can go to this website to browse the code or download the source code as a zip file: **https://reactlikea.pro/ch12-skeleton**.

This project contains only the user model and the pages related to the user (welcome, login, and signup)—nothing else. The app looks like figure 12.4. You have to take it from there to achieve the full functionality displayed in figures 12.1 through 12.3 earlier in this chapter.

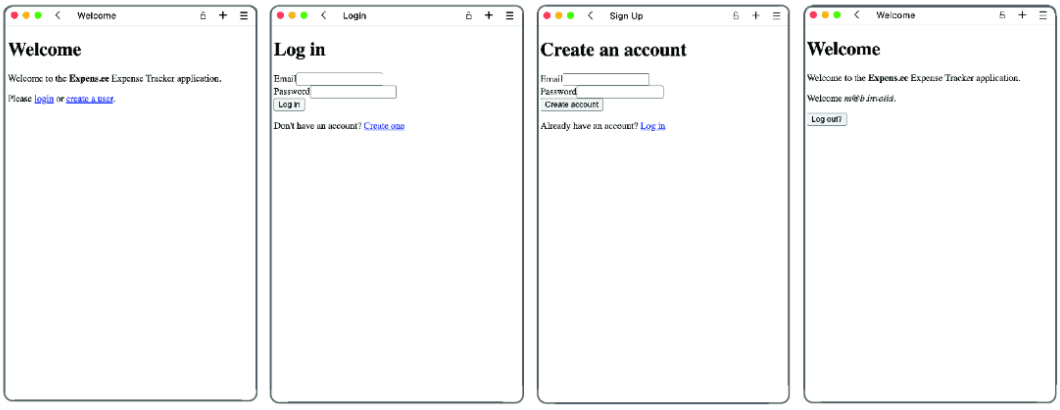

Figure 12.4 In the skeleton project, you have only a welcome screen (which varies a bit depending on whether you're logged in, as you can see in the first and last screens), as well as basic login and signup forms. There are no menus or fancy components, and the whole data layer for expenses and categories is missing.

Then go through sections 12.2, 12.3, and 12.4 in order, slowly building up the required structure, functionality, and the visual UI library to

make the application pop and make it pleasant to use.

LEVEL 2: BUILD ONLY THE DATA LAYER

If you choose level 2, please go to the project located in the `frontend-only` folder.

EXAMPLE: FRONTEND-ONLY

This example is in the `frontend-only` folder. Note that this example is nonstandard and doesn't use the regular Vite setup, but a custom Remix setup. Before you run this example locally, make sure to run the setup (only once for this example):

```
$ npm run setup -w ch12/frontend-only
```

Then you can run the example by running this command:

```
$ npm run dev -w ch12/frontend-only
```

Alternatively, you can go to this website to browse the code or download the source code as a zip file: **https://reactlikea.pro/ch12-frontend-only**.

As the name indicates, this project includes all the frontend bits required to complete the project. But the data structure and object-relational mapping (ORM) wrappers are missing, as well as all loaders and actions.

This project looks like figure 12.5. It seems to be complete, but all the data is dummy data, and you can't update or change anything.

Then go straight to section 12.3, which contains all the guidance you need to complete the project. You can ignore section 12.4, as it is irrelevant to this challenge.

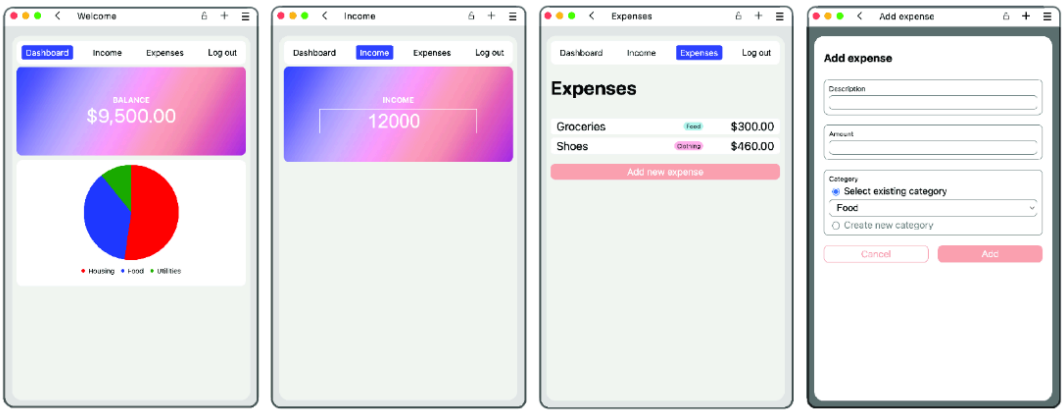

Figure 12.5 All the pages are there, and all the right UI components are displayed, but the data is static and noninteractive. It's your job to add the data layer to make this application fully functional. You can see that the data isn't real because the information on the various pages doesn't add up.

LEVEL 3: BUILD ONLY THE UI LAYER

Finally, if you choose level 3, please look at the project located in the `backend-only` folder.

EXAMPLE: BACKEND-ONLY

This example is in the `backend-only` folder. Note that this example is nonstandard and doesn't use the regular Vite setup, but a custom Remix setup. Before you run this example locally, make sure to run the setup (only once for this example):

```
$ npm run setup -w ch12/backend-only
```

Then you can run the example by running this command:

```
$ npm run dev -w ch12/backend-only
```

Alternatively, you can go to this website to browse the code or download the source code as a zip file: **https://reactlikea.pro/ch12-backend-only**.

As the name indicates, this project includes all the bits related to the database and to reading and writing data, but the presentation layer

leaves a lot to be desired. In this project, all the pages are there, and all the data is present, but you can't see it too well. The login/signup forms are very basic, as you see in figure 12.6.

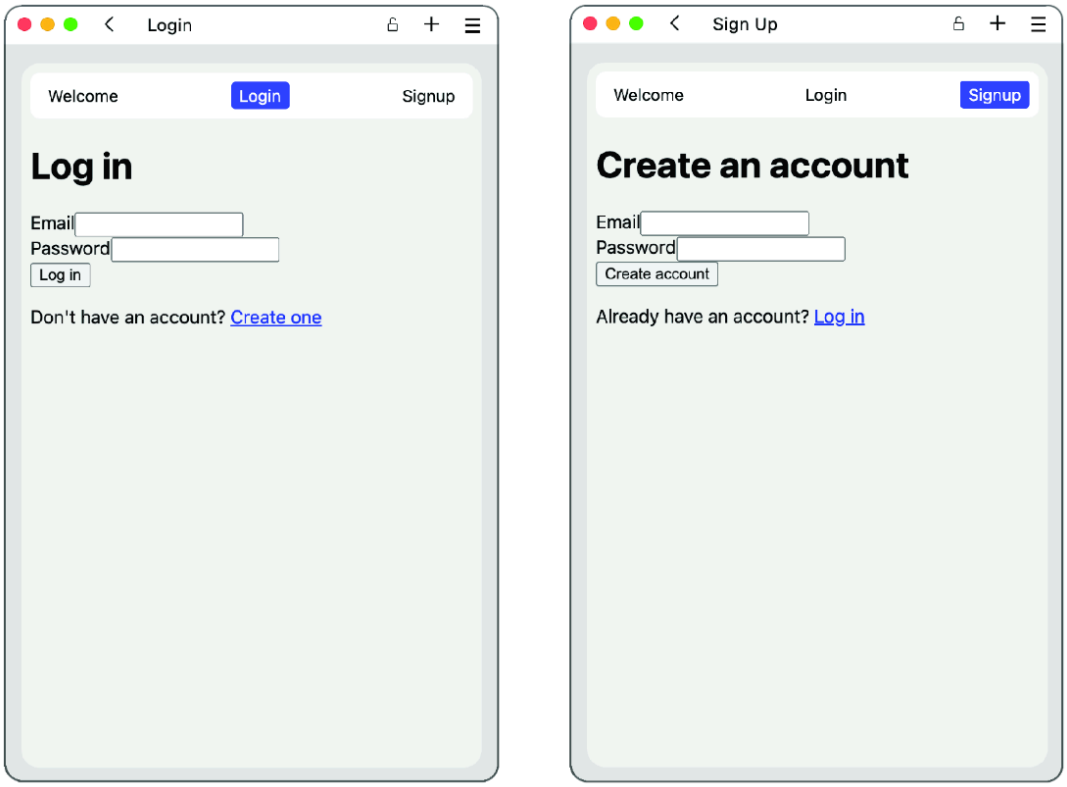

Figure 12.6 Albeit fully functional, these forms leave a bit to be desired UI-wise. The menus are there, though, so that's a start.

When you do log in, all the data pages contain only raw JSON data. No lovely data visualization or usable forms are in sight, as you see in figure 12.7.

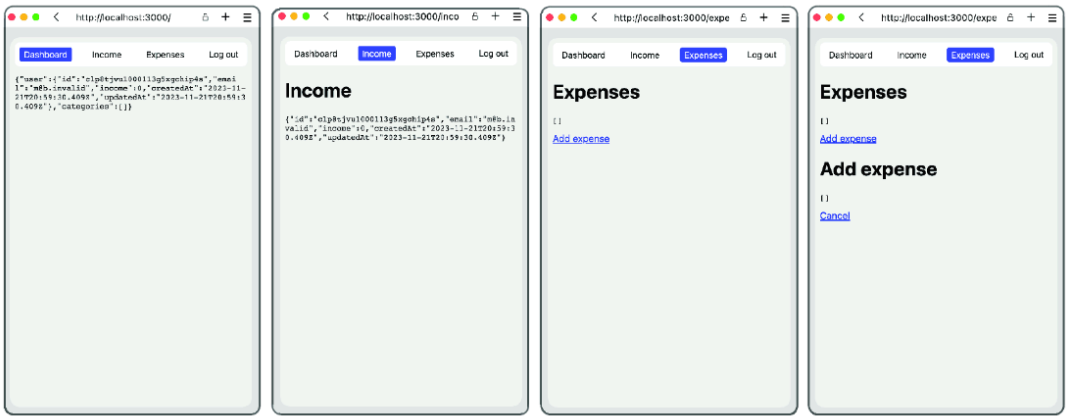

Figure 12.7 The backend-only project needs some visual UI components to make the application look great and be functional. The last screenshot doesn't even contain a dialog box—only more code. I mean, what is this, the Matrix?

Then go straight to section 12.4 and complete that section to get to the desired result.

SOLUTIONS

Whatever complexity level you choose, feel free to compare your solution with mine in the `complete` folder when you're done. You can also use my solution as a reference if you get stuck.

EXAMPLE: COMPLETE

This example is in the `complete` folder. Note that this example is non-standard and doesn't use the regular Vite setup, but a custom Remix setup. Before you run this example locally, make sure to run the setup (only once for this example):

```
$ npm run setup -w ch12/complete
```

Then you can run the example by running this command:

```
$ npm run dev -w ch12/complete
```

Alternatively, you can go to this website to browse the code or download the source code as a zip file: **https://reactlikea.pro/ch12-complete**.

12.2 Starting from scratch

If you start from scratch, please open the `ch12/skeleton` example in your editor. Then I recommend that you complete these steps before tackling anything else:

1. Create the basic visual framework for all the pages.
2. Extend the route system.

After completing these two tasks, you will be at the stage where you can add the data layer for the new types (described in section 12.3)

and then add the visual layer (as described in section 12.4).

12.2.1 Creating the basic visual framework

As you saw in figure 12.4 earlier in this chapter, the application is completely unstyled and even missing the base styles for the entire application, as well as the menu. Adding these two bits is a good effort at this stage, allowing you to shape the architecture around them.

The root styles are in `app/root.tsx`. Here, you see the main HTML structure for the entire web page, including the application. (Unlike in a Vite application, React also renders the `<html>` and `<body>` elements in Remix!) You can add styles directly in this file or import them from another file, as always. I suggest that you put the styles in a file such as `components/Root.tsx`, but anything is valid.

To add the menu, you can create a component for the menu itself and another for the individual menu items. The menu items should extend the `Link` component from Remix. You probably need to create two different menus, depending on the user's authorization status, but that task should be fairly straightforward. Remember that you need to know which menu item is the current one because it is highlighted differently.

One final bit of advice: when you add the logout button (to the menu for authorized users), you have to make that button a submit button inside a form, not a link. See the current `app/routes/_index.tsx` to see how this form should be defined. You can use the same styles as for the regular menu item; simply render the item as a button rather than a link and then wrap it in a Remix `Form` component with the expected properties.

12.2.2 Extending routes

Next, we need to add all the routes we're going to need to make this application serve our needs. We currently have routes for the front page, login page, signup page, and logout page. But we also need routes for the dashboard page (which is at the same route as the front page!), the income page, the expenses page, and the add-expense page.

The extra pages should reside at these URLs, so the file structure should resemble the following:

- *Dashboard page*
  - *URL*: `/`
  - *File*: `app/routes/_index.tsx`
- *Income page*
  - *URL* : `/income`
  - *File* : `app/routes/income.tsx`
- *Expenses page*
  - *URL* : `/expenses`
  - *File* : `app/routes/expenses.tsx`
- *Add-expense page*
  - *URL* : `/expenses/add`
  - *File* : `app/routes/expenses.add.tsx`

Because the dashboard page has to reside inside the same file as the front page, extend that route to handle the two different scenarios (using the `useOptionalUser` hook) and render the appropriate version of this page.

You can include some dummy content on each page. It would make sense to include the proper menu from the preceding step and make sure that it renders the correct menu item as the current one.

With this work done, we're ready to add the data types and migrations and then add actions and loaders to the routes so we can start reading and manipulating data.

12.3 Adding the backend

To add the backend, you need to complete these steps:

1. Extend the database.
2. Add new server-side wrapper functions for the ORM.
3. Update components to use Prisma-generated types.
4. Update the four new pages to use real database data rather than dummy data.

12.3.1 Extending the database

Next, update the schema with the extra attributes for the user and add the new types. The schema is defined in the file `prisma/schema.prisma`. The user has to be extended with an income property as well as lists of categories and expenses. The two new types are

- Categories that have an autogenerated ID, a name, and a color belong to a user and have a list of expenses.
- Expenses that have an autogenerated ID, an item field, and a value field belong to both a user and a category.

It is customary to add `createdAt` and `updatedAt` fields if for no other purpose than sorting. Please see how these same fields are defined for the user.

Next, you can add some default categories and expenses to the default user created in the seed file in `prisma/seed.ts`.

Finally, add and apply a migration to the database, and make sure to regenerate all types. To do all this, run these two commands:

```
$ npx prisma migrate dev
$ npx prisma generate
```

If you added new fixtures to the seed file, you can reset the database and run the seed by running this command:

```
$ npx prisma migrate reset
```

Note that this command clears the database.

With all that done, you should be ready to import the database and start working with the new types directly. If you're ever in doubt, remember to check the relevant documentation pages:

- *Prisma schema* — **https://www.prisma.io/docs/orm/prisma-schema**
- *Prisma seed* —**https://mng.bz/GZ0N**

TIP It can also be helpful to compare your types with the types from the simple blogging app in the Remix introductory tutorial: **https://remix.run/docs/en/main/tutorials/blog**.

12.3.2 Defining ORM wrappers

It is a general best practice in Remix to add model files in the `app/models/` folder for each type in your system to create higher-level utility functions around your ORM rather than access the ORM directly in your loaders and actions. First, you want to add a new function to the user model in `models/user.server.ts` to allow the updating of the income attribute. Then you want to create new files (probably based on the user model) named `models/category.server.ts` and `models/expense.server.ts` with all the functions required to read and write to these model types. Remember that the interactions are as follows:

- *Category*
  - You need to create a category for a given user with a given name and color.
  - You need to read all categories by a given user.
- *Expense*
  - You need to create an expense for a given user with a given item, value, and category ID.
  - You need to read all expenses and for each expense also read the name and color of its category (required for the expense list).
  - You need to get the sum of expenses grouped by category (required for the pie chart).

Again, refer to the Prisma documentation to see how to achieve the more complex bits:

- *Nested queries (for the category of an expense)* — https://mng.bz/rVwx
- *Aggregations and group by* —https://mng.bz/VxeN

These two files may be tricky to create, and you probably won't create them correctly the first time around. But as you develop the application further, you'll figure out the mistakes and make sure to add all the right arguments and return properties.

12.3.3 Updating components

NOTE This subsection does not apply if you are starting from scratch. If so, simply skip this section and go straight to section 12.3.4.

Update the types for the `Expense` and `PieChart` components to be based on the actual object types generated by Prisma rather than inline unconnected types. Remember that you can import these generated types directly from the Prisma client or from your own model ORM wrappers.

12.3.4 Adding server-side data to routes

The last bit required to make the routes fully data powered is to connect the data and models to the routes. You use loaders to load data and actions to manipulate data. The four pages need the following connections:

- *Dashboard* —On the dashboard page, add a loader for the categories (preferably with expenses summarized in an aggregation), and read income from the user object.
- *Income* —On the income page, add an action to update the income in the user object, and read the income from the same object.
- *Expenses* —On the expenses page, add a loader to read all the expenses with category information.
- *Add-expense* —On the add-expense page, add a loader to read the list of categories, and add an action to create the new expense with an existing category or a new category.

If you're in doubt, here are the relevant pages in the Remix documentation:

- *Loaders* —**https://remix.run/docs/en/main/guides/data-loading**
- *Actions* —**https://remix.run/docs/en/main/guides/data-writes**

12.4 Adding the frontend

To add the frontend, you need to complete these steps:

- Create a form component library.
- Create components for the dashboard page.
- Create components for the income page.
- Create components for the expenses list page.
- Create components for the add-expense form page.

12.4.1 Form library

First, let's make the login and signup forms look like they're supposed to. One way is to create a new form component library, perhaps in `app/components/Form.tsx`, where you can create the necessary elements. You can structure and name your components however you like, but the following list is common nomenclature:

- `Form` —The form itself, which probably wraps the `Form` from Remix
- `Group` —A wrapper for an input with a label
- `Label` —The label text that goes above the input (which doesn't have to be a `<label>` element; the `Group` might be the `<label>` element to allow you to skip the `htmlFor` attribute)
- `Input` —The input itself
- `Submit` —The submit button

Using these five components in the right structure should allow you to reuse the components perfectly and create lovely forms on both the login and signup pages.

12.4.2 Dashboard components

The dashboard page needs two components: the balance and a pie chart with a legend. The balance is fairly simple; the most complex part is crafting the lovely gradient behind it. You can go to a website such as CSS Gradient (**https://cssgradient.io/**) to generate a suitable gradient.

The pie chart is a bit more complex. You can render a pie chart manually by crafting a Scalable Vector Graphics (SVG) image, using an existing charting library, or (if you don't mind the chart's being completely static and noninteractive) using a conic gradient. You can find a great codepen for creating pie charts with conic gradients at **https://codepen.io/chriscoyier/pen/RPLqMg**. The legend is "only" a properly styled ordered list with a small colored bullet for each category.

Make sure to use the proper types for the inputs of these components. The balance component takes a number, and the pie chart component take a list of categories, with the sum of expenses being a property of each category. You can define the types for the pie chart component in TypeScript by combining the type of a `Category` with a `Pick` of the `value` property from the `Expense` type:

```
type CategoryWithValue = Category & Pick<Expense, "value">;
type Props = { items: CategoryWithValue[] };
function PieChart({ items }: Props) {
  ...
```

That example should be enough to get you started. But creating that pie chart is probably going to be tricky!

12.4.3 Income component

The income page needs an income-display component, which is similar to the balance- display component from the dashboard page, but here, the value will be displayed in an input. You probably don't want

to include the form itself inside the income component; you can create that on the income page.
Remember that you have two ways to render forms: controlled and uncontrolled. For this component, it makes sense to use an uncontrolled form because we probably don't need to edit the input value from the script if we make it a `type="number"` input. So, passing in a `defaultValue` (and a `name`, of course) may be the easiest way to go.

12.4.4 Expenses component

You need to style two things on the expenses page: a list of expenses and a styled link to the add form. The latter is easier; you can reuse the submit button style from the form library that you created but apply the CSS class name to a Remix `<Link />` component instead of a regular `<button>`.

You can make the expenses list in either of two ways:

- Create a single `ExpenseList` component that takes a list of expenses as an argument. (Remember that for the types, each expense has a category with a title and color.)
- Map the expenses directly on the expenses page to individual `Expense` components.

12.4.5 Add-expense component

On the add-expense page, extend the form element library with the required elements to add the extra input configurations, including radio groups and drop-down menus. Also, make sure the whole page renders with a dialog box on top of the page by positioning it absolutely and adding a semitransparent scrim behind it. (I used a 50% black background to make it stand out.)

12.5 Future work

Congratulations on building your first React + Remix application, regardless of which approach you took. Although the resulting application is decent, incorporating a lot of great features of Remix and show-

ing how you can use the platform, it may not be the most feature-complete application. If you haven't done so already, feel free to compare your solution with mine, located in the `complete` folder.
This section contains ideas for expanding the application to be more useful. I've provided some relevant hints but haven't completed any of the steps, so you're on your own!

12.5.1 Showing error messages

You may have noticed that we return JSON error messages in the actions on various pages—particularly on the login and signup pages but also on the add-expense form. But currently, we don't use those error messages for anything.

It would be helpful to display these error messages so that users know why their request failed. Perhaps their new password is too short, or they forgot to enter the amount for a new expense item.

To add this functionality, use the `useActionData` hook, which allows you to dip into the return values from a (failed) action. The documentation even describes this hook as being ideal for form-validation error messages and includes examples of how to use it; see **https://remix.run/docs/en/main/guides/form-validation**.

12.5.2 Editing and deleting objects

Allowing the user to create expenses is necessary to make the app functional, but you might make an error, and in the current state of the application, you can't do anything about it. You can't edit an expense; you can't even delete it. The same thing goes for categories. So, adding editing/deleting functionality would be useful.

There's nothing mysterious about adding this functionality, but remember that you have to go through the server. The Remix way is to use a form for that purpose, so you can't have a delete link; you must have a submit button inside a form. Hidden fields are a common way to inform the form handler of the user's intent. It is commonplace to use a field like this one inside the form:

```
<Form method="post">
  <input type="hidden" name="intent" value="delete" />
  <input type="hidden" name="expenseId" value={expenseId} />
  <button>Delete expense</button>
</Form>
```

Then, in your action, you can switch on the value of the `intent` parameter and handle the situation appropriately. This way, you can have many actions on the same route, such as update expense, delete expense, and delete category.

12.5.3 Making the pie chart interactive

The current pie chart is a graphic that shows the sizes of different slices, but it is static and not all that useful. Using SVG, you can draw a proper pie chart and annotate each slice to show what it represents, and you can allow the user to click a slice for more information.

You can do this job on your own, of course, but it's going to be complex. Using a library would probably be easier. You have many libraries to choose from, so feel free to search the internet for your favorite charting solution.

12.5.4 Filtering, ordering, and paginating the expense list

If you have many expenses—say, hundreds—the expense list becomes a bit unwieldy, as it lists all expenses in reverse order of creation (newest first). Adding filtering, sorting, and perhaps even pagination would make it much more useful.

You can easily do all these things by using the ORM. Please check the documentation on Prisma's `findMany` method (**https://mng.bz/x2w7** to see how to add `where` clauses for filtering, `orderBy` clauses for sorting, and `take` / `skip` properties for pagination.