Development of Robust SDKs for REST APIs in PHP

How to Effectively Develop, Maintain and Release REST API SDKs

Yaroslav Shestakov

BACHELOR’S THESIS
January 2020

Business Information Systems
Software Production

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

SHESTAKOV, YAROSLAV:
Development of robust SDKs for REST APIs in PHP
How to effectively develop, maintain and release REST API SDKs

Opinnäytetyö 40 sivua
Tammikuu 2020

Nykyaikaisessa talouselämässä ohjelmointirajapintojen (API) suosio on noussut sen ansiosta, että ne mahdollistavat digitaalisen ja automatisoidun vuorovaikutuksen organisaatioiden välillä. Usein, kun järjestelmä integroidaan APIn kanssa, kehitetään kaksi osaa: SDK (Software Development Kit) ja yhdistävä koodi. SDK on ohjelmistokehitystyökalujen kokoelma, jonka tavoitteena on yksinkertaistaa kehittäjän työtä APIn kanssa ja tehdä siitä intuitiivisempaa.

Tämä opinnäytetyö tutki, mallinsi ja dokumentoi toimintavarman SDK:n kehitysprosessia PHP-ohjelmointikielellä sekä osoitti, mitä positiivisia vaikutuksia hyvin suunnitellulla ja dokumentoidulla SDK:lla on integraation laatuun, kehitysnopeuteen ja kehittäjäkokemukseen.

Tutkimus toteutettiin kehittäjän näkökulmasta havainnoituna ohjelmistoprojektina, jossa kehitettiin Tampere Journey Planner APIa kattava SDK. Tutkimuksen teoreettisessa osassa paneuduttiin API-konseptiin, käytettyihin teknologioihin ja mallinnettiin SDK:n rakennetta. Käytännön osuudessa kuvattiin SDK:n kehitystä, dokumentointia, testausta, pakkaamista ja julkaisemista Composer-paketinhallintatyökalun avulla sekä selvitettiin, kuinka SDK:ta on käytetty uudessa projektissa.

Tulokset osoittavat, että teknisten yksityiskohtien sisällyttäminen SDK:hon sekä intuitiivisten abstraktioiden ja funktioiden tarjoaminen yksinkertaistavat integraatioiden kehitysprosesseja ja tekevät niistä siksi nopeampia. Koodin dokumentoinnin ansiosta automaattinen täydennys koodieditorissa on mahdollista, mikä puolestaan auttaa kehittämään integraatiota ja vähentää virheiden määrää. Ohjelmistopakettina julkaiseminen ja versiohallinta mahdollistavat ohjelmiston tehokkaan uudelleenkäyttämisen ja ylläpitämisen tiimien ja projektien välillä. Lisäksi liiketoiminnan näkökulmasta laadukas SDK voi tehostaa API-palvelun omaksumista markkinoille ja olla tärkeä menestyvän liiketoiminnan osatekijä.

Avainsanat: php, api, sdk, rest, integraatio

ABSTRACT

Tampere University of Applied Sciences
Business Information Systems
Software Production

SHESTAKOV, YAROSLAV:
Development of Robust SDKs for REST APIs in PHP
How to Effectively Develop, Maintain and Release REST API SDKs

Bachelor's thesis 40 pages
January 2020

In modern economy, APIs (Application Programming Interface) have gained popularity due to the fact that they enable digital automation of cross-organizational interactions. In order to integrate a system with an API, commonly two parts are required: an SDK (Software Development Kit) and a glue code. SDK is a set of software tools aiming to simplify a developer's work with an API and making the work more intuitive.

This thesis researches, wireframes and documents the process of robust SDK creation in PHP programming language, and demonstrates what positive impact a well-designed, documented SDK could have on the integration quality, speed and developer experience.

The research was carried out as a software project, where an SDK was developed for Tampere Journey Planner API and then analyzed through observation from an end-developer's perspective. The theoretical aspect covered the API concept, the technologies used and multiple SDK design approaches. The practical aspect covered programming, documenting, testing and packaging of an SDK, releasing it as a Composer dependency and using the package in a new project.

The results suggest that encapsulating technicalities of the API and providing intuitive abstractions and methods, significantly simplifies the process of integration development and therefore makes development faster. Code documentation enables autocompletion in code editors, which in turn helps with the integration development and reduces the number of errors. Releasing an SDK as a package and versioning it allows for better re-use and maintenance across teams and projects. Furthermore, from a business perspective a high-quality SDK can boost market adoption for an API service, potentially making it one of the key tools of a successful business.

Key words: php, api, sdk, rest, integration

CONTENTS

ABBREVIATIONS AND TERMS

API	Application Programming Interface
CLI	Command Line Interface (shell or terminal)
Composer	Package manager for PHP
Docker	Virtualization tool, enables containerized software
DTO	Data Transfer Object
FTP	File Transfer Protocol
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
JMS	Java Message Service (API protocol)
JSON	JavaScript Object Notation
Laravel	Web framework for PHP
PHP	Hypertext Preprocessor, Programming language
SDK	Software Development Kit
SemVer	Semantic Versioning convention
SMTP	Simple Mail Transfer Protocol
TDD	Test Driven Development
UI	User Interface
XML	Extensible Markup Language

1 INTRODUCTION

1.1 Rising popularity of APIs

The term API (Application Programming Interface) has gained much of popularity in the world of technology and business in the past decade and it happened for a reason. As of 2019, there were more than 4,5 billion Internet users, which amounted approximately to 58% of the global population (Internet World Stats 2019). Such a large number of digitally connected people has led to an emergence of a network, where participants can conveniently transact with each other on a global scale, and the potential of such a network is enormous.

What participants of the network needed was to find a way to communicate with each other utilizing the digitalization efficiently. Email services were extensively used as a means of exchanging information, and while it is still an important technology, it usually requires human interaction on both sides for sending and receiving information. For large scale businesses, this was still inefficient, because much manual work needed to be done. To facilitate a seamless communication and to automate processes, web APIs came into being.

Essentially, a web API represents a digital socket for information exchange virtually accessible to anyone with an Internet connection. An API can also represent a set of digital contracts, solidifying specific interactions between parties.

For instance, a shipping service provider would expose an API that accepts information about shipment and returns a PDF packing slip back to the customer. An E-commerce platform would expose APIs for order and product management. IoT devices expose APIs to exchange sensor data. The benefit of it is that the service becomes easily accessible, while at the same eliminating communication overhead as no human labor is required.

APIs are mostly used in data transfer and do not dictate what kind of user interface is used as compared to monolithic systems with built-in user interface. This approach gives flexibility to developers to design graphic user interfaces specifically optimized for certain business workflows.

Organizations started to see the value in this concept and to develop their own APIs. A research was conducted showing that the number of public APIs has increased from none to 22000 over the years 2005-2019 (figure 1). Given that there are many more private and unregistered APIs, the pattern is clear – the API space is flourishing. (Programmable Web 2019.)

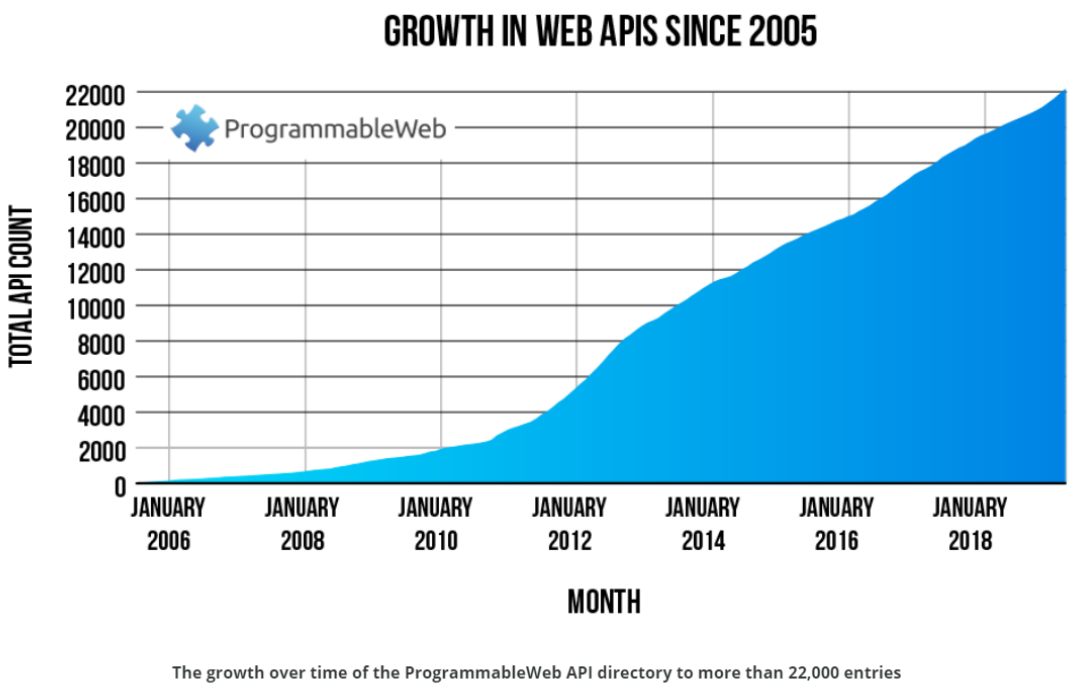


FIGURE 1. Rapid growth of public APIs since 2005 (Programmable Web 2019)

Another important aspect of APIs is enabling the possibility to create long and sophisticated value chains. Value chain is a set of activities or processes needed to be done in order to deliver a service or product. An application can consume multiple APIs in order to aggregate multiple services into one longer value chain.

Modern web shops are an example of such a value chain - they are typically integrated to payment and shipment services. It is convenient for customers, because the whole purchase process can be completed in one place. It is also convenient for merchants, because they can utilize ready-made solutions and focus on what is important for them. With some creativity, those services can be extended. The merchant could add an analytics service, make purchase orders directly from their web shop, or publish their products on an external platform such as Amazon. The business flow can be augmented with additional full-spectrum services, enabled by the API concept.

These value chains can be long linear chains (figure 2) or sophisticated networks. Some of the value chains serve a function of combining multiple services to create a single streamlined business flow, while others aggregate similar-purpose services under one API. Unifaun is an example of such an aggregator, uniting services of over 200 carriers under a single API (Unifaun). The benefit of such services is that only one API needs to be integrated to a system instead of many, in order to get access to a large variety of services.

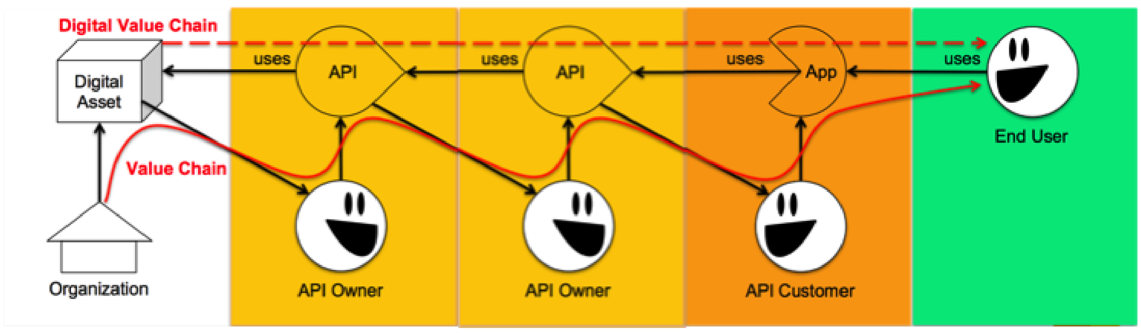


FIGURE 2. Scheme of a digital value chain (Paloviita S. 2018 on Medium.com)

The impact of APIs on business life has become so apparent that nowadays building an API is viewed as a strategic move to operate on the market. Such a move needs to be carefully planned, designed and implemented. (Woods, Brail, Jacobson 2011, ch. 2.)

Terms “API Economy” and “API Management” refer to the process of crafting and maintaining an API of an organization (figure 3).

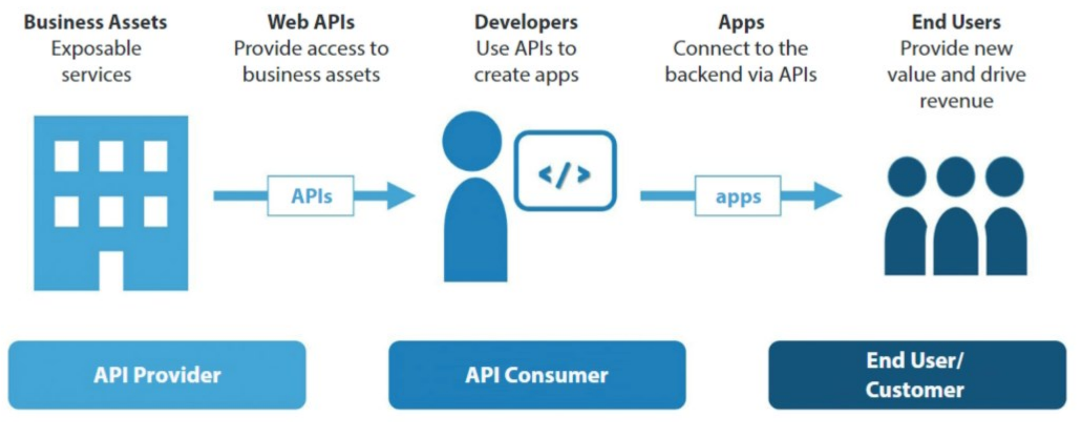


FIGURE 3. The API economy value chain (IBM, 2016)

To put the scale in perspective, in 2015 Salesforce.com generated 50% of its revenue through API, eBay and Expedia.com generated 60% and 90% respectively (Harvard Business Review 2015).

1.2 SDKs help to tap into API potential

APIs are a powerful tool, since they have a massive scalable impact and are gaining traction. APIs enable building entire service-augmented ecosystems and turning them into revenue streams. Therefore, as a developer or a business owner, it is worth learning more about APIs and learn how to integrate them into systems in order to tap into their potential.

To simplify integration process to API services, Software Development Kits (SDK) can be created. They resemble a set of software tools, intuitive abstractions and functions written in a specific programming language. SDKs can be created by an API service provider or unaffiliated developers, who intend to consume the API.

Depending on the market or platform conditions, multiple SDKs might need to be developed, each for separate platform or programming language. Not all API service providers release SDKs for their APIs due to resource limitations. When this is the case, other parties consuming the APIs develop and maintain their own SDKs. Some of them are Open Source, some available on marketplaces and most of them are kept private. In any case, there are significantly more APIs out there than SDKs implementing them, therefore making SDK development an important skill in software development.

Although SDKs bring API functionality closer to end-developers, building them can be a labor-costly process. Therefore, one needs to ensure that the best type of SDK is chosen when developing an integration.

2 AIMS

This thesis aims at researching and formulating a consistent method of robust SDK creation in PHP programming language in order to efficiently integrate REST API services. Such an SDK has a clear structure, automated tests and documentation. The SDK is intuitive to use and intended to significantly speed up integration process.

The thesis consists of theoretical and practical parts. Theoretical part covers aspects of API concepts and SDK structure in general. Practical part demonstrates step-by-step the process of SDK creation, release and re-use as a package.

PHP is the programming language of choice, because as of 2019, it powers 78,9% of all sites on the Internet (W3Techs 2019).

It is assumed that readers have an IT background and have some knowledge of modern PHP programming in order to understand the subject.

The SDK architecture approaches described in this work can be especially beneficial for developers, who work on API integration tasks to improve efficiency and code quality. Teams can adopt this approach in order to better understand each other’s code patterns, and to maintain and re-use modular parts using Composer package manager. Business owners of API-augmented services may find an insight how publishing a well-designed SDK can be a strategic move to boost market adoption.

3 THEORY BEHIND API

In order to integrate an API into a system, various important information needs to be gathered first. API service providers typically offer a full documentation with technical information and examples on API endpoint usage. This part covers theoretical aspects of APIs and their common types.

3.1 Definition of API

An API is a software interface or communication protocol, that defines how computer applications should communicate with each other over a network. An API defines a contract in terms of protocol, data format and endpoint. (Brajesh 2017, ch. 1.)

The main difference of APIs from web sites is that while web sites publish information, which is consumed by user, they do not have contracts. The site's layout, structure and content may be changed at any time without prior notice to users. An API, on the other hand, resembles a contract, which cannot be changed after the release because many external applications may rely on it. (Brajesh 2017, ch. 1.)

3.2 Characteristics of API

Essential characteristics, which should be specified by API providers:

- Functionality description (business logic)
- Location of the API (typically URL endpoint)
- Input and output parameters (names, data types and formats)
- Service-level agreements (SLA) such as response time, throughput, availability
- Technical requirements about the rate limits
- Legal constraints, such as licensing terms and fees
- Documentation

(Brajesh 2017, ch. 1.)
All of these characteristics are useful for understanding the nature of API and for developing Software Development Kits.

APIs can be categorized by exposure (figure 4). APIs can be private: used internally or shared to partners, and public – anyone with Internet connection can have access.

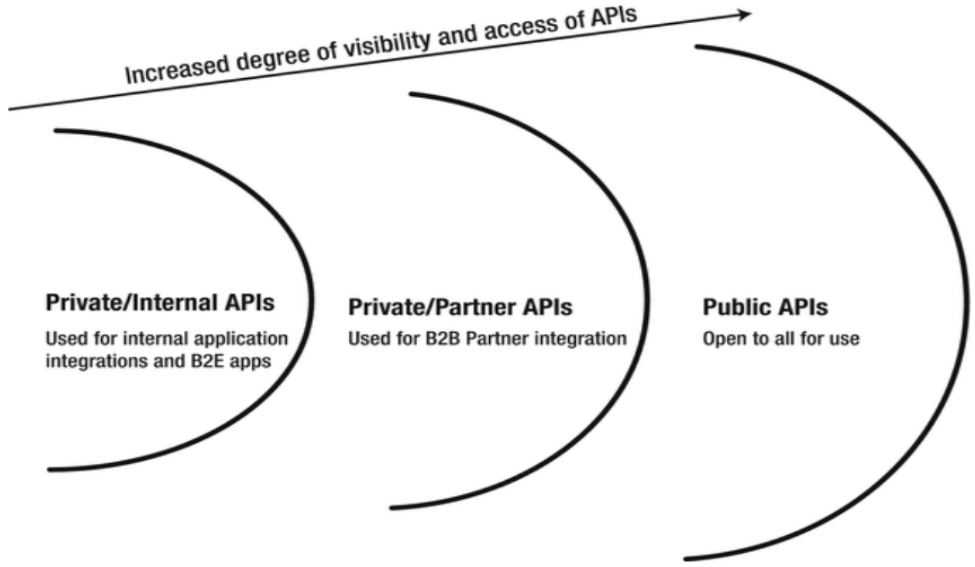
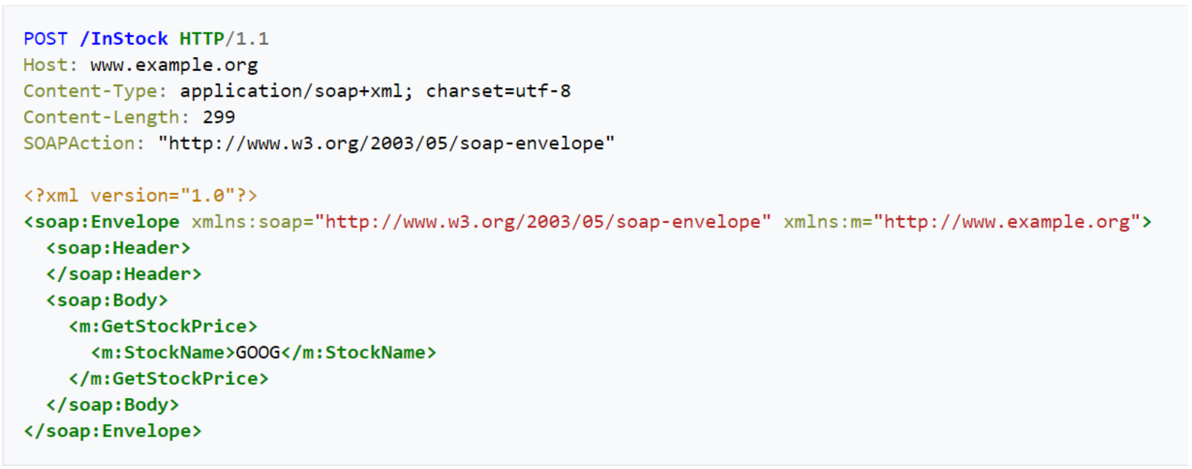

Increased degree of visibility and access of APIs
Private/Internal APIs
Used for internal application integrations and B2E apps
Private/Partner APIs
Used for B2B Partner integration
Public APIs
Open to all for use

FIGURE 4. Types of API by publicity/exposure (Brajesh 2017)

3.2 API protocol: SOAP

SOAP (Simple Object Access Protocol) was designed in 1998. SOAP web services usually use HTTP as a transport protocol, although they can operate over JMS/FTP/SMTP protocol. A SOAP message structure (figure 5) consists of a SOAP envelope, containing SOAP headers and the body. The body contains the actual payload and is based on XML format. As a standard, SOAP is mature and is used in many systems, although it does not utilize many features of HTTP protocol. (Brajesh 2017, ch. 1.)



```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xmlns:m="http://www.example.org">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice>
      <m:StockName>GOOG</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

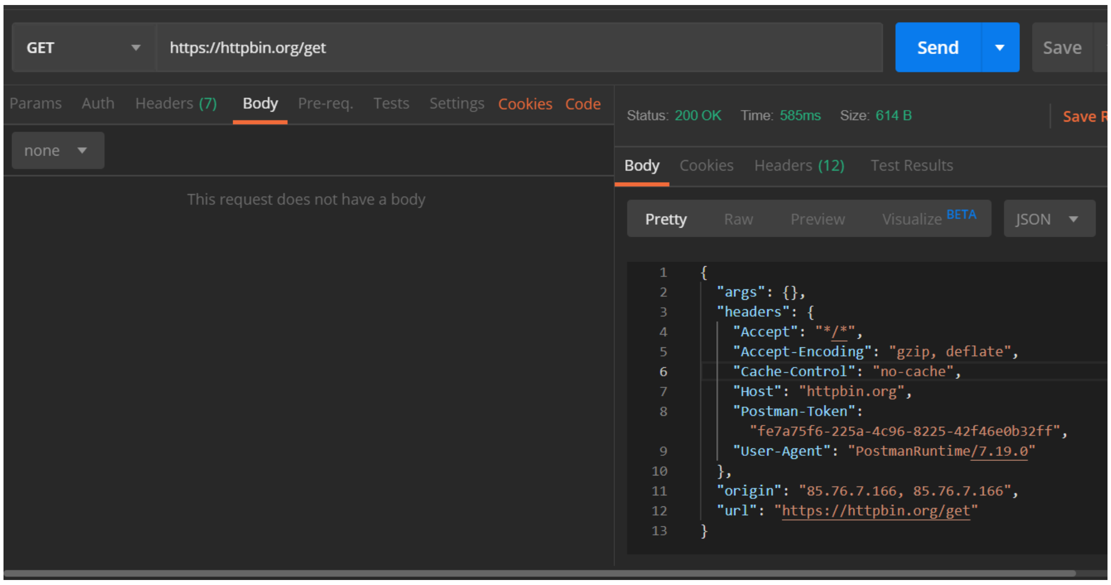
FIGURE 5. Example of an outgoing SOAP message (Wikipedia)

SOAP API is usually strictly typed and validated, thus diminishing amount of errors. However, verbosity of the protocol and slow parsing speed of XML stand as disadvantages and adopting this protocol requires a sufficiently fast network bandwidth and a fair amount of computing power for optimal performance. Due to the performance requirements, SOAP is becoming more obsolete and is being replaced by more agile counterparts. (Wikipedia.)

3.3 API protocol: REST

REST (Representational State Transfer) was first defined in 2000 and can exchange different types of data including XML and JSON (figure 6). REST extensively utilizes HTTP features in order to function. For instance, using a full range of HTTP verbs (GET, POST, PUT, PATCH, DELETE), header-based authentication, cacheability indication in responses and others. It was widely adopted because of its simplicity and performance, can be easily implemented and does not require specific software. (Brajesh 2017, ch. 1.) URL endpoints usually represent a data entity, that a client wants to interact with.

Since it is relatively easy to create an API using REST architecture and it is a popular approach at the time of writing, this thesis covers building SDKs for this API type.

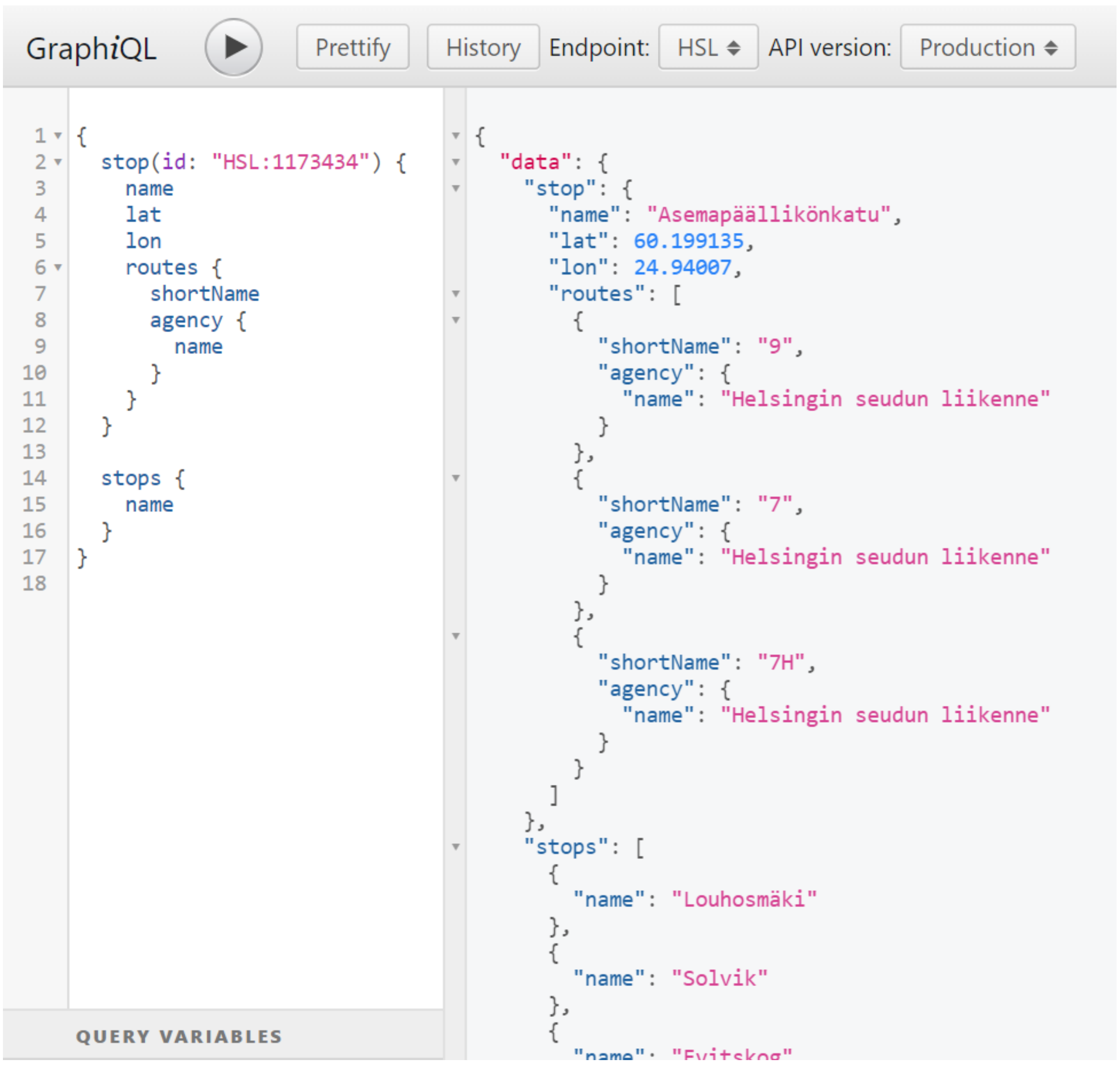


```
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Cache-Control": "no-cache",
    "Host": "httpbin.org",
    "Postman-Token":
      "fe7a75f6-225a-4c96-8225-42f46e0b32ff",
    "User-Agent": "PostmanRuntime/7.19.0"
  },
  "origin": "85.76.7.166, 85.76.7.166",
  "url": "https://httpbin.org/get"
}
```

FIGURE 6. Example of JSON REST GET request and response

3.4 API protocol: GraphQL

GraphQL was internally developed by Facebook in 2012 and publicly released in 2015. This is the newest API protocol, which is not yet widely adapted but getting a lot of attention - it allows to fetch the exact needed data in one HTTP request (figure 7). This optimizes server and network performance as less data needs to be computed and transferred. (Wikipedia.)



```
{
  stop(id: "HSL:1173434") {
    name
    lat
    lon
    routes {
      shortName
      agency {
        name
      }
    }
  }

  stops {
    name
  }
}
```

```
{
  "data": {
    "stop": {
      "name": "Asemapäällikönkatu",
      "lat": 60.199135,
      "lon": 24.94007,
      "routes": [
        {
          "shortName": "9",
          "agency": {
            "name": "Helsingin seudun liikenne"
          }
        },
        {
          "shortName": "7",
          "agency": {
            "name": "Helsingin seudun liikenne"
          }
        },
        {
          "shortName": "7H",
          "agency": {
            "name": "Helsingin seudun liikenne"
          }
        }
      ]
    },
    "stops": [
      {
        "name": "Louhosmäki"
      },
      {
        "name": "Solvik"
      },
      {
        "name": "Evitskog"
```

FIGURE 7. Example of GraphQL nested request and response from HSL API

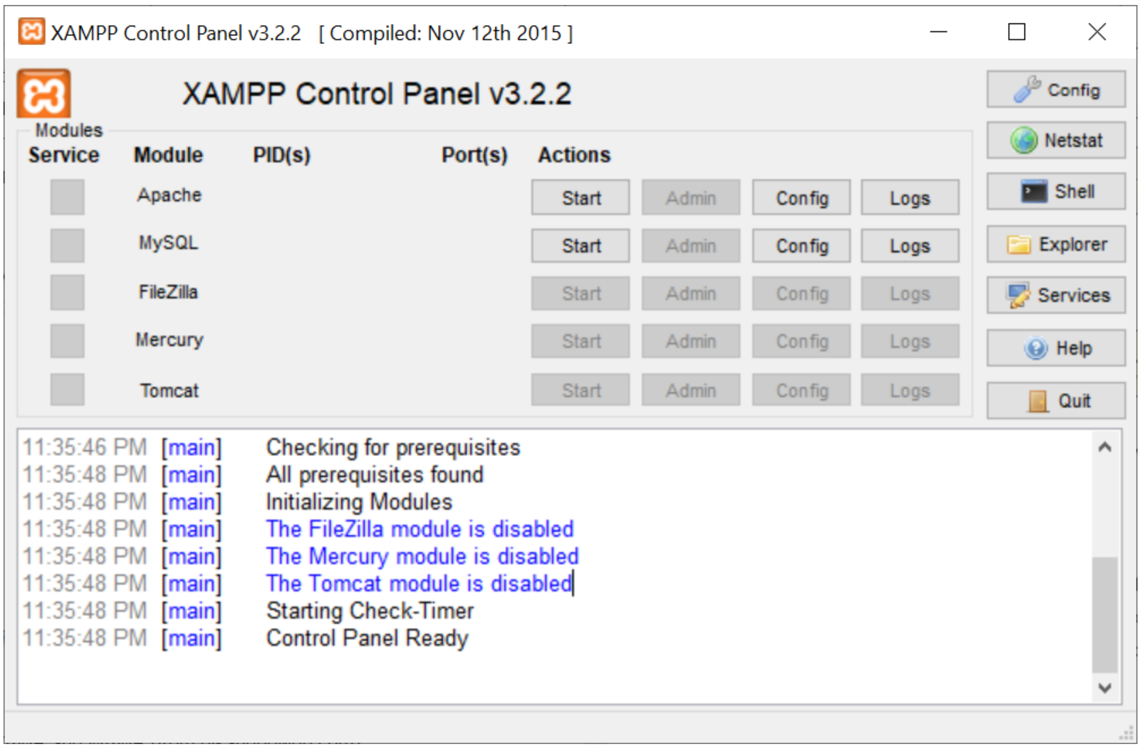
4 DEVELOPMENT TOOLS FOR SDK CREATION

This chapter covers a set of basic tools required to develop a REST API SDK.

4.1 Choosing the right development environment

Local development stack

There are multiple methods for running a PHP application locally. The first method involves installing a pre-packaged server software on the computer and hosting a website locally. Ready-made solutions such as LAMP, WAMP and MAMP are available for download and are easy to install. Abbreviations *AMP stand for the stack of technologies Apache, MySQL and PHP, and the first letter indicates an operating system Linux, Windows and MacOS). XAMPP is another version of such an application, X stands for cross-platform approach. Figure 8 displays XAMPP control panel with stack module controls.



```
11:35:46 PM  [main]   Checking for prerequisites
11:35:48 PM  [main]   All prerequisites found
11:35:48 PM  [main]   Initializing Modules
11:35:48 PM  [main]   The FileZilla module is disabled
11:35:48 PM  [main]   The Mercury module is disabled
11:35:48 PM  [main]   The Tomcat module is disabled
11:35:48 PM  [main]   Starting Check-Timer
11:35:48 PM  [main]   Control Panel Ready
```

FIGURE 8. XAMPP control panel

This development environment approach is the most performant, because computer's operating system communicates directly to the stack software. However, it lacks in flexibility in case of need to develop other projects having different version requirements, as the stack parts cannot be easily updated.

Vagrant

Vagrant is a software that utilizes VirtualBox and is able to create a virtual operating system on a computer. A provision script containing installation and configuration is fed to the virtual operating system. Multiple virtual machines can coexist on the same hard drive. (Lambert, Aulakh & Rickard 2015.)

This approach is more advanced than a local development stack, as it allows to create an isolated development environment for each separate project. Sharing provision script as part of project code ensures that every member in the team has up-to-date development environment. The drawback of using the virtual machine approach includes a diminished performance due to increased communication complexity in the system and a requirement to have enough storage on the computer. Figure 9 demonstrates a common workflow of Vagrant.

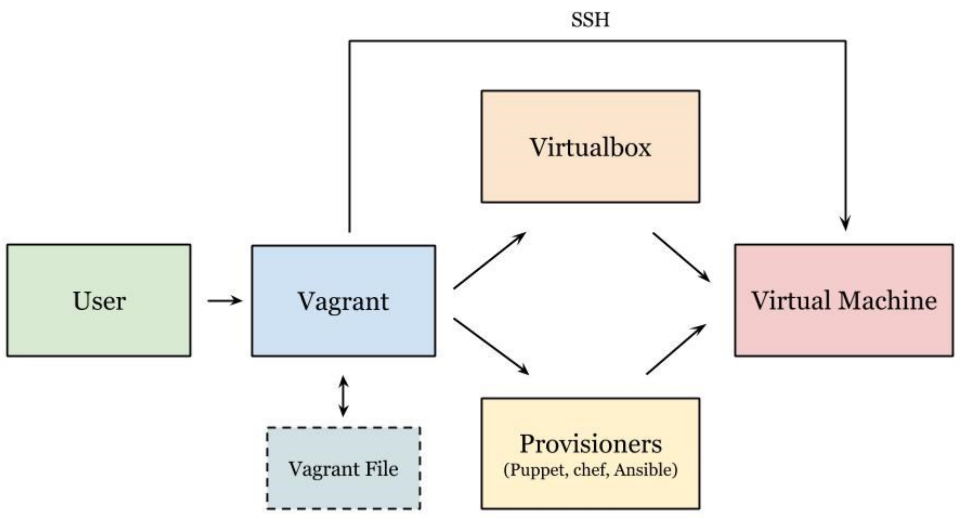


FIGURE 9. Scheme of a Vagrant workflow (Lambert, Aulakh & Rickard 2015)

Docker

The third method involves using Docker, which is similar to the virtual machine approach in terms of flexibility, however different in infrastructure (figure 10). Docker creates a network of isolated software containers that are able to communicate with each other and do not require a virtual operating system. In current technology stack there would be interconnected containers for PHP, MySQL and Apache modules.

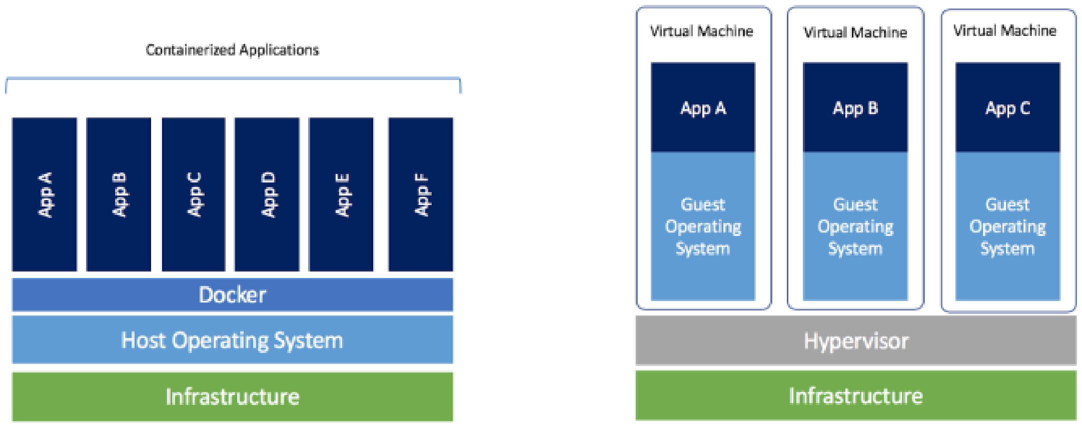
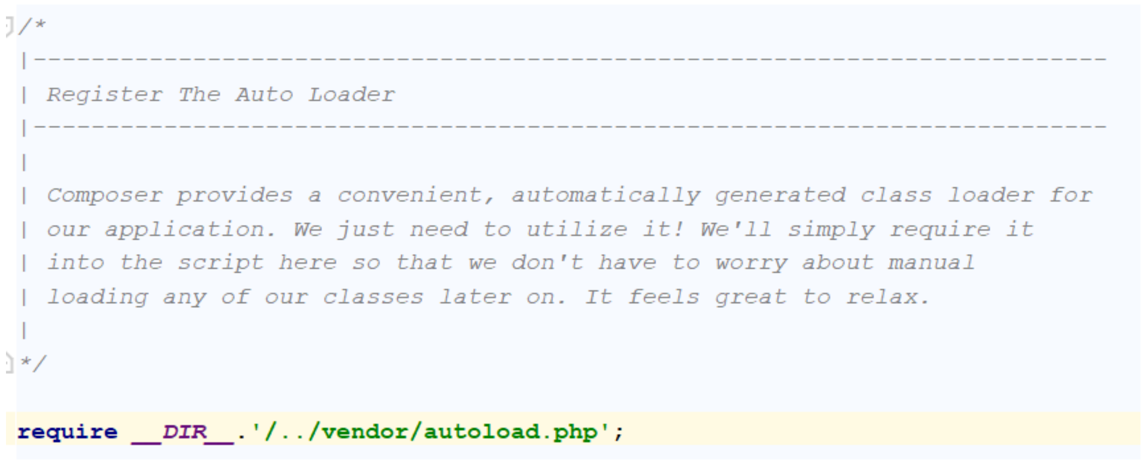


FIGURE 10. Docker vs Virtual Machines (from Docker.com)

The Docker approach offers the best flexibility for teamwork - containers are light-weight as data snapshots contain only app-related information and container configuration can be precisely tweaked.

4.2 Composer - PHP package manager

PHP world has its own package manager, Composer. It is a great addition to any PHP project, as it allows installing ready-to-use packages as well as provides class autoloading and an array of other useful features to augment development process. Autoloading is especially important for efficient development as PHP class files need to be included – and Composer does it automatically on demand with only one line of code (figure 11).



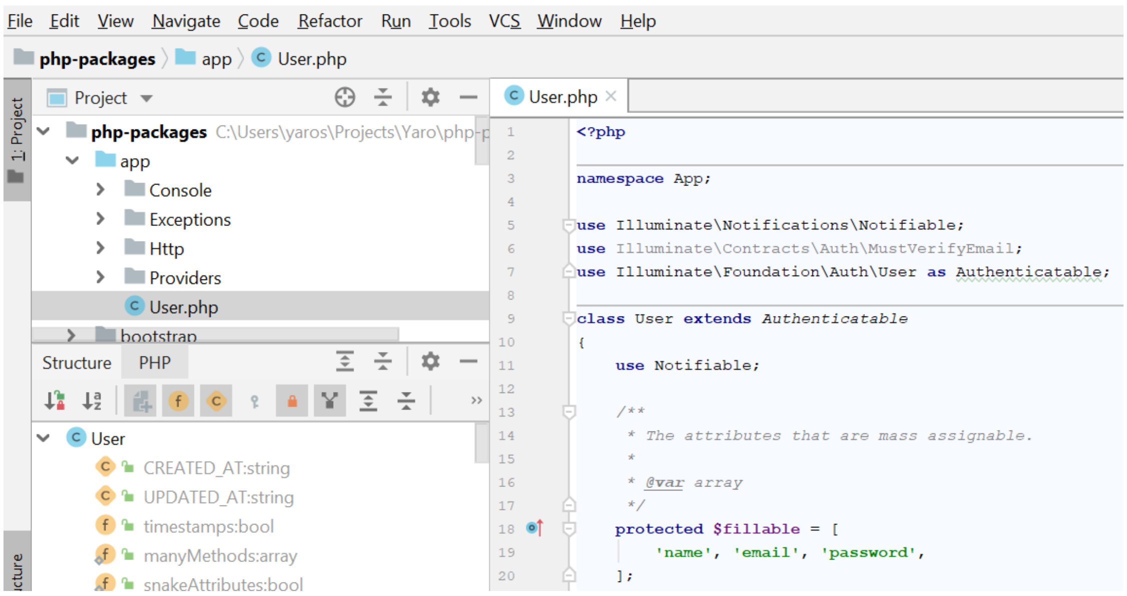
```
/*
|--------------------------------------------------------------------------
| Register The Auto Loader
|--------------------------------------------------------------------------
|
| Composer provides a convenient, automatically generated class loader for
| our application. We just need to utilize it! We'll simply require it
| into the script here so that we don't have to worry about manual
| loading any of our classes later on. It feels great to relax.
|
*/

require __DIR__.'/../vendor/autoload.php';
```

FIGURE 11. Enabling PHP class autoloading via Composer

4.3 IDE to make programming interactive

No specific code editor is required for PHP programming as PHP files contain plain text and can be modified with any text editor. However, for professional development it is wise to utilize professional tools that make navigation and code inspection easier and augment the development with auto-completion. With the help of an IDE, the development process becomes more efficient and interactive (Wikipedia).

PHPStorm was the IDE used in the research, being one of the most robust tools for professional PHP programming at the time of writing. It understands parses the code in real time, has a plugin system and provides a powerful navigation system. Figure 12 shows an example of PHPStorm's graphical user interface.



```php
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'name', 'email', 'password',
    ];
```

FIGURE 12. PHPStorm GUI

4.4 Version control

Working in a team in software development requires a version control system in order to keep simultaneous work organized. A history of changes tracked by a version control makes tracing issues easier. Git allows working on separate branches and merge them. Example of such a workflow is called Gitflow and demonstrated in figure 13.

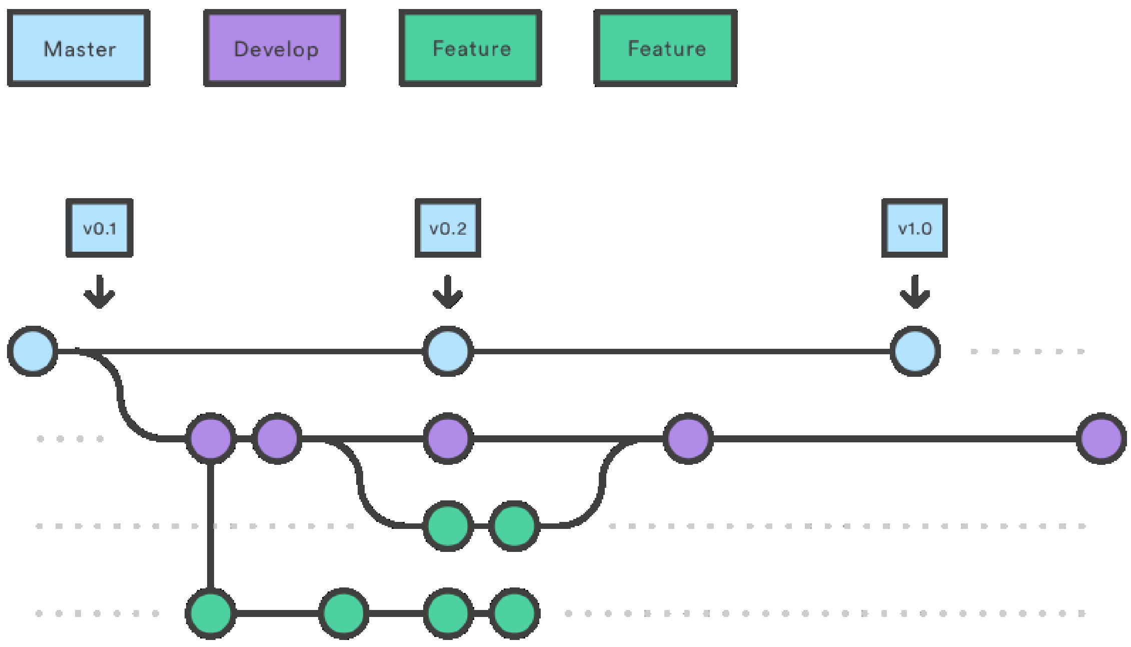


FIGURE 13. Example of Gitflow (from Atlassian.com)

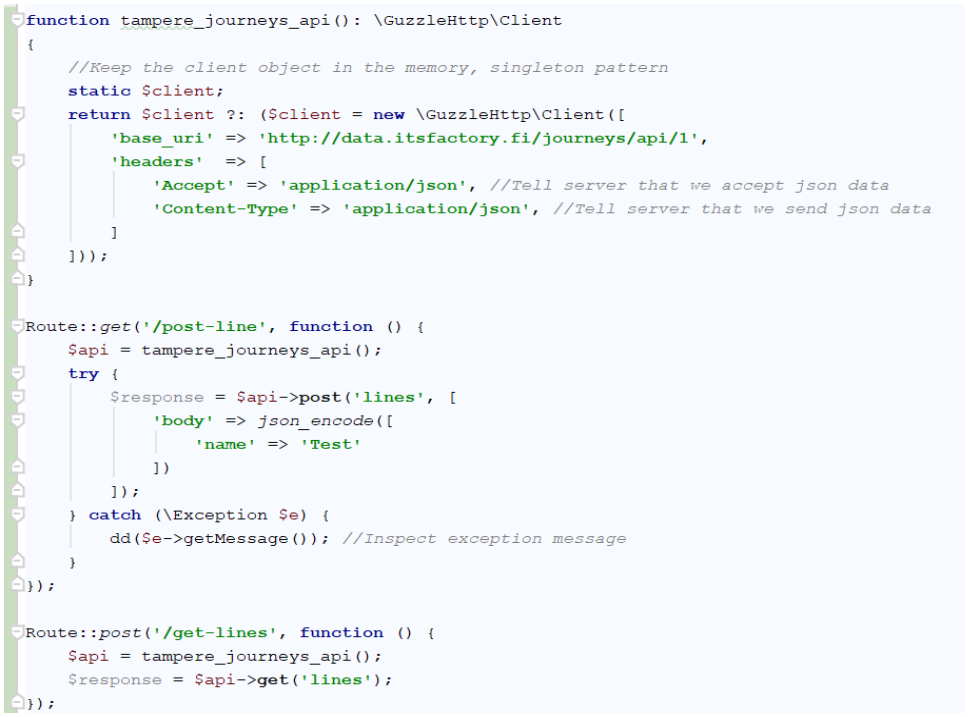
Git is a very robust and perhaps the most popular version control system. Package management service Packagist.org supports integration with GitHub (Git repository service) and therefore Git is the version control system of choice.

5 REST API SDK DESIGN APPROACHES

API integrations may be implemented in various ways, depending on technical aspects of API, team's needs and resources. This chapter covers aspects of API integration design and suggests a number of design approaches. A decision to choose a specific design should be based on multiple factors such as API schema complexity, need to use the SDK across teams and applications, workload availability and the purpose of the SDK (internal / public).

5.1 Simple straightforward integration using HTTP Client

Since REST API is strongly bound to HTTP protocol, the bare minimum for writing REST API integrations would be creating a **HTTP Client** abstraction (figure 14), which exposes methods for making HTTP requests. The data received from the API will be typically an **array** or **stdClass** (simple object) in PHP. In order to keep the code in a DRY (Don't Repeat Yourself) fashion, the **HTTP Client** would accept essential parameters, such as base URL and authentication credentials. This way the object can be readily passed to other functions for reuse. In PHP world, there is a composer package guzzle/guzzle, which provides a configurable HTTP Client which well-suited for this purpose.



```
function tampere_journeys_api(): \GuzzleHttp\Client
{
    //Keep the client object in the memory, singleton pattern
    static $client;
    return $client ?: ($client = new \GuzzleHttp\Client([
        'base_uri' => 'http://data.itsfactory.fi/journeys/api/1',
        'headers'  => [
            'Accept' => 'application/json', //Tell server that we accept json data
            'Content-Type' => 'application/json', //Tell server that we send json data
        ]
    ]));
}

Route::get('/post-line', function () {
    $api = tampere_journeys_api();
    try {
        $response = $api->post('lines', [
            'body' => json_encode([
                'name' => 'Test'
            ])
        ]);
    } catch (\Exception $e) {
        dd($e->getMessage()); //Inspect exception message
    }
});

Route::post('/get-lines', function () {
    $api = tampere_journeys_api();
    $response = $api->get('lines');
});
```

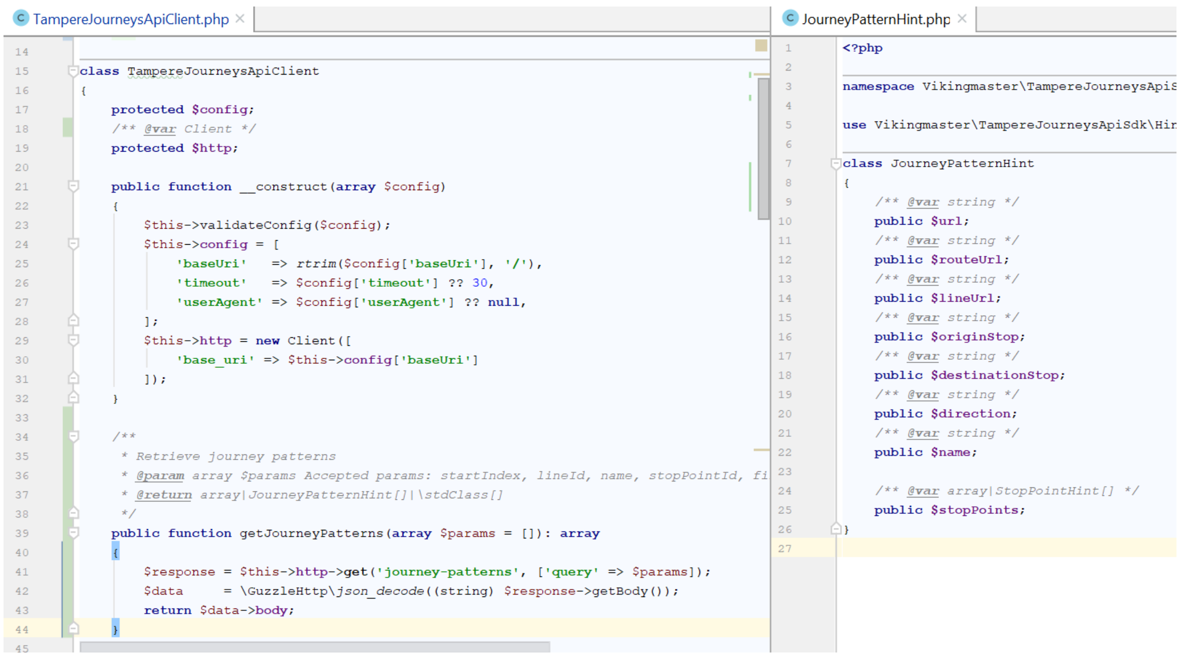
FIGURE 14. Example of straightforward integration with HTTP Client

The simplicity of this approach lies in the fact that the Client object does not contain any other functionality than making HTTP functions accessible. Therefore, it is fast and easy to implement at the expense of lacking abstractions and documentation. This approach is favorable where interactions with a specific API are minimal and straightforward.

5.2 Minimalistic SDK with type-hinted response data

When API response schema has a sophisticated structure and the API is used extensively across application, it is advisable to provide abstractions. Modern IDEs understand the programming code and provide autocompletion for methods and properties. The autocompletion allows developers to quickly modify the code without referring to original documentation and to avoid mistakes.

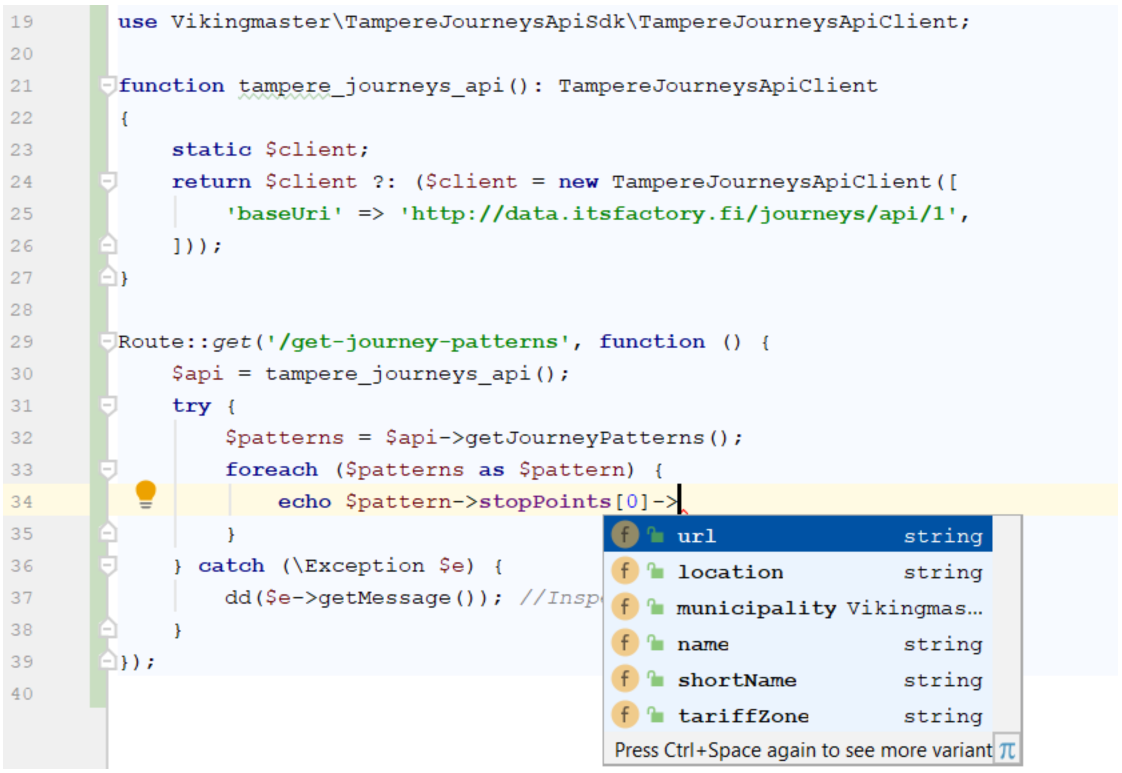
This integration model contains **API Client** abstraction (figure 15), which contains **HTTP Client** and exposes intuitive methods for receiving type-hinted API data. Data returned by the exposed API methods can be type-hinted with simple **Hint** classes in order to make IDE aware of the API schema to offer autocompletion for a developer as demonstrated in figure 16. Type-hinting on documentation level rather than code level allows saving time on integration development as well as a fair amount of computing power.



```php

class TampereJourneysApiClient
{
    protected $config;
    /** @var Client */
    protected $http;

    public function __construct(array $config)
    {
        $this->validateConfig($config);
        $this->config = [
            'baseUri'   => rtrim($config['baseUri'], '/'),
            'timeout'   => $config['timeout'] ?? 30,
            'userAgent' => $config['userAgent'] ?? null,
        ];
        $this->http = new Client([
            'base_uri' => $this->config['baseUri']
        ]);
    }

    /**
     * Retrieve journey patterns
     * @param array $params Accepted params: startIndex, lineId, name, stopPointId, fi
     * @return array|JourneyPatternHint[]|\stdClass[]
     */
    public function getJourneyPatterns(array $params = []): array
    {
        $response = $this->http->get('journey-patterns', ['query' => $params]);
        $data     = \GuzzleHttp\json_decode((string) $response->getBody());
        return $data->body;
    }

```

```php
<?php

namespace Vikingmaster\TampereJourneysApiS

use Vikingmaster\TampereJourneysApiSdk\Hir

class JourneyPatternHint
{
    /** @var string */
    public $url;
    /** @var string */
    public $routeUrl;
    /** @var string */
    public $lineUrl;
    /** @var string */
    public $originStop;
    /** @var string */
    public $destinationStop;
    /** @var string */
    public $direction;
    /** @var string */
    public $name;

    /** @var array|StopPointHint[] */
    public $stopPoints;
}

```

FIGURE 15. Simple API Client providing type-hinted response data



```
use Vikingmaster\TampereJourneysApiSdk\TampereJourneysApiClient;

function tampere_journeys_api(): TampereJourneysApiClient
{
    static $client;
    return $client ?: ($client = new TampereJourneysApiClient([
        'baseUri' => 'http://data.itsfactory.fi/journeys/api/1',
    ]));
}

Route::get('/get-journey-patterns', function () {
    $api = tampere_journeys_api();
    try {
        $patterns = $api->getJourneyPatterns();
        foreach ($patterns as $pattern) {
            echo $pattern->stopPoints[0]->
        }
    } catch (\Exception $e) {
        dd($e->getMessage()); //Insp
    }
});
```

```
url                 string
location            string
municipality Vikingmas…
name                string
shortName           string
tariffZone          string
Press Ctrl+Space again to see more variant
```

FIGURE 16. Type-hinting makes PHPStorm aware of the API schema

Minimalistic SDK approach allows to create a REST API integration relatively easily as it consists mostly from **ApiClient** and **Hints** (simple classes with public properties). This approach is the most optimal for integrations where speed of development, API schema awareness and reliance on read-only operations are required.

5.3 Advanced REST API SDK architecture

When an API contains many different endpoints, nested data structures, accepts nested input and returns a large variety of errors, there needs to be an advanced REST API SDK architecture, which is able to tackle the complexity of such an API. These are requirements for SDK: it must encapsulate technical details of message transportation, while providing integration API documentation in the IDE.

Although file structure becomes more complex, the architecture utilizes intuitive abstractions which are related to HTTP and REST. The whole workflow starts with just one entry – **ApiClient**, which has all the needed methods to access full spectrum of API functions. **ApiException** classes are purposefully designed for easy troubleshooting - after a request is made, it is possible to access all objects involved in the API call: **ApiClient**, **Request**, **Response** and distinguish between API-related and server/network errors. This architecture allows developer to be in control at every stage of interaction with an API.

In a nutshell, the structure would look like this:

- **API Client**, entry point
  - Configurable via constructor method
  - Holds an **HTTP Client** object, encapsulates transport logic
  - Exposes methods for creation of various **Request** objects
  - Expose **DTO Factory** singleton instance (optional)
- **Request** objects (Abstraction for API requests)
  - Contains fluent **set**-methods, which allow method-chaining
  - Exposes **getHeaders, getBody**, **getParams**, **getEndpoint** methods
  - Exposes **send** method, to return corresponding **Response** object
  - Exposes **getApiClient** method for back-tracing
- **Response** objects (Abstraction for API responses)
  - Exposes **get**-methods
  - Exposes **getBody**, **getHeaders** methods
- **DTO**s (Data Transfer Object)
  - Data containers with **get-se**t methods for nested data
  - Can be part of **Request** and **Response** objects' nested data
- **DTO Factory**
  - Exposes methods for creating API-schema related **DTO** objects
  - Allows to avoid namespace imports and **new** keyword while creating objects
  - Provides documentation for data types
- **ApiException**
  - Allows retrieving **Request** and **Response** for back-tracing
  - Exposes **getApiError** method, which allows to distinguish between API and non-API errors (for networking and server errors the **API Error** is empty)

UML scheme for Tampere Journey API SDK is displayed on figure 17.

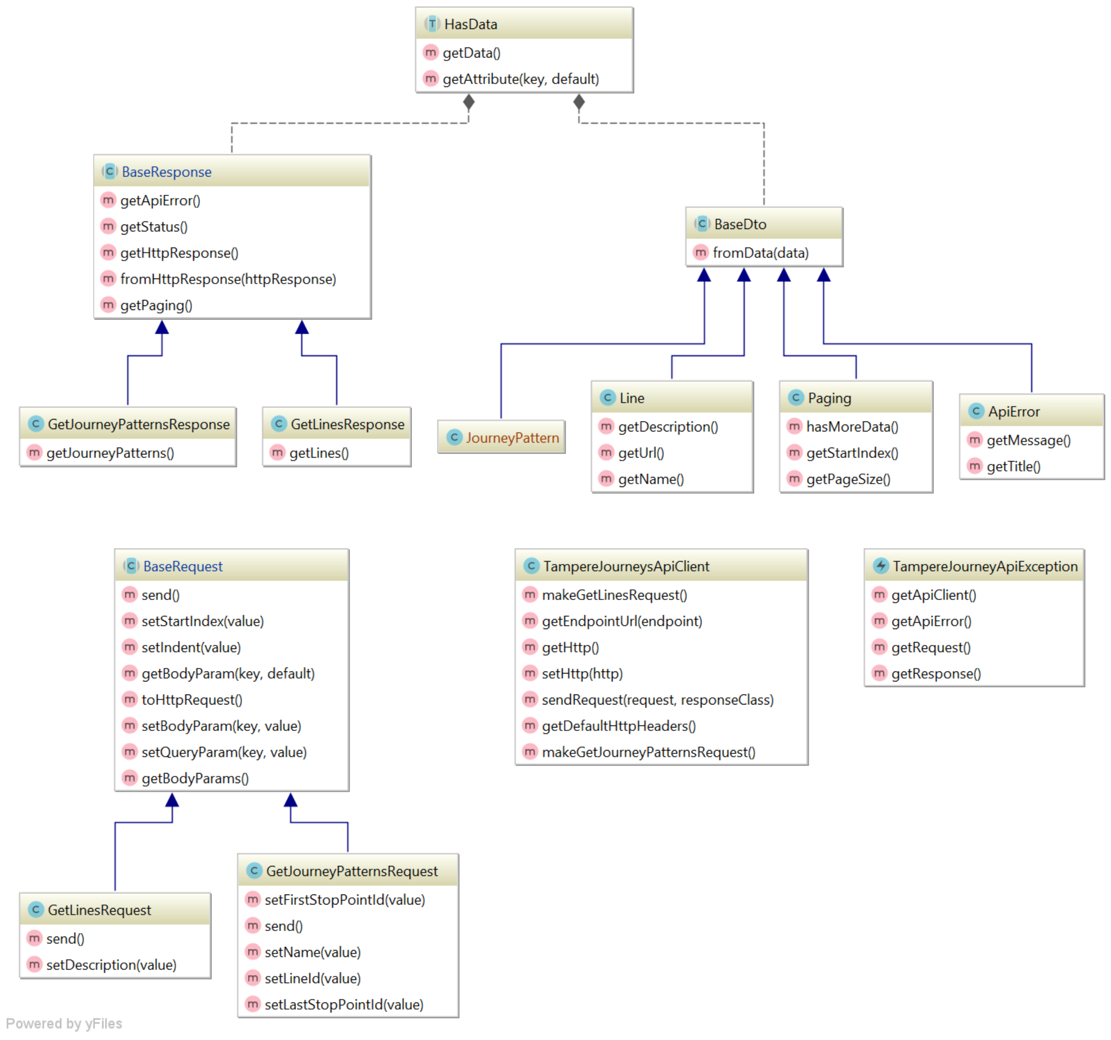


FIGURE 17. UML example of advanced REST API SDK architecture

This approach is the more labor-costly than simpler models, as it requires multiple abstractions and test coverage. However, if a team rigorously uses an API across one or multiple applications or plans to release the SDK into public, this type of SDK is the most optimal, as it brings the API very close to a developer.

6 DEVELOPING SDK FOR TAMPERE JOURNEYS API

This chapter covers practical aspects of integration development and demonstrates how to implement an example SDK for Tampere Journeys API from scratch. For the sake of demonstration, only a few API methods are the covered by the SDK. The package is covered with Unit tests and released to packagist.org service, where it is publicly available. After that the package is tested from end developer's perspective in a new project.

Full source code:
https://github.com/vikingmaster/tampere-journeys-api-sdk
Composer package:
https://packagist.org/packages/vikingmaster/tampere-journeys-api-sdk

6.1 Setting up a package development project

Framework installation

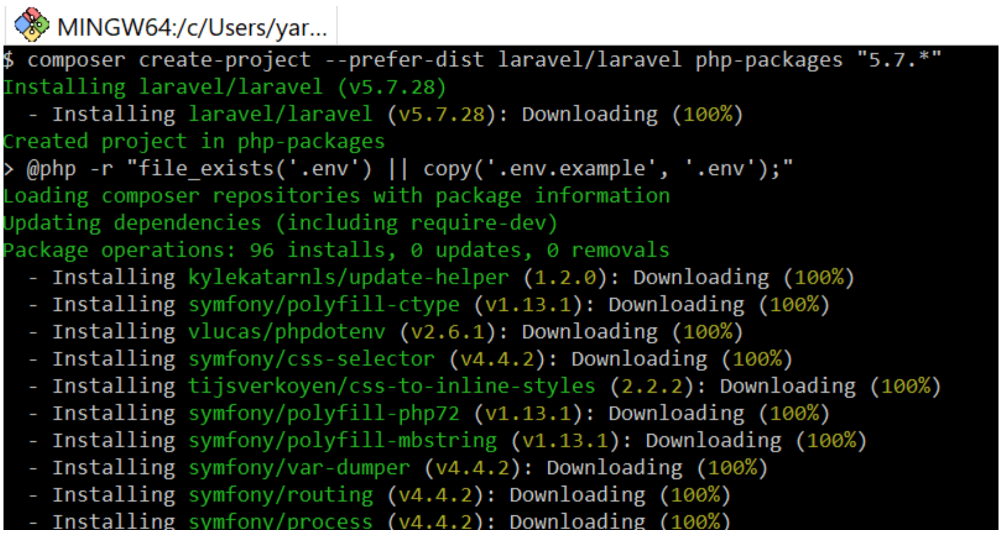
Although no framework is required to create a software package in PHP, a framework can simplify testing software parts. The choice of framework could be also made depending on compatibility requirements.

There are many frameworks available for PHP, for example Symfony, PHPCake, CodeIgniter and Laravel. Due to my extensive experience with Laravel framework as well as it being the most robust and popular at the time of writing (Clariontech 2019), Laravel is the framework of choice.

To set up a new Laravel-based project, the following command is executed:

```
composer create-project --prefer-dist laravel/laravel php-packages "5.7.*"
```

The command will create a new project directory **php-packages**, setup the project structure and install packages into **/vendor** directory as shown in figure 18. In Laravel, **/public** directory is the web folder, therefore it should be configured in a server software (e.g. Apache) as a web root. Alternative to a full server setup, Laravel provides a CLI command which can be used to instantly host an application locally: `php artisan serve`.



MINGW64:/c/Users/yar...

```
$ composer create-project --prefer-dist laravel/laravel php-packages "5.7.*"
Installing laravel/laravel (v5.7.28)
  - Installing laravel/laravel (v5.7.28): Downloading (100%)
Created project in php-packages
> @php -r "file_exists('.env') || copy('.env.example', '.env');"
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 96 installs, 0 updates, 0 removals
  - Installing kylekatarnls/update-helper (1.2.0): Downloading (100%)
  - Installing symfony/polyfill-ctype (v1.13.1): Downloading (100%)
  - Installing vlucas/phpdotenv (v2.6.1): Downloading (100%)
  - Installing symfony/css-selector (v4.4.2): Downloading (100%)
  - Installing tijsverkoyen/css-to-inline-styles (2.2.2): Downloading (100%)
  - Installing symfony/polyfill-php72 (v1.13.1): Downloading (100%)
  - Installing symfony/polyfill-mbstring (v1.13.1): Downloading (100%)
  - Installing symfony/var-dumper (v4.4.2): Downloading (100%)
  - Installing symfony/routing (v4.4.2): Downloading (100%)
  - Installing symfony/process (v4.4.2): Downloading (100%)
```

FIGURE 18. Installation of a fresh Laravel project

Working with Composer

Composer resolves dependencies recursively from a file called **composer.json**. The file contains version constraints, which tell composer what version to use and up to which version a package can be updated. After sources and versions for all packages are resolved, they are downloaded and installed into **/vendor** folder.

After packages are installed for the first time, Composer stores precise version information to **composer.lock** file. When no lock file is present, Composer will read requirements from **composer.json** and generate a new lock file. Subsequently, Composer checks precise version requirements form the lock file for dependency installation. The lock file must be under version control, so all members of a team will have exactly the same versions of dependencies installed.

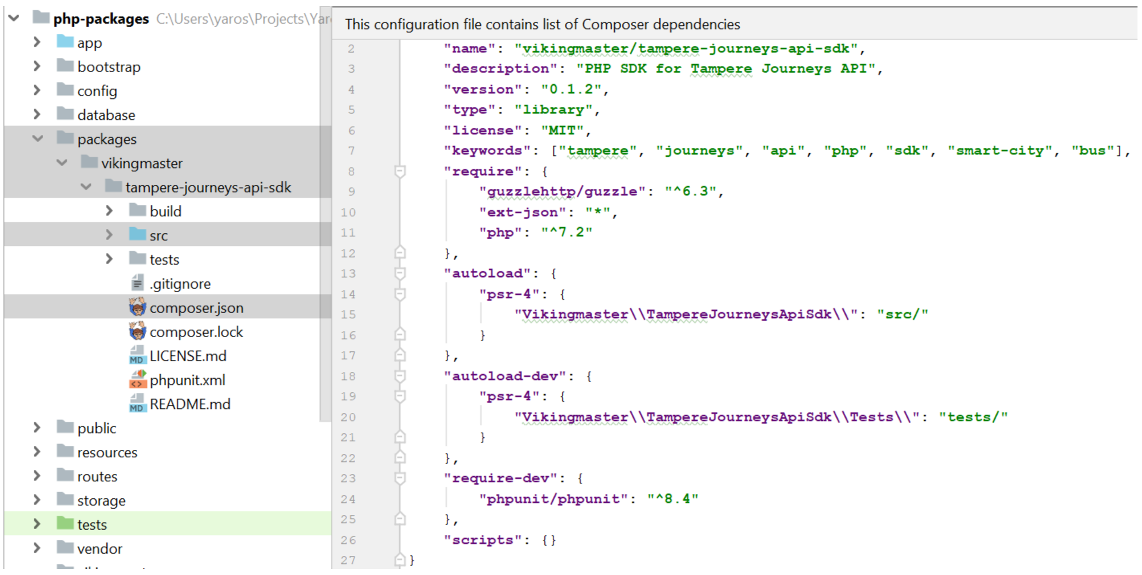
Since we aim to develop our SDK as a separate Composer package, the package needs to have its composer.json file in the package root directory.

To initialize a package, the following needs to be decided:

- Package name in format **vendor/package**
  - vikingmaster/tampere-journeys-api-sdk
- Namespace (advisable to mirror the package name, however arbitrary)
  - \Vikingmaster\TampereJourneysApiSdk
- Package directory in the project (should mirror the package name)
  - /packages/vikingmaster/tampere-journeys-sdk

In modern PHP development, the package root namespace should be mapped to a source directory according to PSR-4 convention. When directory structure follows the namespaces, it allows Composer to seamlessly autoload classes. (PSR-4)

The information needs to be registered in **composer.json** of the package. Executing command `composer init` and following the prompts will create the file, which looks like in figure 19.



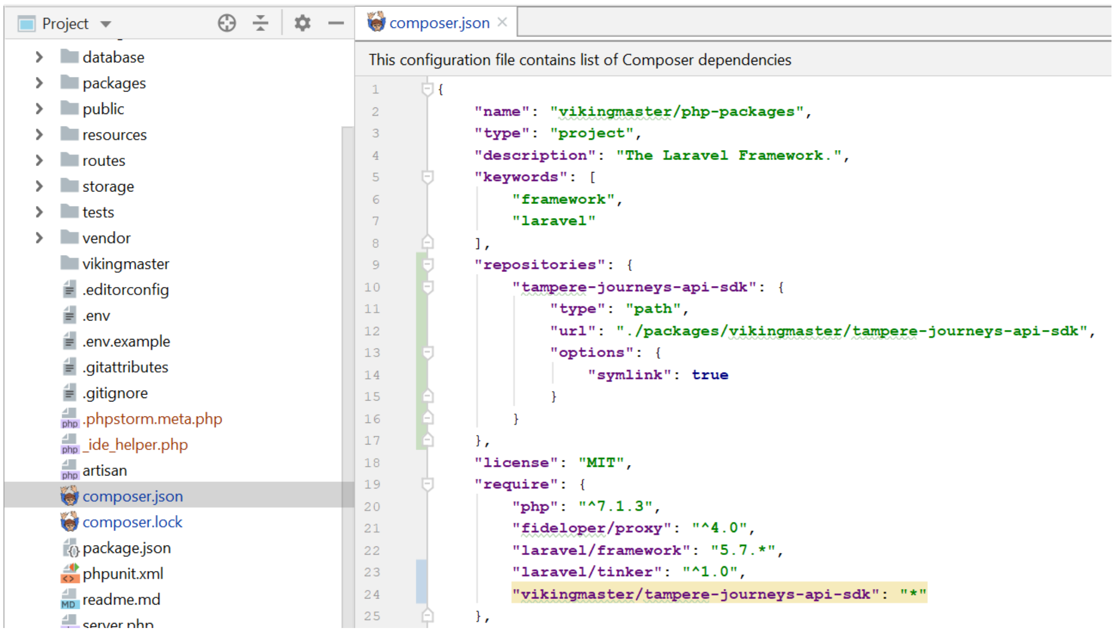
```
php-packages C:\Users\yaros\Projects\Yar
  app
  bootstrap
  config
  database
  packages
    vikingmaster
      tampere-journeys-api-sdk
        build
        src
        tests
        .gitignore
        composer.json
        composer.lock
        LICENSE.md
        phpunit.xml
        README.md
  public
  resources
  routes
  storage
  tests
  vendor
```

This configuration file contains list of Composer dependencies

```json
    "name": "vikingmaster/tampere-journeys-api-sdk",
    "description": "PHP SDK for Tampere Journeys API",
    "version": "0.1.2",
    "type": "library",
    "license": "MIT",
    "keywords": ["tampere", "journeys", "api", "php", "sdk", "smart-city", "bus"],
    "require": {
        "guzzlehttp/guzzle": "^6.3",
        "ext-json": "*",
        "php": "^7.2"
    },
    "autoload": {
        "psr-4": {
            "Vikingmaster\\TampereJourneysApiSdk\\": "src/"
        }
    },
    "autoload-dev": {
        "psr-4": {
            "Vikingmaster\\TampereJourneysApiSdk\\Tests\\": "tests/"
        }
    },
    "require-dev": {
        "phpunit/phpunit": "^8.4"
    },
    "scripts": {}
}
```

FIGURE 19. Composer.json file of the SDK package

In the on project-level, the package is located in **/packages** directory and not **/vendor** because **/vendor** directory contains source code of external packages and is usually ignored by version control. In order to preserve the package from accidental deleting, the source code is kept in a separate directory and connected to the project level **composer.json** using a **symlink** feature. The following figure demonstrates how packages can be virtually present by utilizing the **symlink** feature as demonstrated on figure 20.



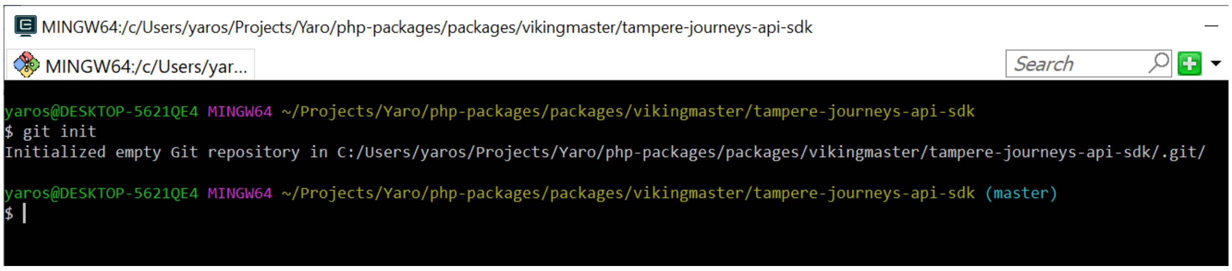
```
{
    "name": "vikingmaster/php-packages",
    "type": "project",
    "description": "The Laravel Framework.",
    "keywords": [
        "framework",
        "laravel"
    ],
    "repositories": {
        "tampere-journeys-api-sdk": {
            "type": "path",
            "url": "./packages/vikingmaster/tampere-journeys-api-sdk",
            "options": {
                "symlink": true
            }
        }
    },
    "license": "MIT",
    "require": {
        "php": "^7.1.3",
        "fideloper/proxy": "^4.0",
        "laravel/framework": "5.7.*",
        "laravel/tinker": "^1.0",
        "vikingmaster/tampere-journeys-api-sdk": "*"
    },
```

FIGURE 20. Project-level composer.json registering the package locally

Upon requiring the package locally in the project-level **composer.json**, running a command composer require vikingmaster-tampere-journeys-api-sdk in the project directory will do installation, map namespaces to directories and create **symlink** under project-level **/vendor** folder.

6.2 Setting up Git for version control

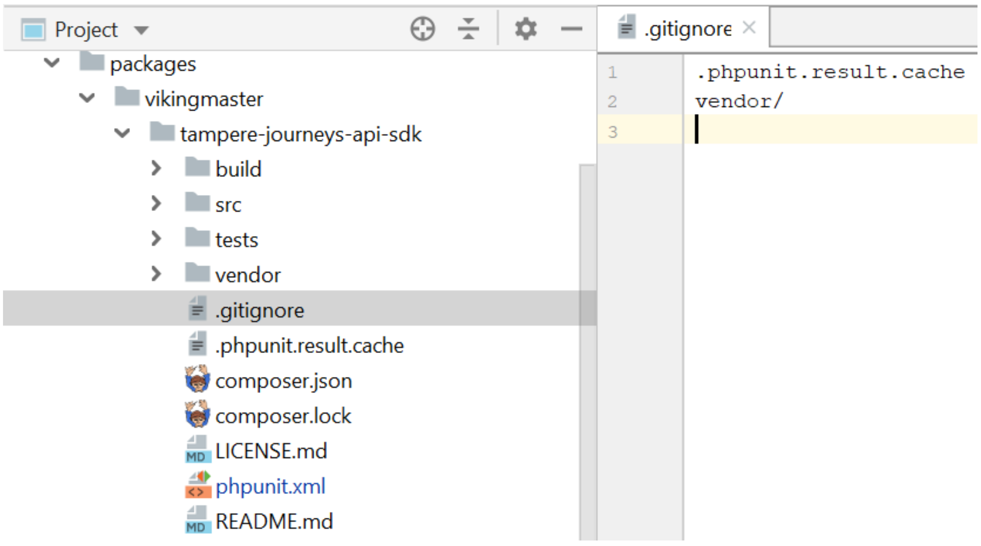
Version control should be set up for a proper maintenance of the software.
It is optional whether the packages project should be under Git version control, however the SDK package should have its own Git repository. To do so, Git repository needs to be initialized in **/packages/vikingmaster/tampere-journeys-api-sdk** using command git init (figure 21).


MINGW64:/c/Users/yaros/Projects/Yaro/php-packages/packages/vikingmaster/tampere-journeys-api-sdk
MINGW64:/c/Users/yar...
Search

```
yaros@DESKTOP-5621QE4 MINGW64 ~/Projects/Yaro/php-packages/packages/vikingmaster/tampere-journeys-api-sdk
$ git init
Initialized empty Git repository in C:/Users/yaros/Projects/Yaro/php-packages/packages/vikingmaster/tampere-journeys-api-sdk/.git/

yaros@DESKTOP-5621QE4 MINGW64 ~/Projects/Yaro/php-packages/packages/vikingmaster/tampere-journeys-api-sdk (master)
$ |
```

FIGURE 21. Initializing Git repository for the package

To make sure, that **/vendor** folder does not accidently go under the source control, a **.gitignore** file needs to be present and have a line **/vendor** (figure 22).



```
Project
packages
  vikingmaster
    tampere-journeys-api-sdk
      build
      src
      tests
      vendor
      .gitignore
      .phpunit.result.cache
      composer.json
      composer.lock
      LICENSE.md
      phpunit.xml
      README.md
```

```
.gitignore
.phpunit.result.cache
vendor/
```

FIGURE 22. Initializing .gitignore file for the new package

The rest of the files can be committed to Git.

6.3 Writing source code

The SDK was implemented using advanced SDK architecture. The UML schema is demonstrated in figure 23 and the file structure in figure 24.

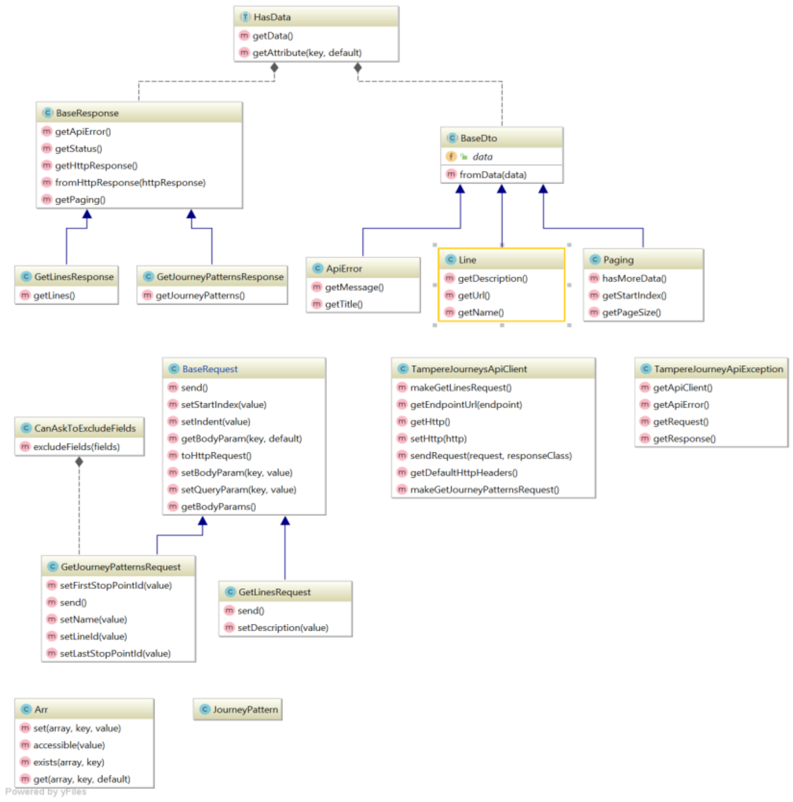
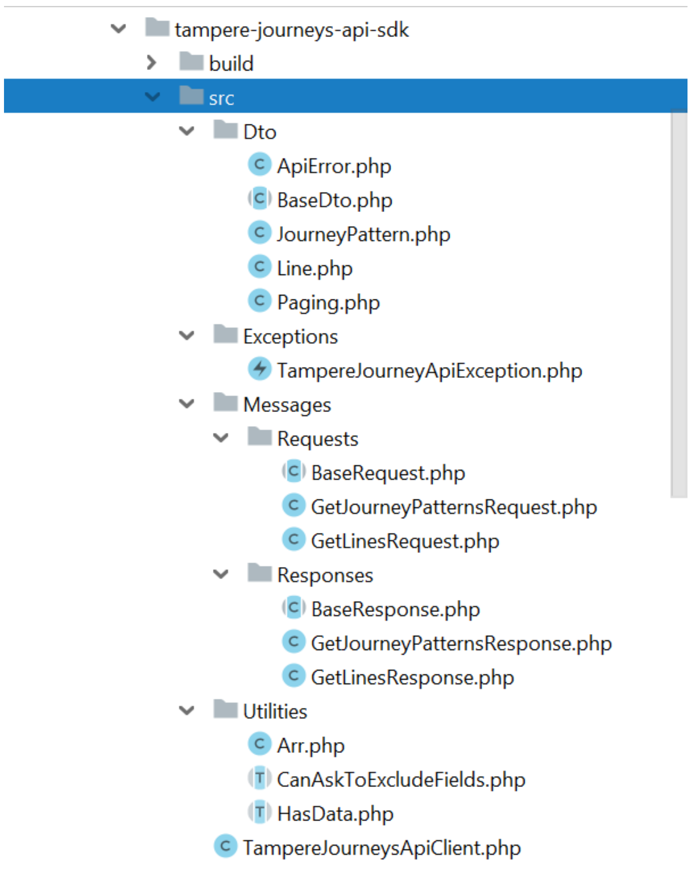


FIGURE 23. UML structure of Tampere Journeys API SDK



```
tampere-journeys-api-sdk
  build
  src
    Dto
      ApiError.php
      BaseDto.php
      JourneyPattern.php
      Line.php
      Paging.php
    Exceptions
      TampereJourneyApiException.php
    Messages
      Requests
        BaseRequest.php
        GetJourneyPatternsRequest.php
        GetLinesRequest.php
      Responses
        BaseResponse.php
        GetJourneyPatternsResponse.php
        GetLinesResponse.php
    Utilities
      Arr.php
      CanAskToExcludeFields.php
      HasData.php
    TampereJourneysApiClient.php
```

FIGURE 24. File structure of Tampere Journeys API SDK

6.4 Writing automated tests with PHPUnit

The more complexity a software has, the more potential issues it can contain, therefore it is a good idea to cover the SDK with automated tests. The modular structure of the SDK makes it relatively easy to write unit tests.

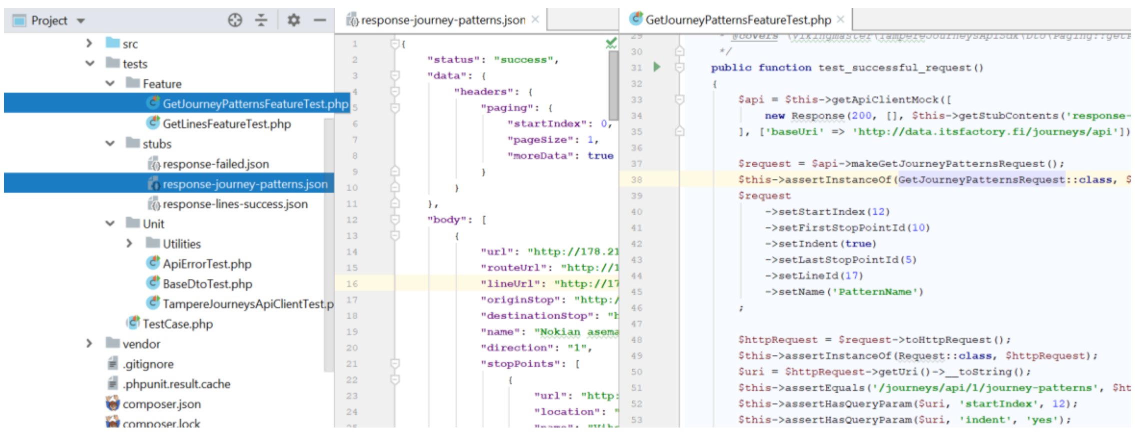
Tests are executed by **PHPUnit**, which can be downloaded as a Composer dependency by requiring **phpunit/phpunit**. PHPUnit is a standard module for testing software in PHP.

In order to enable **PHPUnit** testing, a configuration file **phpunit.xml** is required. It defines various parameters such as location of test files and target directory for test coverage report (figure 25).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<phpunit backupGlobals="false"
         backupStaticAttributes="false"
         bootstrap="vendor/autoload.php"
         colors="true"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         processIsolation="false"
         stopOnFailure="false">
    <testsuites>
        <testsuite name="Unit">
            <directory suffix="Test.php">./tests/Unit</directory>
        </testsuite>
        <testsuite name="Feature">
            <directory suffix="Test.php">./tests/Feature</directory>
        </testsuite>
    </testsuites>
    <filter>
        <whitelist processUncoveredFilesFromWhitelist="true">
            <directory suffix=".php">./src</directory>
        </whitelist>
    </filter>
    <logging>
        <log type="coverage-html" target="./build/coverage"/>
    </logging>
</phpunit>
```

FIGURE 25. Contents of phpunit.xml at the root directory of the package

In order to make sure that the SDK works correctly when its components interact with each other, Feature tests are written. Feature tests are great for testing software features from top-down approach. However, such Feature tests are not supposed to make real API calls – HTTP responses are simulated or “mocked” with response stubs instead (figure 26).



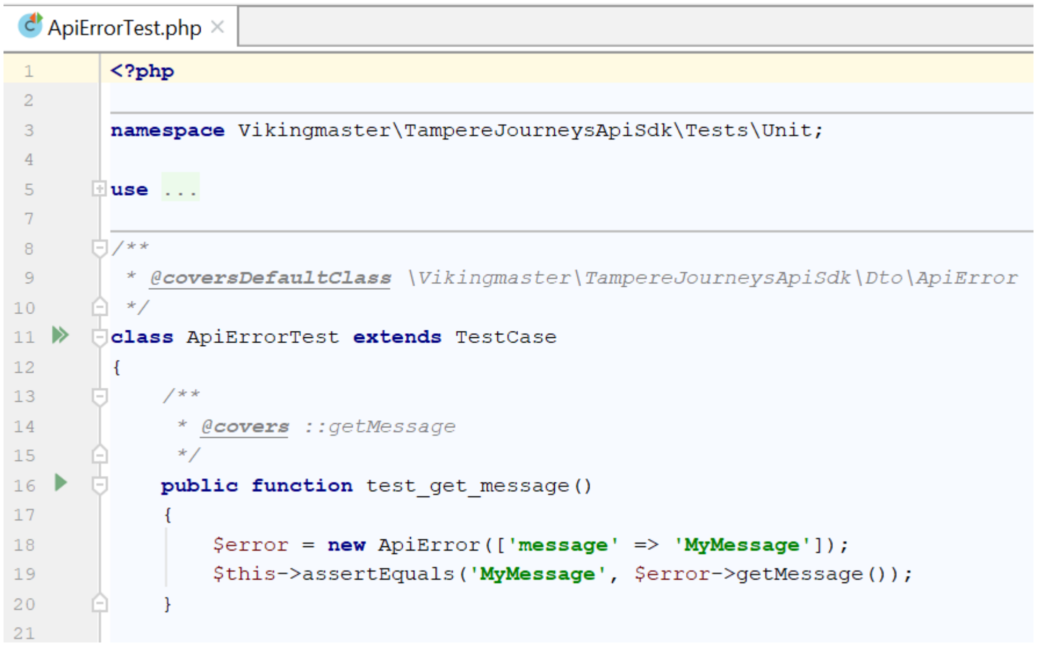
```json
{
    "status": "success",
    "data": {
        "headers": {
            "paging": {
                "startIndex": 0,
                "pageSize": 1,
                "moreData": true
            }
        }
    },
    "body": [
        {
            "url": "http://178.21
            "routeUrl": "http://1
            "lineUrl": "http://17
            "originStop": "http:/
            "destinationStop": "h
            "name": "Nokian asema
            "direction": "1",
            "stopPoints": [
                {
                    "url": "http:
                    "location": "
```

```php
     */
    public function test_successful_request()
    {
        $api = $this->getApiClientMock([
            new Response(200, [], $this->getStubContents('response-
        ], ['baseUri' => 'http://data.itsfactory.fi/journeys/api'])

        $request = $api->makeGetJourneyPatternsRequest();
        $this->assertInstanceOf(GetJourneyPatternsRequest::class, $
        $request
            ->setStartIndex(12)
            ->setFirstStopPointId(10)
            ->setIndent(true)
            ->setLastStopPointId(5)
            ->setLineId(17)
            ->setName('PatternName')
        ;

        $httpRequest = $request->toHttpRequest();
        $this->assertInstanceOf(Request::class, $httpRequest);
        $uri = $httpRequest->getUri()->__toString();
        $this->assertEquals('/journeys/api/1/journey-patterns', $ht
        $this->assertHasQueryParam($uri, 'startIndex', 12);
        $this->assertHasQueryParam($uri, 'indent', 'yes');
```

FIGURE 26. Mocking API calls with stubs in Feature tests

When writing automated tests, developers should be interested in **code coverage**. **Code coverage** tells, to which degree a software is used during automated testing. More code coverage means more software parts are tested. Although code coverage does not necessarily tell about the quality of tests, one should strive to maximize code coverage as it indicates the reliability of software to end-developers.

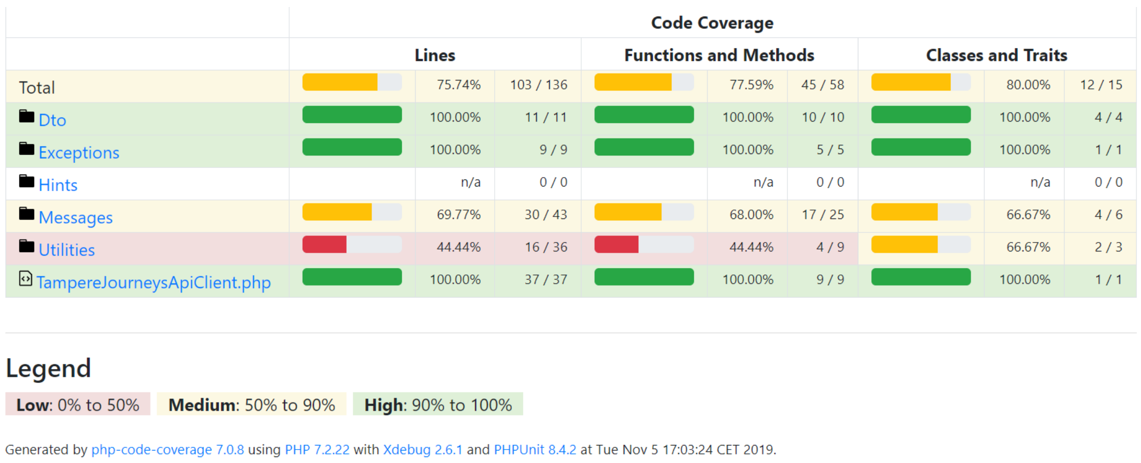
PHPUnit has an ability to automatically generate code coverage reports after testing. **PHPDoc** notations such as **@coversDefaultClass** and **@covers** register specific software components for code coverage reporting (figure 27).



```php
<?php

namespace Vikingmaster\TampereJourneysApiSdk\Tests\Unit;

use ...

/**
 * @coversDefaultClass \Vikingmaster\TampereJourneysApiSdk\Dto\ApiError
 */
class ApiErrorTest extends TestCase
{
    /**
     * @covers ::getMessage
     */
    public function test_get_message()
    {
        $error = new ApiError(['message' => 'MyMessage']);
        $this->assertEquals('MyMessage', $error->getMessage());
    }

```

FIGURE 27. @coversDefaultClass and @covers notations

When the notations are correctly placed, a code coverage report can be generated by executing `vendor/bin/phpunit --coverage-html`. The generated code coverage report is shown in figure 28.



	Code Coverage								
	Lines			Functions and Methods			Classes and Traits		
Total	🟨	75.74%	103 / 136	🟨	77.59%	45 / 58	🟨	80.00%	12 / 15
Dto	🟩	100.00%	11 / 11	🟩	100.00%	10 / 10	🟩	100.00%	4 / 4
Exceptions	🟩	100.00%	9 / 9	🟩	100.00%	5 / 5	🟩	100.00%	1 / 1
Hints		n/a	0 / 0		n/a	0 / 0		n/a	0 / 0
Messages	🟨	69.77%	30 / 43	🟨	68.00%	17 / 25	🟨	66.67%	4 / 6
Utilities	🟥	44.44%	16 / 36	🟥	44.44%	4 / 9	🟨	66.67%	2 / 3
TampereJourneysApiClient.php	🟩	100.00%	37 / 37	🟩	100.00%	9 / 9	🟩	100.00%	1 / 1

Legend

Low: 0% to 50% **Medium**: 50% to 90% **High**: 90% to 100%

Generated by php-code-coverage 7.0.8 using PHP 7.2.22 with Xdebug 2.6.1 and PHPUnit 8.4.2 at Tue Nov 5 17:03:24 CET 2019.

FIGURE 28. Code coverage report generated by PHPUnit

6.5 Preparing the package for release

Knowing how to publish a package is very useful as it allows for better sharing and collaboration. Additionally, it enables microarchitecture approach to software development, when parts are modularized and developed separately. Business-wise releasing a ready-made software allows to increase market adoption for a specific API service.

Creating README.md

Package maintainers should provide a **README** file, which provide explanation what the package is meant for and how it can be used. Documentation means sharing of accumulated knowledge in an accessible form for others. The file must be committed to the repository.

Creating LICENSE.md

LICENSE file is especially important for public software releases, as it will describe in what ways the software can be used and what are the terms and agreements. Service **Choosealicense.com** offers a simple way to choose the needed license. My SDK package has an MIT license, which is very permissive – allows to modify, distribute, use software privately and commercially, provides no warranty and includes a limitation on liability. Users are only required to include copyright information. Like the README.md, the LICENSE.md must be committed to the repository.

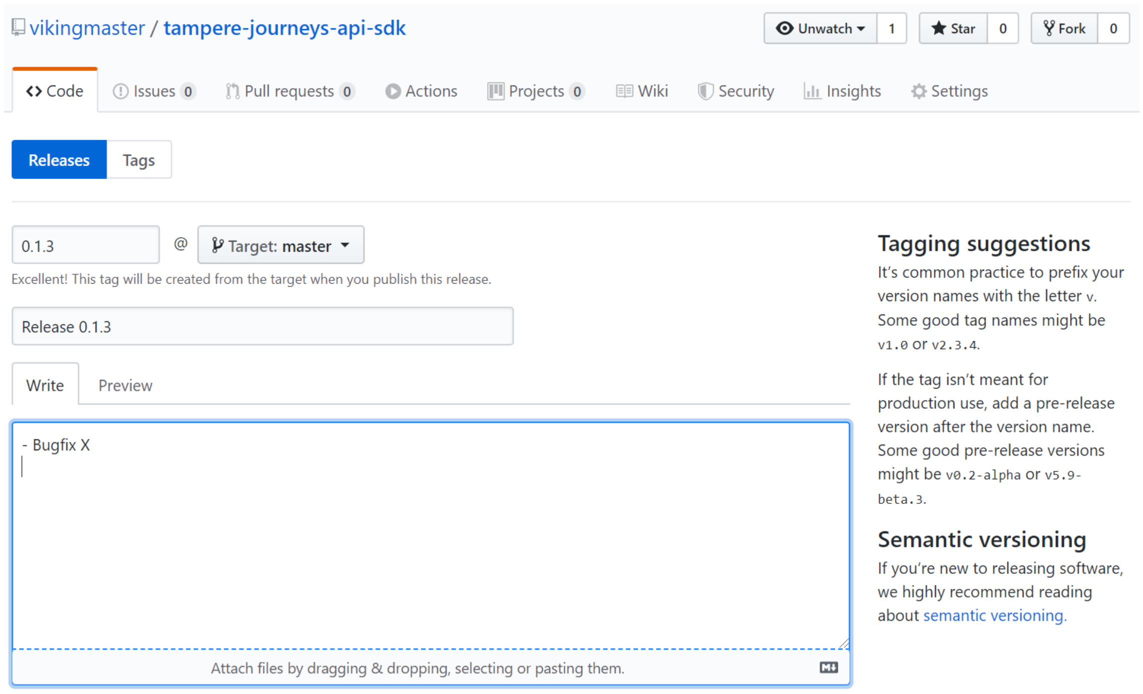
Semantic versioning and tagging

Composer uses **semantic versioning** convention (SemVer) in format: **MAJOR.MINOR.PATCH**

- **MAJOR** is incremented, when **backwards incompatible** changes are introduced.
- **MINOR** is incremented, when **backwards compatible** features are introduced
- **PATCH** is incremented when **backwards compatible** bug fixes are introduced

Version number should be specified in **SemVer** format in **composer.json** file. After version increment, the file must be committed to version control and pushed GitHub for new release drafting.

Version releases on **Github** are bound to tags in Git. Tags resemble labels pointing to a specific point in Git history. When drafting a release (figure 29), a new tag must be created and it must be the same as version number specified in **composer.json**. This way, Packagist.org will be able to serve package versions correctly.



0.1.3 @ Target: master

Excellent! This tag will be created from the target when you publish this release.

Release 0.1.3

- Bugfix X

Attach files by dragging & dropping, selecting or pasting them.

Tagging suggestions

It's common practice to prefix your version names with the letter v. Some good tag names might be `v1.0` or `v2.3.4`.

If the tag isn't meant for production use, add a pre-release version after the version name. Some good pre-release versions might be `v0.2-alpha` or `v5.9-beta.3`.

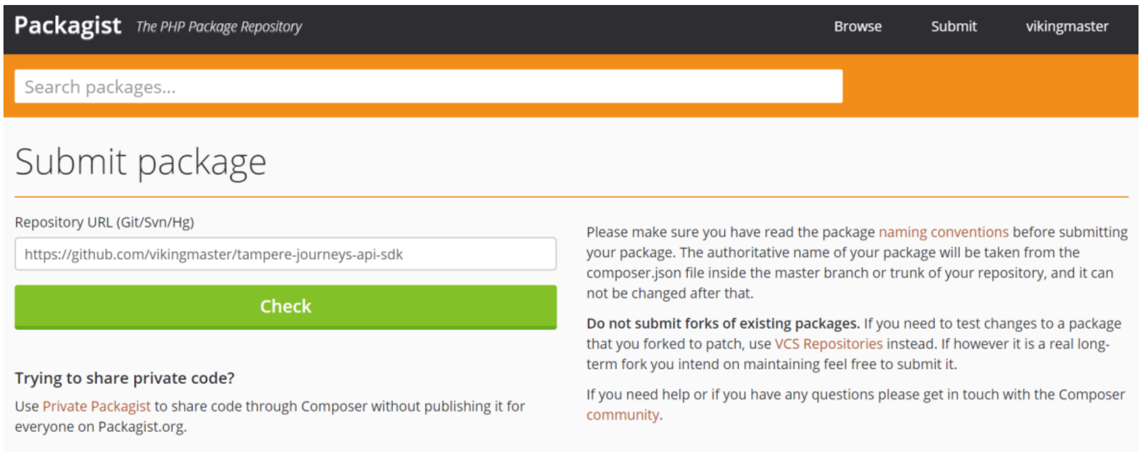
Semantic versioning

If you're new to releasing software, we highly recommend reading about semantic versioning.

FIGURE 29. Creating a new version release on GitHub

6.6 Submitting the package to Packagist.org

After the release have been created, we head to Packagist.org, go through authentication process and submit the package (figure 30). The service will register existing releases and subscribe to the repository events such as new release creation.



Submit package

Repository URL (Git/Svn/Hg)

https://github.com/vikingmaster/tampere-journeys-api-sdk

Check

Trying to share private code?

Use Private Packagist to share code through Composer without publishing it for everyone on Packagist.org.

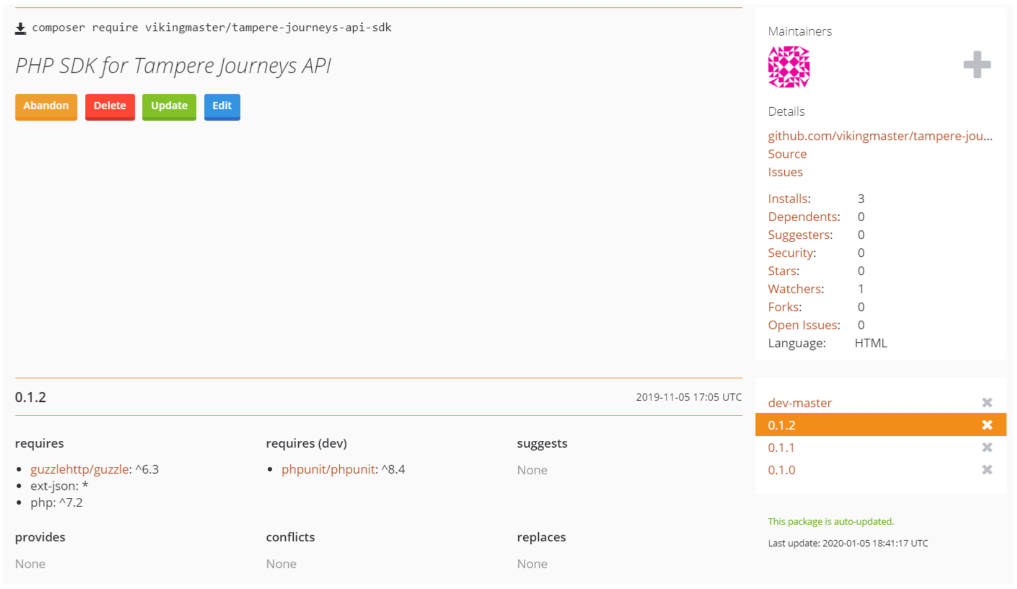
Please make sure you have read the package naming conventions before submitting your package. The authoritative name of your package will be taken from the composer.json file inside the master branch or trunk of your repository, and it can not be changed after that.

Do not submit forks of existing packages. If you need to test changes to a package that you forked to patch, use VCS Repositories instead. If however it is a real long-term fork you intend on maintaining feel free to submit it.

If you need help or if you have any questions please get in touch with the Composer community.

FIGURE 30. Submitting the SDK package to Packagist.org

When submission is completed, Packagist.org displays full information about the package for anyone interested. It shows version numbers, required dependencies, usage statistics and other information (figure 31).



```
composer require vikingmaster/tampere-journeys-api-sdk
```

PHP SDK for Tampere Journeys API

Maintainers

Details

github.com/vikingmaster/tampere-jou...
Source
Issues

Installs:	3
Dependents:	0
Suggesters:	0
Security:	0
Stars:	0
Watchers:	1
Forks:	0
Open Issues:	0
Language:	HTML

0.1.2 2019-11-05 17:05 UTC

requires	requires (dev)	suggests
guzzlehttp/guzzle: ^6.3 ext-json: * php: ^7.2	phpunit/phpunit: ^8.4	None

provides	conflicts	replaces
None	None	None

dev-master
0.1.2
0.1.1
0.1.0

This package is auto-updated.

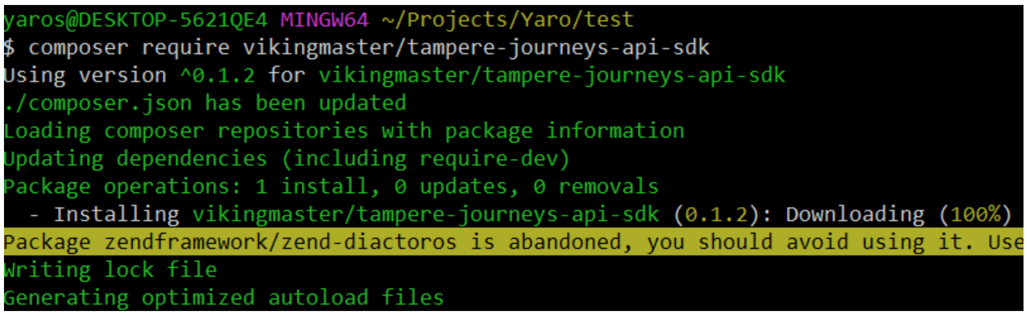
Last update: 2020-01-05 18:41:17 UTC

FIGURE 31. View of a submitted Composer package

After submission, the package **vikingmaster/tampere-journeys-api-sdk** is available for public use and can be required as a Composer dependency.

6.7 Using the package in a new project

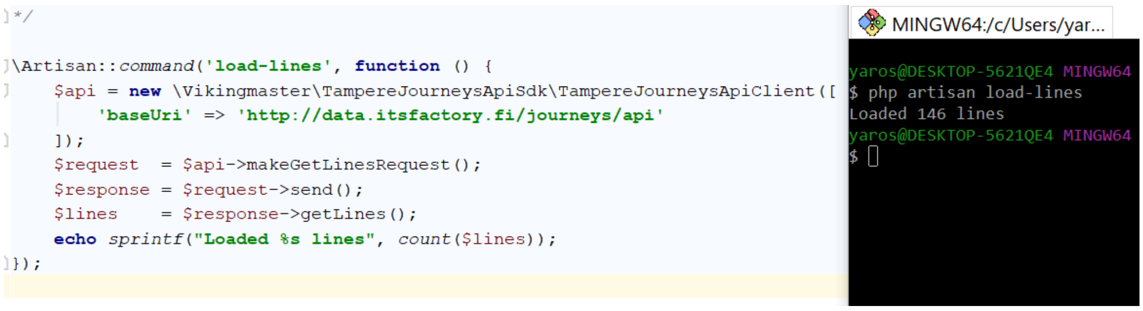
In order to test the published package, a new Laravel project was created. To test the latest version of the package, a command composer require vikingmaster/tampere-journeys-api-sdk was executed (figure 32), after which the package appeared under **/vendor** directory.



```
yaros@DESKTOP-5621QE4 MINGW64 ~/Projects/Yaro/test
$ composer require vikingmaster/tampere-journeys-api-sdk
Using version ^0.1.2 for vikingmaster/tampere-journeys-api-sdk
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing vikingmaster/tampere-journeys-api-sdk (0.1.2): Downloading (100%)
Package zendframework/zend-diactoros is abandoned, you should avoid using it. Use
Writing lock file
Generating optimized autoload files
```

FIGURE 32. Installation of new SDK package in a new project

After installation, the SDK functionality was tested by creating a Laravel-based CLI command which used SDK to make an API call. As it is shown on the right side of the figure 33, the result indicates that the feature functions correctly. Autocompletion makes getting the API data a quick and trivial process.



```
*/

\Artisan::command('load-lines', function () {
    $api = new \Vikingmaster\TampereJourneysApiSdk\TampereJourneysApiClient([
        'baseUri' => 'http://data.itsfactory.fi/journeys/api'
    ]);
    $request  = $api->makeGetLinesRequest();
    $response = $request->send();
    $lines    = $response->getLines();
    echo sprintf("Loaded %s lines", count($lines));
});
```

```
MINGW64:/c/Users/yar...
yaros@DESKTOP-5621QE4 MINGW64
$ php artisan load-lines
Loaded 146 lines
yaros@DESKTOP-5621QE4 MINGW64
$
```

FIGURE 33. Testing an API call of SDK via a CLI command

6.8 Future development and maintenance

In order to make changes to a package, a developer needs to release a new version, so it will be available to others. In a nutshell a new version is released through these steps:

- Branch out of the desired version
- Make code changes
- Change version number in **composer.json** according to SemVer

- Commit and push changes
- Create a new release in GitHub with change documentation

Packagist.org will register a new release, after which, depending on version constraints, the new version of package can be installed by executing `composer update vikingmaster/tampere-journeys-api-sdk`.

This concludes the cycle of REST API SDK package development in PHP.

7 DISCUSSION

Scarcity of reference material for practical part

While most of the theory could be referenced to external sources, there was not enough relevant material to back up the practical part of REST API SDK creation. Wireframing different types of SDK design emerged from my web development experience and therefore may be considered as subjective.

Practical part requires advanced web development knowledge

Originally it was thought that the thesis could be beneficial for beginners, however it was later decided that the focus must be on SDK architecture rather than beginner-friendly steps for getting the project up and running. Practical part might contain missing intermediate steps, which can be obvious for experienced PHP developers but confusing for beginners.

Slight mismatch of actual SDK code and architecture in practical part

The actual implementation has a hybrid approach to SDK creation: advanced SDK architecture was used in addition to type-hinting as a part of research.

Quantitative research not available

The results of the impacts of a well-written SDK are judged with a common sense and observations, however no quantitative research regarding speed is available to tell the degree of the impact.

Advanced ORM SDK architecture revealed but not covered

One more REST API SDK design was revealed during writing of this thesis. ORM (Object Relation Model) approach would be suited the best for integrations needing to support full spectrum of CRUD operations on API endpoints and heavy API usage. This type of architecture was not covered due the need for extensive research of its complex architecture.

The revealed SDK designs will be useful in the future

As a web developer I had done plenty of API integration tasks and there did not seem to be a single perfect SDK design approach for all of them. The research revealed and solidified multiple satisfactory approaches for the future adoption in order to improve the development efficiency.

REFERENCES

Brajesh D. 2017. API Management: An Architect's Guide to Developing and Managing APIs for Your Organization. Read 20.11.2019.

Clariontech. 2019. 10 Reasons Why Laravel Is the Best PHP Framework for 2019. Accessed 19.1.2020. https://www.clariontech.com/blog/10-reasons-why-laravel-is-the-best-php-framework-for-2019

Harvard Business Review. 2015. The Strategic Value of APIs. Accessed 10.1.2020. https://hbr.org/2015/01/the-strategic-value-of-apis

IBM. 2016. API Monetization. Accessed 10.1.2020. https://www.ibm.com/downloads/cas/L5Q82XR0

Internet World Stats. 2019. Accessed 8.1.2020. https://www.internetworldstats.com/stats.htm

Lambert J., Aulakh N. & Rickard T. 2015. Virtualization with Vagrant. Accessed 20.01.20. https://computationalmodelling.bitbucket.io/tools/vagrant.html

McKinsey. 2017. What it really takes to capture the value of APIs. Accessed 9.1.2020. https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/what-it-really-takes-to-capture-the-value-of-apis

Paloviita, S. 2018. Digital Value Chain in API Economy. Accessed 9.1.2020. https://medium.com/apinf/digital-value-chain-in-api-economy-37ce8771b54e

Programmable Web. 2019. Accessed 8.1.2020. https://www.programmableweb.com/news/apis-show-faster-growth-rate-2019-previous-years/research/2019/07/17

PSR-4. Class autoloading convention for PHP. Accessed 16.1.2020. https://www.php-fig.org/psr/psr-4

Semantic Versioning 2.0.0. Accessed 16.1.2020. https://semver.org

Unifaun. Example of value chain service. Accessed 9.1.2020. https://www.unifaun.com/fi/tavarantoimittaja/saatavilla-olevat-kuljetusliikkeet

W3Techs. 2020. Market share of PHP. Accessed 11.1.2020. https://w3techs.com/technologies/details/pl-php

Wikipedia. GraphQL. Accessed 21.01.20. https://en.wikipedia.org/wiki/GraphQL

Wikipedia. SOAP. Accessed 19.1.2020. https://en.wikipedia.org/wiki/SOAP

Wikipedia. Integrated Development Environment. Accessed 19.1.2020. https://en.wikipedia.org/wiki/Integrated_development_environment

Woods D., Brail G., Jacobson D. 2011. APIs: A Strategy Guide. Read 28.11.2019.