Review Rating

The following documentation goes through an example implementation of a review rating platform for the food service industry on Sui. Unlike traditional review rating platforms that often do not disclose the algorithm used to rate reviews, this example uses an algorithm that is published on-chain for everyone to see and verify. The low gas cost of computation on Sui make it financially feasible to submit, score, and order all reviews on-chain.

There are four actors in the typical workflow of the Reviews Rating example.

Service owners are entities like restaurants that list their services on the platform. They want to attract more customers by receiving high-rated reviews for their services.

Service owners allocate a specific amount of SUI as a reward pool. Assets from the pool are used to provide rewards for high-rated reviews. A proof of experience (PoE) NFT confirms a reviewer used the service, which the reviewer can burn later to provide a verified review. Service owners provide their customers with unique identifiers (perhaps using QR codes) to identify individual reviewers.

Reviewers are consumers of services that use the review system. Reviewers provide feedback in the form of comments that detail specific aspects of the service as well as a star rating to inform others. The reviews are rated, with the most effective reviews getting the highest rating. Service owners award the 10 highest rated reviews for their service. How often the rewards are distributed is up to the service owner's discretion; for example, the rewards can be distributed once a week or once a month.

Review readers access reviews to make informed decisions on selecting services. Readers rate reviews by casting votes. The review readers' ratings are factored into the algorithm that rates the reviews, with the authors of the highest-rated reviews getting rewarded. Although it is not implemented as part of this guide, this example could be extended to award review readers a portion of the rewards for casting votes for reviews.

Moderators monitor content of the reviews and can delete any reviews that contain inappropriate content.

The incentive mechanism for moderators is not implemented for this guide, but service owners can all pay into a pool that goes to moderators on a rolling basis. People can stake moderators to influence what portion of the reward each moderator gets, up to a limit (similar to how validators are staked on chain), and moderator decisions are decided by quorum of stake weight. This process installs incentives for moderators to perform their job well.

The reviews are scored on chain using the following criteria:

There are several modules that create the backend logic for the example.

The dashboard.move module defines the Dashboard struct that groups services.

The services are grouped by attributes, which can be cuisine type, geographical location, operating hours, Google Maps ID, and so on. To keep it basic, the example stores only service_type (for example, fast food, Chinese, Italian).

A Dashboard is a shared object , so any service owner can register their service to a dashboard. A service owner should look for dashboards that best match their service attribute and register. A dynamic field stores the list of services that are registered to a dashboard. Learn more about dynamic fields in The Move Book . A service may be registered to multiple dashboards at the same time. For example, a Chinese-Italian fusion restaurant may be registered to both the Chinese and Italian dashboards.

See Shared versus Owned Objects for more information on the differences between object types.

This module defines the Review struct.

In addition to the content of a review, all the elements that are required to compute total score are stored in a Review object.

A Review is a shared object , so anyone can cast a vote on a review and update its total_score field. After total_score is updated, the update_top_reviews function can be called to update the top_reviews field of the Service object.

This module defines the Service struct that service owners manage.

The same amount is rewarded to top reviewers, and the reward is distributed to 10 participants at most. The pool of SUI tokens to be distributed to reviewers is stored in the reward_pool field, and the amount of SUI tokens awarded to each participant is configured in reward field.

Because anyone can submit a review for a service, Service is defined as a shared object. All the reviews are stored in the reviews field, which has ObjectTable type. The reviews are stored as children of the shared object, but they are still accessible by their ID . See Dynamic Collections in The Move Book for more information on ObjectTables .

In other words, anyone can go to a transaction explorer and find a review object by its object ID, but they won't be able to use a review as an input to a transaction by its object ID.

See Dynamic Collections in The Move Book for more information on the differences between Table and ObjectTable .

The top rated reviews are stored in top_reviews field, which has vector type. A simple vector can store the top rated reviews because the maximum number of reviews that can be rewarded is 10. The elements of top_reviews are sorted by the total_score of the reviews, with the highest rated reviews coming first. The vector contains the ID of the reviews, which can be used to retrieve content and vote count from the relevant reviews .

A reader can cast a vote on a review to rate it as follows:

Whenever someone casts a vote on a review, the total_score of the review is updated and the update_top_reviews function updates the top_reviews field, as needed. Casting a vote also triggers a reordering of the top_reviews field to ensure that the top rated reviews are always at the top.

This example follows a capabilities pattern to manage authorizations. For example, SERVICE OWNERS are given AdminCap and MODERATORS are given Moderator such that only they are allowed to perform privileged operations.

To learn more about the capabilities pattern, see The Move Book .

Navigate to the setup folder of the repository and execute the publish.sh script. Refer to the README instructions for deploying the smart contracts.

The frontend module is written in React, and is structured to provide a responsive user experience for interacting with a review rating platform. The page component supports user log in as a SERVICE OWNER , a MODERATOR , or a REVIEWER . A REVIEW READER role is not implemented for this example, but a REVIEWER can also read reviews and cast votes.

The frontend is a NextJS project, that follows the NextJS App Router project structure . The main code of the frontend is located in the app/src/ directory.

The main sub-directories are:

The Wallet Kit comes with a pre-built React.js component called ConnectButton that displays a button to connect and disconnect a wallet. The component handles connecting and disconnecting wallet logic.

Place the ConnectButton in the navigation bar for users to connect their wallets:

All the type definitions are in src/app/types/ .

Review and Service represent the review and service objects.

In the frontend, you might need to execute a transaction block in multiple places, hence it's better to extract the transaction execution logic and reuse it everywhere. Let's examine the execute transaction hook.

A Transaction is the input, sign it with the current connected wallet account, execute the transaction block, return the execution result, and finally display a basic toast message to indicate whether the transaction is successful or not.

Use the useWalletKit() hook from the Wallet Kit to retrieve the Sui client instance configured in WalletKitProvider . The signTransaction() function is another hook that helps to sign the transaction block using the currently connected wallet. It displays the UI for users to review and sign their transactions with their selected wallet. To execute a transaction block, the executeSignedTransaction() on the Sui client instance of the Sui TypeScript SDK. Use react-hot-toast as another dependency to toast transaction status to users.

Custom hooks: To keep the code as structured as possible, multiple custom hooks are utilized to manage the list of reviews associated with a service. The useGetReviews custom hook encapsulates the service, exposing all the required information (with fields such as nameOfService , listOfReviews , listOfStars) to display the reviews in a table. Multiple additional custom hooks, such as useDashboardCreation , and useServiceReview are encapsulating their own piece of state and logic to make the code readable and maintainable.

Component for adding a new review: The AddReview component is implemented to facilitate the creation of a new review. It is rendered by the servicePage to collect a review entry from a USER and uses the signAndExecuteTransaction function of the [useWalletKit] hook to execute the transaction.

Proof of experience generation: PoE is an NFT that is minted by SERVICE OWNER for customers after they dine at the restaurant; customers can then burn the PoE to write a high-rated review later. Minting an NFT is facilitated by the ownedServicePage component. This component is using the useServicePoEGeneration custom hook.

Delete a review: The moderator can delete a review that contains inappropriate content. moderatorRemovePage component is used to delete a review.

Reviews Rating repository .

Personas

There are four actors in the typical workflow of the Reviews Rating example.

Service owners are entities like restaurants that list their services on the platform. They want to attract more customers by receiving high-rated reviews for their services.

Service owners allocate a specific amount of SUI as a reward pool. Assets from the pool are used to provide rewards for high-rated reviews. A proof of experience (PoE) NFT confirms a reviewer used the service, which the reviewer can burn later to provide a verified review. Service owners provide their customers with unique identifiers (perhaps using QR codes) to identify individual reviewers.

Reviewers are consumers of services that use the review system. Reviewers provide feedback in the form of comments that detail specific aspects of the service as well as a star rating to inform others. The reviews are rated, with the most effective reviews getting the highest rating. Service owners award the 10 highest rated reviews for their service. How often the rewards are distributed is up to the service owner's discretion; for example, the rewards can be distributed once a week or once a month.

Review readers access reviews to make informed decisions on selecting services. Readers rate reviews by casting votes. The review readers' ratings are factored into the algorithm that rates the reviews, with the authors of the highest-rated reviews getting rewarded. Although it is not implemented as part of this guide, this example could be extended to award review readers a portion of the rewards for casting votes for reviews.

Moderators monitor content of the reviews and can delete any reviews that contain inappropriate content.

The incentive mechanism for moderators is not implemented for this guide, but service owners can all pay into a pool that goes to moderators on a rolling basis. People can stake moderators to influence what portion of the reward each moderator gets, up to a limit (similar to how validators are staked on chain), and moderator decisions are decided by quorum of stake weight. This process installs incentives for moderators to perform their job well.

The reviews are scored on chain using the following criteria:

There are several modules that create the backend logic for the example.

The dashboard.move module defines the Dashboard struct that groups services.

The services are grouped by attributes, which can be cuisine type, geographical location, operating hours, Google Maps ID, and so on. To keep it basic, the example stores only service_type (for example, fast food, Chinese, Italian).

A Dashboard is a shared object , so any service owner can register their service to a dashboard. A service owner should look for dashboards that best match their service attribute and register. A dynamic field stores the list of services that are registered to a dashboard. Learn more about dynamic fields in The Move Book . A service may be registered to multiple dashboards at the same time. For example, a Chinese-Italian fusion restaurant may be registered to both the Chinese and Italian dashboards.

See Shared versus Owned Objects for more information on the differences between object types.

This module defines the Review struct.

In addition to the content of a review, all the elements that are required to compute total score are stored in a Review object.

A Review is a shared object , so anyone can cast a vote on a review and update its total_score field. After total_score is updated, the update_top_reviews function can be called to update the top_reviews field of the Service object.

This module defines the Service struct that service owners manage.

The same amount is rewarded to top reviewers, and the reward is distributed to 10 participants at most. The pool of SUI tokens to be distributed to reviewers is stored in the reward_pool field, and the amount of SUI tokens awarded to each participant is configured in reward field.

Because anyone can submit a review for a service, Service is defined as a shared object. All the reviews are stored in the reviews field, which has ObjectTable type. The reviews are stored as children of the shared object, but they are still accessible by their ID . See Dynamic Collections in The Move Book for more information on ObjectTables .

In other words, anyone can go to a transaction explorer and find a review object by its object ID, but they won't be able to use a review as an input to a transaction by its object ID.

See Dynamic Collections in The Move Book for more information on the differences between Table and ObjectTable .

The top rated reviews are stored in top_reviews field, which has vector type. A simple vector can store the top rated reviews because the maximum number of reviews that can be rewarded is 10. The elements of top_reviews are sorted by the total_score of the reviews, with the highest rated reviews coming first. The vector contains the ID of the reviews, which can be used to retrieve content and vote count from the relevant reviews .

A reader can cast a vote on a review to rate it as follows:

Whenever someone casts a vote on a review, the total_score of the review is updated and the update_top_reviews function updates the top_reviews field, as needed. Casting a vote also triggers a reordering of the top_reviews field to ensure that the top rated reviews are always at the top.

This example follows a capabilities pattern to manage authorizations. For example, SERVICE OWNERS are given AdminCap and MODERATORS are given Moderator such that only they are allowed to perform privileged operations.

To learn more about the capabilities pattern, see The Move Book .

Navigate to the setup folder of the repository and execute the publish.sh script. Refer to the README instructions for deploying the smart contracts.

The frontend module is written in React, and is structured to provide a responsive user experience for interacting with a review rating platform. The page component supports user log in as a SERVICE OWNER , a MODERATOR , or a REVIEWER . A REVIEW READER role is not implemented for this example, but a REVIEWER can also read reviews and cast votes.

The frontend is a NextJS project, that follows the NextJS App Router project structure . The main code of the frontend is located in the app/src/ directory.

The main sub-directories are:

The Wallet Kit comes with a pre-built React.js component called ConnectButton that displays a button to connect and disconnect a wallet. The component handles connecting and disconnecting wallet logic.

Place the ConnectButton in the navigation bar for users to connect their wallets:

All the type definitions are in src/app/types/ .

Review and Service represent the review and service objects.

In the frontend, you might need to execute a transaction block in multiple places, hence it's better to extract the transaction execution logic and reuse it everywhere. Let's examine the execute transaction hook.

A Transaction is the input, sign it with the current connected wallet account, execute the transaction block, return the execution result, and finally display a basic toast message to indicate whether the transaction is successful or not.

Use the useWalletKit() hook from the Wallet Kit to retrieve the Sui client instance configured in WalletKitProvider . The signTransaction() function is another hook that helps to sign the transaction block using the currently connected wallet. It displays the UI for users to review and sign their transactions with their selected wallet. To execute a transaction block, the executeSignedTransaction() on the Sui client instance of the Sui TypeScript SDK. Use react-hot-toast as another dependency to toast transaction status to users.

Custom hooks: To keep the code as structured as possible, multiple custom hooks are utilized to manage the list of reviews associated with a service. The useGetReviews custom hook encapsulates the service, exposing all the required information (with fields such as nameOfService , listOfReviews , listOfStars) to display the reviews in a table. Multiple additional custom hooks, such as useDashboardCreation , and useServiceReview are encapsulating their own piece of state and logic to make the code readable and maintainable.

Component for adding a new review: The AddReview component is implemented to facilitate the creation of a new review. It is rendered by the servicePage to collect a review entry from a USER and uses the signAndExecuteTransaction function of the [useWalletKit] hook to execute the transaction.

Proof of experience generation: PoE is an NFT that is minted by SERVICE OWNER for customers after they dine at the restaurant; customers can then burn the PoE to write a high-rated review later. Minting an NFT is facilitated by the ownedServicePage component. This component is using the useServicePoEGeneration custom hook.

Delete a review: The moderator can delete a review that contains inappropriate content. moderatorRemovePage component is used to delete a review.

Reviews Rating repository .

How reviews are scored

The reviews are scored on chain using the following criteria:

There are several modules that create the backend logic for the example.

The dashboard.move module defines the Dashboard struct that groups services.

The services are grouped by attributes, which can be cuisine type, geographical location, operating hours, Google Maps ID, and so on. To keep it basic, the example stores only service_type (for example, fast food, Chinese, Italian).

A Dashboard is a shared object , so any service owner can register their service to a dashboard. A service owner should look for dashboards that best match their service attribute and register. A dynamic field stores the list of services that are registered to a dashboard. Learn more about dynamic fields in The Move Book . A service may be registered to multiple dashboards at the same time. For example, a Chinese-Italian fusion restaurant may be registered to both the Chinese and Italian dashboards.

See Shared versus Owned Objects for more information on the differences between object types.

This module defines the Review struct.

In addition to the content of a review, all the elements that are required to compute total score are stored in a Review object.

A Review is a shared object , so anyone can cast a vote on a review and update its total_score field. After total_score is updated, the update_top_reviews function can be called to update the top_reviews field of the Service object.

This module defines the Service struct that service owners manage.

The same amount is rewarded to top reviewers, and the reward is distributed to 10 participants at most. The pool of SUI tokens to be distributed to reviewers is stored in the reward_pool field, and the amount of SUI tokens awarded to each participant is configured in reward field.

Because anyone can submit a review for a service, Service is defined as a shared object. All the reviews are stored in the reviews field, which has ObjectTable type. The reviews are stored as children of the shared object, but they are still accessible by their ID . See Dynamic Collections in The Move Book for more information on ObjectTables .

In other words, anyone can go to a transaction explorer and find a review object by its object ID, but they won't be able to use a review as an input to a transaction by its object ID.

See Dynamic Collections in The Move Book for more information on the differences between Table and ObjectTable .

The top rated reviews are stored in top_reviews field, which has vector type. A simple vector can store the top rated reviews because the maximum number of reviews that can be rewarded is 10. The elements of top_reviews are sorted by the total_score of the reviews, with the highest rated reviews coming first. The vector contains the ID of the reviews, which can be used to retrieve content and vote count from the relevant reviews .

A reader can cast a vote on a review to rate it as follows:

Whenever someone casts a vote on a review, the total_score of the review is updated and the update_top_reviews function updates the top_reviews field, as needed. Casting a vote also triggers a reordering of the top_reviews field to ensure that the top rated reviews are always at the top.

This example follows a capabilities pattern to manage authorizations. For example, SERVICE OWNERS are given AdminCap and MODERATORS are given Moderator such that only they are allowed to perform privileged operations.

To learn more about the capabilities pattern, see The Move Book .

Navigate to the setup folder of the repository and execute the publish.sh script. Refer to the README instructions for deploying the smart contracts.

The frontend module is written in React, and is structured to provide a responsive user experience for interacting with a review rating platform. The page component supports user log in as a SERVICE OWNER , a MODERATOR , or a REVIEWER . A REVIEW READER role is not implemented for this example, but a REVIEWER can also read reviews and cast votes.

The frontend is a NextJS project, that follows the NextJS App Router project structure . The main code of the frontend is located in the app/src/ directory.

The main sub-directories are:

The Wallet Kit comes with a pre-built React.js component called ConnectButton that displays a button to connect and disconnect a wallet. The component handles connecting and disconnecting wallet logic.

Place the ConnectButton in the navigation bar for users to connect their wallets:

All the type definitions are in src/app/types/ .

Review and Service represent the review and service objects.

In the frontend, you might need to execute a transaction block in multiple places, hence it's better to extract the transaction execution logic and reuse it everywhere. Let's examine the execute transaction hook.

A Transaction is the input, sign it with the current connected wallet account, execute the transaction block, return the execution result, and finally display a basic toast message to indicate whether the transaction is successful or not.

Use the useWalletKit() hook from the Wallet Kit to retrieve the Sui client instance configured in WalletKitProvider . The signTransaction() function is another hook that helps to sign the transaction block using the currently connected wallet. It displays the UI for users to review and sign their transactions with their selected wallet. To execute a transaction block, the executeSignedTransaction() on the Sui client instance of the Sui TypeScript SDK. Use react-hot-toast as another dependency to toast transaction status to users.

Custom hooks: To keep the code as structured as possible, multiple custom hooks are utilized to manage the list of reviews associated with a service. The useGetReviews custom hook encapsulates the service, exposing all the required information (with fields such as nameOfService , listOfReviews , listOfStars) to display the reviews in a table. Multiple additional custom hooks, such as useDashboardCreation , and useServiceReview are encapsulating their own piece of state and logic to make the code readable and maintainable.

Component for adding a new review: The AddReview component is implemented to facilitate the creation of a new review. It is rendered by the servicePage to collect a review entry from a USER and uses the signAndExecuteTransaction function of the [useWalletKit] hook to execute the transaction.

Proof of experience generation: PoE is an NFT that is minted by SERVICE OWNER for customers after they dine at the restaurant; customers can then burn the PoE to write a high-rated review later. Minting an NFT is facilitated by the ownedServicePage component. This component is using the useServicePoEGeneration custom hook.

Delete a review: The moderator can delete a review that contains inappropriate content. moderatorRemovePage component is used to delete a review.

Reviews Rating repository .

Smart contracts

There are several modules that create the backend logic for the example.

The dashboard.move module defines the Dashboard struct that groups services.

The services are grouped by attributes, which can be cuisine type, geographical location, operating hours, Google Maps ID, and so on. To keep it basic, the example stores only service_type (for example, fast food, Chinese, Italian).

A Dashboard is a shared object , so any service owner can register their service to a dashboard. A service owner should look for dashboards that best match their service attribute and register. A dynamic field stores the list of services that are registered to a dashboard. Learn more about dynamic fields in The Move Book . A service may be registered to multiple dashboards at the same time. For example, a Chinese-Italian fusion restaurant may be registered to both the Chinese and Italian dashboards.

See Shared versus Owned Objects for more information on the differences between object types.

This module defines the Review struct.

In addition to the content of a review, all the elements that are required to compute total score are stored in a Review object.

A Review is a shared object , so anyone can cast a vote on a review and update its total_score field. After total_score is updated, the update_top_reviews function can be called to update the top_reviews field of the Service object.

This module defines the Service struct that service owners manage.

The same amount is rewarded to top reviewers, and the reward is distributed to 10 participants at most. The pool of SUI tokens to be distributed to reviewers is stored in the reward_pool field, and the amount of SUI tokens awarded to each participant is configured in reward field.

Because anyone can submit a review for a service, Service is defined as a shared object. All the reviews are stored in the reviews field, which has ObjectTable type. The reviews are stored as children of the shared object, but they are still accessible by their ID . See Dynamic Collections in The Move Book for more information on ObjectTables .

In other words, anyone can go to a transaction explorer and find a review object by its object ID, but they won't be able to use a review as an input to a transaction by its object ID.

See Dynamic Collections in The Move Book for more information on the differences between Table and ObjectTable .

The top rated reviews are stored in top_reviews field, which has vector type. A simple vector can store the top rated reviews because the maximum number of reviews that can be rewarded is 10. The elements of top_reviews are sorted by the total_score of the reviews, with the highest rated reviews coming first. The vector contains the ID of the reviews, which can be used to retrieve content and vote count from the relevant reviews .

A reader can cast a vote on a review to rate it as follows:

Whenever someone casts a vote on a review, the total_score of the review is updated and the update_top_reviews function updates the top_reviews field, as needed. Casting a vote also triggers a reordering of the top_reviews field to ensure that the top rated reviews are always at the top.

This example follows a capabilities pattern to manage authorizations. For example, SERVICE OWNERS are given AdminCap and MODERATORS are given Moderator such that only they are allowed to perform privileged operations.

To learn more about the capabilities pattern, see The Move Book .

Navigate to the setup folder of the repository and execute the publish.sh script. Refer to the README instructions for deploying the smart contracts.

The frontend module is written in React, and is structured to provide a responsive user experience for interacting with a review rating platform. The page component supports user log in as a SERVICE OWNER , a MODERATOR , or a REVIEWER . A REVIEW READER role is not implemented for this example, but a REVIEWER can also read reviews and cast votes.

The frontend is a NextJS project, that follows the NextJS App Router project structure . The main code of the frontend is located in the app/src/ directory.

The main sub-directories are:

The Wallet Kit comes with a pre-built React.js component called ConnectButton that displays a button to connect and disconnect a wallet. The component handles connecting and disconnecting wallet logic.

Place the ConnectButton in the navigation bar for users to connect their wallets:

All the type definitions are in src/app/types/ .

Review and Service represent the review and service objects.

In the frontend, you might need to execute a transaction block in multiple places, hence it's better to extract the transaction execution logic and reuse it everywhere. Let's examine the execute transaction hook.

A Transaction is the input, sign it with the current connected wallet account, execute the transaction block, return the execution result, and finally display a basic toast message to indicate whether the transaction is successful or not.

Use the useWalletKit() hook from the Wallet Kit to retrieve the Sui client instance configured in WalletKitProvider . The signTransaction() function is another hook that helps to sign the transaction block using the currently connected wallet. It displays the UI for users to review and sign their transactions with their selected wallet. To execute a transaction block, the executeSignedTransaction() on the Sui client instance of the Sui TypeScript SDK. Use react-hot-toast as another dependency to toast transaction status to users.

Custom hooks: To keep the code as structured as possible, multiple custom hooks are utilized to manage the list of reviews associated with a service. The useGetReviews custom hook encapsulates the service, exposing all the required information (with fields such as nameOfService , listOfReviews , listOfStars) to display the reviews in a table. Multiple additional custom hooks, such as useDashboardCreation , and useServiceReview are encapsulating their own piece of state and logic to make the code readable and maintainable.

Component for adding a new review: The AddReview component is implemented to facilitate the creation of a new review. It is rendered by the servicePage to collect a review entry from a USER and uses the signAndExecuteTransaction function of the [useWalletKit] hook to execute the transaction.

Proof of experience generation: PoE is an NFT that is minted by SERVICE OWNER for customers after they dine at the restaurant; customers can then burn the PoE to write a high-rated review later. Minting an NFT is facilitated by the ownedServicePage component. This component is using the useServicePoEGeneration custom hook.

Delete a review: The moderator can delete a review that contains inappropriate content. moderatorRemovePage component is used to delete a review.

Reviews Rating repository .

Deployment

Navigate to the setup folder of the repository and execute the publish.sh script. Refer to the README instructions for deploying the smart contracts.

The frontend module is written in React, and is structured to provide a responsive user experience for interacting with a review rating platform. The page component supports user log in as a SERVICE OWNER , a MODERATOR , or a REVIEWER . A REVIEW READER role is not implemented for this example, but a REVIEWER can also read reviews and cast votes.

The frontend is a NextJS project, that follows the NextJS App Router project structure . The main code of the frontend is located in the app/src/ directory.

The main sub-directories are:

The Wallet Kit comes with a pre-built React.js component called ConnectButton that displays a button to connect and disconnect a wallet. The component handles connecting and disconnecting wallet logic.

Place the ConnectButton in the navigation bar for users to connect their wallets:

All the type definitions are in src/app/types/ .

Review and Service represent the review and service objects.

In the frontend, you might need to execute a transaction block in multiple places, hence it's better to extract the transaction execution logic and reuse it everywhere. Let's examine the execute transaction hook.

A Transaction is the input, sign it with the current connected wallet account, execute the transaction block, return the execution

result, and finally display a basic toast message to indicate whether the transaction is successful or not.

Use the useWalletKit() hook from the Wallet Kit to retrieve the Sui client instance configured in WalletKitProvider . The signTransaction() function is another hook that helps to sign the transaction block using the currently connected wallet. It displays the UI for users to review and sign their transactions with their selected wallet. To execute a transaction block, the executeSignedTransaction() on the Sui client instance of the Sui TypeScript SDK. Use react-hot-toast as another dependency to toast transaction status to users.

Custom hooks: To keep the code as structured as possible, multiple custom hooks are utilized to manage the list of reviews associated with a service. The useGetReviews custom hook encapsulates the service, exposing all the required information (with fields such as nameOfService , listOfReviews , listOfStars) to display the reviews in a table. Multiple additional custom hooks, such as useDashboardCreation , and useServiceReview are encapsulating their own piece of state and logic to make the code readable and maintainable.

Component for adding a new review: The AddReview component is implemented to facilitate the creation of a new review. It is rendered by the servicePage to collect a review entry from a USER and uses the signAndExecuteTransaction function of the [useWalletKit] hook to execute the transaction.

Proof of experience generation: PoE is an NFT that is minted by SERVICE OWNER for customers after they dine at the restaurant; customers can then burn the PoE to write a high-rated review later. Minting an NFT is facilitated by the ownedServicePage component. This component is using the useServicePoEGeneration custom hook.

Delete a review: The moderator can delete a review that contains inappropriate content. moderatorRemovePage component is used to delete a review.

Reviews Rating repository .

Frontend

The frontend module is written in React, and is structured to provide a responsive user experience for interacting with a review rating platform. The page component supports user log in as a SERVICE OWNER , a MODERATOR , or a REVIEWER . A REVIEW READER role is not implemented for this example, but a REVIEWER can also read reviews and cast votes.

The frontend is a NextJS project, that follows the NextJS App Router project structure . The main code of the frontend is located in the app/src/ directory.

The main sub-directories are:

The Wallet Kit comes with a pre-built React.js component called ConnectButton that displays a button to connect and disconnect a wallet. The component handles connecting and disconnecting wallet logic.

Place the ConnectButton in the navigation bar for users to connect their wallets:

All the type definitions are in src/app/types/ .

Review and Service represent the review and service objects.

In the frontend, you might need to execute a transaction block in multiple places, hence it's better to extract the transaction execution logic and reuse it everywhere. Let's examine the execute transaction hook.

A Transaction is the input, sign it with the current connected wallet account, execute the transaction block, return the execution result, and finally display a basic toast message to indicate whether the transaction is successful or not.

Use the useWalletKit() hook from the Wallet Kit to retrieve the Sui client instance configured in WalletKitProvider . The signTransaction() function is another hook that helps to sign the transaction block using the currently connected wallet. It displays the UI for users to review and sign their transactions with their selected wallet. To execute a transaction block, the executeSignedTransaction() on the Sui client instance of the Sui TypeScript SDK. Use react-hot-toast as another dependency to toast transaction status to users.

Custom hooks: To keep the code as structured as possible, multiple custom hooks are utilized to manage the list of reviews associated with a service. The useGetReviews custom hook encapsulates the service, exposing all the required information (with fields such as nameOfService , listOfReviews , listOfStars) to display the reviews in a table. Multiple additional custom hooks, such as useDashboardCreation , and useServiceReview are encapsulating their own piece of state and logic to make the code readable and maintainable.

Component for adding a new review: The AddReview component is implemented to facilitate the creation of a new review. It is rendered by the servicePage to collect a review entry from a USER and uses the signAndExecuteTransaction function of the [useWalletKit] hook to execute the transaction.

Proof of experience generation: PoE is an NFT that is minted by SERVICE OWNER for customers after they dine at the restaurant; customers can then burn the PoE to write a high-rated review later. Minting an NFT is facilitated by the ownedServicePage component. This component is using the useServicePoEGeneration custom hook.

Delete a review: The moderator can delete a review that contains inappropriate content. moderatorRemovePage component is used to delete a review.

Reviews Rating repository .

Related links

Reviews Rating repository .