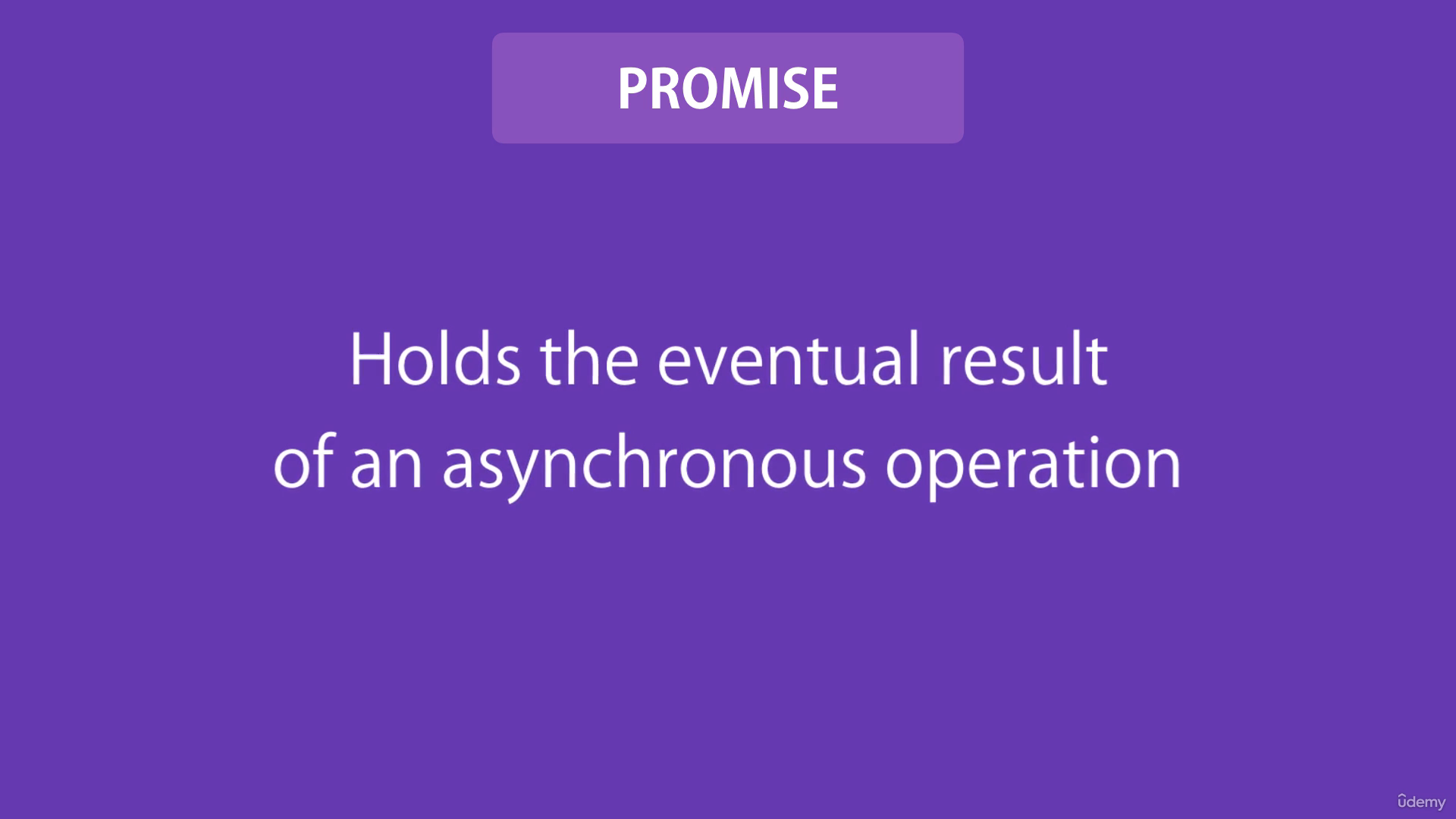
Promises

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

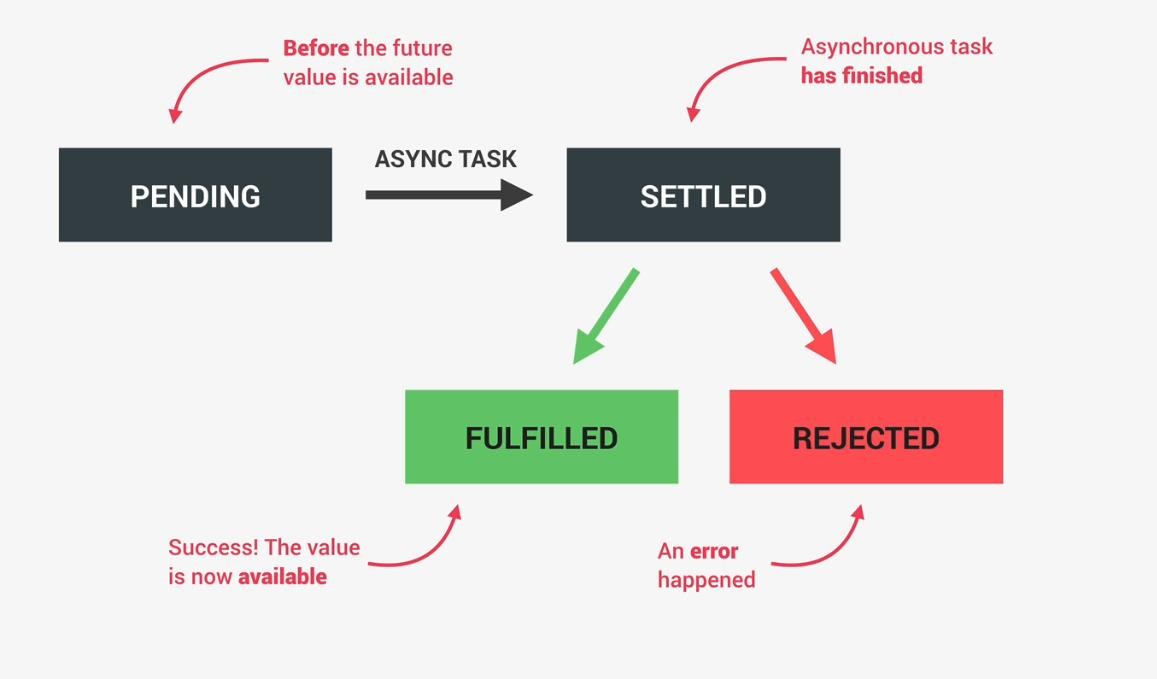
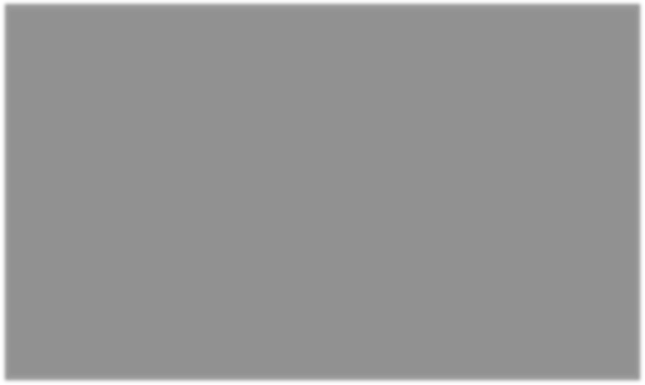


A Promise is in one of these states:

pending: initial state, neither fulfilled nor rejected.

fulfilled: meaning that the operation was completed successfully.

rejected: meaning that the operation failed.



const promise1 = () =>

{

  return new Promise((resolve, reject) =>

  {

    setTimeout(resolve, 500, "Promise1 is resolved");

  });

};

const promise2 = () =>

{

  return new Promise((resolve, reject) =>

  {

    setTimeout(resolve, 600, "Promise2 is resolved");

  });

};

const promise3 = () =>

{

  return new Promise((resolve, reject) =>

  {

    setTimeout(resolve, 700, "Promise3 is resolved");

  });

};

const fetchResults = async () =>

{

  try

  {

    const beforeTime = new Date();

    const p1 = await promise1();

    const p2 = await promise2();

    const p3 = await promise3();

    const afterTime = new Date();

    console.log(p1);

    console.log(p2);

    console.log(p3);

    console.log(`Time taken is: ${afterTime - beforeTime}`)

  }

  catch (error)

  {

    console.log(error.message);

  }

}

fetchResults();

Output:

//Promise1 is resolved

//Promise2 is resolved

//Promise3 is resolved

//Time taken is: 1826

all

It short circuits when an input value is rejected.

const fetchResults = async () =>

{

  try

  {

    const beforeTime = new Date();

    const results = await Promise.all([promise1(), promise2(), promise3()]);

    const afterTime = new Date();

    results.forEach((result) => console.log(result));

    console.log(`Time taken is: ${afterTime - beforeTime}`)

  }

  catch (error)

  {

    console.log(error.message);

  }

}

fetchResults();

Output:

//Promise1 is resolved

//Promise2 is resolved

//Promise3 is resolved

//Time taken is: 704

const promise1 = () =>

{

  return new Promise((resolve, reject) =>

  {

    setTimeout(resolve, 500, "Promise1 is resolved");

  });

};

const promise2 = () =>

{

  return new Promise((resolve, reject) =>

  {

    setTimeout(reject, 600, new Error("Promise2 is rejected"));

  });

};

const promise3 = () =>

{

  return new Promise((resolve, reject) =>

  {

    setTimeout(resolve, 700, "Promise3 is resolved");

  });

};

const fetchResults = async () =>

{

  const beforeTime = new Date();

  try

  {

    const results = await Promise.all([promise1(), promise2(), promise3()]);

    console.log(results, "results");

    results.forEach((result) => console.log(result));

  }

  catch (error)

  {

    console.log(error.message);

  }

  finally

  {

    const afterTime = new Date();

    console.log(`Time taken is: ${afterTime - beforeTime}`)

  }

}

fetchResults();

Output:

//Promise2 is rejected

//Time taken is: 609

race

It short circuits when an input value is settled(rejected or resolved).

const fetchResults = async () =>

{

  const beforeTime = new Date();

  try

  {

    const result = await Promise.race([promise1(), promise2(), promise3()]);

    console.log(result);

  }

  catch (error)

  {

    console.log(error.message);

  }

  finally

  {

    const afterTime = new Date();

    console.log(`Time taken is: ${afterTime - beforeTime}`)

  }

}

fetchResults();

Output:

//Promise1 is resolved

//Time taken is: 519

allSettled

It does not short circuit.

const fetchResults = async () =>

{

  const beforeTime = new Date();

  try

  {

    const result = await Promise.allSettled([promise1(), promise2(), promise3()]);

    console.log(result);

  }

  catch (error)

  {

    console.log(error.message);

  }

  finally

  {

    const afterTime = new Date();

    console.log(`Time taken is: ${afterTime - beforeTime}`)

  }

}

fetchResults();

/\*

[

  { status: 'fulfilled', value: 'Promise1 is resolved' },

  { status: 'rejected', reason: 'Promise2 is rejected' },

  { status: 'fulfilled', value: 'Promise3 is resolved' }

]

Time taken is: 723

\*/

any

It short circuits when an input value is fulfilled.

const fetchResults = async () =>

{

  const beforeTime = new Date();

  try

  {

    const result = await Promise.any([promise1(), promise2(), promise3()]);

    console.log(result);

  }

  catch (error)

  {

    console.log(error.message);

  }

  finally

  {

    const afterTime = new Date();

    console.log(`Time taken is: ${afterTime - beforeTime}`)

  }

}

fetchResults();

Output:

//Promise3 is resolved

//Time taken is: 519

Reject

The Promise.reject() static method returns a Promise object that is rejected with a given reason.

resolve

The Promise.resolve() static method "resolves" a given value to a [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise).

If the value is a promise, that promise is returned; if the value is a [thenable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#thenables), Promise.resolve() will call the then() method with two

callbacks it prepared; otherwise the returned promise will be fulfilled with the value. This function flattens nested layers of promise-like objects (e.g. a promise that fulfills to a promise that fulfills to something) into a single layer — a promise that fulfills to a non-thenable value.

then

The then() method of [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) instances takes up to two arguments: callback functions for the fulfilled and rejected cases of the Promise. It immediately returns an equivalent [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) object, allowing you to [chain](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#chaining) calls to other promise methods.

catch

The catch() method of [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) instances schedules a function to be called when the promise is rejected. It immediately returns an equivalent [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) object, allowing you to [chain](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#chaining) calls to other promise methods. It is a shortcut for [Promise.prototype.then(undefined,](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) [onRejected)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then).

finally

The finally() method of [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) instances schedules a function to be called when the promise is settled (either fulfilled or rejected). It immediately returns an equivalent [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) object, allowing you to [chain](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#chaining) calls to other promise methods. This lets you avoid duplicating code in both the promise's [then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) and [catch()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch) handlers.

const getUser = (id) => {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      const user = { id, user: `User ${id}` };

      const rd = Math.random();

      if (0.5 < rd) {

        resolve(user);

      } else {

        reject(new Error("Error while fetching user Info"));

      }

    }, 2000);

  });

};

const getRepos = (username) => {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      const repoInfo = {

        user: username,

        repos: ["Repo1", "Repo2", "Repo3", "Repo4"],

      };

      const rd = Math.random();

      if (0.5 < rd) {

        resolve(repoInfo);

      } else {

        reject(new Error("Error while fetching user repository Info"));

      }

    }, 2000);

  });

};

const getCommits = (repo) => {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      const commitsInfo = {

        repo,

        commits: ["Commit 1", "Commit 2", "Commit 3", "Commit 4"],

      };

      const rd = Math.random();

      if (0.5 < rd) {

        resolve(commitsInfo);

      } else {

        reject(new Error("Error while fetching repository  commits Info"));

      }

    }, 2000);

  });

};

console.log("Before");

getUser(10)

  .then((user) => {

    const username = user.username;

    return getRepos(username);

  })

  .then((repoInfo) => {

    const repo = repoInfo.repos[2];

    return getCommits(repo);

  })

  .then((commitsInfo) => {

    const commits = commitsInfo.commits;

    commits.forEach((commit) => console.log(commit));

  })

  .catch((error) => {

    console.log(error.message);

  });

Promise.resolve("I am working").then((data) => {

  console.log(data);

});

Promise.reject(new Error("I am a bug")).catch((error) => {

  console.log(error.message);

});

console.log("After");

//user > multiple repositories > repo > commits