JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Julian Kaindl

Submitted at
Institute for System Software

Supervisor and First Examiner
DI **Sebastian Kloibhofer**

Second Examiner
Dr. **Christian Wirth**

January 10, 2023

IMPLEMENTATION OF THE FETCH API FOR GRAAL.JS

Bachelor Thesis

to obtain the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Sworn Declaration

I hereby declare under oath that the submitted bachelors thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited.

The submitted document here present is identical to the electronically submitted text document.

Linz, January 10, 2023 

JOHANNES KEPLER
UNIVERSITY LINZ

Abstract

The Fetch API for JavaScript provides a standardized way of requesting resources asynchronously across the network. It defines network behavior and interfaces for accessing and manipulating the request and response data. Most notably, it introduces the global `fetch()` function and the global types `Request`, `Response`, and `Headers`. To unify the behavior across different implementations, the Fetch API standard was developed and is maintained by the Web Hypertext Application Technology Working Group (WHATWG) as a living document. GraalVM JavaScript or Graal.js is a high-performance JavaScript implementation on the GraalVM platform. It is compliant to the latest ECMAScript specification and is implemented in Java, utilizing the language implementation framework Truffle. This work introduces a first native implementation of the Fetch API, as specified by the WHATWG, for Graal.js. Networking functionality is implemented using the networking capabilities of our host language, Java. Our implementation is integrated into the Graal.js JavaScript interpreter by utilizing the Truffle framework. The goal of this implementation is to cover as much of the Fetch API standard as possible, choose sensible alternatives when we can not conform to the specification and document these deviations.

JOHANNES KEPLER
UNIVERSITY LINZ

Kurzfassung

Die Fetch API bietet JavaScript Applikationen eine standartisierte Art, asynchron über das Netzwerk auf Ressourcen zuzugreifen. Sie definiert Netzwerkabläufe und Schnittstellen für Abfrage- und Antwortdaten. Die wichtigsten neuen globalen Typen sind `Request`, `Response` und `Headers`, sowie die globale `fetch()` Funktion. Um die API für verschiedene Implementierungen zu vereinheitlichen, wurde der Fetch API Standard entworfen. Er wird von der Web Hypertext Application Technology Working Group (WHATWG) als lebendes Dokument geführt. GraalVM JavaScript, oder Graal.js, ist eine performante JavaScript-Implementierung auf der GraalVM Platform. Es ist konform mit der aktuellen ECMAScript-Spezifikation und ist mit Java und dem Sprachimplementierungsframework Truffle umgesetzt. Im Rahmen dieser Arbeit wird eine erste native Implementierung der Fetch API, wie von WHATWG spezifiziert, für Graal.js vorgestellt. Netzwerkbezogene Funktionalität ist mithilfe der von Java zur Verfügung gestellten Möglichkeiten implementiert. Das Truffle-Framework wurde genutzt, um die Implementierung korrekt in den Graal.js Interpreter zu integrieren. Das Ziel dieser Implementierung ist es, den Fetch-API-Standard möglichst vollständig umzusetzen. Falls eine Umsetzung von speziellen Teilen des Standards nicht möglich ist, sollen sinnvolle Alternativen genutzt und die Abweichungen dokumentiert werden.

JOHANNES KEPLER
UNIVERSITY LINZ

Contents

1 Introduction

This thesis aims to implement the Fetch API in a JavaScript engine that lacked support for it so far, the GraalVM JavaScript implementation Graal.js. By adopting the API in Graal.js, applications have access to features and functionality as specified by the Fetch API standard, most notably the network I/O it provides. In contrast to the current solution, which would be to import the Fetch API via a third-party extension module, this native approach aims for better performance and potentially better security.

JavaScript is one of the core technologies of the world wide web. As of 2022, 98% of websites use it on the client side[1]. Outside of the browser, it has also become a well-established general purpose programming language. It is specified as *ECMAScript* by an ever evolving standard [1], that aims to ensure interoperability between browsers and other JavaScript platforms. Even though the language is mature and complete, ECMAScript does not standardize any functionality around I/O. Implementations of the language typically provide that functionality in a proprietary fashion. Different web browsers have a different internal I/O implementation. This lack of unification, however, creates problems for websites that rely on I/O functionality, for example to request data from a server over the network. If I/O implementations of different browsers were incompatible, the client side JavaScript would have to deal with multiple ways of accessing these proprietary APIs. To mitigate these problems, a number of additional standards, such as the Fetch API, are used to specify behavior around I/O. They are implemented by web browsers and server-side JavaScript platforms, like Node.js[2] and Graal.js.

The Fetch API, standardized by the WHATWG[3], offers functionality related to network I/O. It provides an interface for *fetching*, with the goal of standardizing this process across different JavaScript platforms. Fetching is the process of accessing a resource that lies on the network. Requesting data from a server endpoint or when simply loading a website. To facilitate this process, the Fetch API provides `Request` and `Response` types. They are the input and output of the central `fetch()` function, which actually performs the fetching process and acquires the resource over the network. A typical usage of the API consists of (1) defining the desired resource using a URL and other request options, (2) executing the `fetch()` function, and (3) reading the response data from the return value.

[1] `https://w3techs.com/technologies/details/cp-javascript/`
[2] `https://nodejs.org/en/`
[3] `https://whatwg.org/`

As focus of this work, we present a first native implementation of the Fetch API in the *Graal.js* JavaScript engine. Graal.js[4] is a fully ECMAScript-compliant implementation of JavaScript built on the GraalVM and developed by Oracle Labs. It aims to execute JavaScript with the best possible performance and allows language interoperability with other GraalVM languages. Graal.js is implemented in Java and utilizes the language implementation framework *Truffle*[5]. Truffle is a library for creating language interpreters for self-modifying Abstract Syntax Trees [11]. Furthermore, the GraalVM Compiler can greatly optimize the performance of languages implemented with the Truffle framework [10].

While our implementation of the Fetch API in Graal.js follows the WHATWG specification, it is not feasible to completely support the Fetch API standard. This is because the Fetch API specification assumes a client-side context, concretely a web browser. As Graal.js is a JavaScript runtime for executing JavaScript outside of the browser, this implementation has a server-side context, thus a number of aspects, that the Fetch specification includes, do not apply.

Our implementation utilizes the Truffle programming model to correctly integrate the Fetch API interface into the Graal.js interpreter. To implement the Fetch network functionality, we use the networking capabilities of our host language, Java. Specifically, classes from the standard library package `java.net.*`. We use node-fetch[6] as a reference implementation, which is an already existing implementation of the Fetch API standard. It is a third-party module for applications using the Node.js JavaScript runtime and therefore also operates on a server-side platform. It is well established and includes an extensive list of unit tests for standardized Fetch API behavior, which we adopted to validate our implementation.

[4] `https://github.com/oracle/graaljs`
[5] `https://www.graalvm.org/22.0/graalvm-as-a-platform/language-implementation-framework/`
[6] `https://github.com/node-fetch/node-fetch`

2 Background

In this chapter, the technologies and concepts this thesis is based upon, are introduced. First, Section 2.1 introduces ECMAScript and language-specific concepts relevant for this implementation. In Section 2.2, the GraalVM JavaScript engine is introduced. The core concepts of the Fetch API standard are discussed in Section 2.3. The Java networking capabilities and other networking related background information is presented in Section 2.4. Finally, Section 2.5 introduces node-fetch, an already existing implementation of this standard for a similar use case.

2.1 ECMAScript

JavaScript is a general-purpose programming language and a central technology in the world-wide-web. It conforms to the *ECMAScript* (ECMA 262)[1] standard, which is a language specification for JavaScript standardized by *Ecma International*[2]. The goal of the specification is to ensure compatibility between different browsers. Only language syntax and semantics of the core API are specified in the ECMA Language Specification. Implementations of JavaScript that are valid under the ECMA standard add their own functionality, like I/O handling. In other words, the term ECMAScript describes a standardized version of the JavaScript language that adheres to the ECMA 262 standard. For simplicity, when JavaScript is mentioned in the next chapters, we always refer to an ECMAScript compliant implementation.

The following sections introduce language concepts that are important for the implementation of the Fetch API.

2.1.1 The Global Object

The ECMAScript standard defines the *unique global object* as an object that is created before any execution context is entered. Upon entering the global execution context, the properties of the global object are accessible. While the ECMAScript standard specifies a number of different value and function properties, implementations may have additional properties. [2]

[1] `https://www.ecma-international.org/publications-and-standards/standards/ecma-262/`
[2] `https://www.ecma-international.org/`

2.1.2 Objects and Prototypes

Version 6 of ECMAScript (*ES6*), released in 2015, introduced significant new syntax for writing complex applications using object oriented programming such as the keywords `class` and `extends`. Although ES6 syntactically includes class definitions, the objects are not created using a class-based approach like in Java or C++. Instead, classes are just object templates and constructor functions create the objects. To create an object using a constructor function, the `new` keyword is needed, for instance `new Date()` [3]. All objects have an internal property that links to another object, called its `prototype`. The prototype then is used for *prototype-based inheritance* via the *prototype chain*. The object that the prototype links to has a `prototype` itself, and so on - this is called the prototype chain. It terminates when an object has `null` as its prototype value. When accessing a property on an object, the property is not only searched in the object itself, but also by following the prototype chain until either a matching property name is found or the end of the chain is reached. Furthermore, it is possible to mutate prototypes and properties of prototypes at runtime.

The example in Figure 2.1 illustrates the concept of prototype-based inheritance. `CF` represents a constructor function. The objects $\texttt{cf}_1$ to $\texttt{cf}_5$ are created with the expression `new CF()`. All created objects contain the properties `q1` and `q2` and implicitly link to the `prototype` of their constructor function `CF`. For example, the prototype of $\texttt{cf}_1$ is $\texttt{CF}_\texttt{p}$. The constructor also has two properties `P1` and `P2`, which are not visible to the objects $\texttt{cf}_1$ to $\texttt{cf}_5$ and also not to the prototype $\texttt{CF}_\texttt{p}$. The property `CFP1` of $\texttt{CF}_\texttt{p}$ is shared by $\texttt{cf}_1$ to $\texttt{cf}_5$ but not by `CF`. In other words, objects have access to the properties of their prototype, if they do not already have properties with that exact name. The constructor function prototype $\texttt{CF}_\texttt{p}$ also has an implicit prototype link to a (second) prototype, which in turn could have a link to another (third) prototype. This prototype chain terminates at the `Object` prototype. All properties that are defined by prototypes when following this chain are shared.

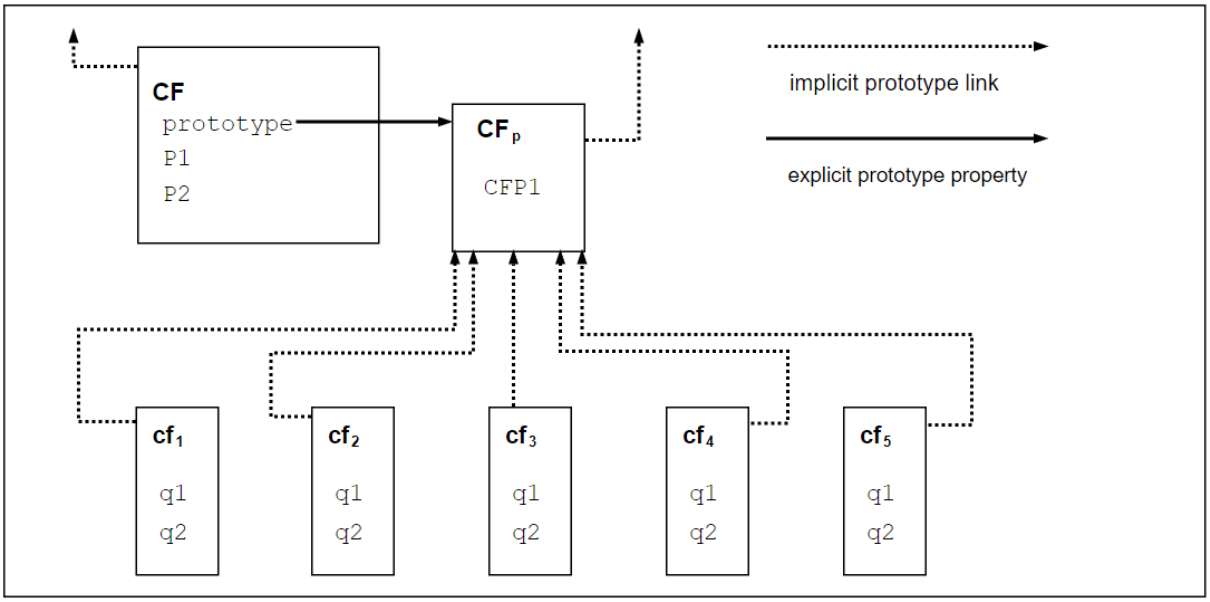


Figure 2.1: JavaScript prototypes example [4]

2.2 Graal.js

This section introduces *GraalVM JavaScript*, in short *Graal.js*[3], a high performance JavaScript implementation in Java. It gives an overview of a number of concepts regarding the GraalVM platform which are relevant background information for Chapter 4. Graal.js is built on the GraalVM, which is an extended Java HotSpot VM with support for additional programming languages. It is developed by Oracle Labs. The major goals of Graal.js are:

- High performance JavaScript execution
- Full compatibility with the latest version of ECMAScript
- Fast language interoperability using *Polyglot Programming*[6]

GraalVM allows users to write polyglot applications that are able to inter-operate between two or more languages using the *Truffle Language Implementation Framework*, shorthand *Truffle* [11]. Truffle is a library for implementing programming languages by building a self-modifying abstract syntax tree (AST) .

2.2.1 Truffle Nodes and DSL

Truffle facilitates the implementation of a programming language via the concept of Truffle nodes. These represent the vertices of an AST and the abstract `Node` class should be extended to implement

[3] `https://github.com/oracle/graaljs`

desired functionality for the created language. A Truffle node always has an `execute()` method that performs the operation. Operations, e.g., an addition in JavaScript, could allow arguments of multiple different types. This is where Truffle *specializations* come in, which provide multiple `execute()` methods that fit the possible parameter type combinations of an operation. Since that would result in a large amount of boilerplate code, the Truffle *domain specific language* (DSL) uses the Java annotation system to alleviate the amount of boilerplate code [7].

2.3 The Fetch Living Standard

This section gives an idea of what the Fetch Standard encompasses and the goals it aims to achieve. It gives some background information about the history, introduce the vital classes on a high level, but will not contain much implementation specific requirements or challenges. In Chapter 3 these are discussed in greater detail.

During the early stages of developing web applications it was a difficult task to perform asynchronous requests across websites, for example, when requesting data from a server. Implementations of JavaScript, by the browsers, included no such functionality. In 1998, the `XMLHttpRequest` class, introduced with Internet Explorer 5, provided the first standardized API. It was initially designed to fetch XML resources but support for JSON and other data formats was added later. As web applications increased in size and complexity, this API got too difficult to work with. To overcome limitations of the `XMLHttpRequest` a modern successor was introduced in 2015. The Fetch API became the de facto standard for handling asynchronous requests in web applications. One significant advantage is that `Promises` are used, which allows for a much cleaner API. It is defined and standardized by the WHATWG[5]. The *WHATWG* (Web Hypertext Application Technology Working Group) is an organization dedicated to developing, maintaining and updating a multitude of web standards. This includes the Fetch APIs Living Standard document. A document that is a living standard is continually being edited and updated, potentially deprecating features or introducing new ones. The reference for this implementation is a snapshot of the standard as of 28th July 2022 [4].

2.3.1 Concept and Goals

Fetching is the process of accessing a resource that lies on the network. For example, such resources could be public APIs, websites or resources on the local network. The Fetch API provides the classes `Request` and `Response` which are connected via fetching. A typical use case would look like this:

1. User specifies a desired resource by creating a `Request` *`request`*.

[4] https://fetch.spec.whatwg.org/commit-snapshots/9bb2ded94073377ec5d9b5e3cda391df6c769a0a/

2. `fetch(request)` is invoked.

3. The call returns a `Response` *response* containing the desired resource.

On a high level this operation may seem trivial. A request goes in and a response is returned to access the resource. The specifics can, however, be quite intricate. Involving, for example, redirects or different URL schemes. Therefore, fetching should be clearly defined and written-down. This is where WHATWG sets the goal post for the Fetch Standard: It should provide a consistent way of handling the fetching process across all web platforms. [5]

2.3.2 HTTP Fetch

While not being strictly limited to the HTTP protocol, the Fetch API uses a lot of HTTP concepts and applies them to resources obtained via other means. This section introduces these concepts on a high level.

- *Method*: A request method is a string that describes the action the user wants to perform on a resource. In addition to the standard HTTP methods[5] the Fetch API supports custom, user defined request methods.

- *Headers*: A header list, included in both request and response, enables specifying additional information, like authorization- or meta-data. It can contain multiple headers where a header consists of a unique name and a value. For example, the following header field is used to indicate the type of body content that is sent: `"Content-Type":"application/json"`.

- *Status*: A response status is an integer indicating if the request has been successfully completed or if errors occurred. The Fetch API can handle HTTP status codes[6] as well as all values in the inclusive range from 0 to 999.

- *Body*: Contains the actual data that is being sent or retrieved and is a part of both request and response.

- *Request*: A `Request` object is the input to the fetch method. It defines an operation to be performed on a given resource.

- *Response*: A `Response` object is the result of `fetch`. It makes the response data accessible and is built as the resource is fetched.

[5] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
[6] https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

2.3.3 The Fetch Method

This is the central method of the Fetch API. It takes a `Request` and starts the process of fetching the resource over the network. It runs asynchronously and resolves to a `Response` once it is available. Listing 2.1 shows typical usage of the `fetch` method. In this case, the request is defined by just the URL and the `Request` object is created implicitly.

Listing 2.1: Fetch usage in JavaScript

```
1 const response = await fetch("https://example.com");
```

2.4 Java Network API

In this section, we look at the networking capabilities that the Java standard library provides and go into more detail on relevant classes for this implementation.

2.4.1 Java Networking Capabilities

The `java.net`[7] package contains classes for implementing network applications. It can roughly be divided into a low-level and a high-level API which deal with different levels of abstraction.

Lower-Level API

- IP addresses
- Network interfaces, defining multiple protocols and interfaces
- Network sockets, for lower level bi-directional communication

Higher-Level API

- *URIs* (Universal Resource Identifiers) and *URLs* (Universal Resource Locators), for resource identification and location
- Connections, that represent the network connections to resources

[7] `https://docs.oracle.com/javase/8/docs/api/java/net/package-summary.html`

For implementing the Fetch API functionality, the higher-level API, dealing with URLs and resource connections, is applicative. The `URL` class is useful for validating user input and, in particular, the class `HttpURLConnection`[8] provides a lot of out of the box functionality for the fetching process. The next section explains the class in more detail and contains an example of how it fits a basic Fetch API use case.

2.4.2 HttpURLConnection

While the underlying connection can stay open for multiple requests, an instance of this class represents a single HTTP request. Listing 2.2 shows how to send a basic `GET` request and retrieve the response status code.

Listing 2.2: GET request with HttpURLConnection

```
1 URL url = new URL("http://example.com");
2 HttpURLConnection connection = (HttpURLConnection) url.openConnection();
3 connection.setRequestMethod("GET");
4 int status = connection.getResponseCode();
```

As described in Section 2.3.3, this is basically the functionality required for the Fetch API; a request goes in and a response comes out. We just have to pass in the request data via different setter methods, such as `setRequestMethod()`, and collect the response data via different getter methods, such as `getReponseCode()`. The point where the response data is extracted from the `HttpURLConnection` class is where a `Response` object could be created.

2.4.3 HTTP Redirection

A *HTTP redirect* is used to give more than one URL to a resource and is triggered when the server sends a specific redirect response. Redirect responses have a status code beginning with `3`, which can indicate different reasons for the redirect where the main difference is between permanent and temporary redirection.

Figure 2.2 shows the requests and responses involved in the most basic redirect. The initial request to a resource receives a response with status code `301`, indicating a permanently moved resource, and the new URL, where the resource is actually located, in the `Location` header field. In this case, the location is a relative path on the same domain. A subsequent request is then sent to this new location to actually fetch the resource.

[8] `https://docs.oracle.com/javase/8/docs/api/java/net/HttpURLConnection.html`

Figure 2.2: Example of a redirect [8]

2.5 Existing Implementation: node-fetch

This section introduces the *node-fetch*[9] implementation of the standard. It brings the Fetch API functionality to the *Node.js*[10] platform, which is a runtime for JavaScript. As of September 2022, the module is being downloaded roughly 36 million times weekly from the node package registry[11]. Node-fetch is developed as an open source project with over 100 contributors.

Since it also implements the standard from a server-side context, not from the client or browser context as assumed by the WHATWG, node-fetch is a good reference for this implementation. In addition, the module defines an extensive list of Fetch API unit tests.

2.5.1 Non-standard Extensions

Node-fetch implements useful extensions to the standard-defined functionality. Mainly the `Request` class is extended to give the user more control over the fetching process. The implementation also tries to be more transparent with errors during fetching. The following options and default values are non-standard extensions:

- `follow = 20`: The maximum redirect count. WHATWG defines a constant value of `20`. This option can change that behavior. A value of `0` will allow no redirects.

[9] `https://github.com/node-fetch/node-fetch`
[10] `https://nodejs.org/`
[11] `https://www.npmjs.com/package/node-fetch`

- `compress = true`: node-fetch supports *gzip* and *deflate*, which are content encoding algorithms used for the response body.
- `size = 0`: Defines the maximum size of the response body in bytes. A value of `0` disables the option.
- `agent = null`: A http(s). Agent[12] instance that allows to specify networking options that are out of scope for the Fetch API, such as using only IPv6 or performing custom DNS lookups.
- `highWaterMark = 16384`: The maximum size of the internal body buffer.
- `insecureHttpParser = false`: To use an insecure HTTP parser that allows invalid HTTP header fields.

These fields are neither specified nor mandated by the Fetch API standard. As the node-fetch module is an open source project, these features were probably requested by the community and seen as useful additions by the maintainers. They provide extra functionality that is often needed when using the Fetch API in a Node.js application.

[12] `https://nodejs.org/api/http.html#http_new_agent_options`

JOHANNES KEPLER
UNIVERSITY LINZ

3 Fetch API for Graal.js

This chapter discusses the scope of the implementation by outlining the goals and non-goals in Section 3.1. Section 3.2 lists the Fetch API definitions from the WHATWG standard which serve as concrete implementation requirements for Chapter 4. Finally, Section 3.3 highlights a number of interesting challenges which might arising from these requirements.

3.1 Scope and Goals

Although the in Section 2.3 described basic concept and functionality might seem simple, the Fetch API standard defines a lot of in-depth behavior and specifics which might not be feasible or are out of scope for this implementation.

Main goal is to implement the Fetch API based on the living standard. All functionality should be well integrated into the Graal.js interpreter and follow its programming model. The implementation should cover as much as possible, but does not have to cover all cases or details of the standard. The incomplete and missing parts of the standard are summarized in Section 4.7.

The Java package `java.net` should be used to implement all of the fetch networking functionality. It is also not expected to have excellent I/O performance and does not have to be further optimized.

To test and verify completeness, tests from node-fetch[1] should be used.

3.2 Implementation Requirements

This section describes the required interfaces, classes and methods as defined by the WHATWG living standard. The code in the following sections is Web IDL[2] code from [5] adapted for readability. Also, since the WHATWG definitions assume a browser context some fields or methods that are not sensible outside of a browser are omitted, as discussed in Section 4.7.

[1] `https://github.com/node-fetch/node-fetch/tree/main/test`
[2] `https://webidl.spec.whatwg.org/`

3.2.1 The Fetch Method

The Fetch API provides a function `fetch()` which should be globally accessible in the Javascript realm with a method signature as in Listing 3.1.

Listing 3.1: Fetch method signature

```
Promise<Response> fetch(RequestInfo input, optional RequestInit init);
```

The `input` parameter can be a string representing the URL or a `Request` object. The `init` parameter is optional and allows the user to set options on the request like the method or headers. The parameter types are described in Section 3.2.2. The parameters are the same as for the constructor of the `Request` class.

The first step of the method is to create a new `Request` by calling the constructor with `input` and `init` as arguments. While the resources are fetched, a `Response` object should be built. As mentioned in Section 2.3.3, the method should return a `Promise` which resolves to a `Response` object or rejects in case of a network error with a `FetchError`.

3.2.2 Request Class

As described in Section 2.3.2, a request is the start of the fetching process. This class should allow to specify a resource and different fetching related options. The `Request` type is defined as in Listing 3.2 and should be accessible globally. It should not only be usable with the Fetch API but also serve as a generic request implementation.

One of the parameters to the constructor is of the `RequestInit` type. It contains options to be set when creating a new instance. The `method` field sets the request method. A `Header` object is created from the value in the `headers` field. Same with the `body` field. The `referrer` and `referrerPolicy` fields are used to set the respective request properties. `RequestRedirect` is a enum type with `"follow"`, `"manual"` or `"error"` as values which set the request redirect mode.

The `referrer` and `referrerPolicy` fields are required for setting the `Referrer` and `ReferrerPolicy` HTTP header of a request. The user should also be able to specify those headers via the `headers` field, but these distinct properties allow for handling default values and conversions. The referrer headers allow the server to identify where the request comes from and are used for analytics, optimized caching, and logging [9].

A request is constructed with a `RequestInfo`, which is either a string representing a URL or a `Request` object, and also the already described optional `RequestInit` object. The `Request` constructor has the same parameter list as the fetch method. The user can just skip the step of calling the constructor since the fetch method should do this implicitly with the provided arguments. If a `Request`

Listing 3.2: Request definition

```
typedef (Request or String) RequestInfo;

dictionary RequestInit {
  String method;
  HeadersInit headers;
  BodyInit body;
  String referrer;
  ReferrerPolicy referrerPolicy;
  RequestRedirect redirect;
};

interface Request {
  constructor(RequestInfo input, optional RequestInit init);
  readonly attribute String method;
  readonly attribute String url;
  readonly attribute Headers headers;
  readonly attribute String referrer;
  readonly attribute ReferrerPolicy referrerPolicy;
  readonly attribute RequestRedirect redirect;
  Request clone();
};
Request includes Body;
```

object (*request*) and a `RequestInit` object (*init*) are passed to the constructor, the values from *init* should overwrite the values in *request* and create a new instance. All the properties should be set in the constructor and be immutable afterwards. The constructor should validate the arguments and throw an appropriate `TypeError` if invalid values are provided. In case no value is given for a specific field, the constructor should set the following defaults:

- `method = "GET"`
- `headers = {}`
- `referrer = "client"`
- `referrerPolicy = ""`
- `redirect = "follow"`

With value v, the `referrer` getter should return the empty string if v is `"no-referrer"`, `"about:client"` if v is `"client"` or v otherwise. Other getters should return the internal value.

The `clone()` method returns a cloned object of this request. The `Request` class extends the `Body` class and should provide all properties and methods defined by `Body` as described in Section 3.2.5.

3.2.3 Response Class

Listing 3.3: Response definition

```
dictionary ResponseInit {
  short status = 200;
  String statusText = "";
  HeadersInit headers;
};

interface Response {
  constructor(optional BodyInit body, optional ResponseInit init);
  static Response error();
  static Response redirect(String url, optional short status = 302);
  static Response json(any data, optional ResponseInit init);
  readonly attribute ResponseType type;
  readonly attribute String url;
  readonly attribute boolean redirected;
  readonly attribute short status;
  readonly attribute boolean ok;
  readonly attribute String statusText;
  readonly attribute Headers headers;
  Response clone();
};
Response includes Body;

enum ResponseType { "basic", "cors", "default", "error", "opaque", "opaqueredirect" };
```

As described in Section 2.3.2, a `Response` should be the result of fetching. It is built during fetching and should make all relevant response data accessible. The type `Response` should be accessible globally with an interface as defined in Listing 3.3 and described in this section.

The optional `body` parameter of the constructor is of type `BodyInit` as described in Section 3.2.5. It should allow specifying the response's body. The optional `init` parameter of the constructor is of type `ResponseInit`. It should allow setting options for the response being created. The `status` and `statusText` fields set the respective response fields. A `Header` object should be created from the value in the `headers` field. Similar to the request class all, properties should be set in the constructor and be immutable afterwards. The following defaults should be set by the constructor:

- `status = 200`
- `statusText = ""`
- `headers = {}`

- `type = "default"`

The static `error()` method should create a new `Response r` with `r.type === 'error'`. The static `redirect(url, status)` method should validate the URL and status to be a redirection status. Then create a new `Response r` with `r.headers.get('Location') === url` and `r.status === status`. The default status code should be 302. The static `json(data, init)` method should create a new `Response r` with options `init` and the body defined by `data`.

The `ok` getter is a shorthand and should return `true` if `status` is in the inclusive range from 200 to 299. The `redirected` getter should return `true` if a redirect occurred. Other getters should return the internal value.

The `clone()` method should return a cloned object of this response. Same as the `Request` class, the `Response` class also extends the `Body` class and should provide all properties and methods defined by `Body` as described in Section 3.2.5.

3.2.4 Headers Class

Listing 3.4: Headers definition

```
typedef (sequence<sequence<String>> or record<String, String>) HeadersInit;

interface Headers {
  constructor(optional HeadersInit init);
  undefined append(String name, String value);
  undefined delete(String name);
  String? get(String name);
  boolean has(String name);
  undefined set(String name, String value);
  iterable<String, String>;
};
```

The `Headers` type is defined as in Listing 3.4 and is used in request and response but should also be accessible globally. A header consists of a unique name and a value. As described in Section 2.3.2, a `Headers` object should have an internal headers list. This internal list should be able to store multiple values per name. The constructor should take a `HeadersInit` object as argument and create the internal header list from this argument. In terms of creating a new `Headers` object, the values in Listing 3.5 are equal. If no argument is provided, the internal list should be empty. Since names are unique it has a map-like interface.

At every method call the arguments should be validated using the respective header-field production[3] for both header name and value.

[3] `https://datatracker.ietf.org/doc/html/rfc7230#section-3.2`

Listing 3.5: HeadersInit example

```
const init1 = { "Accept": "*/*", "foo": "bar" };
const init2 = [
    ["Accept", "*/*"],
    ["foo", "bar"]
];
```

The `append(name, value)` method should append a header. Either by adding the value to the list if `name` already exists or by creating a new list if `name` does not exist.

The `delete(name)` method should delete the entry with identifier `name`.

The `get(name)` method should return all values of identifier `name` as string separated by comma and space or `undefined` if `name` does not exist.

The `has(name)` method should return `true` if `name` exists or `false` otherwise.

The `set(name, value)` method should set `name` to a new list with `value` as its only element.

Headers should be iterable and provide a `forEach()` method. Even though it does not matter how the headers are organized internally, they should be returned in alphabetical order and header names should be normalized to lowercase.

3.2.5 Body-mixin

The `Body` class, as defined in Listing 3.6, should purely be used to define common behavior related to the HTTP body for the `Request` and `Response` classes. It holds a readable stream which it should be able to consume and convert to a specific data type. Multiple methods are defined that achieve this. All methods should consume the body and return a `Promise` that resolves to the desired data type. The default value of `body` should be `null`.

The `bodyUsed` getter should return `true` if the body has been consumed and `false` otherwise.

The `arrayBuffer()` method should consume the body and return an `ArrayBuffer`[4] object, that is used to represent fixed length binary data.

The `blob()` method should consume the body and return a `Blob`[5] object.

The `formData()` method should consume the body and return a `FormData`[6] object.

[4] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer`
[5] `https://developer.mozilla.org/en-US/docs/Web/API/Blob`
[6] `https://developer.mozilla.org/en-US/docs/Web/API/FormData`

Listing 3.6: Definition body

```
interface mixin Body {
  readonly attribute ReadableStream? body;
  readonly attribute boolean bodyUsed;
  Promise<ArrayBuffer> arrayBuffer();
  Promise<Blob> blob();
  Promise<FormData> formData();
  Promise<any> json();
  Promise<String> text();
};
```

The `json()` method should consume the body and parse it as a JavaScript object using *JavaScript Object Notation*[7] (JSON).

The `text()` method should consume the body, parse it as plain text and return a string.

3.3 Selected Implementation Challenges

This section lists some challenges these requirements might yield. The following list is not an extensive collection of all the implementation challenges. It should rather give an idea of which aspects of the requirements might be difficult to implement.

- The `Body` should be implemented as a base class to the `Request` and `Response` class. This should not be an issue on the Java side, but for the JavaScript objects it could be difficult to set the types up correctly.
- The `fetch` method and all the body methods should return a `Promise`. Because a `Response` in JavaScript represents the result of a asynchronous operation there could be difficulties if the internal networking implementation would work in a synchronous way.
- HTTP redirects should be handled properly according to the standard. This also includes chains of multiple redirects and possibly dealing with an infinite redirect loop.

[7] `https://developer.mozilla.org/en-US/docs/Glossary/JSON`

4 Implementation

This chapter shows the actual implementation of the Fetch API for Graal.js, as defined in Chapter 3. First, Section 4.1 provides an overview of the implementation. It should give an idea of which classes are created or extended and how they are connected. In Section 4.2, the networking functionality and Java's capabilities in that regard are discussed. The globally accessible entry point for the `fetch()` method is explained in Section 4.3. The implementation of types defined by the standard is documented in Section 4.4 (operational types of the API), Section 4.5 (new error types), and Section 4.6 (promises). Finally, incomplete and missing features are discussed in Section 4.7.

4.1 Overview

The core function of the Fetch API, `fetch()`, should be provided as a global-builtin. In other words, a new function property has to be added to the global object. The class `GlobalBuiltins` contains other such global function properties, for example the `parseInt()` function. The new `fetch` keyword is added to this class by providing a Truffle node that implements the fetch operation. This Truffle node (`JSGlobalFetchNode`) defines a specialization method which has the same signature and represents the fetch method defined in Section 3.2.1. All of the fetching process is then completed inside the context of this specialization method: (1) A request is built from the parameters, (2) the resource is fetched using Java networking features, and (3) a response is built and returned.

However, this requires the `Request`, `Response`, and `Headers` types. These do not exist yet and to use them, as input and output types in the fetching process, they first have to be implemented. To provide a new type, the `ConstructorBuiltins` class, which contains built-in constructor functions, has to be extended to include the three new types. Again, for every type a Truffle node is needed to implement the respective constructor operation and return a new object. To create a new object of a given type the prototype of this type has to be supplied. The classes `JSFetchRequest`, `JSFetchResponse`, and `JSFetchHeaders` build and supply the prototype for the respective type. The actual data for objects of these types is stored in the internal classes `FetchRequest`, `FetchResponse` and `FetchHeaders`. The classes with a `JS` prefix make the data accessible from the JavaScript side, while the actual data that should also be accessible from the Java side is stored in

these internal classes. Also, the `JSContext` and `JSRealm` classes have to be adapted to contain the new types.

To provide the `FetchError` type the classes `Errors` and `JSErrorType` have to be extended.

4.2 Fetch Networking Functionality

This section details how the Fetch API networking functionality is implemented. We utilize classes from the Java network API (`java.net`), as described in Section 2.4. Finally, we discuss some limitations of our approach that are relevant for the Fetch API use case.

4.2.1 Handling Redirects

This section shows the implementation of HTTP redirection, described in Section 2.4.3. The example in Listing 4.1 highlights the redirect handling but also already contains the final method signature and control flow. This static method receives a `FetchRequest` and returns a `FetchResponse`, the internal data classes. This example should outline how redirects are handled. The idea is to recursively call `connect(request)` for redirects. The argument that is passed for recursive calls is the previous request with updated values. For example, the redirect counter is increased for every recursive call. Also, the URL is updated in order to follow the redirects. The recursion stops when a request does not return a redirect response or the maximum amount of redirects is reached.

The `HttpURLConnection` class follows redirects automatically by default. To get the required control we have to use the `setInstanceFollowRedirects()` method as in line 4 of Listing 4.1. After the `connection.connect()` call we check if the status code signals a redirect response. The `isRedirect(status)` method returns true if `status` is one of the following values:

- `301` (Moved Permanently)
- `302` (Found but moved temporarily, for search engine optimization)
- `303` (See Other)
- `307` (Temporary Redirect)
- `308` (Permanent Redirect)

If a redirect response is identified, we extract the value of the `Location` header and create the new URL. It is possible that the URL in the `Location` field also points to a redirection. This results in a chain of redirects that could theoretically loop endlessly if the URLs form a cycle. Therefore, the fetch standard defines a redirect count that keeps track of the amount of redirects followed and terminates the execution at a set maximum. By default, a maximum of 20 redirects are followed before a `FetchError` is returned. In Listing 4.1, this check is done at line 14. At the end, we update the request URL using `request.setUrl(locationURL)` and return the result of recursively calling `connect` which sends a request to the updated URL. If no redirect occurred a `FetchResponse` object is built by extracting the data from `connection` and subsequently returned.

Listing 4.1: Handling redirects

```
1  public static FetchResponse connect(FetchRequest request) {
2      // Setup Connection
3      HttpURLConnection connection = (HttpURLConnection)
       ↪  request.getUrl().openConnection();
4      connection.setInstanceFollowRedirects(false);
5      connection.setRequestMethod(request.getMethod());
6      connection.connect();
7
8      int status = connection.getResponseCode();
9
10     if (isRedirect(status)) {
11         String location = connection.getHeaderField("Location");
12         URL locationURL =
           ↪  URI.create(request.getUrl().toString()).resolve(location).toURL();
13
14         if (request.getRedirectCount() >= request.getFollow()) {
15             throw Errors.createFetchError("maximum redirect reached");
16         }
17
18         request.incrementRedirectCount();
19         request.setUrl(locationURL);
20
21         // following the redirect
22         return connect(request);
23     }
24
25     FetchResponse response = new FetchResponse();
26     // build response
27     return response;
28 }
```

In Listing 4.2, we show the implementation of multiple redirection modes that are defined for a fetch request in Section 3.2.2. The three valid redirect modes are:

- `manual`: Returns the response as is and gives the user control over handling the redirect.
- `error`: Throws a `FetchError` when a redirect is encountered.
- `follow`: The default value, automatically follows redirects as already discussed.

If any other value is passed, a `TypeError` is thrown.

Listing 4.2: Implementation of the redirect modes

```
1  if (isRedirect(status)) {
2    String location = connection.getHeaderField("Location");
3    URL locationURL =
     ↪  URI.create(request.getUrl().toString()).resolve(location).toURL();
4
5    switch (request.getRedirectMode()) {
6        case "manual":  // return response as is
7            break;
8        case "error":   // reject redirection
9            throw Errors.createFetchError("uri requested responds with a redirect,
              ↪  redirect mode is set to error", "no-redirect", node);
10       case "follow":
11           // handle redirect
12           // ...
13           return connect(request);
14       default:
15           throw Errors.createTypeError("Redirect option " +
              ↪  request.getRedirectMode() + " is not a valid value of
              ↪  RequestRedirect");
16   }
17 }
```

Similar to checking if the maximum amount of redirects has been reached, Fetch performs a number of checks during a redirect and modifies the request accordingly. The check in Listing 4.3 deletes a number of sensitive headers from the request, if a redirect requires to change the host or protocol in any way. For example, if the host changes from `example.com` to `foo.bar` or the protocol from `https` to `http`.

Listing 4.3: Sensitive headers when redirecting

```
1  if (!isDomainOrSubDomain(locationURL, request.getUrl()) ||
   ↪  !isSameProtocol(locationURL, request.getUrl())) {
2      Set.of("authorization", "www-authenticate", "cookie", "cookie2").forEach(k ->
       ↪  {
3          request.headers.delete(k);
4      });
5  }
```

4.2.2 Request and Response Body

To fit the requirements, `HttpURLConnection` must be able to send and receive body data. The body data that is sent is stored in the request and the response body is stored in the response. The methods `getOutputStream()` and `getInputStream()` enable this functionality. With `getOutputStream()` we can manipulate the stream that writes to the connection and with `getInputStream()` we can do the same for the stream that reads from the open connection.

Listing 4.4 shows how both these methods are used to setup the connection and extract the response body. If a request body is present, we write it to the output stream using the `OutputStreamWriter` class. To get the response body we use `getInputStream()`. This method throws an exception when the request returns a response with an error status code. To also get the body of erroneous requests we use `getErrorStream()`. At line 23, the `responseBody` is created which defaults to the empty string.

4.2.3 Networking Limitations

The `HttpURLConnection` suits the use case for a Fetch API implementation very well and provides useful abstractions for HTTP connections. However, it has some limitations that are relevant for certain details of the Fetch API. This section lists those limitations and Section 4.7 explains how they correlate to concrete Fetch API features that are not possible or need special setup when using the `HttpURLConnection` class.

A request always has a request method and, as Listing 2.2 shows, the `setRequestMethod` setter method allows setting the value for this request field. But the implementation of `HttpURLConnection` only allows the following method values: `"GET"`, `"POST"`, `"HEAD"`, `"OPTIONS"`, `"PUT"`, `"DELETE"` and `"TRACE"`.

As mentioned the class provides useful behavior out of the box but this also includes some undesired side effects. A number of HTTP headers are set by the class implicitly. This internal be-

Listing 4.4: `HttpURLConnection` content using streams

```
1  HttpURLConnection connection = (HttpURLConnection)
   ↪  request.getUrl().openConnection();
2
3  if (request.body != null) {
4      OutputStreamWriter out = new OutputStreamWriter(connection.getOutputStream());
5      out.write(request.body);
6      out.flush();
7      out.close();
8  }
9
10 connection.connect();
11
12 InputStream inputStream = null;
13 try {
14     inputStream = connection.getInputStream();
15 } catch (IOException exception) {
16     inputStream = connection.getErrorStream();
17 }
18
19 BufferedReader br = null;
20 if (inputStream != null) {
21     br = new BufferedReader(new InputStreamReader(inputStream));
22 }
23 String responseBody = br != null ? br.lines().collect(Collectors.joining()) : "";
```

haviour overwrites any user defined headers. Consider using an output stream to set the body for a `HttpURLConnection` as in Listing 4.4. The `Content-Length` is automatically calculated when setting the body. If we now try to manually set the `Content-Length` header, the internally calculated value overwrites the user-provided one. Essentially, the `HttpURLConnection` class prevents the user from lying and sending wrong data in headers. Also, by default the class restricts the user from setting certain headers for security reasons, which requires workarounds. This includes, for example, the `Host` header.

While it is not directly a limitation of the `HttpURLConnection` class, it does not provide any functionality for asynchronous HTTP requests. Using Java concurrency features to work with multiple threads could be a way to provide this functionality to the current implementation. Other networking classes, for example `HttpClient`, support asynchronous request out of the box. But none of these classes provide as much control and flexibility as the `HttpURLConnection` class.

4.2.4 Final Networking Implementation

The method in Listing 4.5 is a shortened version of the final networking implementation. It is a static method in the `FetchHttpConnection` class, which is used by the global built-in `fetch()`. As the previous sections already presented parts of this method they are omitted and replaced by comments here.

The method takes a `FetchRequest` as argument and returns a `FetchResponse`. If a network error occurs an `IOException` is thrown which is later converted to a `FetchError`.

A *scheme* is the first part of an URL and is used as an identifier for launching specific applications with the URL. For example, a URL starting with `https://` opens the web browser while `mailto://` could open a mail client. The supported schemes by the Fetch API are `http`, `https` and `data`. The first check in Listing 4.5 validates the URL the user provided to only include a supported scheme. Otherwise, the call is rejected with a `TypeError`. After that, the request is built by setting up the `HttpURLConnection` class. At line 11, the method `setRequestHeaders()` is called. Before the request is sent, the request body is written to the connection's output stream as explained in Section 4.2.2.

At line 14 in Listing 4.5, a request is performed using the `connect()` method. As already discussed in Section 4.2.1, the response could potentially be a redirect that is handled here. Then, the response is built by extracting data from the `HttpURLConnection` and filling it into a new `FetchResponse`. The `responseBody` is extracted as already shown in Section 4.2.2. Finally, the response is returned by the method.

The `setRequestHeaders()` method, shown in Listing 4.6, is used by the final networking code. It sets all the default and user-defined headers for a given `HttpURLConnection`. First, default values for the `Accept` and `User-Agent` headers are set. Also, the `setFixedLengthStreamingMode()` method is called with the body length. By default the `HttpURLConnection` class buffers everything that is written to its output stream and calculate the `Content-Length` header that way. By already providing the final body length and using a fixed length streaming mode the connection is able to set the `Content-Length` header sooner. After that, the `Referrer` and other user-defined headers are set. Since they are set last the user-defined headers should overwrite all default headers set before but as mentioned in Section 4.2.3 this is not possible for all headers.

Listing 4.5: Final networking implementation

```
public static FetchResponse connect(FetchRequest request) throws IOException {
    if (!SUPPORTED_SCHEMA.contains(request.getUrl().getProtocol())) {
        throw Errors.createTypeError("unsupported schema");
    }

    HttpURLConnection connection = (HttpURLConnection)
     ↪  request.getUrl().openConnection();
    connection.setInstanceFollowRedirects(false);
    connection.setRequestMethod(request.getMethod());
    connection.setDoOutput(true);

    setRequestHeaders(connection, request);
    // Set Requests Body

    connection.connect();

    int status = connection.getResponseCode();

    if (isRedirect(status)) {
        // Handle redirect
    }

    FetchResponse response = new FetchResponse();
    // Get Response Body
    response.setBody(responseBody);
    response.setUrl(request.getUrl());
    response.setCounter(request.getRedirectCount());
    response.setStatusText(connection.getResponseMessage());
    response.setStatus(connection.getResponseCode());
    response.setHeaders(new FetchHeaders(connection.getHeaderFields()));

    return response;
}
```

Listing 4.6: Setting the request headers

```
1  private static void setRequestHeaders(HttpURLConnection connection, FetchRequest
    ↪  req) {
2      connection.setRequestProperty("Accept", "*/*");
3      if (req.body != null) {
4          int length = req.getBodyBytes();
5          connection.setFixedLengthStreamingMode(length);
6      } else if (Set.of("POST", "PUT").contains(req.getMethod())) {
7          connection.setFixedLengthStreamingMode(0);
8      }
9      connection.setRequestProperty("User-Agent", "graaljs-fetch");
10
11     if (req.isReferrerUrl()) {
12         connection.setRequestProperty("Referrer", req.getReferrer());
13     }
14
15     // user specified headers
16     req.headers.keys().forEach(key -> {
17         connection.setRequestProperty(key, req.headers.get(key));
18     });
19 }
```

4.3 The Fetch Global Builtin

This section discusses the implementation of the global object properties in Graal.js and how the Fetch API's `fetch()` function is integrated.

As described in Section 2.1.1, the global object contains a number of value and function properties. The built-in function properties[1] are implemented by Graal.js in the `GlobalBuiltins` class. Next to the functions specified in the ECMAScript standard, the fetch function is also implemented as a built-in function property of the global object in this class. To make the fetch method available globally, we need the parser to recognize the function name (`fetch`). This is done by extending the `GlobalBuiltins` class. If the parser matches a function name, a Truffle node is returned. Then, the method of this node with a specialization that is compatible to the function arguments is executed or an error is thrown if no specialization is compatible.

Listing 4.7 shows our implementation of such a Truffle node that is returned when the `fetch` keyword is parsed. Our `JSGlobalFetchNode` extends `JSBuiltinNode`, which is a class defining shared behavior for JavaScript built-ins. All member fields of our node are related to creating promises and the private `toPromise()` method, which is detailed in Section 4.6. In the constructor, we pass all arguments to the base class and instantiate fields needed for promise creation. Most of the Fetch API related functionality of this node is implemented in its only specialization method, which is described in the next section.

4.3.1 The Fetch Specialization Method

The new node `JSGlobalFetchNode` defines one specialization method. This method, depicted in Listing 4.8, is the entry point of a `fetch()` call. As required and described in Section 3.2.1, the method takes two arguments: An `input` that is either a string or a request object and an `options` argument. The `toString` argument is internal to the parser and allows to parse other values as strings.

The `options` argument of type `RequestInit` is optional, so at line 3 in Listing 4.8 we check if the argument is present. If it is not present, an empty object is created and used instead. After that, we check if the `input` argument is a `JSFetchRequest` object, which represents a `Request` object on the JavaScript side. If that is the case, we extract the internal `FetchRequest` and apply the parsed options to it. This overwrites already existing headers values. If it is not a `JSFetchRequest` object, the `input` argument is parsed as string and a new `FetchRequest` is created using this string and the parsed options.

The `FetchRequest` object is then used as argument to `FetchHttpConnection.connect()`, which is described in Section 4.2.4. In short, this method performs the actual fetching process. It takes a

[1] `https://262.ecma-international.org/6.0/#sec-function-properties-of-the-global-object`

Listing 4.7: Fetch global Truffle node

```
public abstract static class JSGlobalFetchNode extends JSBuiltinNode {
    @Child NewPromiseCapabilityNode newPromiseCapability;
    @Child JSFunctionCallNode promiseResolutionCallNode;
    @Child TryCatchNode.GetErrorObjectNode getErrorObjectNode;
    private final BranchProfile errorBranch = BranchProfile.create();

    protected JSGlobalFetchNode(JSContext context, JSBuiltin builtin) {
        super(context, builtin);
        this.newPromiseCapability = NewPromiseCapabilityNode.create(context);
        this.promiseResolutionCallNode = JSFunctionCallNode.createCall();
    }

    protected JSDynamicObject toPromise(Object resolution) { ... }

    @Specialization
    protected JSDynamicObject fetch(Object input, Object options,
     ↪  @Cached("create()") JSToStringNode toString) { ... }
}
```

`FetchRequest` and returns a `FetchResponse`. This response is used to create a new `JSFetchResponse`, which represents the `Response` object on the JavaScript side. In case the method throws an `IOException`, a new `FetchError` is thrown on the JavaScript side. To adhere to the interface definitions, the response is wrapped in a `Promise` using the `toPromise()` method, as described in Section 4.6.

Listing 4.8: Fetch specialization method

```
@Specialization
protected JSDynamicObject fetch(Object input, Object options, @Cached("create()")
 ↪  JSToStringNode toString) {
    JSObject parsedOptions;
    if (options == Null.instance || options == Undefined.instance) {
        parsedOptions = JSOrdinary.create(getContext(), getRealm());
    } else {
        parsedOptions = (JSObject) options;
    }

    FetchRequest request;
    if (JSFetchRequest.isJSFetchRequest(input)) {
        request = JSFetchRequest.getInternalData((JSObject) input);
        request.applyRequestInit(parsedOptions);
    } else {
        request = new FetchRequest(toString.executeString(input), parsedOptions);
    }

    FetchResponse response = null;
    try {
        response = FetchHttpConnection.connect(request);
    } catch (IOException e) {
        throw Errors.createFetchError(e.getMessage(), "system", this);
    }

    return toPromise(JSFetchResponse.create(getContext(), getRealm(), response));
}
```

4.4 The Fetch API Types

This section describes the implementation of the `Request`, `Response`, and `Headers` type. As all of the three types have common patterns in their implementation, the first section about the `Request` implementation is referenced by the other sections for these common parts. The implementation of body functionality is discussed in Section 4.4.4.

4.4.1 Request Type

This section documents the implementation of the `Request` type defined in Section 3.2.2. It serves as input to a `fetch()` call but should also be usable outside of the Fetch API as a generic request type.

The Constructor

The `Request` type should be globally accessible via its constructor. To create a new constructor, that is recognized by the parser as such, the `ConstructorBuiltins` class has to be extended. Similar to how the parser matches the `fetch()` function name in Section 4.3, also the constructor name for this type is matched. If the parser matches the `Request` constructor, a Truffle node is created that handles the construction of the new object. This node (`ConstructFetchRequestNode`) then contains a specialization method that is invoked to actually create a new `Request` object.

This specialization method is implemented as in Listing 4.9. The two constructor arguments are `input` and `options`. Parsing the `options` argument, which is optional and of type `RequestInit`, is implemented in the same way as it is for the `fetch()` function call. If the argument is not present, an empty object is created and used instead. The `input` argument can be a string or a `Request` object. In line 10, we check if a `Request` object was provided and use it to create a new `FetchRequest`, a class that represents the data of a `Request` object internally. To overwrite the existing data with the potentially provided `options` argument, the `applyRequestInit()` method is invoked. If any other value is provided as `input`, we parse it as a string and create a new `FetchRequest` instance. In both cases, the new `Request` object is created with `JSFetchRequest.create()`.

Internal Data Class

The implementation of the request constructor in Listing 4.9 makes use of the `FetchRequest` class. It is a helper class that provides a convenient way to store the request data and to make it accessible during the fetching process. The class has a single constructor that takes a string representing the URL and a options object as arguments. Since this class is where the data is stored, the `FetchRequest` class implements a majority of the request constructor steps defined by [5]. Other steps are either implicitly completed or contained in the specialization method. The following points are the most important functionality implemented by the `FetchRequest` class:

- Parse and validate the URL.
- Set default values.
- Set values specified via the options argument in the constructor.

For the use case of wrapping a `Request` object in another `Request` object, as required by the `Request` constructor in Listing 4.9, the `FetchRequest` class provides the `applyRequestInit()` method that takes a `RequestInit` object as an argument. This provides the flexibility to set and overwrite the options even if there is already a `FetchRequest` instance.

Listing 4.9: Construct request specialization method

```
@Specialization
protected JSDynamicObject constructFetchRequest(JSDynamicObject newTarget, Object input, Object options, @Cached("create()") JSToStringNode toString) {
    JSObject parsedOptions;
    if (options == Null.instance || options == Undefined.instance) {
        parsedOptions = JSOrdinary.create(getContext(), getRealm());
    } else {
        parsedOptions = (JSObject) options;
    }

    if (JSFetchRequest.isJSFetchRequest(input) && input != Null.instance && input != Undefined.instance) {
        FetchRequest request = JSFetchRequest.getInternalData((JSObject) input);
        request.applyRequestInit(parsedOptions);
        return swapPrototype(JSFetchRequest.create(getContext(), getRealm(), request), newTarget);
    }

    TruffleString url = toString.executeString(input);
    FetchRequest request = new FetchRequest(url, parsedOptions);
    return swapPrototype(JSFetchRequest.create(getContext(), getRealm(), request), newTarget);
}
```

JavaScript Object and Prototype

The `JSFetchRequest` class implements the `Request` object for the JavaScript side which conforms to the interface given in Listing 3.2. The static `create()` method, used in Listing 4.9, creates a new `Request` object. It takes the JavaScript context, realm, and a `FetchRequest` as arguments and creates an object that fulfills the defined interface. To implement the interface itself, we have to supply the correct prototype for the `Request` type. Read-only properties are implemented as getter methods and are added to the prototype using `putBuiltinAccessorProperty()`. The `clone()` method is a built-in function for the prototype and is implemented in the `FetchRequestPrototypeBuiltins` class. All functions defined in this container class are then added to the prototype by using the `putFunctionsFromContainer()` method.

Non-standard extensions

Inspired by the node-fetch implementation, we include the `follow` option as an additional property for `RequestInit` objects. This option gives the user control over the maximum amount of

allowed redirects that should be followed, as described in Section 4.2.1. We adopted it because it is a simple addition and generically useful in a lot of scenarios. Other extensions made by node-fetch, as listed in Section 2.5.1, could be provided if necessary but are omitted in this implementation.

4.4.2 Response Type

This section describes the implementation of the `Response` type, defined in Section 3.2.3. It resembles the output of a `fetch` call but is also usable outside of the Fetch API as a generic response type. To make the type available by creating the constructor and prototype, the implementation is essentially the same as for the `Request` class. This section highlights the most important parts of the `Response` implementation. For a more detailed description refer to the `Request` implementation in Section 4.4.1.

- The `Response` constructor is added to the `ConstructorBuiltins` class.
- The node `ConstructFetchResponseNode` is created and contains a specialization method that creates a new `Response` object.
- The `JSFetchResponse` class implements the interface defined in Listing 3.3 and provides the correct prototype.
- Read-only properties are defined as getter methods and the `clone()` method is implemented in the `FetchResponsePrototypeBuiltins` class.
- Internal response and body data is managed by the `FetchResponse` class, which is used to create a `JSFetchResponse` object.

The constructor has two optional parameters, a `BodyInit` and a `ResponseInit` object. These are represented by the arguments `body` and `init`. First, the `init` argument that contains options for the response. If it is not present, an empty object is used instead. Then a `FetchResponse` is created by calling the constructor with `body` and the parsed options. Finally, the new `FetchResponse` is used to create a `JSFetchResponse`.

Static Response Methods

Static methods are implemented in the `FetchResponseFunctionBuiltins` class. The functions are then added to the prototype similar to how the `FetchResponsePrototypeBuiltins` are added. All static methods of the `Response` class return a `Response` object, each with properties set specific to the method.

Listing 4.10: Construct response specialization method

```
@Specialization
protected JSDynamicObject constructFetchResponse(JSDynamicObject newTarget, Object
 ↪ body, Object init) {
    JSObject parsedOptions;
    if (init == Null.instance || init == Undefined.instance) {
        parsedOptions = JSOrdinary.create(getContext(), getRealm());
    } else {
        parsedOptions = (JSObject) init;
    }
    FetchResponse res = new FetchResponse(body, parsedOptions);
    return swapPrototype(JSFetchResponse.create(getContext(), getRealm(), res),
     ↪ newTarget);
}
```

The `Response.error()` method in Listing 4.11 takes no arguments and returns a new `Response` object. Properties of this object are set as follows: (1) The status code is set to `0`, (2) the status text is the empty string, and (3) the response type is set to `"error"`.

Listing 4.11: Static response error method

```
protected JSFetchResponseObject error() {
    FetchResponse response = new FetchResponse();
    response.setStatus(0);
    response.setStatusText("");
    response.setType(FetchResponse.FetchResponseType.error);
    return JSFetchResponse.create(getContext(), getRealm(), response);
}
```

The `Response.json()` method in Listing 4.12 takes the same arguments as the `Response` constructor. It creates a new `Response` object with the given arguments and sets the content type of the body to `"application/json"`.

The `Response.redirect()` method in Listing 4.13 takes a URL and a status code as arguments. It creates a redirect response with the given arguments as it would look like if an actual redirect occurred. The content of the `Location` header is the given URL and the status code must be a redirect status. If a non-redirect status code is provided, a `RangeError` is thrown.

Listing 4.12: Static response JSON method

```
protected JSFetchResponseObject json(JSObject data, JSObject init) {
    FetchResponse response = new FetchResponse(data, init);
    response.setContentType("application/json");
    return JSFetchResponse.create(getContext(), getRealm(), response);
}
```

Listing 4.13: Static response redirect method

```
protected JSFetchResponseObject redirect(TruffleString url, int status) {
    URL parsedURL;
    try {
        parsedURL = new URL(url.toString());
    } catch (MalformedURLException exp) {
        throw Errors.createTypeError(exp.getMessage());
    }

    if (!FetchHttpConnection.REDIRECT_STATUS.contains(status)) {
        throw Errors.createRangeError("Failed to execute redirect on response:
        ↪ Invalid status code");
    }

    FetchResponse response = new FetchResponse();
    response.setStatus(status);
    response.headers.set("Location", parsedURL.toString());
    return JSFetchResponse.create(getContext(), getRealm(), response);
}
```

4.4.3 Headers Type

This section describes the implementation of the `Headers` type, as defined in Section 3.2.4. While both `Request` and `Response` have a member property of this type, it is also usable outside of the Fetch API types. To provide the constructor and prototype for `Headers`, the implementation is the same as for `Request` and `Response`. Here, the most important parts of the `Headers` implementation are highlighted. For a more detailed descriptions refer to the `Request` implementation in Section 4.4.1.

- The `Headers` constructor is added to the `ConstructorBuiltins` class.
- The new `ConstructFetchHeadersNode` contains a specialization that creates a new `Headers` object.

- The `JSFetchHeaders` class implements the interface defined in Listing 3.4 and provides the prototype.

- Methods are implemented in the `FetchHeadersPrototypeBuiltins` class.

- Internally, the data is managed in the `FetchHeaders` class. This class is a wrapper around a Java `Map<String, List<String>>`, with the header names as keys and multiple header values per name stored in a list.

- Also, the standard requires the header names to be lowercase and sorted alphabetically. A `TreeMap` automatically sorts the inserted entries by header names.

All the interface methods defined in `FetchHeadersPrototypeBuiltins` extract the internal `FetchHeaders` object, which in turn provides a method to actually execute the operation. The operation itself is handled by the internal Java `Map`. The following example shows how the `append()` method is implemented. Other methods have a similar implementation. Listing 4.14 shows the specialization that is entered when the method is called on a `Headers` object. As mentioned, the internal data is extracted and the arguments, `name` and `value`, are passed to the `append()` method of the `Headers` class.

Listing 4.14: JSHeaders append method

```
1 @Specialization
2 protected Object append(Object thisHeaders, TruffleString name, TruffleString
   ↪ value) {
3     asFetchHeaders(thisHeaders).getHeadersMap().append(name.toString(),
      ↪ value.toString());
4     return Undefined.instance;
5 }
```

The `append()` method of the `FetchHeaders` class in Listing 4.15 implements the behavior. It first converts the `name` to lowercase and normalizes the `value`. Then, both header name and value are validated. Validation is performed according to the `header-field` token production[2], as specified by the standard. Concrete regular expressions for the validation implementation are inspired by the Node.js standard http library[3]. Finally, the internal `Map` is used to perform the append operation.

[2] `https://datatracker.ietf.org/doc/html/rfc7230#section-3.2`
[3] `https://nodejs.org/api/http.html`

Listing 4.15: Headers append method

```
public void append(String name, String value) {
    name = name.toLowerCase();
    String normalizedValue = value.trim();
    validateHeaderName(name);
    validateHeaderValue(name, normalizedValue);
    headers.computeIfAbsent(name, v -> new ArrayList<>()).add(normalizedValue);
}
```

4.4.4 Body Functionality

Body Mixin

Since the `Request` and `Response` types share some characteristics, this section explains how the `Body` class is implemented for both types. As defined in Listing 3.6, the `Body` class should serve as a parent for `Request` and `Response` as it contains common behavior. In the case of this implementation, the properties of the body mixin are added to the `Request` and `Response` types as member properties for simplicity. This naturally comes with the downside of some amount of duplicated code parts. Here, the body interface implementation has to be duplicated for `Request` and `Response`.

Both the `body` and `bodyUsed` properties are implemented as getter methods in the `JSFetchRequest` and `JSFetchResponse` classes and are added to the respective prototype using `putBuiltinAccessorProperty`. The methods that read and consume the body are added to the request and response `PrototypeBuiltins` class, similar to the `clone()` method.

The example in Listing 4.16 shows how the `text()` method of the body mixin is implemented for the `Request` type. For the `Response` type the specialization method is implemented using the same approach. The example can be adapted to work for the response by replacing classes related to `Request` with classes related to `Response`. Since the `text()` method has no parameters, the specialization only receives the `Request` object the method is called on as argument. First, the internal data class `FetchRequest` is extracted. Then the body is consumed to get a string of the body content. To return a promise that resolves to the body string, the auxiliary method `toPromise()` is used. This method is discussed in Section 4.6. The other body methods are implemented similarly and mostly differ in the type of output that is created from the body. The following list gives an overview of the body function implementations:

- `text()`: The body is consumed from the internal data class and is returned as a string.

- `json()`: The body is consumed from the internal data class and is parsed as a JavaScript object using a `TruffleJSONParser`.
- `arrayBuffer()`: The body is consumed from the internal data class, is converted to a sequence of bytes and a new `JSArrayBuffer` containing the byte sequence is returned.
- `blob()` and `formData()`: These methods are both not implemented. They require prerequisite types, mentioned in Section 3.2.5, that are not implemented in GraalJS.

Listing 4.16: Implementation of the text() body method for the request type

```
1  @Specialization
2  protected JSDynamicObject text(Object thisRequest) {
3      FetchRequest request = asFetchRequest(thisRequest).getRequestMap();
4      TruffleString body = request.consumeBody();
5      return toPromise(body);
6  }
```

Internal Body Class

As already mentioned in this section, the body data for requests and responses is stored in the `FetchRequest` and `FetchResponse` internal data classes. Internally, the body functionality is implemented as a base class `FetchBody` for both the internal classes. It provides methods to set and consume the body to the `FetchRequest` and `FetchResponse` class, as shown in Listing 4.16. The body content is stored as a string in the `body` field. The class provides a method to consume the body, which updates the `bodyUsed` boolean field. An attempt to consume a body twice is rejected with a `TypeError`.

4.5 The FetchError Type

The Fetch API specification is not designed to be transparent about the cause of request errors during fetching. A *network error* is defined as a `Response` object with `response.type === "error"`. If at the end of fetching the response is a network error, the fetch method will throw a `TypeError`[4] to indicate that fetching failed, without further details about the cause. Therefore, the error handling in this implementation is inspired by the node-fetch implementation with the following baseline idea.

[4] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError`

Programmer errors that occur before fetching, for example, during argument validation, are a `TypeError`. This also is compliant with the specification. However, operational errors that occur during the fetching process, for example, a network failure, are not handled as the specification describes. A new error type `FetchError` captures the cause of the error.

To create a new error type, the class `JSErrorType` has to be adapted to contain the new type. Also, the error constructor has to be added to the `ConstructorBuiltins` class. A new utility function, shown in Listing 4.17, is added to the `Errors` class and can create a `FetchError` object during fetching. It takes a message, type, and the originating Truffle node as arguments. The node is used to get the correct JavaScript context. Next to the message, a type is included for more detailed error reporting. Line 6 in Listing 4.17 shows where this type property is set. The used error types are:

- `"system"`, for system or network errors
- `"no-redirect"`, if the request's redirect mode is set to `"error"`
- `"max-redirect"`, if the maximum amount of redirects has been reached
- `"unsupported-redirect"`, other redirect errors

Listing 4.17: The createFetchError() method

```
1  public static JSException createFetchError(String message, String type, Node
   ↪  originatingNode) {
2      JSContext context = JavaScriptLanguage.get(originatingNode).getJSContext();
3      JSRealm realm = JSRealm.get(originatingNode);
4      JSErrorObject errorObj = JSError.createErrorObject(context, realm,
       ↪  JSErrorType.FetchError);
5      JSError.setMessage(errorObj, message);
6      JSObjectUtil.putDataProperty(context, errorObj, JSError.ERRORS_TYPE, type,
       ↪  JSError.ERRORS_ATTRIBUTES);
7      JSException exception = JSException.create(JSErrorType.FetchError, message,
       ↪  errorObj, realm);
8      JSError.setException(realm, errorObj, exception, false);
9      return exception;
10 }
```

4.6 Promises

This section describes the implementation of the `toPromise()` helper method that is used when a `Promise` should be returned. This does not change a synchronous implementation to work asyn-

chronously. It just wraps the returned type in a `Promise`, when it is required by the standard to conform with the defined interfaces. Usages of `toPromise()` are the `fetch` method (in Listing 4.8) and all the body methods (e.g. `text()` in Listing 4.16).

The implementation of the `toPromise()` method is shown in Listing 4.18. The method has one parameter, that defines the value that the promise resolves to. In case of the `fetch()` method, it would be a `JSFetchResponseObject`. The `PromiseCapabilityRecord` class provides the promise functionality and by executing a function call with `promiseResolutionCallNode` we put the resolution value into the promise. We expect to enter the catch clause extremely rarely. With a Truffle `BranchProfile` (`errorBranch`) we can speculate on this and exclude it from optimized compilation. When `errorBranch.enter()` is invoked, the optimized code is invalidated and the branch is enabled for compilation. Otherwise, if `errorBranch.enter()` is never invoked, the branch never gets compiled.

Listing 4.18: Creating a promise

```
protected JSDynamicObject toPromise(Object resolution) {
    JSRealm realm = getRealm();
    PromiseCapabilityRecord promiseCapability =
     ↪ newPromiseCapability.execute(realm.getPromiseConstructor());
    try {
        promiseResolutionCallN-
         ↪ ode.executeCall(JSArguments.createOneArg(Undefined.instance,
         ↪ promiseCapability.getResolve(), resolution));
    } catch (AbstractTruffleException ex) {
        errorBranch.enter();
        if (getErrorObjectNode == null) {
            CompilerDirectives.transferToInterpreterAndInvalidate();
            getErrorObjectNode =
             ↪ insert(TryCatchNode.GetErrorObjectNode.create(getContext()));
        }
        Object error = getErrorObjectNode.execute(ex);
        promiseResolutionCallN-
         ↪ ode.executeCall(JSArguments.createOneArg(Undefined.instance,
         ↪ promiseCapability.getReject(), error));
    }
    return promiseCapability.getPromise();
```

4.7 Incomplete and Missing Features

While missing or incomplete features are also discussed in the applicable implementation sections, we also summarize them in this section to provide an organized overview.

4.7.1 Body Functionality

There are a number of limited and missing features regarding the `Body` class:

- `Body` as base class for `Request` and `Response` from the JavaScript side.
- Internal body is not a `ReadableStream`.
- The `blob()` and `formData()` body methods are not implemented.

Defined in Section 3.2.5 and implemented in Section 4.4.4, the `Body` class contains common behavior for `Request` and `Response` and these types should be implemented by using `Body` as base class. Because of how new types are implemented in Graal.js, it is easier to handle common functionality internally, but duplicate the implementation of body properties for both the `Request` and `Response` type in the `JSFetchRequest` and `JSFetchResponse` classes. In other words, for the JavaScript side there is no `Body` base class and the body properties are contained in request and response objects as direct member properties. The standard does not require the `Body` mixin to be exported as a type and is only intended for internal use, so for most use cases this implementation detail does not actually make a difference.

The standard requires the `body` property to be an instance of `ReadableStream`, which is a web type. The implementation uses a reasonable workaround by storing the actual body data in a Java `String` and using Java I/O classes (`InputStream` and `OutputStream`) with the `HttpURLConnection`.

As already discussed in Section 4.4.4, the `blob()` and `formData()` methods are not implemented because their required return types are not yet implemented for Graal.js.

4.7.2 Browser Features

The standard defines the Fetch API for the implementation in a browser. All functionality related to the topics listed below are not relevant for this implementation, because they only apply in a browser context. As these limitations were already known before implementing, details regarding these topics were already omitted from the implementation requirements in Section 3.2.

- Cross-Origin

- Content Security Policy
- Mixed Content
- Service Workers

Caching is not considered in this implementation, because server-side caching can differ per use case and the decision is left to the user. The implementation lacks a built-in cookie store. Cookies have to be manually extracted from the headers, for example, with `headers.get('Set-Cookie')`.

4.7.3 Miscellaneous

Other known limitations are:

- Aborting a request via `AbortSignal`.
- Synchronous implementation.
- Support for custom HTTP methods when using `HttpURLConnection`.
- Limited control over request headers when using `HttpURLConnection`.

For the `Request` class a `signal` property that is an instance of `AbortSignal` is missing in this implementation because the required `AbortSignal` type is a web type and not implemented for Graal.js.

As mentioned in Section 4.2.3, the networking implementation works synchronously, thus blocking the main thread. Ideally, the networking part of the fetching process would run in a separate thread. That would also allow a more sensible use of the `Promise` return type as discussed in Section 4.6.

The `HttpURLConnection` provides a lot of control, but comes with some restrictions. It does not accept custom request methods and only provides limited control over request headers. For example, the `Content-Length` header is calculated automatically and a user-specified value cannot overwrite it. A more detailed explanation of the `HttpURLConnection` limitations can be found in Section 4.2.3.

4.7.4 Non-standard Features

While it is not our goal to be fully compatible to node-fetch, we adapted some non-standard features from node-fetch. A full list of extensions that node-fetch contains can be found in Section 2.5.1. For this implementation we decided to implement the request option `follow`, as it provides a lot of utility for the user. Other extensions made by node-fetch are not implemented.

5 Testing

This chapter discusses the testing of the implementation. As mentioned in Section 2.5, the node-fetch implementation is well tested, both by an extensive suite of unit tests and actual usage of the package by millions of applications. For the purpose of testing this implementation, initially a subset of the already existing tests from node-fetch are utilized and extended when needed. Tests regarding non-standard extension made by the module, that are also not part of this implementation, are omitted.

Unit tests from node-fetch use a Node.js testing framework. While it would be possible to set up the tests by using Graal.js to execute Node.js code, the simpler approach is to rewrite tests and use the default JavaScript suite. Since networking related tests require a local testing server, they are implemented as JUnit tests and a Java `HttpServer` is used as testing server. Section 5.1 contains the network behaviour testing. Other non-networking tests for the `Request`, `Response` and `Headers` class are implemented in JavaScript, these tests are discussed in Section 5.2.

5.1 Networking Testing

The behavior of the `fetch()` function is tested by a suite of 73 unit test. 71 of the tests pass, while two tests are ignored. The ignored tests would test behavior that this implementation can't produce because of the internally used `HttpURLConnection` class. Both tests involve custom HTTP methods, which is a known limitation of this class, as discussed in Section 4.2.3.

To have a controlled testing environment, a local HTTP server is used for testing. The server is implemented in the `FetchTestServer` class and defines a number of endpoints to test against, for example, routes that test redirection behavior, returning appropriate status codes and headers, or a general purpose `/inspect` endpoint that returns the request it received as JSON response body, containing the requests url, method, headers, and body. Exemplary, Listing 5.1 shows how the tests are implemented. JavaScript code is passed to the `async` method, which creates an execution context and evaluates the given code in an `async` function. This test case specifically tests the behavior when a custom header is sent. The `/inspect` endpoint returns the request as JSON so it can be read from the `Response` object using `.json()` and logged with `console.log()`. Output of `async()` is a string containing the logs, which is then asserted to validate to test.

Listing 5.1: Custom header JUnit test

```
@Test
public void testSendRequestWithCustomHeader() {
    String out = async(
        "const res = await fetch('http://localhost:8080/inspect', {" +
        "   headers: { 'x-custom-header': 'abc' }" +
        "});" +
        "const result = await res.json();" +
        log("result.headers['x-custom-header']")
    );
    assertEquals("abc\n", out);
}
```

5.2 Fetch Types Testing

Different from the networking related tests, the tests for the `Request`, `Response`, and `Headers` class do not need a HTTP server for testing. They can be implemented purely in JavaScript. Exemplary, Listing 5.2 shows a test for the `Request` class that tests the behavior when `Request` objects wrap each other using the constructor.

Listing 5.2: Request JavaScript test

```
(function shouldSupportWrappingOtherRequest() {
    const r1 = new Request(baseURL, {
        method: 'POST',
    });
    const r2 = new Request(r1, {
        method: 'POST2',
    });
    assertSame(baseURL, r1.url);
    assertSame(baseURL, r2.url);
    assertSame('POST2', r2.method);
})();
```

6 Benchmarking and Evaluation

This chapter examines the performance of the implementation. A basic benchmarking script uses the Fetch API's `fetch()` function to perform a large number of plain `GET` requests to a local HTTP server and measure the time it takes to successfully read from the response of these requests. The following benchmarks using the same starting variables and setup are run by the benchmarking script:

- **B1**: Using the Graal.js JavaScript interpreter to run the built-in `fetch()` function provided by this implementation.
- **B2**: Using the GraalVM compiler to compile the benchmarking code that also runs the built-in `fetch()` provided by this implementation.
- **B3**: The node-fetch implementation is also benchmarked as reference. It is run using the Node.js runtime at version 16.

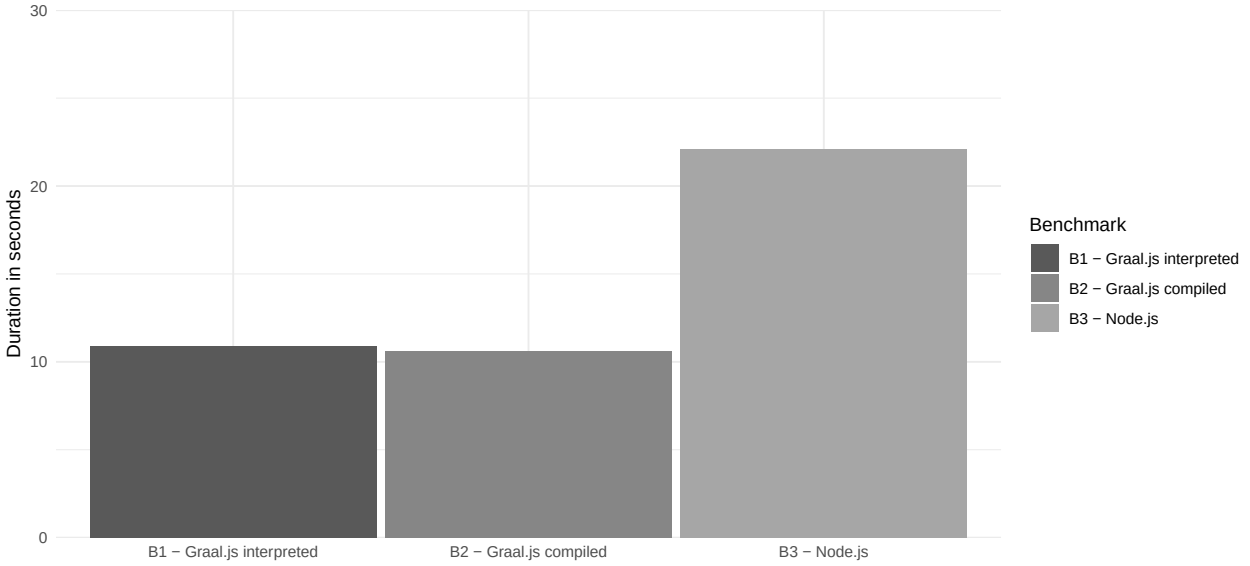
Listing 6.1: Benchmark setup

```
1  async function run(fetchFunc, N) {
2    const start = performance.now();
3    for (let i = 0; i < N; i++) {
4      const result = await fetchFunc(URL);
5      if (!result.ok) {
6        throw new Error('Request not successful');
7      }
8    }
9    return performance.now() - start;
10 }
```

Listing 6.1 shows the core of the benchmarking script. It receives a function and the amount of requests as arguments. The function is either the built-in `fetch()` function or imported from the node-fetch module, depending on which benchmark is actually executed. Then it performs requests and measures the time it took using the JavaScript internal high precision time API `Per-`

formance[1]. Multiple runs are executed per benchmark. For the setting of 10 total runs with 1000 requests each, these are the average times to perform the requests of one run:

- **B1**: `fetch()` with the interpreter: 10.9 seconds
- **B2**: `fetch()` with compiled JavaScript: 10.6 seconds
- **B3**: node-fetch using Node.js: 22.1 seconds



The compiled benchmark (**B2**) is barely faster than the interpreted code (**B1**). This is the expected result, since most of the time is spent in the internal Java implementation that handles network I/O and not in runtime code that the GraalVM optimizing compiler can improve significantly.

The results for node-fetch (**B3**) might seem implausibly bad. The reasons for this result could be (1) JavaScript concurrency, (2) runtime engine related aspects, or (3) differences in the underlying network implementation (e.g. `java.net.*` and `node:http`). However, that is just speculation and would require more specific testing. Because of those issues the performance comparison can only give preliminary results. They prove, however, that a native implementation of the Fetch API has potential to compete with, if not outperform, an implementation provided as an extension module.

[1] `https://developer.mozilla.org/en-US/docs/Web/API/Performance`

7 Conclusions

In this work we presented an initial implementation of the WHATWG Fetch API standard for Graal.js. The `Request`, `Response`, and `Headers` classes as well as the `fetch()` function are provided to the JavaScript interpreter. Networking functionality is implemented using the `HttpURLConnection` class from the standard library package `java.net`. To correctly integrate the Fetch API into the JavaScript implementation, we utilized the Truffle programming model and DSL.

Since the standard assumes a client or browser context and this implementation has a server side context, there inherently are parts of the specification that are infeasible for this implementation. Although the core functionality of the Fetch API is implemented, as mentioned above, we also describe a number of limitations and some missing features. Nonetheless, we were able to provide an implementation of the Fetch API, with all major classes and functionality to JavaScript applications in Graal.js.

Finally, we tested the validity and performance of this implementation. To verify completeness, unit test from the node-fetch implementation were adapted. Tests involving extended behavior implemented by node-fetch were ignored. To explore the performance of the implementation, we created a basic benchmark script. It compares the Fetch API performance of the Graal.js interpreter, compiled code using the GraalVM compiler and the node-fetch implementation using Node.js as runtime. The results of our performance comparison show that this native implementation has the potential to outperform currently existing approaches.

JOHANNES KEPLER
UNIVERSITY LINZ

Bibliography

[1] *ECMAScript® 2022 Language Specification*. URL: https://262.ecma-international.org.

[2] *ECMAScript® 2022 Language Specification*. URL: https://262.ecma-international.org/#sec-global-object.

[3] *ECMAScript® 2022 Language Specification*. URL: https://262.ecma-international.org/#sec-objects.

[4] *ECMAScript® 2022 Language Specification*. URL: https://262.ecma-international.org/#figure-1.

[5] *Fetch API - Living Standard*. URL: https://fetch.spec.whatwg.org/.

[6] *GraalVM - Polyglot Programming*. URL: https://www.graalvm.org/22.0/reference-manual/polyglot-programming/.

[7] Christian Humer et al. "A Domain-Specific Language for Building Self-Optimizing AST Interpreters". In: (2014), pp. 123–132. URL: https://doi.org/10.1145/2775053.2658776.

[8] *Redirections in HTTP - HTTP | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Redirections.

[9] *Referer - HTTP | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referer.

[10] Christian Wimmer and Thomas Würthinger. "Truffle: A Self-Optimizing Runtime System". In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. SPLASH '12. Tucson, Arizona, USA: Association for Computing Machinery, 2012, pp. 13–14. ISBN: 9781450315630. DOI: 10.1145/2384716.2384723. URL: https://doi.org/10.1145/2384716.2384723.

[11] Thomas Würthinger et al. "One VM to Rule Them All". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 187–204. ISBN: 9781450324724. DOI: 10.1145/2509578.2509581. URL: https://doi.org/10.1145/2509578.2509581.

JOHANNES KEPLER
UNIVERSITY LINZ

List of Figures

JOHANNES KEPLER
UNIVERSITY LINZ

List of Listings