6 Среда разработки Connect

- Создание Connect-приложения
- Принципы работы программного обеспечения промежуточного уровня в Connect
- Важность порядка следования программных компонентов промежуточного уровня
- Монтирование программного обеспечения промежуточного уровня и серверов
- Создание настраиваемого программного обеспечения промежуточного уровня
- Использование программного обеспечения промежуточного уровня для обработки ошибок

Connect — это среда разработки приложений, в которой с помощью модульных компонентов, *называемых программным обеспечением промежуточного уровня* (middleware), создается многократно используемый код веб-приложений. Программный компонент промежуточного уровня в Connect представляет собой функцию, которая перехватывает объекты `request` (запрос) и `response` (ответ), предоставляемые HTTP-сервером, выполняет программную логику, а затем либо завершает ответ, либо передает управление другому программному компоненту промежуточного уровня. В Connect программные компоненты промежуточного уровня объединяются вместе с помощью *диспетчера* (dispatcher).

В Connect можно создавать собственное программное обеспечение промежуточного уровня, а также использовать обычные компоненты, реализующие в приложениях сбор данных запросов, обслуживание статических файлов, синтаксический разбор тел запросов, управление сеансами и т. п. Благодаря простоте расширения и надстройки среда Connect с точки зрения разработчиков представляет собой некий абстрактный уровень, позволяющий им создавать собственные высокоуровневые

среды веб-разработки. На рис. 6.1 показано, каким образом диспетчер строит Connect-приложение и организует программное обеспечение промежуточного уровня.

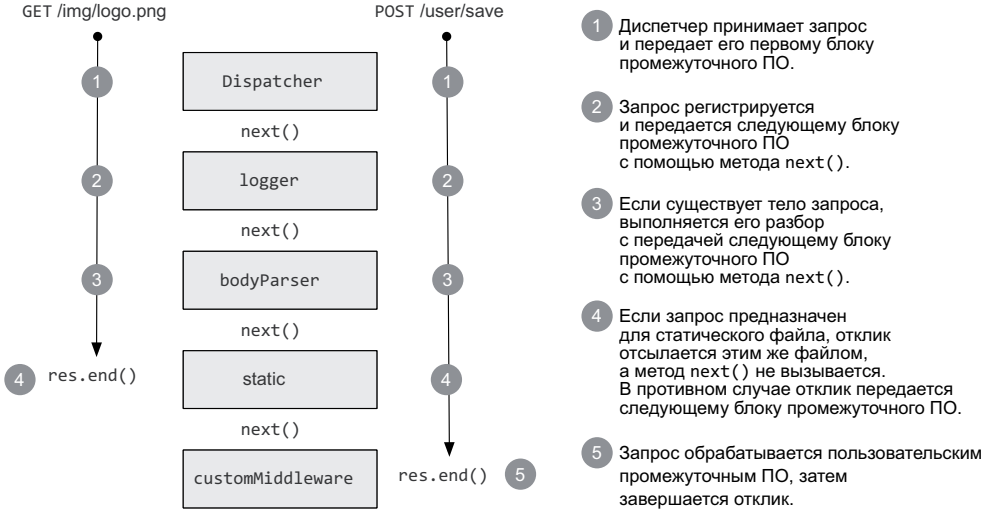

Рис. 6.1. Жизненный цикл двух HTTP-запросов, обрабатываемых Connect-сервером

CONNECT И EXPRESS

Концепции, рассматриваемые в этой главе, непосредственно применимы и к более высокоуровневой среде разработки Express, поскольку, по сути, она представляет собой среду Connect, расширенную и надстроенную дополнительными высокоуровневыми «примочками». Прочитав эту главу, вы получите полное представление о том, как работает программное обеспечение промежуточного уровня и как объединять компоненты друг с другом при создании приложения.

В главе 8 с помощью среды Express мы будем создавать симпатичные веб-приложения с более высокоуровневым, чем в Connect, API-интерфейсом. Фактически большая часть функциональности, поддерживаемая в настоящий момент в Connect, изначально появилась в Express, однако после абстрагирования низкоуровневые «строительные блоки» остались в Connect, а выразительная высокоуровневая «начинка» была зарезервирована за Express.

Для начала давайте создадим простое Connect-приложение.

6.1. Создание Connect-приложения

Поскольку модуль `connect` разработан сторонним производителем, по умолчанию он при установке Node не устанавливается. Воспользуйтесь следующей командой, чтобы загрузить и установить среду Connect из mpm-хранилища:

```
$ npm install connect
```

Завершив установку, приступим к созданию простого Connect-приложения. Для создания такого приложения понадобится модуль `connect`, представляющий собой функцию, которая при вызове возвращает Connect-приложение.

В главе 4 показано, как метод `http.createServer()` принимает функцию обратного вызова, которая обрабатывает входящие запросы. Фактически «приложение», создаваемое в Connect, представляет собой JavaScript-функцию, которая принимает HTTP-запрос, а затем выполняет его диспетчеризацию выбранному программному компоненту промежуточного уровня.

В листинге 6.1 приведен код простого Connect-приложения. В этом приложении нет программных компонентов промежуточного уровня, поэтому диспетчер в ответ на любой принятый HTTP-запрос отвечает сообщением 404 Not Found.

Листинг 6.1. Простейшее Connect-приложение

```
var connect = require('connect');
var app = connect();
app.listen(3000);
```

После запуска сервера и передачи ему HTTP-запроса (с помощью команды `curl` или в окне веб-браузера) появится текст `"Cannot GET /"`. Это означает, что приложение не сконфигурировано для обработки запрошенного URL-адреса. По первому примеру видно, как работает диспетчер в Connect: поочередно вызываются связанные между собой программные компоненты промежуточного уровня. Этот процесс длится до тех пор, пока один из компонентов не ответит на запрос. Если ни один из компонентов не отвечает, завершив перебор списка компонентов, приложение выводит сообщение об ошибке со статусом 404.

Теперь, создав простейшее Connect-приложение и поняв, как работает диспетчер, давайте попробуем построить приложение, делающее хоть *что-нибудь*. Чтобы создать подобное приложение, нужно определить и добавить программное обеспечение промежуточного уровня.

6.2. Принципы работы программного обеспечения промежуточного уровня в Connect

Программный компонент промежуточного уровня в Connect представляет собой JavaScript-функцию, которая по умолчанию принимает три аргумента. Первый аргумент — это запрос (объект `request`), второй аргумент — ответ (объект `response`), а третий аргумент, который обычно называется `next`, задает функцию обратного

вызова, показывающую, что компонент завершил работу и может выполняться следующий программный компонент промежуточного уровня.

Концепция программного обеспечения промежуточного уровня изначально была реализована в среде Ruby на платформе Rack. Эта среда разработки предоставляет очень простой модульный интерфейс, но из-за потоковой природы Node API-интерфейсы в Node и Rack не идентичны. Программные компоненты промежуточного уровня — это прекрасное решение, поскольку они невелики, самодостаточны и могут совместно использоваться приложениями.

В этом разделе мы познакомимся с программным обеспечением промежуточного уровня, с помощью которого к созданному в предыдущем разделе простейшему Connect-приложению добавим два простых программных компонента промежуточного уровня, которые наделят приложение новыми возможностями:

- программный компонент промежуточного уровня `logger` предназначен для записи на консоль данных запросов;
- программный компонент промежуточного уровня `hello` должен отвечать на запрос сообщением «hello world».

Начнем с создания простого программного компонента промежуточного уровня, который собирает данные запросов, поступающих на сервер.

6.2.1. Компонент logger

Предположим, что нужно создать файл журнала, в котором для каждого запроса, поступающего на сервер, записывать метод и URL-адрес запроса. Для решения этой задачи создадим функцию, называемую `logger`, которая в качестве параметров принимает объекты запроса и ответа, а также функцию обратного вызова `next`.

Функция `next` может быть вызвана программным компонентом промежуточного уровня, чтобы сообщить диспетчеру о том, что он завершил свою работу и управление можно передать следующему компоненту. Поскольку вместо возвращаемого значения используется функция обратного вызова, внутри программного компонента промежуточного уровня может быть реализована асинхронная логика, тем не менее диспетчер переходит к следующему компоненту только после завершения предыдущего. С помощью функции `next()` реализуется превосходный механизм обслуживания последовательности операций, выполняемых программными компонентами промежуточного уровня.

В компоненте `logger` вызывается метод `console.log()`, в качестве аргументов которого указываются метод и URL-адрес запроса, затем выводится текст `"GET /user/1"` и вызывается функция `next()`, передающая управление следующему компоненту:

```
function logger(req, res, next) {
   console.log('%s %s', req.method, req.url);
   next();
}
```

Итак, мы только что завершили разработку корректно функционирующего программного компонента промежуточного уровня, который для каждого полученного HTTP-запроса выводит на консоль метод и URL-адрес запроса, а затем вызывает функцию `next()`, возвращающую управление диспетчеру. Чтобы воспользоваться

этим компонентом в приложении, вызовите метод `.use()`, передав его функции промежуточного уровня:

```
var connect = require('connect');
var app = connect();
app.use(logger);
app.listen(3000);
```

После отправки нескольких запросов серверу (как и раньше, с помощью команды `curl` или веб-браузера) на консоли вы увидите следующее:

```
GET /
GET /favicon.ico
GET /users
GET /user/1
```

Компонент сбора данных запросов — это лишь один программный компонент промежуточного уровня. Теперь нужно как-то отправить ответ клиенту. А для этого предназначен другой программный компонент промежуточного уровня.

6.2.2. Компонент, отвечающий сообщением «hello world»

Второй программный компонент промежуточного уровня нашего приложения будет отвечать на HTTP-запрос. Это тот же код, который используется в серверной функции обратного вызова «hello world» на домашней странице в Node:

```
function hello(req, res) {
   res.setHeader('Content-Type', 'text/plain');
   res.end('hello world');
}
```

Чтобы воспользоваться вторым компонентом нашего приложения, вызовите метод `.use()`. Количество вызовов этого метода, служащего для добавления компонентов промежуточного уровня, неограниченно.

В листинге 6.2 представлен код всего приложения. Добавление в код компонента `hello` приводит к тому, что сначала сервер вызывает компонент `logger`, который печатает текст на консоли, а затем отвечает на каждый HTTP-запрос текстом «hello world».

Листинг 6.2. Использование нескольких программных компонентов промежуточного уровня в Connect

```
var connect = require('connect');

// Вывод на консоль HTTP-метода и URL-адреса запроса и вызов метода next()
function logger(req, res, next) {
   console.log('%s %s', req.method, req.url);
   next();
}

// Завершение ответа на HTTP-запрос словами "hello world"
function hello(req, res) {
   res.setHeader('Content-Type', 'text/plain');
   res.end('hello world');
}
```

продолжение ⇙

Листинг 6.2 (*продолжение*)

```
connect()
.use(logger)
.use(hello)
.listen(3000);
```

В данном случае у компонента `hello` нет аргумента обратного вызова `next`. Это связано с тем, что данный компонент завершает HTTP-ответ и не нуждается в том, чтобы передавать управление обратно диспетчеру. В подобных случаях обратный вызов `next` не нужен. Это удобно, поскольку он соответствует сигнатуре функции обратного вызова `http.createServer`. Сказанное означает, что если вы уже написали код HTTP-сервера с применением модуля `http`, в вашем распоряжении оказывается полностью работоспособный программный компонент промежуточного уровня, который можно многократно использовать в Connect-приложении.

Функция `use()` возвращает экземпляр Connect-приложения, поддерживая формирование цепочки вызовов методов, как отписывалось ранее. Обратите внимание, что цепочка вызовов `.use()` не нужна, как показано в следующем примере кода:

```
var app = connect();
app.use(logger);
app.use(hello);
app.listen(3000);
```

Теперь, получив простое работоспособное приложение «hello world», давайте выясним, почему столь важен порядок вызовов методов `.use()` и каким образом этот порядок меняет поведение приложения.

6.3. Почему важен порядок вызова программных компонентов промежуточного уровня

Чтобы предоставить разработчикам приложений и сред разработки максимальную степень гибкости, в Connect не предусмотрен заранее заданный порядок вызова программных компонентов промежуточного уровня — вы сами может указать, в какой очередности они должны выполняться. Это очень простая концепция, но она не всегда принимается во внимание.

В этом разделе рассказывается, как порядок выполнения программного обеспечения промежуточного уровня в вашем приложении может радикально изменить его поведение. В частности, будут рассмотрены следующие вопросы:

- остановка выполнения оставшихся программных компонентов путем исключения функции `next()`;
- использование мощного средства упорядочения программного обеспечения промежуточного уровня;
- использование программного обеспечения промежуточного уровня для идентификации.

Сначала посмотрим, как Connect работает с программным компонентом промежуточного уровня, который явно вызывает функцию `next()`.

6.3.1. Когда функция next() не вызывается

В предыдущем примере приложения «hello world» сначала выполняется компонент `logger`, а за ним — компонент `hello`. В этом примере Connect записывает данные в поток `stdout`, а затем отвечает на HTTP-запрос. А теперь посмотрим, что произойдет, если изменить этот порядок (листинг 6.3).

Листинг 6.3. Неправильно: компонент hello выполняется до компонента logger

```
var connect = require('connect');

// Всегда вызывается функция next() для перехода к следующему
// компоненту промежуточного уровня
function logger(req, res, next) {
   console.log('%s %s', req.method, req.url);
   next();
}

// Функция next() не вызывается, поскольку компонент отвечает на запрос
function hello(req, res) {
   res.setHeader('Content-Type', 'text/plain');
   res.end('hello world');
}

var app = connect()
   // Компонент logger никогда не вызывается, поскольку
   // в hello нет вызова функции next()
   .use(hello)
   .use(logger)
   .listen(3000);
```

В этом примере компонент `hello` вызывается первым и отвечает на HTTP-запрос, как и ожидалось. Но поскольку компонент `hello` не вызывает функцию `next()`, он не возвращает управление диспетчеру, чтобы выполнить следующий программный компонент промежуточного уровня. В результате компонент `logger` вообще не выполняется. Если компонент промежуточного уровня не вызывает функцию `next()`, следующие за ним в цепочке компоненты промежуточного уровня не выполняются.

В данном случае помещение компонента `hello` перед компонентом `logger` вряд ли поможет, но при правильном использовании соответствующих методик от выбранного порядка размещения компонентов можно извлечь реальную пользу.

6.3.2. Выполнение аутентификации путем расположения компонентов в нужном порядке

Расположив компоненты промежуточного уровня в нужной последовательности, можно выполнять некоторые операции, например аутентификацию. Эта операция требуется почти в любом приложении. Пользователям приложения обычно нужно проходить процедуру входа (аутентификацию), а тем, кто не прошел аутентификацию, доступ к контенту блокируется. Реализовать аутентификацию нам поможет нужный порядок следования компонентов промежуточного уровня.

Предположим, что создан компонент `restrictFileAccess`, разрешающий доступ к файлу только аутентифицированным пользователям, которым разрешается получить доступ к следующему компоненту промежуточного уровня. Если же пользователь не проходит аутентификацию, функция `next()` не вызывается. В листинге 6.4 код компонента `restrictFileAccess` должен располагаться за кодом компонента `logger`, но перед кодом компонента `serveStaticFiles`.

Листинг 6.4. Ограничение доступа к файлу за счет правильного выбора порядка следования компонентов

```
var connect = require('connect');
connect()
   .use(logger)
   // Функция next() вызывается только в том случае, если пользователь
   // проходит аутентификацию
   .use(restrictFileAccess)
   .use(serveStaticFiles)
   .use(hello);
```

Теперь, когда вы получили представление о программном обеспечении промежуточного уровня и о том, какой это важный инструмент для управления прикладной логикой, рассмотрим другие средства среды Connect, помогающие использовать программное обеспечение промежуточного уровня.

6.4. Монтирование программного обеспечения промежуточного уровня и серверов

При создании приложений в Connect вы столкнетесь с концепцией *монтирования* (mounting). Эта концепция описывает простой, но мощный организационный инструмент, позволяющий определять префикс пути для программного обеспечения промежуточного уровня или приложений в целом. Путем монтирования можно создавать компоненты промежуточного уровня, как будто они находятся в корневой папке (значение `/base` свойства `req.url`), а затем использовать вместе с произвольным префиксом пути, не изменяя код.

Например, если компонент промежуточного уровня или сервер смонтирован в папке /blog, указанная в коде ссылка `req.url` на папку /article/1 будет доступна для клиентского запроса в виде /blog/article/1. С помощью подобной методики можно обращаться к серверу блога из разных мест, не меняя код вызова. Например, если вы примете решение хранить посты в папке /articles (/articles/article/1) вместо папки /blog, понадобится лишь изменить префикс пути монтирования.

А теперь рассмотрим другой пример использования монтирования, характерный для приложений, обладающих собственными областями администрирования, такими как модерирование комментариев и принятие новых пользователей. В рассматриваемом примере область администрирования находится в папке /admin приложения. Нужно сделать так, чтобы папка /admin была доступна только пользователям, обладающим соответствующими полномочиями, а другие папки сайта были доступны всем пользователям.

Помимо перезаписи запросов из папки /base (значение `/base` свойства `req.url`), при монтировании можно вызывать программное обеспечение промежуточного уровня или приложение только в том случае, если запрос выполняется с определенным префиксом пути (точка монтирования). В листинге 6.5 второй и третий вызовы функции `use()` включают строку `'/admin'` в качестве первого аргумента, за которым следует компонент промежуточного уровня. Это означает, что следующие компоненты будут использованы только в том случае, если запрос выполняется с указанием префикса `/admin`. Так выглядит синтаксис, реализующий монтирование компонента промежуточного уровня или сервера в Connect.

Листинг 6.5. Синтаксис монтирования компонента промежуточного уровня или сервера

```
var connect = require('connect');

connect()
   .use(logger)
   // Если первым аргументом метода .use() является строка, Connect
   // будет вызывать программное обеспечение промежуточного уровня
   // только в случае соответствия префикса в URL-адресе
   .use('/admin', restrict)
   .use('/admin', admin)
   .use(hello)
   .listen(3000);
```

Используя технологии монтирования программного обеспечения промежуточного уровня и серверов, дополним приложение «hello world» областью администрирования. Мы будем использовать монтирование и добавим два новых программных компонента промежуточного уровня:

- компонент `restrict` гарантирует, что только зарегистрированный пользователь получает доступ к странице;
- компонент `admin` предоставляет пользователю доступ к области администрирования.

И начнем мы с компонента, ограничивающего доступ к ресурсам пользователей, не имеющих соответствующих полномочий.

6.4.1. Программный компонент промежуточного уровня для аутентификации

Первый компонент промежуточного уровня должен выполнять аутентификацию. Речь идет об обобщенном компоненте аутентификации, который не привязан каким-либо образом к значению `/admin` свойства `req.url`. Однако при его монтировании в приложение компонент аутентификации будет вызываться только в том случае, если URL-адрес запроса начинается префиксом `/admin`. В этом случае выполняется аутентификация только тех пользователей, которые попытаются получить доступ с URL-адресом `/admin`; остальные пользователи входят в систему обычным образом.

В листинге 6.6 представлен «сырой» код базовой аутентификации. В этом простейшем механизме аутентификации используется поле заголовка HTTP-авторизации

вместе с полномочиями, закодированными по алгоритму Base64. Дополнительные сведения о базовой аутентификации можно найти в статье Википедии по адресу http://wikipedia.org/wiki/Basic_access_authentication. Когда полномочия декодируются компонентом промежуточного уровня, имя пользователя и пароль проверяются на корректность. Если имя пользователя и пароль корректны, компонент вызывает функцию `next()`. Этот означает, что запрос выполнен и обработка данных может продолжаться. В противном случае генерируется ошибка.

Листинг 6.6. Компонент промежуточного уровня, выполняющий базовую HTTP-аутентификацию

```
function restrict(req, res, next) {
   var authorization = req.headers.authorization;
   if (!authorization) return next(new Error('Unauthorized'));

   var parts = authorization.split(' ')
   var scheme = parts[0]
   var auth = new Buffer(parts[1], 'base64').toString().split(':')
   var user = auth[0]
   var pass = auth[1];

   // Функция, проверяющая полномочия на доступ к базе данных
   authenticateWithDatabase(user, pass, function (err) {
      // Информируем диспетчера о происшедшей ошибке
      if (err) return next(err);
      // При наличии корректных полномочий вызывается функция next()
      // без аргументов
      next();
   });
}
```

И снова обратите внимание на то, что этот компонент промежуточного уровня не проверяет значение свойства `req.url`, пытаясь убедиться в том, что запрашивается именно папка /admin. Подобная проверка выполняется средой Connect. Благодаря этому обстоятельству можно создавать обобщенное программное обеспечение промежуточного уровня. Компонент `restrict` может применяться для аутентификации другой части сайта или другого приложения.

ВЫЗОВ ФУНКЦИИ NEXT С ОБЪЕКТОМ ОШИБКИ В КАЧЕСТВЕ АРГУМЕНТА

Обратите внимание, как в предыдущем примере функция next вызывается с объектом ошибки, передаваемым в качестве аргумента. В результате выполнения этой операции Connect получает уведомление о том, что произошла ошибка приложения. А это означает, что в ответ на оставшуюся часть HTTP-запроса может вызываться только программное обеспечение промежуточного уровня, предназначенное для обработки ошибок. Эта разновидность программного обеспечения промежуточного уровня рассматривается в этой главе чуть позже. Ну а на данный момент вам достаточно знать, что Connect получает сообщение о том, что произошла ошибка и программное обеспечение промежуточного уровня завершает работу.