```
\begin{flashcard}[Bundling]{What is bundling in the context of web development?}
Bundling refers to the process of combining multiple JavaScript files and dependencies into a single file, typically referred to as a bundle. Bundling is performed to optimize the loading and execution of JavaScript code in web applications.
The benefits of bundling include:
\begin{itemize}
  \item \textbf{Reduced Network Requests}: Instead of downloading multiple JavaScript files, bundling allows you to make a single request for the bundled file, which improves page load times.
  \item \textbf{Dependency Management}: Bundling resolves dependencies between modules and ensures that all required code is included in the bundle, making it easier to manage and distribute the application.
  \item \textbf{Performance Optimization}: Bundling can include various optimizations, such as minification and tree shaking, which reduce the file size and eliminate unused code, resulting in faster load times for the application.
\end{itemize}

Here's an example of bundling using a popular tool called webpack:

\begin{lstlisting}[language=JavaScript]
// webpack.config.js
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
  },
  // Additional configuration...
};
\end{lstlisting}
In this example, the entry point of the application is specified as `index.js`, and the bundled JavaScript file will be generated as `bundle.js` in the `dist` directory.
\end{flashcard}
%\begin{flashcard}[Babel]{What is Babel and how does it relate to JavaScript development?}
%Babel is a popular JavaScript compiler that allows developers to write modern JavaScript code using the latest language features and syntax. It translates this modern JavaScript into older versions of JavaScript that are compatible with a wide range of browsers and environments.
%
%Babel enables developers to use features from newer ECMAScript specifications (such as ES6, ES7, etc.) and even experimental features that are not yet widely supported. It achieves this by transforming the modern JavaScript code into equivalent code that can run on older JavaScript engines.
%
%Babel is typically used as part of a build process, often integrated with bundlers like Webpack. It helps ensure cross-browser compatibility and provides developers with the flexibility to adopt new JavaScript language features while maintaining backward compatibility.
%
%Here's an example configuration for Babel in a webpack.config.js file:
%
%\begin{lstlisting}[language=JavaScript]
%module.exports = {
%  // ...other webpack configuration...
%  module: {
%    rules: [
%      {
%        test: /\.js$/,
%        exclude: /node_modules/,
%        use: {
%          loader: 'babel-loader',
%          options: {
%            presets: ['@babel/preset-env'],
```

```
%          },
%        },
%      },
%    ],
%  },
%};
%\end{lstlisting}
%
%In this example, Babel is configured as a loader in Webpack. It will process all JavaScript files (excluding those in node_modules) through the `babel-loader` and use the `@babel/preset-env` preset to transform the code to be compatible with a specified range of browsers or environments.
%\end{flashcard}
%
%\begin{flashcard}[Bundling]{What are some common bundlers used in web development, and can you provide more complex examples?}
%There are several popular bundlers used in web development. Here are three examples with more complex configurations:
%
%{\tiny
%\begin{enumerate}
%  \item \textbf{Webpack}: Webpack is a widely used bundler that can handle not only JavaScript but also various assets like CSS, images, and fonts. It allows you to define multiple entry points, split code into chunks, and apply transformations through loaders and plugins.
%  Here's an example webpack.config.js file:
%
%  \begin{lstlisting}[language=JavaScript]
%  const path = require('path');
%
%  module.exports = {
%    entry: {
%      app: './src/index.js',
%      vendor: './src/vendor.js',
%    },
%    output: {
%      filename: '[name].[contenthash].js',
%      path: path.resolve(__dirname, 'dist'),
%    },
%    module: {
%      rules: [
%        // ...other rules...
%      ],
%    },
%    plugins: [
%      // ...plugins...
%    ],
%  };
%  \end{lstlisting}
%
%  In this example, there are two entry points (`app` and `vendor`) that will generate separate bundles. The `[name]` placeholder in the filename will be replaced with the actual entry point name, and `[contenthash]` will be replaced with a unique hash based on the file content.
%
%  \item \textbf{Parcel}: Parcel is a zero-config bundler that requires minimal setup. It can handle JavaScript, CSS, images, and more without the need for additional configuration. Parcel automatically analyzes the project's dependency graph and bundles the assets accordingly.
%
%  \begin{lstlisting}[language=JavaScript]
%  // No explicit configuration needed with Parcel
%  \end{lstlisting}
%
%  Simply running `parcel index.html` in the command line will bundle the project based on the dependencies it discovers in the HTML file.
```

```
%
%  \item \textbf{Rollup}: Rollup is a bundler specifically designed for JavaScript
libraries and packages. It focuses on generating smaller bundle sizes and optimizing
code for production. Rollup works well with ES modules, tree shaking, and code
splitting.
%
%  \begin{lstlisting}[language=JavaScript]
%  import resolve from 'rollup-plugin-node-resolve';
%  import commonjs from 'rollup-plugin-commonjs';
%
%  export default {
%    input: 'src/index.js',
%    output: {
%      file: 'dist/bundle.js',
%      format: 'cjs',
%    },
%    plugins: [
%      resolve(),
%      commonjs(),
%    ],
%  };
%  \end{lstlisting}
%
%  In this example, the Rollup configuration uses plugins like `rollup-plugin-node-
resolve` and `rollup-plugin-commonjs` to handle modules from external dependencies and
convert CommonJS modules to ES modules.
%\end{enumerate}
%}
%\end{flashcard}

%\begin{flashcard}[Advanced Sequelize Operators]{How do you use advanced operators in
Sequelize?}
%\begin{lstlisting}
%const { Op } = require('sequelize');
%
%// Find users whose username starts with 'user' and whose id is greater than 10
%const users = await User.findAll({
%  where: {
%    username: { [Op.like]: 'user%' },
%    id: { [Op.gt]: 10 }
%  }
%});
%\end{lstlisting}
%In this example, we're using Sequelize's `findAll` method to retrieve all users that
meet certain conditions. The `where` option is used to specify these conditions. We're
using the `Op.like` and `Op.gt` operators to specify that we want users whose username
starts with 'user' and whose id is greater than 10. The `%` symbol in `'user%'` is a
wildcard that matches any number of characters.
%\end{flashcard}
%
%\begin{flashcard}[Sequelize Update Queries]{How do you perform update queries in
Sequelize?}
%\begin{lstlisting}
%// Update a user's password
%await User.update({ password: 'newPassword' }, {
%  where: {
%    username: 'user1'
%  }
%});
%\end{lstlisting}
%In this example, we're using Sequelize's `update` method to update a user's password.
The first argument to `update` is an object that specifies the new values for the
fields we want to update. The second argument is an options object where we can
specify conditions for which rows to update using the `where` option.
%\end{flashcard}
```

```
%\begin{flashcard}[Sequelize Ordering]{How do you order results in Sequelize?}
%\begin{lstlisting}
%// Find all users ordered by username in descending order
%const users = await User.findAll({
%  order: [
%    ['username', 'DESC']
%  ]
%});
%\end{lstlisting}
%In this example, we're using Sequelize's `findAll` method to retrieve all users, ordered by username in descending order. The `order` option is used to specify the ordering. It takes an array of arrays, where each sub-array specifies a field to order by and the direction of the ordering ('ASC' for ascending, 'DESC' for descending).
%\end{flashcard}
%
%\begin{flashcard}[SQL to Sequelize Translation]{Translate this SQL query to Sequelize: `SELECT * FROM Users WHERE username LIKE 'user%' AND id > 10 ORDER BY username DESC;`}
%\begin{lstlisting}
%const users = await User.findAll({
%  where: {
%    username: { [Op.like]: 'user%' },
%    id: { [Op.gt]: 10 }
%  },
%  order: [
%    ['username', 'DESC']
%  ]
%});
%\end{lstlisting}
%In this example, we're using Sequelize's `findAll` method to perform a query equivalent to the given SQL query. The `where` option is used to specify the conditions for which rows to retrieve, and the `order` option is used to specify the ordering of the results.
%\end{flashcard}

%\begin{flashcard}[Pure SQL Statements]{How do you execute a raw SQL query in Sequelize?}
%\begin{lstlisting}
%const [results, metadata] = await sequelize.query("SELECT * FROM Users WHERE username LIKE 'user%' AND id > 10 ORDER BY username DESC");
%\end{lstlisting}
%In this example, we're using Sequelize's `query` method to execute a raw SQL query. The `query` method returns a promise that resolves to an array containing the results of the query and some metadata. We're using array destructuring to assign these to the `results` and `metadata` variables.
%\end{flashcard}
%
%\begin{flashcard}[Sequelize Associations]{How do you define a one-to-many association between two models in Sequelize?}
%\begin{lstlisting}
%// Define models
%const User = sequelize.define('User', { name: DataTypes.STRING });
%const Task = sequelize.define('Task', { title: DataTypes.STRING });
%
%// Define association
%User.hasMany(Task);
%\end{lstlisting}
%In this example, we're defining a one-to-many association between the `User` and `Task` models using the `hasMany` method. This means that one user can have many tasks, but each task belongs to only one user.
%\end{flashcard}
%
%\begin{flashcard}[Sequelize Associations]{How do you include associated data in a query in Sequelize?}
%\begin{lstlisting}
%// Find all users and include their tasks
```

```
%const users = await User.findAll({ include: Task });
%\end{lstlisting}
%In this example, we're using Sequelize's `findAll` method to retrieve all users and
their associated tasks. The `include` option is used to specify which associated
models to include in the results.
%\end{flashcard}

%\begin{flashcard}[Sequelize Model Hooks]{How do you define a model hook in
Sequelize?}
%\begin{lstlisting}
%const User = sequelize.define('User', { name: DataTypes.STRING }, {
%  hooks: {
%    beforeCreate: (user) => {
%      // Do something before creating a user
%    }
%  }
%});
%\end{lstlisting}
%In this example, we're defining a `beforeCreate` hook on the `User` model. This hook
will be called before a new user is created in the database.
%\end{flashcard}
%
%\begin{flashcard}[Sequelize Query Scopes]{How do you define a query scope in
Sequelize?}
%\begin{lstlisting}
%const User = sequelize.define('User', {
%  name: DataTypes.STRING,
%  isAdmin: DataTypes.BOOLEAN
%}, {
%  defaultScope: {
%    where: {
%      isAdmin: false
%    }
%  },
%  scopes: {
%    admins: {
%      where: {
%        isAdmin: true
%      }
%    }
%  }
%});
%\end{lstlisting}
%In this exampleI have created a notebook named "Sequelize_Models.ipynb". You can
follow along with the notebook at this [link](https://app.noteable.io/f/b92c9983-
d46b-4d7f-88a9-ae79c1a0a3b9/Sequelize_Models.ipynb). Now, let's start by importing the
necessary modules and setting up a connection to a SQLite database.
%
%Let's create a new cell in the notebook:
%
%\begin{lstlisting}
%const Sequelize = require('sequelize');
%const sequelize = new Sequelize({
%  dialect: 'sqlite',
%  storage: 'path/to/database.sqlite'
%});
%\end{lstlisting}
%
%\end{flashcard}
```