jamk

Increasing Full Stack Development Productivity via Technology Selection

Mike Koder

Master's thesis
November 2021
Information and communication technologies
Master's Degree Programme in Information and communication technologies
Full Stack Software Development

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Description

Koder, Mike

Increasing Full Stack Development Productivity via Technology Selection

Jyväskylä: JAMK University of Applied Sciences, November 2021, 78 pages.

Information and communication technologies. Master's Degree Programme in Information Technology, Full Stack Software Development. Master's Thesis.

Permission for web publication: Yes

Language of publication: English

Abstract

Building web application prototypes is a common project type for consulting companies. Developers can have hard time selecting the best technologies from dozens of options. The primary objective was to find backend and frontend technologies to improve the productivity of full stack development. The secondary goals were determining the extent of features available in modern frontend and backend technologies and studying which are the most significant features for technology evaluation.

Research papers on software development productivity were analyzed to find factors suitable for guiding the technology selection process. The most popular programming languages and their web frameworks and libraries were collected for comparative analysis. Technologies' features were gathered from official documentation websites to gain a good understanding of the spectrum of features. Finally, technologies were compared by how well each feature was supported.

Reuse, adequate documentation, automatization and community support were identified to be the few productivity factors relevant for technology selection process. JavaScript, TypeScript, Python, C#, Java and PHP were found to be the most popular programming languages for web development. Feature comparison revealed backend technologies having great differences in the available features. Especially request binding and the ability to automatically infer OpenAPI documentation were detected to reduce manual repetitive work. ASP.NET Core, NestJS, Laravel, FastAPI and Spring were found to be the most feature rich frameworks for different programming languages. Frontend technologies were found to have only minor differences.

Comparison results can be used to evaluate technologies for new full stack development projects today. The feature evaluation process can also be utilized in the future to compare how well new technologies measure up with prior ones.

Keywords/tags (subjects)

Web development, Full stack development, Backend development, Frontend development, Productivity, RESTful API, SPA, Single page application, OpenAPI

Miscellaneous (Confidential information)

n/a

Contents

Figures

Tables

Code Blocks

Terms

AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
CORS	Cross-Origin Resource Sharing
CRUD	Create, read, update and delete operations
DOM	Document Object Model
DSL	Domain Specific Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JWT	JSON Web Token
OpenAPI	Specification for describing RESTful APIs
PAAS	Platform as a Service
REST	Representational State Transfer
SPA	Single-page application
URL	Uniform Resource Locator
XML	Extensible Markup Language

1 Introduction

1.1 Background

Shortage of software developers has appeared frequently in the Finnish news headlines for the last couple of years. Educating more developers has often been seen as the solution, but that's not the only option. In addition to increasing the number of developers, enabling them to complete more tasks in the same time would equally ease the shortage.

During my career I've worked on dozens of web development projects. In my experience building small 1-18 man-month web application prototypes have been quite common projects for consulting companies in Finland. These prototypes could have often been described as "glorified CRUD applications", a term commonly used in web and social media for applications consisting of mostly basic data manipulation operations (Hacker news, 2020; Reddit, 2013; Team blind, 2020).

Given that dozens of technologies are available for web development, developers can have hard time deciding which ones to use. Time constraints limit the depth of the evaluation and web articles provide only shallow comparisons. Technologies are advertised as having been used in popular web applications having millions of users. It's possible that small teams in Finnish software development industry building applications for vastly smaller audience could be more productive using other technologies.

1.2 Research objectives and questions

The objective of this thesis is to research how the selection of technologies could increase productivity in full stack development. In order to gain both immediate and future benefits, the objective is split into three concrete goals:

1. Find the best frontend and backend technologies currently in terms of productivity.
2. Gain knowledge on the spectrum of features available in frontend and backend technologies.
3. Determine which characteristics are the most crucial when evaluating capabilities of frontend and backend technologies in terms of productivity.

This study builds upon four research questions:

RQ1. What software development productivity factors could be used to guide the technology selection process?

RQ2. What are the programming languages, frameworks and libraries commonly used for web development?

RQ3. What are the characteristics and features of modern backend and frontend (SPA) technologies?

RQ4. What are the best technologies productivity-wise currently?

1.3 Scope

Technologies like low-code/no-code tools and code generators are left out from the scope of this thesis. Although they are claimed to increase productivity (Alpha Software, 2021; Hackernoon, 2021; Long, T. 2021; Spendel, T. 2020; Stangarone, J. 2019; Tay, N. 2021;) they also have the possible drawbacks of **limited customization** (Alpha Software, 2021; Hackernoon, 2021; Long, T. 2021; Spendel, T. 2020; Stangarone, J. 2019; Tay, N. 2021; Tozzi, C. 2019) and **vendor lock-in** (Alpha Software, 2021; Hackernoon, 2021; Long, T. 2021; Stangarone, J. 2019; Tay, N. 2021; Tozzi, C. 2019). They are also marketed to enable building software **without coding experience** by **non-technical business users** (Alpha Software, 2021; Long, T. 2021; Spendel, T. 2020; Tay, N. 2021; Tozzi, C. 2019) which conflicts with the idea of selling coding work.

Desired technologies should also be *general purpose*, i.e. enable developing a wide range of web applications and/or APIs for mobile applications and be deployable to any major cloud provider. In detail, criterion used to define general purpose technologies in this thesis are:

- Frontend and backend must be independent of each other.
- Backend must be database independent.
- Technologies must be vendor and platform independent.
- Technologies must be free and open source.

Focus is on developing prototypes with a small team or even by a sole developer. Productivity factors related to teamwork, process and other non-technical factors are not in the scope of this research.

2 Research setting

In order to research the main objective of finding the best technologies in terms of productivity, it is first needed to answer the first research question (RQ1) of finding the factors affecting software development productivity. Literature review on prior studies is done to find productivity factors suitable for the technology selection process. Studies are searched from online databases Google Scholar (https://scholar.google.com) and IEEE Xplore Digital Library (https://ieeexplore.ieee.org). In software development new technologies emerge frequently. In order to answer the second research question (RQ2) viable technologies are collected from online resources. As this phase only involves collecting the names of the technologies, all sources found with web searches are considered credible. Further details on technologies are collected from various online sources to evaluate their vitality and suitability for the feature level analysis.

Web searches “backend features” or “frontend features” don’t return any relevant results for forming lists of features to be used in the technology evaluation. Comprehensive literature reviews on backend and frontend technologies` features are done to find the answer to research question 3 (RQ3) of determining the spectrum of available features. Technologies’ official documentation websites are used as sources as they are considered having most up to date and correct information.

Lastly, technologies are evaluated on the extent of features they have available. Analyzing the evaluation results answers the most important research question (RQ4) of naming the best full stack technologies today in terms of productivity.

3 Technical factors in software development productivity

Wagner and Ruhe (2008) reviewed literature on software development productivity made in years 1970-2007 and identified various technical and soft factors affecting development productivity. Of those factors only *product complexity*, *reuse*, *use of tools*, *programming language* and *documentation quality* could guide the selection of technologies while others were more related to project management, process, business requirements and other non-technical matters. Study didn’t clearly state whether those factors had a positive or negative impact on productivity.

Various other studies also mentioned similar factors including their impacts and found *decreasing complexity* (de Barros Sampaio et al., 2010; Maxwell & Forselius, 2000; Mota et al., 2021), *increasing reuse* (de Barros Sampaio et al., 2010; Murphy-Hill et al., 2021; Melo et al, 2011), *use of suitable tools* (de Barros Sampaio et al., 2010; Maxwell & Forselius, 2000; Mota et al., 2021; Murphy-Hill et al., 2021), *higher level programming language* (de Barros Sampaio et al., 2010; Mota et al., 2021; Jiang & Comstock, 2007) and *adequate documentation* (Mota et al., 2021; Tomaszewski & Lundberg, 2005) having positive impacts on productivity. *Competence* with used technologies was also observed increasing productivity in various studies (Canedo & Santos, 2019; de Barros Sampaio et al., 2010; Maxwell & Forselius, 2000).

Pano et al. (2018) studied the factors leading to the adaption of JavaScript frameworks and found *automatization*, *learnability*, *complexity* and *understandability* having impact on the effort needed to achieve programming tasks. Precise documentation with good examples was found to be important for understandability.

It's notable that many of the studies have collected productivity factors from quite old sources and software development has been evolving rapidly. Programming languages commonly used for web development are high level and have a broad range of tools available. The concept of reuse has also expanded from reusing code within a company to using open source libraries available from package managers such as NPM, which was released in 2010 (GitHub 2021).

In addition to effort expectancy factors, Pano et al. (2018) also found *social influence* and *facilitating condition* factors affecting the adoption of JavaScript frameworks. Of these factors *community size*, *community responsiveness*, *updates*, and *extensibility* can also be considered affecting productivity by increasing reuse and reducing the time needed to find help.

Given that *learnability* and *understandability* are highly subjective, getting meaningful results for them would require gathering experience from a wide range of developers which is not in the scope of this thesis. It seems selecting technologies having a wide range of features, adequate documentation and good community support and then sticking to them would result in increased productivity. Table 1 describes those factors, their implications and how they are evaluated in more detail.

Table 1. Productivity factors suitable for technology selection

Factor	Implication	Evaluation method
Reuse	Less time writing code	Available features built-in Available features as extensions
Adequate documentation	Less time finding example code	Feature examples in documentation
Automatization	Less time doing repetitive work	Eliminating repetitive code
Community size	More help available More contributors adding new features	Popularity Stack Overflow questions Github contributors
Community responsiveness	Questions are answered faster New features are added more frequently	Stack overflow questions Feature examples by community
Updates	New features are added more frequently	Recently updated
Extensibility	3rd party extensions increase reuse	Features available as extensions

In addition to increasing productivity these factors also make technologies more appealing to developers (Pano et al., 2018), thus making them more likely relevant also in the future. By reducing the need to change technologies developers are more likely able to use longer those technologies they are competent with. While technology's *vitality* might not affect productivity today, it's an important factor to consider for future productivity.

4 Full stack development

Full stack development definition varies and has evolved during the years. In its most narrow definition full stack development includes only backend and frontend development. Broader definitions contain a wide variety of technologies and practices including different kinds of databases, cloud services, mobile apps and continuous integration/deployment among others (roadmap.sh, 2021a; roadmap.sh, 2021b).

This chapter describes the essential technologies used in full stack development in the scope of this thesis. More specifically, what are RESTful APIs and single page applications (SPA), how they communicate using HTTP, and how OpenAPI can be used to reduce manual work.

4.1 HTTP

Hypertext Transfer Protocol (HTTP) is a protocol used for communication between client (e.g. browser or mobile application) and server. Messaging consists of requests sent by the client and server responses. The main elements of request are *method*, *URL*, *headers* and *body*. (Gaitatzis, 2019)

Method describes the action. GET method is intended for retrieving data and should not have side effects. POST, PUT, PATCH and DELETE are commonly used for creating, replacing, updating and deleting data respectively. HEAD is used to fetch the headers only. (Gaitatzis, 2019)

Other methods CONNECT, OPTIONS and TRACE are used for establishing a tunnel to the server, describing the communication options and loop-back testing respectively (MDN Web Docs, 2021a).

URLs are addresses for resources on the web. URL itself consists of many parts. E.g. URL *https://example.com/products/123/reviews?start=2020-01-01* contains the following information

Protocol	https
Host	example.com
Path	/products/123/reviews
Query	?start=2020-01-01

Headers consist of name-value pairs. Dozens of standard header fields exist to describe authentication, caching and message body among others (MDN Web Docs, 2021b). Table 2 lists common request headers.

Table 2. Common HTTP request headers (Gaitatzis, 2019; MDN Web Docs, 2021b)

HTTP Header	**Purpose**
Accept	Acceptable response types
Accept-Charset	Acceptable character sets
Accept-Language	Acceptable languages
Authorization	Authentication credentials
Content-Length	Length of the data
Content-Type	Content type of the body

In addition to standard headers, clients and servers can send custom headers. E.g. clients could use an *Accept-Version* header to indicate the desired version of an API. Request body can contain text and data in various formats. JSON is commonly used for objects and binary data for uploaded files (Gaitatzis, 2019). HTTP responses contain *version*, *status*, *headers* and *body*. Status has a 3-digit response code and textual description (MDN Web Docs, 2021c). Table 3 lists some common response codes.

Table 3. Common HTTP status codes (Gaitatzis, 2019)

HTTP status code	**Message**	**Description**
200	OK	Request succeeded
201	Created	Resource was created
204	No Content	Request succeeded, but no data is returned
301	Moved Permanently	Resources has moved to another location
400	Bad Request	Request cannot be processed
401	Unauthorized	Client doesn’t have sufficient access rights
404	Not Found	Resource is not found
500	Internal Server Error	Server encountered unexpected error

Headers are similar to those of request's with some exceptions such as caching which are only relevant to responses (MDN Web Docs, 2021b). Table 4 lists some common response headers.

Table 4. Common HTTP response headers (Gaitatzis, 2019)

HTTP Header	**Purpose**
Content-Disposition	Indicates whether the data should be displayed inline in the browser or downloaded. File name for downloaded content.
Content-Language	Language of the content
Content-Length	Length of the data
Content-Type	Content type of the body
Expires	Data expiration time

Response bodies can be textual or binary data. As in responses, JSON is a common format for objects (Gaitatzis, 2019).

4.2 RESTful API

Representational State Transfer (REST) defines a set of constraints for the architecture of web services. Fielding (2000) lists the following principles for REST

Client-Server

Separating data and UI improves portability and scalability allowing client and server to evolve independently of each other.

Stateless

No client information is stored on the server. All data required for an operation is included in the request.

Cache

Response data can be marked cacheable or non-cacheable. Cacheable data can be reused later in place of an equivalent request.

Uniform Interface

Information is exchanged in a standardized form allowing components to evolve independently and decoupling provided services from their implementation.

Layered System

Components can't tell whether they are communicating with the end system or some intermediate component such as proxy.

Code-on-Demand

Client functionality can be extended with code, such as JavaScript, sent from the server.

Uniform interface further has the following set of constraints of its own

Resources and Resource Identifiers

Resources can be practically any information that can be named, e.g. document or image. Individual resources are identified e.g. by URI.

Representations

Representations can contain the current or intended state of a resource. E.g. server can send the current state of a resource as a response to the client and the client can perform an action by sending the intended state back to the server.

Self-descriptive message

Messages should contain enough information for the other end to be able to process it. E.g. media type is used to tell whether the data should be rendered as an image or an HTML document.

Hypermedia as the engine of application state (HATEOAS)

Clients can discover other resources by following links provided in the message.

Web APIs are commonly called REST APIs although often they don't fulfill the constraints defined for REST (Fielding, 2008). Web APIs use HTTP as a communication protocol. HTTP methods and paths to describe resources and their operations (Gaitatzis, 2019). Table 5 shows some common RESTful routing conventions.

Table 5. Common RESTful routing conventions

HTTP Method	Path pattern	Operation
GET	/{resource}/	Get list of all resources
GET	/{resource}/{id}/	Get a single resource by id
POST	/{resource}/	Create a new resource
PUT	/{resource}/{id}/	Replace a resource
PATCH	/{resource}/{id}/	Update a resource (partially)
DELETE	/{resource}/{id}/	Delete a resource

Representation in modern APIs usually means JSON and less often XML for objects and various media types for binary data such as images. Content-Type headers are used to describe the kind of data. Statelessness is achieved by passing all the data in the request. Client information is encoded as JSON Web Token (JWT) and sent in the request headers. Cookies are also used but using bearer tokens eliminate CSRF (Cross-Site Request Forgery) attacks.

Web applications in the past used to be more server driven, meaning that business logic was mostly in the backend. Servers built the HTML for browsers to display for example. That way a generic client (browser) had all the information needed to navigate between resources (hyperlinks) and invoke actions (e.g. send forms). Web APIs today provide mostly just data. Business logic has moved more to the client. Generic clients can no longer navigate between resources as data often doesn't contain links. Data also doesn't contain enough metadata for clients to invoke actions. Clients today are built for specific purposes and need to know URIs to the resources and the data schema in advance in order to work.

4.3 Single page applications

Web frontends in the past were mostly server-built HTML with CSS for styling and JavaScript for improved interactivity. Following a link or sending a form made the browser navigate to another page or reload the current page. As the web evolved from fairly static websites to the direction of applications with more interactivity the processing needs on the server increased. E.g. more database queries were made to populate all the dropdown fields in a form. As every action in the

client caused a new page load, all processing was done again in the server even if only a small part of the page changed as a result.

AJAX (Asynchronous JavaScript and XML) is a web development technique that allows interacting with the server in the background. Applications with rich interactions started to utilize background communication to decrease the load on the server. E.g. if dropdown options were dependent on another field it was possible to load only the values for that particular dropdown and update it in place instead of reloading the whole page.

Single page applications (SPA) push page loads to the bare minimum. Frontend is loaded once in the beginning and the rest of the communication with the server happens in the background. Even navigating to another URL is handled by the client by showing and hiding visible elements in the page.

4.4 OpenAPI

OpenAPI specification defines a standard to describe RESTful APIs. OpenAPI definition can be used by tools to generate human readable documentation, client libraries, server code and testing tools. OpenAPI documents are defined in JSON or YAML format, but requests and responses can be of any type. Code Block 1 shows an example of an OpenAPI document which describes an endpoint used to create a product.

```
{
    "openapi":"3.0.0",
    "paths":{
       "/products":{
          "post":{
             "operationId":"createProduct",
             "parameters":[],
             "requestBody":{
                "required":true,
                "content":{
                   "application/json":{
                      "schema":{
                         "$ref":"#/components/schemas/CreateProductModel"
                      }
                   }
                }
             },
```

```
            "responses":{
               "201":{
                  "description":"Product created successfully"
               }
            }
         }
      },
      "info":{
         "title":"Products example",
         "version":"1.0"
      },
      "components":{
         "schemas":{
            "CreateProductModel":{
               "type":"object",
               "properties":{
                  "name":{
                     "type":"string"
                  }
               },
               "required":[
                  "name"
               ]
            }
         }
      }
}
```

Code Block 1. OpenAPI document example

At a minimum OpenAPI document contains the version number indicated by the openapi element, generic description in the info element and paths describing the endpoints. Schemas describe request and response payloads and can be referenced from the endpoint definitions. Various other elements exist to describe HTTP elements like query strings, headers and content types. In addition, metadata like contacts, servers and licenses among others are available. (Swagger 2021)

Given that just the basic CRUD operations consist of 5 endpoints (list, get one, create, update, delete) per entity and any real-world application has multiple entities, the number of endpoints would be dozens even in a relatively trivial application. Being able to create the client library and all the data models used in response and request bodies would reduce the amount of manual work tremendously.

5 Existing full stack technology comparisons

The Web is bulging with full stack technology comparison articles. Yet, searching “most productive backend/frontend framework” only finds articles listing the “best” or “top” frameworks. Based on the first 50 comparison articles (Appendix 1) of the backend search, articles are highly focused on certain technologies. 34 frameworks were mentioned in total, but only eight of them were mentioned at least ten times. Table 6 shows the most mentioned frameworks. Productivity was one of the mentioned advantages for many frameworks.

Table 6. Most mentioned backend frameworks in comparison web articles

Framework	**Proportion of articles**	**Most common mentioned advantages**
Django	92%	Feature rich 46% Security 43% Scalable 43% Productivity 39%
Laravel	80%	Documentation 28% Feature rich 23% Templates 20%
Ruby on Rails	80%	Productivity 45% Community 33% Extensible 23%
Express	70%	Performance 40% Extensible 20%
Spring (Boot)	64%	Easy to set up 41% Feature rich 25%
Flask	54%	Flexible 33% Documentation 22%
ASP.NET Core	34%	Productivity 47% Performance 41% Maintenance 29% Tools 24%
CakePHP	22%	CRUD development 27% Productivity 27%

Articles were really shallow. Most had only 5-10 sentences describing each framework. One third had also around five bullet points for advantages, disadvantages and/or key features. Only one fifth had more information which was most often listing companies and products using the technology.

Articles didn't explain how productivity or other advantages were determined. Some claims were undoubtedly wrong. E.g. good performance was often mentioned as an advantage for Express (Clark, 2020; JumpGrowth, n.d.; RaftLabs, 2021; Safonov, 2021) although TechEmpower benchmark (TechEmpower, 2021) ranks it in 94th place (of 122) far behind many other frameworks mentioned in the articles like ASP.NET Core (8th) and Spring (51st).

Frontend comparisons had much less variety. Frontend comparison article search also produced millions of results. Based on the first 30 comparison articles (Appendix 2) comparisons were mostly between Angular, React and Vue.js. Svelte was included half of the time along with some older frameworks occasionally. Structure of the articles was similar to backend articles with short descriptions and bullet points for advantages and disadvantages. Most mentioned advantages are listed in Table 7.

Table 7. Most mentioned advantages of frontend frameworks

Framework / Library	**Most common mentioned advantages**
Angular	2-way data binding 33% Community 30% Feature rich 20% Reusable components 20%
React	Virtual DOM 47% Reusable components 43% Browser development tools 20% 1-way data binding 20%
Vue.js	Documentation 53% Simplicity 40% Small 23% 2-way data binding 20%

Frontend articles didn't describe the reasoning behind advantages either. Angular, React and Vue.js are all able to reuse components which makes it a non-distinguishing feature in this set of technologies. Still, it wasn't seen as an advantage equally for each framework. Same could be said for many features available in most of the compared technologies, like 2-way data binding and TypeScript support. As existing technology comparisons are shallow and don't provide reasoning behind the claims they make, the data isn't considered high enough quality to be used in this thesis.

6 Backend technologies

6.1 Programming languages

Programming language popularity ranking websites *Tiobe* (2020) and *PYPL PopularitY of Programming Language* (2020a) were used to compare popularity in general. Number of packages for each platform were retrieved from *Modulecounts* (2020). *Stackshare*, a website where companies and individuals share technologies they are using, was used to determine the most popular frameworks by inspecting technologies in categories *Frameworks (Full Stack)* (Stackshare 2020a) and *Microframeworks (Backend)* (Stackshare 2020b). Frameworks were then paired with corresponding programming languages to get a list of programming languages commonly used for backend development as presented in Table 8.

Table 8. Popular programming languages having at least one popular web framework

Language	**TIOBE**	**PYPL**	**Packages**	**Most popular Stackshare framework**	
Python	3	1	273k	1	**Django**
JavaScript	7	3	1.4M	4	**Express**
Java	2	2	366k	6	**Spring**
C#	5	4	232k	2	**ASP.NET**
PHP	8	5	290k	3	**Laravel**
TypeScript	46	10	1.4M	4	**Express**
Ruby	13	14	163k	5	**Ruby on Rails**
Elixir	50+	-	11k	14	**Phoenix**
Groovy	12	23	366k	22	**Grails**

For further investigation, programming languages being in top 10 in either Tiobe ranking or PYPL index and having a framework listed in Stackshare were selected. Python, JavaScript/TypeScript, Java, C# and PHP fulfilled those criteria.

Development tool availability for selected languages was determined by inspecting Stackshare Build/Test/Deploy (Stackshare 2020c) and PyPL IDE (PyPL, 2020b) indices. Stackshare stack count and PyPL rankings can be seen in Table 9 among the primary languages and operating system support advertised by the tools.

Table 9. Most popular development tools

IDE	SH stacks	PYPL ranking	Primary Language	Windows	Linux	Mac
VS code	57200	4	JavaScript/TypeScript	Yes	Yes	Yes
Visual Studio	20800	1	C#	Yes	No	Yes
IntelliJ IDEA	20400	6	Java	Yes	Yes	Yes
PyCharm	11800	5	Python	Yes	Yes	Yes
PhpStorm	7670	13	PHP	Yes	Yes	Yes
WebStorm	7000		JavaScript/TypeScript	Yes	Yes	Yes
Eclipse	1840	2	Java	Yes	Yes	Yes
NetBeans	550	7	Java	Yes	Yes	Yes
Rider	221		C#	Yes	Yes	Yes

All languages had similar tools available. Each had an extension available for VS Code. IntelliJ IDEA, PyCharm, PhpStorm, WebStorm and Rider are developed by the same company, JetBrains, which would give a reason to believe they provide similar development experience.

Support on cloud platforms was determined by inspecting three largest cloud providers; AWS, Azure and Google Cloud Platform. All cloud providers advertised supporting all selected languages in their PAAS offerings (AWS, 2020; Azure, 2020; Google Cloud, 2020).

6.2 Frameworks & libraries

List of frameworks were collected from Stackshare top lists *Frameworks (Full Stack)* (Stackshare 2020a) and *Microframeworks* (Backend) (Stackshare 2020b), “awesome lists” for each language/platform (see Appendix 3) and making web searches with patterns “*language* rest api” and “*language* web framework”.

Total of 87 frameworks were found. Vitality of the frameworks were evaluated based on the latest *release date*, the *number of contributors* in their *Github* (https://github.com) repositories, *Stackshare (*https://stackshare.io*)* stack count and *Stack Overflow* (https://stackoverflow.com) question count. Relative ranking within this set of technologies was determined for contributor count, Stackshare stack count and Stack Overflow question count. Technologies having latest release within six months and being in the top 50 in all of the relative rankings were considered having a vital community and adequate for further inspection. Lastly, code generators and similar technologies were filtered out as not being enough general purpose. Full list of technologies and vitality evaluation results are presented in Appendix 4. 29 frameworks and libraries fulfilled all criterion and were selected for the feature analysis.

7 Backend features

7.1 Introduction

Frameworks and libraries were analyzed starting with the most popular for each language and platform. ASP.NET repository was found to be archived and the successor ASP.NET Core was used for C# instead. Official documentation pages were browsed through and mentioned features were collected. A total of 46 features were identified in categories *routing*, *middleware*, *handlers*, *authentication*/*authorization*, *logging*, *OpenAPI*, *messaging* and *tasks*. Figure 1 shows where various features could appear in the HTTP request processing pipeline.

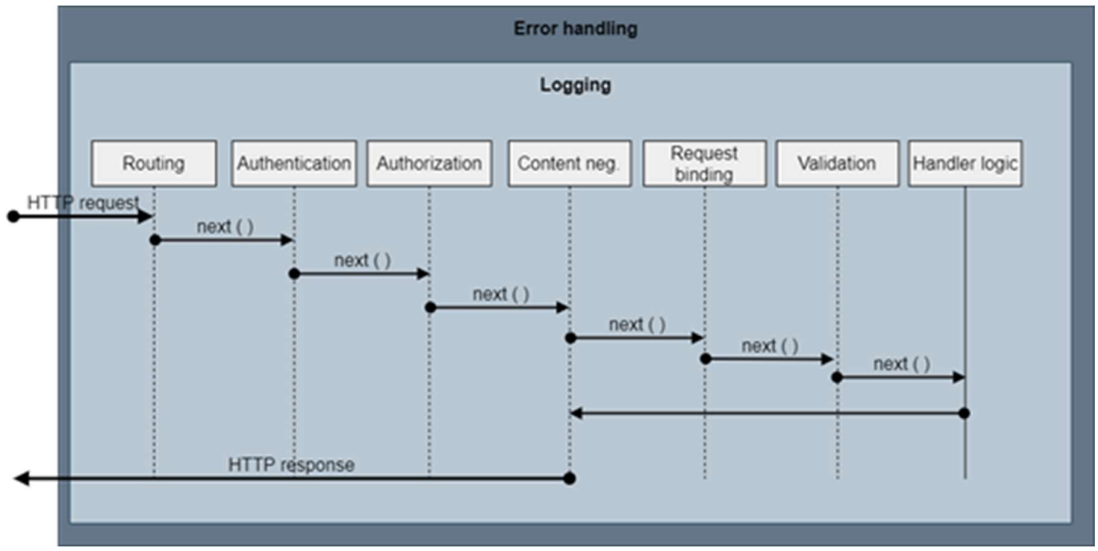

Figure 1. Example of HTTP request handling pipeline

Features non relevant for RESTful API development such as *cookies* (authorization headers preferred), *form handling* (client's responsibility), *session handling* (REST is stateless) and *GraphQL* (alternative to RESTful APIs), were left out from the list.

7.2 Routing

Routing is the process of mapping HTTP request to handling code. Common pattern was adapting RESTful routing conventions of using **HTTP method** and **path** to describe operations. One common form of routing was mapping paths with **parameter** placeholders to functions and using regular expressions to **constrain parameter** values. Regular expressions were also used for **wildcard parameters**. Code Block 2 demonstrates how mapping routes to functions and using regular expression constraints and wildcards can be used in Laravel.

```
Route::prefix('api')->group(function () {
    Route::get('products/{id}', function ($id) {
      // return single product
    })->where('id', '[0-9]+')->name('product-details');

    Route::get('files/{path}', function ($path) {
        // ...
    })->where('path', '.*');
});

Route::domain('{client}.example.com')->group(function () {
    Route::get('products', function ($client) {
```

```
        // ...
    });
});
```

Code Block 2. Laravel routing features

Another common routing method was using classes (often called controllers or resources) and annotating their methods with route patterns. Using statically typed language also often allowed using types to constrain parameter values. Code Block 3 demonstrates these characteristics.

```
app.UseRouting();

app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/products", async context =>
    {
        // get products
    });
    endpoints.MapControllerRoute(
      "default route",
      "api/{controller}/{action}",
      new { controller = "Home", action = "GetAll" });
    endpoints.MapFallbackToFile("/index.html");
});

[Route("api/products")]
[ApiController]
public class ProductsController : ControllerBase
{
  [HttpGet("{id}", Name = "product-details")]
  public Product GetDetails(Guid id)
  {
      // ...
  }

  [HttpGet("files/{*path}")]
  public FileResult GetFile(string path)
  {
    // ...
  }
}
```

Code Block 3. ASP.NET Core routing features

Routes could often have **prefixes** (Code Block 2) for easier segregation e.g. from frontend paths. **Subdomain** routing might become valuable in multi tenant applications. Code Block 2

demonstrates how subdomain is captured to a variable. In Code Block 4 subdomain is used to route to different handlers.

```
$routes->add('products', 'Products::list_client1', ['subdomain' =>
'client1']);
$routes->add('products', 'Products::list_client2', ['subdomain' =>
'client2']);
```

Code Block 4. Subdomain routing in CodeIgniter

Naming routes (Code Block 2 & Code Block 3) allows **generating URLs** (reverse routing) by name and parameter values (Code Block 5) instead of building them manually. This improves maintainability as configuration is in one place.

```
// Laravel
$url = route('product-details', ['id' => 123]);

// ASP.NET Core
var url = Url.Link("product-details", new { id = 123 });
```

Code Block 5. Reverse routing in Laravel and ASP.NET Core

Semantic versioning is an important feature when all client application (such as mobile applications) updates cannot be controlled and multiple versions of an API must be live at the same time. Code Block 6 shows how handlers could be versioned.

```
server.get('/products/:id', restify.plugins.conditionalHandler([
  { version: '1.0.0', handler: getProductByIdV1 },
  { version: '2.0.0', handler: getProductByIdV2 }
]));
```

Code Block 6. Semantic routing in Restify

Static file routing (Code Block 7) is used to provide files from filesystem. When the backend is also serving SPA client files, static files would include scripts and stylesheets.

```
app.UseStaticFiles(new StaticFileOptions
{
    FileProvider = new PhysicalFileProvider("path/to/files"),
    RequestPath = "/assets"
});
```

Code Block 7. Static file provider in ASP.NET Core.

To prevent a single client making too many calls, **rate limiting** (Code Block 8) can be applied.

```
RateLimiter::for('global', function (Request $request) {
    return Limit::perMinute(1000);
});
Route::middleware(['throttle:global'])->group(function () {
    Route::get('/example, function () {
        //
    });
});
```

Code Block 8. Rate limiting in Laravel

Redirect at router level (Code Block 9) removes the need to create handlers for such simple tasks. **Fallback** route can be used to forward frontend paths to the client application (Code Block 3 & Code Block 9) or handle 404 (Not Found) errors.

```
Route::redirect('/old', '/new');

Route::fallback(function () {
    // ...
});
```

Code Block 9. Redirect and fallback routing in Laravel

7.3 Middleware

Middlewares process requests before they are passed to the handlers and also responses before they are returned to the caller. Various features like authentication and content negotiation can be considered to be just predefined specialized middlewares. Cross-Origin Resource Sharing (CORS) is a mechanism to whitelist origins that are permitted to access resources (MDN Web Docs 2021d). Only simple configuration was often needed to enable **CORS** (Code Block 10).

```
app.enableCors(/* configuration */);
```

Code Block 10. Enabling CORS in NestJS

Middlewares commonly had the ability to execute code **before** and **after** executing the handler. More specific examples were seen to **filter**, **terminate** and **decorate** HTTP requests. Various approaches were used to achieve these behaviors. Middleware functions as seen in Code Block 11 were common.

```
var exampleMiddleware = function (req, res, next) {
  // before / filter / decorate / terminate
  next()
  // after
}

app.use(exampleMiddleware)
```

Code Block 11. Middleware functions in Express

Middleware classes (often called filters) with certain methods were also common. Third approach was using lifecycle hooks. Some technologies supported more than one way to define middleware-like functionality. All three approaches are demonstrated in Code Block 12.

```
// Middleware function
app.Use((context, next) =>
{
    // before / filter / decorate / terminate
    next.Invoke();
    // after
});

// Middleware class
public class ExampleFilter : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        base.OnActionExecuting(context);
        // before / filter / decorate / terminate
    }
    public override void OnResultExecuting(ResultExecutingContext context)
    {
        base.OnResultExecuting(context);
        // after
    }
}


// Lifecycle hooks
public abstract class ExampleControllerBase : Controller
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        base.OnActionExecuting(context);
        // before / filter / decorate / terminate
    }
    public override void OnActionExecuted(ActionExecutedContext context)
    {
        base.OnActionExecuted(context);
```

```
        // after
    }
}
```

Code Block 12. Middleware in ASP.NET Core

Middlewares could also have **parameters** (Code Block 13**)** for better reusability.

```
$app->add(new AuthorizeMiddleware('admin'));
```

Code Block 13. Middleware parameterization in Slim

Various ways were used to define **conventions** which routes should middlewares be applied to. Most common was applying middleware globally as seen in Code Block 13. Applying middleware to paths by prefix was also common along with lesser common exclusion as seen in Code Block 14.

```
// Slim
$app->group('/api', function (RouteCollectorProxy $group) {
    // ...
})->add($middleware);

// NestJS
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(MyMiddleware)
      .exclude(
        { path: 'products', method: RequestMethod.PUT },
      )
      .forRoutes('products');
  }
}
```

Code Block 14. Middleware applying conventions in Slim and NestJS

Most technologies applied middlewares in the order they were defined, but also very granular control was seen like the most powerful example seen in Code Block 15.

```
$middleware = new \App\Middleware\ExampleMiddleware;
$middlewareQueue->add($middleware);       // last
$middlewareQueue->prepend($middleware);   // first
$middlewareQueue->insertAt(2, $middleware);
$middlewareQueue->insertBefore(
    'App\Middleware\OtherMiddleware',
    $middleware
```

```
);
$middlewareQueue->insertAfter(
    'App\Middleware\OtherMiddleware',
    $middleware
);
```

Code Block 15. Defining middleware order in CakePHP

Content negotiation allows using the same handler for various media types such as JSON and XML. Request body is deserialized to an object based on *Content-Type* header and the handler result object is serialized to the response body based on the *Accept* header. Some technologies had this feature enabled by default and didn't require any additional work. Some required annotating supported content types (Code Block 16).

```
@Produces(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_XML)
@Consumes(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_XML)
public class ExampleResource {

    @GET
    public ExampleResponse fetch() {
        return new ExampleResponse();
    }

    @POST
    public Response add(ExampleRequest example) {
        // ...
    }
}
```

Code Block 16. Defining content types in Dropwizard

Displaying detailed error messages in the response could reveal sensitive information. Hostile users could use information like library versions to search for known vulnerabilities and target their attack better leading to a possible security breach. Global and **automatic error handling** is used to hide sensitive information when an error happens. Code Block 17 shows how the error handler is registered in Flask.

```
@app.errorhandler(HTTPException)
def handle_exception(e):
    # ...
```

Code Block 17. Registering error handler in Flask

7.4 Handler

Handler is code taking requests as input and producing responses. The following features were identified in the handler category although many technologies had them implemented as middleware functionalities.

Dependency injection is a design pattern where services are injected into components instead of them creating instances by themselves (Wikipedia, 2021). Decreased coupling improves reusability, testability and maintainability. Services can be injected into constructors, functions and properties. First two are demonstrated in Code Block 18.

```
services.AddTransient<IExampleService, ExampleService>();
services.AddSingleton<AnotherService>();

public class ExampleController : Controller
{
    private readonly IExampleService service;
    public ExampleController(IExampleService service)
    {
        this.service = service;
    }

    public ExampleModel GetData([FromService]AnotherService service)
    {
        // ...
    }
}
```

Code Block 18. Dependency injection in ASP.NET Core

Schema based request binding allows converting request values to specific types. This was often paired with **automatization** by specifying types for handler function parameters (Code Block 19).

```
class Product(BaseModel):
    name: str
    description: str
    price: float

@app.post("/products/")
async def create_product(product: Product):
    # ...
```

Code Block 19. Request binding and validation in FastAPI

Request binding could often be paired with **schema-based validation** and to minimize manual work the whole validation process could be **automated** (Code Block 20).

```
export class ProductModel {
  @IsNotEmpty()
  name: string;
  // ...
}

// add validation pipe globally
app.useGlobalPipes(new ValidationPipe());

@Put(':id')
updateProduct(@Param('id') id: string, @Body() product: ProductModel) {
  // ...
}
```

Code Block 20. Request binding and validation in NestJS

Output cache stores the result of the handler for efficient access. Output cache was most often enabled by annotating the handler. For more granular caching **key-value caches** were used. Both cache types are demonstrated in Code Block 21.

```
# Django
@cache_page(5 * 60)
def most_read_news(request):
    # ...

// Symfony
$value = $cache->get('key', function (ItemInterface $item) {
    $item->expiresAfter(5*60);
    $computedValue = '';
    return $computedValue;
});
```

Code Block 21. Output cache in Django and key-value cache in Symfony

Cache invalidation is a common problem. Marking **cache dependencies** with e.g. tags (Code Block 22) was one way to invalidate related cache values.

```
Cache::tags(['example-tag'])->put('key', $value, $seconds);

Cache::tags(['example-tag'])->flush();
```

Code Block 22. Cache dependencies in Laravel

7.5 Authentication & Authorization

Authentication is the process of identifying the user and authorization the process of determining whether they have rights to perform operations. Technologies supported various login methods like **username+password**, **social media accounts** and **Azure AD**. Code Block 23 shows Google login flow in Laravel.

```
'google' => [
    'client_id' => '...',
    'client_secret' => '...',
    'redirect' => 'http://example.com/callback-url',
],

// …

Route::get('/auth/redirect', function () {
    return Socialite::driver('google')->redirect();
});

Route::get('/auth/callback', function () {
    $user = Socialite::driver('google')->user();
    $email = $user->getEmail();
    // ...
});
```

Code Block 23. Google login flow in Laravel

JSON Web Token (JWT) is a standard way of securely sharing user information between client and server. JWT payload contains encoded claims which can be used to grant access to resources. (JWT, 2021). Code Block 24 show how JWT is created in Vert.x.

```
JWTAuthOptions config = new JWTAuthOptions()
  .setKeyStore(new KeyStoreOptions()
  .setPath("keystore.jceks")
  .setPassword("secret"));

AuthenticationProvider provider = JWTAuth.create(vertx, config);

String token = provider.generateToken(new JsonObject().put("key", "value"),
new JWTOptions());
```

Code Block 24. Creating JWT in Vert.x

Policies are a way to control access by defining rules. Rules could contain inspecting claims in the JWT data for example. Code Block 25 shows how email claim is checked in an ASP.NET Core custom policy.

```
services.AddAuthorization(options =>
{
    options.AddPolicy("OnlyExampleDotCom", policy =>
        policy.RequireAssertion(ctx =>
            ctx.User.Claims.First(c => c.Type ==
ClaimTypes.Email).Value.EndsWith("@example.com")));
});

[Authorize(Policy = "OnlyExampleDotCom")]
public class ProductController : ControllerBase { }
```

Code Block 25. Custom policy in ASP.NET Core

Roles are used to grant access to resources to groups of users. **Conventions** allow applying authentication and authorization rules to a variety of endpoints with ease. This removes repetitive work as not every endpoint needs to be handled separately. Code Block 26 shows how role-based authorization can be applied granularly in ASP.NET Core.

```
[Authorize(Roles = "reader")] // apply to all subclasses
public abstract class ExampleControllerBase : ControllerBase {}

[Authorize(Roles = "contributor")] // apply to all actions in this class
public class ExampleController : ControllerBase
{
  [Authorize(Roles = "admin")] // apply to single action
  [HttpGet("")]
  public IActionResult SomeAction() {}
}

// apply globally
services.AddControllers(options =>
{
    options.Filters.Add(typeof(AuthorizeFilter));
});

//
[AllowAnonymous]
public IActionResult Login(string username, string password)
{
    // ...
}
```

Code Block 26. Middleware conventions in ASP.NET Core

7.6 Logging

Logging is a crucial part of problem solving in any application. While developing it might be useful to log all the details. In a production environment the amount of data could be reduced by logging only the errors. **Logging levels** control how much information is logged. Code Block 27 shows how logging level is set in FastAPI.

```
logger.setLevel(logging.DEBUG)
```

Code Block 27. Setting log level in FastAPI

7.7 OpenAPI

OpenAPI schemas can be inferred from the endpoints or explicitly defined. Inferring decreases the manual work and can be one of the biggest time savers when the number of endpoints is large. As with schemas, also the **document API** can be automatically generated or explicitly defined. Code Block 28 demonstrates how annotations are used to infer the schema and automatically generate the document API in NestJS.

```
@Controller({ path: 'products' })
@ApiTags('products')
export class ProductController {

  @Get()
  @ApiOkResponse({
    type: [ProductModel]
  })
  async listProducts(): Promise<ProductModel[]> {
    // ...
  }
}

export class ProductModel {

  @ApiProperty()
  id: string;

  @ApiProperty()
  name: string;

  // ...
}

const options = new DocumentBuilder()
```

```
  .setTitle('My API')
  .setVersion('1.0')
  .build();

const document = SwaggerModule.createDocument(app, options);
SwaggerModule.setup('api', app, document);
```

Code Block 28. OpenAPI configuration in NestJS

7.8 Messaging

Technologies had two kinds of messaging. **Push-based** methods like WebSocket or SSE (Server-Sent Events) were used to send messages from server to client. This removes the need for clients to constantly poll the server for new information. Code Block 29 demonstrates how messages are broadcasted to multiple clients in ASP.NET Core.

```
public class ExampleHub : Hub
{
    public async Task SendMessage(string message)
    {
        await Clients.All.SendAsync("ReceiveMessage", message);
    }
}

services.AddSignalR();

app.UseEndpoints(endpoints =>
{
    endpoints.MapHub<ExampleHub>("/example");
});
```

Code Block 29. Message broadcasting in ASP.NET Core

For messaging happening within the server application **events** were used**.** Although events are available as language features and libraries, some technologies had their own event system. Code Block 30 shows an example of event emitting and subscribing in Flask.

```
my_signals = Namespace()
example_signal = my_signals.signal('example')

# subscribe
def handle_signal(sender, template, context, **extra):
    # ...

example_signal.connect(handle_signal, app)
```

```
# emit
def save(self):
    example_signal.send(self)
```

Code Block 30. Events in Flask

7.9 Tasks

Background tasks could be **scheduled** either with cron-like expressions to happen at certain times or by certain intervals. Code Block 31 demonstrates both ways as they are used in Spring.

```
public class ScheduledFixedRateExample {
    @Scheduled(fixedDelay = 1000)
    public void handleFixedDelayTask() {
        // ...
    }

    @Scheduled(fixedRate = 1000)
    public void scheduleFixedRateTask() {
        // ...
    }

    @Scheduled(cron = "0 */5 * * * *")
    public void handleCronTask() {
        // ...
    }
}
```

Code Block 31. Scheduled tasks in Spring

8 Frontend frameworks & libraries

In addition to well-known popular JavaScript/TypeScript SPA technologies Angular, React, Svelte and Vue.js (State of js, 2020) Vue.js based frameworks Vuetify and Quasar Framework were studied. WebAssembly frameworks and libraries were also searched using phrases “WebAssembly SPA” and “WebAssembly framework”. Many programming languages had compilers to WebAssembly. E.g.

- Emscripten (C, C++)
- Rust WebAssembly (Rust)
- AssemblyScript (TypeScript)
- Kotlin Native (Kotlin)

- SwiftWasm (Swift)

Only a few had SPA capabilities such as components and routing. For these vitality was determined by Stackshare (https://stackshare.io) stack count and Stack Overflow (https://stackoverflow.com) question count. Results are shown in Table 10.

Table 10. WebAssembly frameworks.

Framework	Language	Stackshare stacks	Stack overflow questions
Blazor	C#	233	6239
Yew	Rust	8	18
Seed	Rust	0	1
Vugu	Go	0	0
Bolero	F#	0	7
Vecty	Go	0	0

Only Blazor was considered having a vital community and was selected for comparison along with mentioned JavaScript/TypeScript technologies.

9 Frontend features

9.1 Components

Current SPA technologies are component based. Components can have **state** (internal data) and take values as input **parameters from parent components** (often called props). Passing **data to the parent** component can be done in an event-like manner. Values can be derived from other values to make reactive **computed properties**. Various lifecycle events are used to handle component **creation**, **update** and **destroyal**. Code Block 32 shows an example of a component in Vue.js.

```
<template>
  Markup
</template>
```

```
<script lang="ts">
export default {
  setup (props, { emit }) {
    // data from parent
    const number = props.data;

    // data to parent
    const valueChanged = (newVal: string) => {
      emit('change', newVal)
    }

    // state
    const firstName = ref('');
    const lastName = ref('');

    // computed state
    const name = computed(() =>`${firstName.value} ${lastName.value}`);

    // lifecycle event
    onMounted(() => {
      // dom ready
    })

  }
}
</script>
```

Code Block 32. Example of Vue.js component

Components can be **reused** like custom html elements. Ingoing data and events are defined as attributes with special syntax to separate them from normal HTML attributes. Code Block 33 shows how components are used in various frameworks.

```
// Angular
<MyComponent [data]="123" (change)="handleChange" />

// Blazor
<MyComponent Data="123" OnValueChanged="@HandleChange" />

// React
<MyComponent data={123} onchange={handleChange} />

// Svelte
<MyComponent data={123} on:change={handleChange} />

// Vue
<MyComponent :data="123" @change="handleChange" />
```

Code Block 33. Using components in various frameworks

Data can also be passed to another component deep down in the component hierarchy. This removes the need to define input parameters in the components between which increases decoupling and reusability. Code Block 34 shows how data is passed deep down the component hierarchy in Blazor.

```
// pass value
<CascadingValue Value="@ExampleValue" Name="someData">
  // ...
</CascadingValue>

// read value
@code {
    [CascadingParameter(Name = "someData")]
    protected ExampleType ExampleParameter { get; set; }
}
```

Code Block 34. Passing data deep down in Blazor

DOM reference is used when component needs access to an element in the template. Element can then be manipulated like any other HTML node. Common use cases would include rendering UI widgets such as maps or charts. Code Block 35 shows how DOM reference is used in Vue.js

```
<template>
  <div ref="map"></div>
</template>
<script>
  import { ref, onMounted } from 'vue'

  export default {
    setup() {
      const map = ref(null)
      onMounted(() => {
        // render map
      })

      return {
        map
      }
    }
  }
</script>
```

Code Block 35. DOM reference in Vue.j

Sometimes the desired component cannot be determined in the development time. **Dynamic component selection** can be used to select rendered component at runtime. Code Block 36 shows how component is dynamically rendered in Blazor.

```
@componentToDisplay

@code {
    var componentToDisplay = someCondition ? ComponentA : ComponentB;

    static readonly RenderFragment ComponentA = _ =>
    {
        <ComponentA />
    };

    static readonly RenderFragment ComponentB = _ =>
    {
        <ComponentB />
    };
}
```

Code Block 36. Dynamic component selection in Blazor

9.2 Templates

Technologies had two ways to handle templating. In Angular, Svelte and Vue.js HTML was decorated with special attributes and similar DSL. Blazor and React had it the other way around and HTML was placed inside the code.

Interpolation is the process of evaluating and replacing expressions with their values within string literals. To prevent XSS (cross site scripting) attacks, strings within templates are encoded by default. Special methods are used to render **raw HTML**. **Conditionals** are used to change the rendered elements based on some condition. **Loops** are used to render elements for collection items. Basic templating features are shown in Code Block 37.

```
<span>Today is {{ formatDate(today) }}</span>
<div [innerHTML]="text"></div>
<div>
  <span *ngIf="someCondition; else else_content">This is rendered when
condition is true</span>
  <ng-template #else_content>This is rendered when condition is false</ng-
template>
</div>
```

```
<ul>
  <li *ngFor="let item of items">{{ item.text }}</li>
</ul>
```

Code Block 37. Angular template example

Templates can have dynamic **attributes**. Various ways are used to make attributes evaluable. Code Block 38 shows how attribute bindings are used in Angular and Blazor.

```
<!-- Angular -->
<img [src]="product.imageUrl" />

<!-- Blazor -->
<img src="@product.imageUrl" />
```

Code Block 38. Attribute binding in Angular and Blazor

Binding input **values** and **events** work the same way. **Event modifiers** can restrict the conditions when events are fired. Code Block 39 shows value and event binding in React, Svelte and Vue.js.

```
<!-- React -->
<input type="text" value={this.state.name} onChange={this.nameChanged} />

<!-- Svelte -->
<input bind:value={name} on:change={nameChanged} >

<!-- Vue.js -->
<input type="text" :value="name" @keydown.enter="nameChanged" />
```

Code Block 39. Value and event binding in React, Svelte and Vue.js

Class bindings have special processing as many values could be desired. Object notation is used to define conditions when classes should be applied. Array notation is used to define multiple classes which are always applied. Code Block 40 shows both ways.

```
<!-- Angular -->
<div [ngClass]="{ active: selected === 'first', another: foo === 'bar' }"></div>

<!-- Vue.js -->
<div :class="['first', 'second']"></div>
```

Code Block 40. Class bindings in Angular and Vue.js

Two-way data binding is a pattern where the value is synchronized between state and editable UI controls. It can be thought as combining value and change event binding as seen in Code Block 39. It reduces boilerplate code as event handling code isn't needed. Code Block 41 shows how two-way data binding is used in Blazor.

```
<input type="text" @bind="name" />

@code {
  private string name;
}
```

Code Block 41. Data binding in Blazor

Transclusion is a feature where a component can define an area in the template where inner content is placed. E.g. layout component can define the area where the main content is placed. Code Block 42 shows how transclusion is handled in React.

```
const Layout = (props) => {
  return (
    <div>
      <div class="menu">
      </div>
      <div class="main">
        {props.children}
      </div>
    </div>
  )
}

<Layout>
  <div>Main content</div>
</Layout>
```

Code Block 42. Transclusion in React

Multi transclusion allows defining multiple child content areas. E.g. layout component can define areas for sidebar and main content. Code Block 43 shows how multi transclusion is handled in Blazor.

```
<div class="menu">
  @Sidebar
</div>
```

```
<div class="main">
  @Content
</div>

@code {
    [Parameter]
    public RenderFragment Sidebar { get; set; }

    [Parameter]
    public RenderFragment Content { get; set; }
}

<ExampleComponent>
    <Sidebar>
      Sidebar content
    </Sidebar>
    <Content>
      Main content
    </Content>
</ExampleComponent>
```

Code Block 43. Multi transclusion in Blazor

Showing **validation** errors right after a component has lost focus improves user experience. Code Block 44 shows how minimal code is needed to validate an input in Quasar Framework.

```
<q-input v-model="name" :rules="[val => !!val || 'Field is required']" />
```

Code Block 44. Input validation in Quasar Framework

Pipes provide reusable formatting capabilities. **Directives** are used to attach functionality to elements having a certain attribute. Code Block 45 shows how pipes and directives are defined and used in Angular.

```
import {formatDate} from './utils'

@Pipe({name: 'formatDate'})
export class FormatDatePipe implements PipeTransform {
  transform(value: Date, format: string): string {
    return formatDate(value, format);
  }
}

@Directive({
  selector: '[exampleDirective]'
})
```

```
export class ExampleDirective {
  @Input('exampleDirective') arg: number;

  constructor(el: ElementRef) {
    // do something with the element
    el.focus();
  }

  @HostListener('mouseenter')
  onMouseEnter() {
    // react to mouse enter event
  }
}

{{ published | formatDate:'dd.MM.yyyy' }}
<div [exampleDirective]="123">...</div>
```

Code Block 45. Pipe and directive in Angular

9.3 Routing

Routing maps **paths** to components. Routes were defined either in a separate file or as part of the component. Paths could contain **parameters** which could then be captured in components. Code Block 46 shows how routes are defined in Svelte.

```
routes: [
{
    path: '/products/:id(\\d+)',
    name: 'PRODUCT_DETAILS',
    component: ProductDetails,
    props: (route) => {
      return {
        id: route.params.id,
      }
    }
  },
  { path: '/old', redirect: '/new' },
  { path: '*', component: NotFound } // wildcard
  // ...
]
```

Code Block 46. Route definitions in Svelte

If a route has a name defined (Code Block 46), paths can be **generated** from route definitions by providing the name and possible parameters. This provides better maintainability than manually

building paths as the configuration is in one place. Code Block 47 shows how URL is generated from route definition in Svelte.

```
<RouterLink to={{name: 'PRODUCT_DETAILS', params:{id: 123}}>
    Product details
</RouterLink>
```

Code Block 47. Generating links in Svelte

Routes can have **metadata** to provide values that are not part of the URL. Route **hooks** can be used to check whether a user has permissions to access a certain route for example. Code Block 48 shows how metadata is defined in Vue.js and how it handles events before and after routing.

```
const router = new VueRouter({
  routes: [
    {
      path: '/bugs',
      component: IssueList,
      meta: { issueType: 'bug' }
    },
    {
      path: '/stories',
      component: IssueList,
      meta: { issueType: 'story', adminOnly: true }
    }
  ]
})
router.beforeEach((to, from, next) => {
  if (to.matched.some(r => r.meta.adminOnly)) {
    // check role
  } else {
    next()
  }
})
router.afterEach((to, from) => {
  // ...
})
```

Code Block 48. Using routing metadata and hooks in Vue.js

Nested routes define a hierarchy of paths and components. It's useful when components use the same layout components. Layout can be defined in the parent route and content components as child routes. Code Block 49 shows how nested routes are defined in Angular and React.

```
// Angular
const routes: Routes = [
  {
    path: 'products/:id',
    component: ProductLayout,
    children: [
      {
        path: '',
        component: ProductDetails
      },
      {
        path: 'reviews',
        component: ProductReviews
      },
    ],
  },
];

<!-- React -->
<Switch>
  <Route path="/products/:id">
    <ProductLayout />
  </Route>
</Switch>
<!-- ProductLayout -->
<Switch>
  <Route exact path="/">
    <ProductDetails />
  </Route>
  <Route path="/reviews">
    <ProductReviews />
  </Route>
</Switch>
```

Code Block 49. Nested routes

Route **placeholders** can be used to define multiple components for the same route. Templates define areas (outlets or slots) where components can be placed and routes define which component goes to which area. Code Block 50 shows how route placeholders are used in React and Vue.js.

```
// React
const routes = [
  {
    path: "/products",
    sidebar: () => <ProductMenu />,
    main: () => <ProductList />
  }
];
```

```
...
<Switch>
{routes.map(route => (
  <Route
    path={route.path}
    exact={route.exact}
    children={<route.sidebar />}
  />
  <Route
    path={route.path}
    exact={route.exact}
    children={<route.main />}
  />
))}
</Switch>

// Vue.js
<router-view name="sidebar"></router-view>
<router-view></router-view>
...
const router = new VueRouter({
  routes: [
    {
      path: '/products',
      components: {
        default: ProductList, // router-view without name
        sidebar: ProductMenu  // router-view named "sidebar"
      }
    }
  ]
})
```

Code Block 50. Route placeholders

9.4 State management

State management is used to store state in a location accessible anywhere in the application. E.g. when user logs in their settings could be stored in a central location. Then the same data would be available in any component like header bar showing profile picture and profile page showing user information in an edit form. State management keeps data synchronized and reduces the need to load data separately in every component. Code Block 51 shows how state management is configured and used in Vue.js with vuex extension.

```
@Module({ namespaced: true, dynamic: true, store, name: 'user' })
export default class UserModule extends VuexModule {
  currentUser: UserProfile;
```

```
  @Action
  async updateProfile(profile: UserProfile) {
    // send to api
    // mutate state
    this.setProfile(profile);
  }
  @Mutation
  private setProfile(profile: UserProfile): void {
    this.currentUser = profile;
  }
}

export default class UserEditComponent extends Vue {
  // load module
  userStore = getModule(UserModule);
  // use getter to return value from store state
  get currentUser() {
    return this.userStore.currentUser;
  }

  async save() {
    // call store action
    await this.userStore.updateProfile({/* ... */ });
  }
}
```

Code Block 51. State management in Vue.js using vuex

9.5 Localization

In addition to simple **key-value** translations, parameter **interpolation** and **pluralization** were identified. Also, more advanced features **date** and **currency** formatting were seen. Code Block 52 shows how localization is handled in Vue.js with vue-i18n extension.

```
const messages = {
  simple: 'text',
  withNamedParameter: 'text {name}',
  withIndexedParameter: 'text {0}',
  simplePluralized: 'one item | many items',
  pluralizedWithNumber: 'no items | one item | {count} items',
}

$t('simple')
$t('withNamedParameter', { name: 'value' })
$t('withIndexedParameter', ['value'])
$tc('simplePluralized', 1)
$tc('pluralizedWithNumber', 5, { count: 5 })
```

```
$d(new Date(), 'long', 'fi-FI')
$n(100, 'currency', 'fi-FI')
```

Code Block 52. Localization in Vue.js using vue-i18n

9.6 UI components

UI component libraries provide prebuilt components with various aspects, like responsiveness, accessibility and styling, already taken into consideration (Figure 2). Various UI component libraries like Material Design (https://material.io) and Bootstrap (https://getbootstrap.com) had implementations and wrappers in studied technologies.

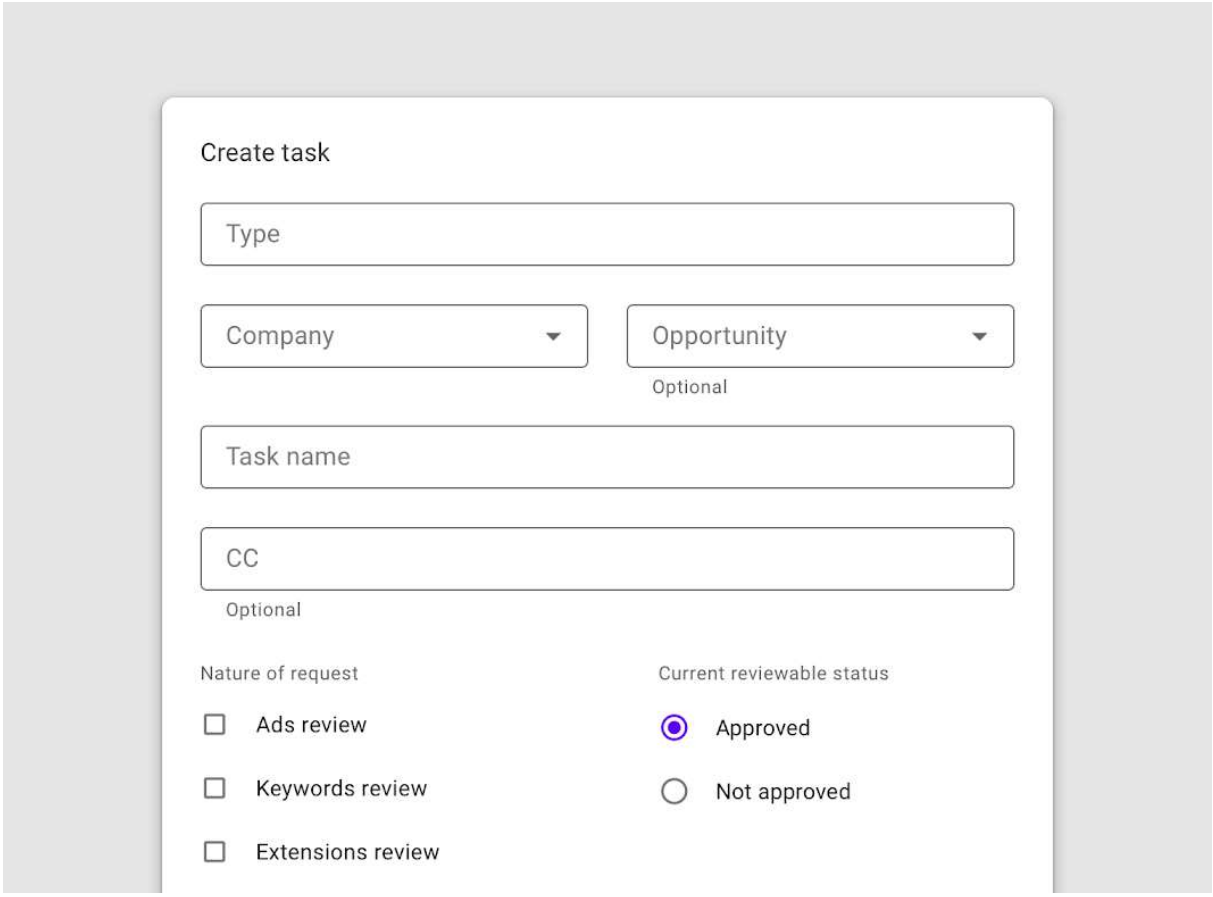

Figure 2. Example of Material Design form (Material Design, 2021)

10 Technology evaluation

10.1 Methodology

Each technology-feature combination was evaluated by first checking whether the official documentation had the feature described. If technology didn't have the feature described in the official documentation Google (https://google.com) searches were made in a format "technology feature" to find extensions and web articles. Searches were made in incognito mode to minimize bias from search history. Only the first page of search results was examined. Based on the search effort and clarity of the solution the effort score was determined for each feature and technology using the criterion presented in Table 11.

Table 11. Feature effort scoring criterion

Effort score	**Definition**
1	● Solution in the official documentation ● Obvious implementation using built-in concept
2	● Obvious solution using an extension ● Simple copy-paste solution from web search
3	● Partial solution from web search ● Complex implementation using built-in concepts
4	● No obvious solution found from web search ● Combining multiple extensions or built-in concepts

10.2 Backend

Full backend evaluation results are shown in Appendix 5. With a couple of exceptions routing features were quite well available. 41% of the technologies didn't have subdomain routing which could mean extra work in a multi-tenant application. 66% didn't have semantic versioning making those bad choices for APIs used by multiple client applications and versions. Routing feature summary is shown in Table 12.

Table 12. Backend routing features availability

Feature / Effort score	**1**	**2**	**3**	**4**
Method	90%	7%	0%	3%
Path	93%	7%	0%	0%
Parameters	93%	7%	0%	0%
Wildcard	93%	7%	0%	0%
Constraints	86%	7%	0%	7%
Prefix	52%	28%	14%	7%
Reversing	52%	28%	0%	21%
Subdomain	14%	28%	17%	**41%**
Semantic versioning	3%	17%	14%	**66%**
Static files	38%	48%	14%	0%
Rate limit	10%	72%	3 %	14%
Redirect	3%	93%	3%	0%
Fallback	52%	48%	0%	0%

Middleware features were quite widely available. Content-negotiation was the most dividing feature in this group. 48% of technologies didn't support it. Content negotiation would be a crucial feature if API is used by systems supporting varying content types. Middleware feature summary is shown in Table 13.

Table 13. Middleware features availability

Feature / Effort score	**1**	**2**	**3**	**4**
CORS	52%	48%	0%	0%
Before hook	86%	10%	3%	0%
After hook	86%	7%	3%	3%
Terminating	86%	10%	3%	0%
Parameters	34%	28%	10%	28%
Filters	86%	10%	3%	0%
Decorators	86%	10%	3%	0%

Table 13 (continued)

Feature / Effort score	**1**	**2**	**3**	**4**
Conventions	52%	21%	14%	14%
Order	72%	10%	3%	14%
Content negotiation	21%	10%	21%	**48%**
Error handling	66%	28%	7%	0%

Handler features had quite a lot of dispersion. Many of the features were not available at all in many of the technologies. 66% of the technologies didn't support request binding and 69% couldn't do it automatically. These groups had the same technologies with one exception supporting request binding, but not doing it automatically. 55% didn't support automatic validation. As request binding and validation are likely requirements for the majority of endpoints and APIs can contain vast amounts of endpoints, the manual work could be quite substantial in these technologies. 41% of technologies had tedious ways to handle response caches and 55% didn't support cache dependencies. These technologies might be a bad fit if caching is essential. Handler feature summary is shown in Table 14.

Table 14. Handler features availability

Feature / Effort score	**1**	**2**	**3**	**4**
Dependency injection	41%	34%	14%	10%
Schema based request binding	34%	0%	0%	**66%**
Automatic request binding	31%	0%	0%	**69%**
Schema based validation	48%	21%	21%	10%
Automatic validation	14%	21%	10%	**55%**
Response cache	28%	31%	**41%**	0%
In-memory cache	34%	66%	0%	0%
Cache dependencies	21%	28%	0%	**52%**

Authentication and authorization were well supported. Finding example code was a bigger problem in this group. Authentication and authorization feature summary is shown in Table 15.

Table 15. Authentication features availability

Feature / Effort score	**1**	**2**	**3**	**4**
Username + password login	31%	45%	**24%**	0%
Social media login	14%	41%	**31%**	**14%**
Azure AD login	3%	62%	**17%**	**17%**
Roles/groups	24%	48%	**24%**	3%
JWT	17%	59%	**21%**	3%
Policies	14%	48%	**31%**	7%
Conventions	21%	48%	**31%**	0%

OpenAPI support was poor in many technologies. 41% couldn't infer schemas from endpoints and 28% couldn't build documentation automatically. As with request binding and validation, manual work increases with every endpoint if OpenAPI creation cannot be automated. OpenAPI feature summary is shown in Table 16.

Table 16. OpenAPI features availability

Feature / Effort score	**1**	**2**	**3**	**4**
Schema creation	10%	28%	21%	**41%**
Documentation API	10%	48%	14%	**28%**

Logging was one of the best supported features. Messaging and task scheduling features were also well supported. These are summarized in Table 17.

Table 17. Logging, messaging and task scheduling features availability

Feature / Effort score	**1**	**2**	**3**	**4**
Automatic logging	62%	34%	3%	0%
Logging levels/Environments	76%	24%	0%	0%
Push-based messaging	38%	45%	10%	7%
Events	41%	59%	0%	0%
Task scheduling	24%	55%	17%	3%

Conclusion

Routing, middleware, authentication/authorization, logging, messaging and task features were generally well supported with either built-in functionality or as extensions. Quite a few features appeared to be unavailable in many technologies:

- route semantic versioning
- content-negotiation
- schema based request binding
- automatic request validation
- inferred OpenAPI documentation.

Static typing and schema-based request binding seemed to correlate with lower effort in validation and OpenAPI documentation. In the worst-case schema logic would have to be defined three times: request parsing, validation and OpenAPI endpoint definition.

Using familiar language has positive impact on productivity. Different languages and ecosystems also have their own advantages. Therefore, no single technology can be designated the best. Table 18 shows frameworks having the most of features per language.

Table 18. Top frameworks for each language.

		Features / Effort score				
Language	**Framework**	**1**	**2**	**3**	**4**	**Sum**
C#	ASP.NET Core	40	10	0	0	60
TypeScript	NestJS	31	17	1	1	72
PHP	Laravel	32	9	6	3	82
Python	FastAPI	24	17	7	2	87
Java	Spring	14	31	5	0	91

10.3 Frontend

Full frontend evaluation results are shown in Appendix 6. Frontend technologies had many different approaches. Angular, Blazor, Vuetify and Quasar Framework are considered frameworks. React and Vue.js are libraries. Svelte is advertised as a compiler. Frameworks had CLIs or similar ways to create projects with common features bundled. Vue.js also had CLI which offered features to include while creating a project (Figure 3). React and Svelte also had templates, but they didn't include additional libraries.

Figure 3. Selecting additional features in Vue CLI

Routing features were quite well available. Frameworks had routing built-in whereas libraries had it as an extension. Full routing results are shown in Table 19. Blazor defines routes in components which makes some routing scenarios more challenging.

Table 19. Routing features availability

Feature / Effort score	**1**	**2**	**3**	**4**
Path	71%	29%	0%	0%
Parameters	71%	29%	0%	0%
Meta	57%	29%	0%	**14%**
Hooks / Guards	71%	29%	0%	0%
URL generation	43%	57%	0%	0%
Nesting	57%	29%	0%	**14%**
Placeholders	57%	14%	14%	**14%**
Wildcards	71%	29%	0%	0%
Redirect	57%	29%	0%	**14%**

No substantial differences were found in the component features. All the features were either available out of the box or an easy workaround was found from web articles. Component feature summary is shown in Table 20.

Table 20. Component features availability

Feature / Effort score	**1**	**2**	**3**	**4**
Reusable components	100%	0%	0%	0%
State	100%	0%	0%	0%
Computed state	86%	14%	0%	0%
Change detection	71%	29%	0%	0%
Created / Mount hook	100%	0%	0%	0%
Destroy hook	100%	0%	0%	0%
Parent-child communication	100%	0%	0%	0%
Child-parent communication	100%	0%	0%	0%
Descendant communication	86%	14%	0%	0%

Table 20 (continued)

Feature / Effort score	**1**	**2**	**3**	**4**
Child/DOM reference	100%	0%	0%	0%
Dynamic component selection	71%	29%	0%	0%

Almost all template features were available in all technologies. React and Blazor didn't have directives or event modifiers. Template feature summary is shown in Table 21.

Table 21. Template features availability.

Feature / Effort score	**1**	**2**	**3**	**4**
Interpolation	100%	0%	0%	0%
Raw HTML	100%	0%	0%	0%
Conditionals	100%	0%	0%	0%
Loops	100%	0%	0%	0%
Value binding	100%	0%	0%	0%
Two-way data binding	86%	14%	0%	0%
Transclusion	100%	0%	0%	0%
Multi transclusion	100%	0%	0%	0%
Input validation	57%	43%	0%	0%
Pipes	57%	43%	0%	0%
Directives	71%	0%	0%	**29%**
Attributes	100%	0%	0%	0%
Class	100%	0%	0%	0%
Events	100%	0%	0%	0%
Event modifiers	57%	0%	14%	**29%**
Scoped styles	86%	14%	0%	0%

State management, localization, UI components and utils were also available for all technologies as summarized in Table 22.

Table 22. State management features availability.

Feature / Effort score	**1**	**2**	**3**	**4**
State management				
State	86%	14%	0%	0%
Modules	71%	29%	0%	0%
Localization				
Key-value	71%	29%	0%	0%
Parameters	71%	29%	0%	0%
Pluralization	57%	43%	0%	0%
Date time	71%	29%	0%	0%
UI / Utils				
Material Design / Bootstrap	29%	71%	0%	0%
Touch gestures	29%	57%	14%	0%
Session Storage / Local Storage	86%	14%	0%	0%
Meta	29%	71%	0%	0%

Tooling

All technologies had CLI or template to get started quickly. Technologies also supported hot reload which enables fast experimentation cycles. All compared technologies had extensions for VS Code enabling basic productivity features like autocomplete, refactoring and type checking. In addition, WebStorm had extensions for TypeScript technologies except Svelte (WebStorm, 2021). Blazor could also be developed with Rider and Visual Studio.

TypeScript technologies had browser developer tools which enable inspecting component hierarchy and state. Modifying state is possible to quickly test components with different data.

React and Vue.js developer tools also enabled inspecting, modifying and exporting state management data. Blazor didn't yet have browser developer tools.

Conclusion

All in all, differences weren't substantial between frontend technologies. TypeScript frameworks Angular, Vuetify and Quasar Framework had more features bundled than C# framework Blazor. UI libraries React and Svelte only had basic component and templating features built in. Vue.js, although being an UI library, had a CLI to bundle the most common libraries in the project template. By installing extensions for just routing, localization and state management libraries were on par with the frameworks. Feature availability is summarized in Table 23.

Table 23. Frontend technology feature availability summary.

		Features / Effort score				
Language	**Framework**	**1**	**2**	**3**	**4**	**Sum**
ES / TS	Quasar Framework	46	0	0	0	46
ES / TS	Vuetify	45	1	0	0	47
ES / TS	Vue.js	42	4	0	0	50
ES / TS	Angular	37	8	1	0	56
ES / TS	Svelte	27	18	1	0	66
C#	Blazor	32	7	1	6	73
ES / TS	React	22	22	0	2	74

11 Retrospective

Initial selection of compared technologies happened a year ago. When looking at the Tiobe (https://www.tiobe.com/tiobe-index/) and PyPL (https://pypl.github.io/PYPL.html) rankings again (Table 24) it can be seen that PHP's ranking has dropped on both indices, Ruby's in PyPL and Java's in Tiobe. Kotlin's ranking has risen in PyPL which could mean Kotlin is replacing Java in JVM development. Whether it's web development or something else is impossible to tell as neither of these indices measures web development popularity specifically. Python has risen to the top on Tiobe also. It could be that Python is replacing PHP and Ruby.

Table 24. Tiobe and PyPL ranking October 2020 and November 2021.

	Tiobe ranking		**PyPL ranking**	
Language	**Oct 2020**	**Nov 2021**	**Oct 2020**	**Nov 2021**
Python	3	**1**	1	1
JavaScript	7	7	3	3
Java	2	**3**	2	2
C#	5	5	4	4
PHP	8	**10**	5	**6**
TypeScript	46	46	10	10
Ruby	13	13	14	**16**
Kotlin	33	33	12	**11**

When comparing community metrics Github (https://github.com) repository contributors, Stackshare (https://stackshare.io) stack count and Stack overflow (https://stackoverflow.com) question count between October 2020 and November 2021 it can be seen that some technologies have increased popularity remarkable (Table 25). **FastAPI** has seen 52% increase in contributors, 900% in Stackshare users and 300% in Stack overflow questions. **NestJS** has had the biggest growth of Node.js technologies increasing contributors by 49%, Stackshare stacks 77% and Stack overflow questions 82%. **ASP.NET Core** also shows significant growth in contributors and Stackshare stacks with 58% and 63% respectively. **Laravel**'s Stackshare stacks increased by 57% being the only significant increase in PHP technologies. Biggest riser in Java technologies has been **Quarkus** with 47% increase in contributors, 106% in Stackshare stacks and 108% in Stack overflow questions.

Table 25. Backend technology community metrics comparison.

		Contributors			Stackshare stacks			Stack overflow questions		
Language	**Framework**	**2020**	**2021**	**Change**	**2020**	**2021**	**Change**	**2020**	**2021**	**Change**
C#	ASP.NET Core	622	981	**58%**	920	1500	**63%**	52486	64481	23%
C#	ServiceStack	254	259	2%	34	46	35%	4981	5115	3%
Java	Akka HTTP	292	310	6%	21	34	**62%**	1292	1398	8%
Java	Dropwizard	352	375	7%	251	284	13%	1891	1916	1%
Java	Micronaut	265	322	22%	67	119	**78%**	774	1218	**57%**
Java	Play framework	760	775	2%	641	688	7%	16944	17123	1%
Java	Quarkus	373	550	**47%**	77	159	**106%**	992	2064	**108%**
Java	Spring	763	576	-25%	2770	3300	19%	177301	190803	8%
Java	Vert.x	187	217	16%	164	214	30%	2029	2255	11%
JS / TS	Express	262	269	3%	14000	19300	38%	69765	80887	16%
JS / TS	Feathers.js	172	178	3%	126	146	16%	782	829	6%
JS / TS	hapi	206	209	1%	371	388	5%	1247	362	-71%
JS / TS	koa	218	225	3%	405	457	13%	1082	1159	7%
JS / TS	NestJS	182	272	**49%**	620	1100	**77%**	3204	5874	**83%**
JS / TS	restify	199	200	1%	60	65	8%	612	612	0%
JS / TS	Sails.js	226	235	4%	290	304	5%	6488	6541	1%
PHP	CakePHP	558	576	3%	560	594	6%	31109	31469	1%
PHP	CodeIgniter	481	483	0%	2860	3000	5%	68315	69546	2%
PHP	Laravel	589	573	-3%	12700	19900	**57%**	159126	183139	15%
PHP	Phalcon	249	259	4%	213	225	6%	1933	1961	1%
PHP	Slim	206	211	2%	228	245	7%	2676	2749	3%
PHP	Symfony	2210	2494	13%	4220	5500	30%	67137	69888	4%
Python	AIOHTTP	497	577	16%	88	109	24%	1057	1335	26%
Python	Django	2048	2130	4%	18400	26100	42%	248041	277810	12%
Python	Django REST Framework	1023	1098	7%	1320	1600	21%	20438	25549	25%
Python	Falcon	157	168	7%	65	76	17%	183	68	-63%
Python	FastAPI	184	279	**52%**	21	210	**900%**	473	1911	**304%**
Python	Flask	621	634	2%	10400	13900	34%	41449	47619	15%
Python	Tornado	339	351	4%	280	310	11%	3590	3681	3%

Except for Spring, the biggest risers are the technologies getting also the best feature evaluation scores. Even in Java technologies the difference between Spring and Quarkus was small. It's impossible to tell whether there's causality, but correlation is clear.

This proves technology doesn't have to be the most popular to be full of features. FastAPI barely made it to the list of evaluated technologies as it had so little Stackshare stacks and Stack overflow questions a year ago. This also raises the questions whether some of the dropped-out technologies would've been worth deeper inspection. Community vitality metrics may still play a role in predicting technologies' longevity.

12 Conclusion

Results

Examining prior research revealed many factors affecting software development productivity, but only few were technical. Reuse, documentation quality and automatization were found to be suitable for the technology selection process. In addition, vital community was found to contribute to those factors by providing reuse and documentation in the form of extensions and web articles.

Python, JavaScript/TypeScript, Java, C# and PHP were found to be the most used programming languages for backend development. Almost 90 backend frameworks and libraries were discovered for these languages, of which 29 were considered vital and enough general purpose. Frontend development had smaller set of technologies available after considering SPA capabilities. Of these 6 were JavaScript/TypeScript based and only one C#/WebAssembly based.

46 features were gathered for both backend and frontend by examining technologies' official documentation websites. Evaluating technologies revealed great differences between feature availability in the backend technologies. Lack of schema-based request binding and inferred OpenAPI documentation were found to result in more repetitive code thus being the most crucial individual features when evaluating productivity. Frontend technologies were more on par and such crucial difference in features couldn't be determined.

ASP.NET Core, NestJS, Laravel, FastAPI and Spring were found to be the most feature rich backend technologies to be used with C#, TypeScript, PHP, Python and Java respectively. In the frontend

technologies Vue.js based frameworks Quasar Framework and Vuetify had the most features built in.

Ethics

This thesis has aimed to be as objective as possible. Research has been done purely for the interest in the studied subject without any affiliation to companies or individuals involved in the development of the studied technologies.

Although technology evaluation was done as if the author hadn't had any prior experience with them, it's impossible to repeal all the knowledge gained from working in the industry over a decade. There's a possibility the terminology used for web searches has been biased towards technologies familiar to the author. Features had various terms in different technologies. E.g. middleware features were also called hooks, interceptors, decorators and filters. It's possible that some features were not discovered due to abnormal naming causing the technology in question having worse score than it actually is.

Discussion

One interesting observation is that TypeScript and C# are the only languages having considerable frameworks available for both backend and frontend. TypeScript has a slight edge in the frontend and C# in the backend. Further studies could be made to research whether using the same language for backend and frontend would have any significant effect on productivity.

This thesis also focused only on developing new application prototypes and theoretical evaluation of technologies. Future studies could be done to compare technologies in practice and evaluate the effects on productivity in the longer term by including maintenance and testing effort.

Summary

This thesis studied how full stack development productivity could be increased via technology selection. Prior studies on software development productivity were reviewed to find factors viable for guiding technology selection. Backend and frontend features were discovered by inspecting

online documentations of selected technologies. Finally, technologies were evaluated on quality of documentation and availability of features.

Backend technologies had quite remarkable differences between feature availability. The biggest deficiencies were found from the request binding capabilities which also resulted in more manual work for validation and OpenAPI documentation. Used programming language may not play a significant role in productivity. Except for JavaScript all languages had good frameworks available.

In the frontend no significant differences were found between technologies' features. Each had only a few minor weaknesses and most of them were easily overcome with extensions. Frontend development productivity is likely more affected by personal preferences with the development style of each technology than the features they provide.

References

Alpha Software. (2021). The Pros and Cons of Low Code Development. Accessed on 28 October 2021. Retrieved from https://www.alphasoftware.com/pros-and-cons-of-low-code-development

AWS. (2020). AWS Elastic Beanstalk. Accessed on 23 October 2020. Retrieved from https://aws.amazon.com/elasticbeanstalk/

Azure. (2020). Web Apps. Accessed on 23 October 2020. Retrieved from https://azure.microsoft.com/en-us/services/app-service/web/

Canedo, E. D., & Santos G. A. (2019). Proceedings of the XXXIII Brazilian Symposium on Software Engineering, 307-316. https://doi.org/10.1145/3350768.3352491

Clark, J. (2020). Top 10 backend frameworks. Accessed on 6 November 2021. Retrieved from https://blog.back4app.com/backend-frameworks/

de Barros Sampaio, S. C., Barros, E. A., de Aquino, G. S. , e Silva, M. J. C., & de Lemos Meira, S. R. (2010). A Review of Productivity Factors and Strategies on Software Development. Fifth International Conference on Software Engineering Advances, 196–204. https://doi.org/10.1109/ICSEA.2010.37

Fielding, R. (2000). Representational State Transfer (REST). Accessed on 1 November 2021. Retrieved from https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Fielding, R. (20 October 2008). REST APIs must be hypertext-driven. Retrieved from https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

Gaitatzis, T. (2019). Learn REST APIs. BackupBrain Publishing.

Github. (2021). Release v0.0.1 · npm/npm · GitHub. Accessed on 22 October 2021. Retrieved from https://github.com/npm/npm/releases/tag/v0.0.1

Google Cloud. (2020). App Engine. Accessed on 20 October 2020. Retrieved from https://cloud.google.com/appengine

Hacker news. (2020). The Prestige Trap: finance, big tech, and consulting. Accessed on 22 October 2021. Retrieved from https://news.ycombinator.com/item?id=25093349

Hackernoon. (6 March 2021). The Pros and Cons of Low-Code Development. https://hackernoon.com/the-pros-and-cons-of-low-code-development-4y2p33g9

Jiang, Z., & Comstock, C. (2007). The Factors Significant to Software Development Productivity. World Academy of Science, Engineering and Technology, Open Science Index 1, International

Journal of Computer and Information Engineering, 1(1), 68-72. https://doi.org/10.5281/zenodo.1083495

JWT. (2021). JSON Web Tokens. Accessed on 26 October 2021. Retrieved from https://jwt.io/

Long, T. (2021). Low-code development platforms: Pros and cons. JDLT. Accessed on 28 October 2021. Retrieved from https://jdlt.co.uk/blog/low-code-development-platforms-pros-and-cons/

Material Design. (2021). Material Design Text fields. Accessed on 29 October 2021. Retrieved from https://material.io/components/text-fields

Maxwell, K. D., & Forselius, P., (2000). Benchmarking Software Development Productivity. IEEE Software 17(1), 80-88. https://doi.org/10.1109/52.820015

MDN Web Docs. (2021a). HTTP request methods. Accessed on 4 November 2021. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

MDN Web Docs. (2021b). HTTP request headers. Accessed on 4 November 2021. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

MDN Web Docs. (2021c). HTTP Messages. Accessed on 4 November 2021. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

MDN Web Docs. (2021d). Cross-Origin Resource Sharing (CORS). Accessed on 25 October 2021. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

Melo, C., Cruzes, D. S., Kon, F., & Conradi, R. (2011). Agile Team Perceptions of Productivity Factors. 2011 Agile Conference, 57-66. https://doi.org/10.1109/AGILE.2011.35

Modulecounts. (2020). Module Counts. Accessed on 20 November 2020. Retrieved from http://www.modulecounts.com

Mota, J. S., Tives, H. A., & Canedo, E. D. (2021). Tool for Measuring Productivity in Software Development Teams. Information, 12(10), 396. https://doi.org/10.3390/info12100396

Murphy-Hill, E., Jaspan, C., Sadowski, C., Shepherd, D., Phillips, M., Winter, C., Knight, A., Smith, E., & Jorde, M. (2021). What Predicts Software Developers' Productivity?. IEEE Transactions on Software Engineering 47(3), 582-594. https://doi.org/10.1109/TSE.2019.2900308

Swagger. (2021). OpenAPI Specification. Accessed on 25 October 2021. Retrieved from https://swagger.io/specification/

Pano, A., Graziotin, D., & Abrahamsson, P. (2018). Factors and actors leading to the adoption of a JavaScript framework. Empirical Software Engineering, 23(6), 3503-3534. https://doi.org/10.1007/s10664-018-9613-x

PYPL PopularitY of Programming Language. (2020a). PYPL PopularitY of Programming Language Oct 2020. https://pypl.github.io/PYPL.htm

PYPL PopularitY of Programming Language. (2020b). Top IDE index Oct 2020. https://pypl.github.io/IDE.html

Reddit. (2013). After CRUD, What's next? Accessed on 22 October 2021. Retrieved from https://www.reddit.com/r/learnprogramming/comments/1fspj0/after_crud_whats_next/

roadmap.sh. (2021a). Backend Developer. Accessed on 1 November 2021. Retrieved from https://roadmap.sh/backend

roadmap.sh. (2021b). Frontend Developer. Accessed on 1 November 2021. Retrieved from https://roadmap.sh/frontend

Spendel, T. (25 May 2020). Low-code development platforms. Pros, cons, use cases. skyrise.tech. https://blog.skyrise.tech/low-code-development-platforms

Stackshare. (2020a) What are the best Frameworks (Full Stack) Tools? Accessed on 13 October 2020. Retrieved from https://stackshare.io/frameworks

Stackshare. (2020b). What are the best Microframeworks (Backend) Tools? Accessed on 13 October 2020. Retrieved from https://stackshare.io/microframeworks

Stackshare. (2020c). Build, Test, Deploy Index. Accessed on 23 October 2020. Retrieved from https://stackshare.io/index/build-test-deploy

State of JS. (2020). Frontend Frameworks. https://2020.stateofjs.com/en-US/technologies/front-end-frameworks/

Stangarone, J. (28 February 2019). Pros and cons of low-code development platforms. mrc's Cup of Joe Blog. https://www.mrc-productivity.com/blog/2019/03/pros-and-cons-of-low-code-development -platforms/

Tay, N. (5 April 2021). 7 Pros and Cons of Low-Code/No-Code. Major Online Business and Marketing. https://blog.hslu.ch/majorobm/2021/04/05/7-pros-and-cons-of-low-code-no-code-ntsy-2-ua-192667621-1/

TechEmpower. (2021). Web Framework Benchmarks. Accessed on 6 November 2021. Retrieved from https://www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=composite

Team blind. (2020). Career Path and Growth : How to rise above glorified CRUD applications? Accessed on 22 October 2021. Retrieved from https://www.teamblind.com/post/Career-Path-and-Growth-How-to-rise-above-glorified-CRUD-applications-ewfv2QbR

TIOBE Index. (2020). TIOBE Index for October 2020. https://www.tiobe.com/tiobe-index/

Tomaszewski, P., & Lundberg, L. (2005). Software development productivity on a new platform: an industrial case study. Information and Software Technology 47(4), 257–269. https://doi.org/10.1016/j.infsof.2004.08.007

Tozzi, C. (1 January 2019). Low-Code Development Is Awesome--Here's When Not to Use It. ITPro Today. https://www.itprotoday.com/devops-and-software-development/low-code-development-awesome-here-s-when-not-use-it

Wagner, S., & Ruhe, M. (2008). A Systematic Review of Productivity Factors in Software Development. Proc. 2nd International Workshop on Software Productivity Analysis and Cost Estimation (SPACE 2008), 1-6.

WebStorm. (2021). Languages and frameworks. Accessed on 4 November 2021. Retrieved from https://www.jetbrains.com/help/webstorm/application-development-guidelines.html

Wikipedia. (2021). Dependency injection. Accessed on 5 November 2021. Retrieved from https://en.wikipedia.org/wiki/Dependency_injection

Appendices

Appendix 1. List of inspected backend framework comparison sites

https://www.monocubed.com/best-backend-frameworks/
https://blog.back4app.com/backend-frameworks/
https://statisticsanddata.org/data/most-popular-backend-frameworks-2012-2021/
https://medium.com/javarevisited/10-best-frontend-and-backend-frameworks-for-java-python-ruby-and-javascript-developers-cce3c951787a
https://hackr.io/blog/web-development-frameworks
https://www.keycdn.com/blog/best-backend-frameworks
https://merehead.com/blog/development-trends-best-backend-frameworks-in-2022/
https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks
https://www.globalmediainsight.com/blog/web-development-frameworks/
https://jumpgrowth.com/top-10-web-development-frameworks/
https://www.intagleo.com/blog/most-popular-backend-frameworks-for-web-development-in-2019/
https://morioh.com/p/d024b656ccc2
https://www.parrolabs.com/blog/2021-07-29-top-5-popular-backend-frameworks-in-2021
https://www.simform.com/blog/best-nodejs-frameworks/
https://www.codesnail.com/backend-web-development-frameworks/
https://acropolium.com/blog/most-popular-backend-frameworks-in-2021-2022-pros-and-cons-what-to-choose/
https://www.technotification.com/2021/07/backend-web-development-frameworks.html
https://www.kelltontech.com/kellton-tech-blog/top-7-backend-web-development-frameworks-in-2021
https://cadabra.studio/blog/best-backend-technologies-list-comparison-examples
https://www.techomoro.com/what-are-the-best-frontend-and-backend-frameworks-to-build-web-apps/
https://the-tech-trend.com/software-development/top-10-backend-frameworks-for-web-development/
https://saamarketing.co.uk/top-10-backend-web-development-frameworks-2021/
https://www.fortunesoftit.com/top-5-backend-frameworks-in-2021/
https://impressit.io/blog/best-backend-frameworks
https://www.developerupdates.com/blog/best-backend-frameworks-for-web-development
https://www.thirdrocktechkno.com/blog/top-5-picks-for-backend-development-in-2021/
https://codete.com/blog/top-web-backend-frameworks-in-2021
https://www.crowdbotics.com/blog/most-compatible-frontend-backend-framework-pairings
https://exceed-team.com/tech/best-frontend-and-backend-frameworks-for-developers

https://monovm.com/blog/backend-development-how-to-choose-the-right-framework/
https://www.asynclabs.co/blog/software-development/how-to-choose-the-right-backend-technology-for-your-app/
https://idego-group.com/best-backend-for-react/
https://www.ateamsoftsolutions.com/top-5-backend-technologies-for-web-application-development/
https://innovature.ai/top-backend-frameworks-2021/
https://chudovo.com/backend-development-how-to-choose-the-best-framework-for-your-project/
https://isoftlab.com.my/10-best-web-development-framework-backend-and-frontend/
https://hackerkernel.com/blog/best-top-backend-frameworks/
https://www.decipherzone.com/blog-detail/top-10-backend-development-frameworks
https://www.sam-solutions.com/blog/web-development-frameworks/
https://www.geeksforgeeks.org/which-one-is-most-demanding-back-end-web-framework-between-laravel-node-js-and-django/
https://masteringbackend.com/posts/top-5-backend-frameworks/
https://www.stackoftuts.com/web-development/best-web-development-frameworks-2019-backend-frontend/
https://www.aalpha.net/articles/top-backend-frameworks-for-web-development/
https://teqnation.com/top-7-backend-web-frameworks-to-use-in-2019/
https://healthgradespro.com/best-backend-frameworks/
https://www.unicodesolutions.com/top-backend-frameworks-to-build-your-web-application/
https://moodup.team/blog/which-backend-framework-is-right-for-your-project/
https://www.slideshare.net/markwilston1/top-12-backend-frameworks-for-web-development-in-2021
https://novateus.com/blog/7-best-backend-framework-in-2021/
https://www.merixstudio.com/blog/backend-development/

Appendix 2. List of inspected frontend framework comparison sites

https://www.monocubed.com/best-front-end-frameworks/
https://www.simform.com/blog/best-frontend-frameworks/
https://www.ideamotive.co/blog/best-frontend-frameworks
https://technostacks.com/blog/best-frontend-frameworks/
https://medium.com/geekculture/best-front-end-frameworks-for-web-development-of-2021-the-complete-guide-ec30098fd1d0
https://medium.com/javarevisited/10-best-frontend-and-backend-frameworks-for-java-python-ruby-and-javascript-developers-cce3c951787a
https://cult.honeypot.io/reads/best-frontend-javascript-frameworks-learn-2021/
https://www.mindbowser.com/best-frontend-frameworks/
https://blog.devgenius.io/best-frontend-frameworks-of-2021-for-web-development-7a183652d81b
https://jumpgrowth.com/top-10-web-development-frameworks/
https://www.communicationcrafts.com/frontend-frameworks-for-web-development-in-2021/
https://www.sitepoint.com/most-popular-frontend-frameworks-compared/
https://www.uptech.team/blog/frontend-frameworks-for-web-product
https://existek.com/blog/top-front-end-frameworks-2021/
https://www.gurutechnolabs.com/top-front-end-frameworks/
https://blog.learncodeonline.in/top-3-frontend-frameworks
https://www.techgeekbuzz.com/front-end-frameworks/
https://www.konstantinfo.com/blog/frontend-frameworks/
https://www.clariontech.com/blog/top-5-frontend-frameworks-to-work-with-in-2019
https://logap.com.br/en/blog/best-frontend-framework/
https://fabrity.com/blog/technical/4-frontend-frameworks-you-should-know-about-in-2021/
https://graffersid.com/best-frontend-frameworks-for-web-development/
https://www.keycdn.com/blog/frontend-frameworks
https://whdb.com/blog/front-end-frameworks-making-best-choice/
https://www.lambdatest.com/blog/best-web-development-frameworks/
https://www.globalmediainsight.com/blog/web-development-frameworks/
https://hackernoon.com/top-7-best-frontend-development-frameworks-and-when-to-use-them-4v3a3wwa
https://enlear.academy/the-5-best-frontend-frameworks-to-learn-in-2021-74b049ed98f1
https://www.vervelogic.com/blog/best-front-end-frameworks/
https://www.ateamsoftsolutions.com/which-is-the-best-javascript-frontend-framework-angular-react-or-vue/

Appendix 3. List of “awesome” websites

Url	Retrieval date
https://github.com/vinta/awesome-python	2020-10-15
https://github.com/trananhkma/fucking-awesome-python	2020-10-15
https://github.com/akullpp/awesome-java	2020-10-15
https://github.com/uhub/awesome-java	2020-10-15
https://github.com/pditommaso/awesome-java	2020-10-15
https://github.com/sindresorhus/awesome-nodejs	2020-10-15
https://github.com/bnb/awesome-awesome-nodejs	2020-10-15
https://github.com/tejasrsuthar/Awesome-NodeJS	2020-10-15
https://github.com/quozd/awesome-dotnet	2020-10-15
https://github.com/thangchung/awesome-dotnet-core	2020-10-15
https://github.com/NajiElKotob/Awesome-DotNET	2020-10-15
https://github.com/danperor/awesome-csharp	2020-10-16
https://github.com/ziadoz/awesome-php	2020-10-16
https://github.com/uhub/awesome-php	2020-10-16

Appendix 4. Studied backend frameworks & libraries

GP	General purpose
REL	Months from latest release
C	Github contributors
SS	Stackshare stack count
SO	Stack overflow question count
n.d.	Not determined

							Ranking											Ranking		
Language	Framework / library	GP	REL	C	SS	SO	C	SS	SO		Language	Framework / library	GP	REL	C	SS	SO	C	SS	SO
C#	ASP.NET	Y	36	149	17300	359597	45	2	1		JS / TS	NestJS	Y	3	182	620	3204	36	14	21
C#	ASP.NET Core	Y	1	622	920	52486	7	12	10		JS / TS	QEWD.js	-	-	3	-	0	83	-	84
C#	Carter	Y	2	37	-	4	67	-	68		JS / TS	restify	Y	4	199	60	612	32	39	45
C#	LightNode	n.d.	49	6	-	0	81	-	82		JS / TS	Sails.js	Y	1	226	290	6488	28	21	18
C#	Nancy	n.d.	28	277	45	1118	21	41	37		JS / TS	ThinkJS	-	57	47	-	0	61	-	73
C#	Restier	n.d.	13	26	4	27	74	60	63		JS / TS	Tinyhttp	Y	1	19	-	0	79	-	80
C#	ServiceStack	Y	3	254	34	4981	24	43	19		JS / TS	total.js	Y	4	34	8	70	68	56	57
Java	Akka HTTP	Y	1	292	21	1292	19	49	34		PHP	API platform	N	2	182	42	845	37	42	41
Java	CRNK	Y	5	49	-	23	60	-	64		PHP	CakePHP	Y	1	558	560	31109	10	16	12
Java	Dropwizard	Y	2	352	251	1891	16	23	29		PHP	CodeIgniter	Y	1	481	2860	68315	13	8	7
Java	Elide	Y	1	41	-	0	65	-	74		PHP	Laminas	N	1	28	233	46	72	24	58
Java	Javalin	Y	1	109	20	35	51	51	60		PHP	Laravel	Y	1	589	12700	159126	9	4	4
Java	Jersey	Y	1	47	64	10100	62	37	17		PHP	Lumen	Y	1	58	323	2281	59	19	24
Java	Jooby	Y	1	64	4	13	58	59	66		PHP	Nette	Y	1	104	127	113	52	29	53
Java	Micro-server	Y	1	20	-	0	77	-	79		PHP	Phalcon	Y	5	249	213	1933	27	26	28
Java	Micronaut	Y	1	265	67	774	22	35	43		PHP	PSX	-	34	-	-	0	-	-	71
Java	Play framework	Y	5	760	641	16944	6	13	15		PHP	Silex	-	30	250	34	1371	26	44	32
Java	Quarkus	Y	1	373	77	992	15	34	40		PHP	Slim	Y	6	206	228	2676	30	25	23
Java	Rapidoid	n.d.	29	19	3	6	78	63	67		PHP	Spiral	Y	1	15	-	0	80	-	81
Java	Ratpack	Y	6	134	11	86	48	55	56		PHP	Symfony	Y	1	2210	4220	67137	1	7	8
Java	Rest.li	N	1	85	13	3	56	54	69		PHP	WaterPipe	Y	3	2	-	0	84	-	85
Java	RestEasy	Y	1	175	-	2225	40	-	25		PHP	Yii	-	9	290	576	16626	20	15	16
Java	RestExpress	n.d.	57	-	-	0	-	-	70		Python	AIOHTTP	Y	1	497	88	1057	12	33	39
Java	Restlet	n.d.	8	41	7	1222	64	57	36		Python	Apistar	-	19	86	-	0	55	-	72
Java	RestX	n.d.	30	27	-	0	73	-	77		Python	Bottle	-	11	175	31	1421	39	46	31
Java	Spark	Y	1	132	-	526	49	-	46		Python	Cornice	Y	2	91	-	31	54	-	62
Java	Spincast	Y	5	1	-	0	85	-	86		Python	Django	Y	1	2048	18400	248041	2	1	2
Java	Spring	Y	1	763	2770	177301	5	9	3		Python	Django REST Framework	Y	6	1023	1320	20438	3	11	14
Java	Spring Boot	Y	1	764	11500	92779	4	5	5		Python	Eve	N	1	164	-	483	42	-	48
Java	Spring MVC	Y	2	506	322	54669	11	20	9		Python	Falcon	Y	3	157	65	183	43	36	50
Java	Vert.x	Y	1	187	164	2029	33	28	27		Python	FastAPI	Y	2	184	21	473	35	50	49
JS / TS	actionhero	Y	1	110	15	18	50	52	65		Python	Flask	Y	6	621	10400	41449	8	6	11
JS / TS	adonis	Y	1	46	97	487	63	32	47		Python	Flask Api	-	11	28	-	0	71	-	76
JS / TS	DerbyJS	N	4	31	-	94	69	-	55		Python	Flask restful	-	9	143	3	1302	46	62	33
JS / TS	Express	Y	5	262	14000	69765	23	3	6		Python	guillotina	Y	1	39	-	0	66	-	75
JS / TS	fastify	Y	1	303	119	173	18	31	51		Python	Hug	Y	1	104	14	35	53	53	61
JS / TS	Feathers.js	Y	1	172	126	782	41	30	42		Python	Sandman2	Y	2	20	-	0	76	-	78
JS / TS	hapi	Y	3	206	371	1247	31	18	35		Python	Sanic	Y	1	253	60	153	25	38	52
JS / TS	hunchwot	Y	1	5	-	0	82	-	83		Python	Tastypie	-	10	180	34	1636	38	45	30
JS / TS	KeystoneJs	N	1	139	48	653	47	40	44		Python	Tornado	Y	1	339	280	3590	17	22	20
JS / TS	koa	Y	4	218	405	1082	29	17	38		Python	Turbogears	-	8	29	7	109	70	58	54
JS / TS	Loopback	N	1	152	208	2796	44	27	22		Python	Vibora	-	27	25	4	0	75	61	87
JS / TS	Meteor	N	1	444	1700	28703	14	10	13		Python	Web2py	-	8	186	28	2128	34	48	26
JS / TS	moleculer	N	1	73	31	43	57	47	59											

Appendix 5. Backend technology evaluation results

Framework	Platform	Routing												Middleware										Handlers								Auth									Logging		OpenApi		Messaging		Tasks	TOTAL
		method	path	parameters	wildcard	constraints	prefix	reversing	subdomain	header versioning (semver)	static files	rate limiting	redirect (before handler)	fallback / catch all	CORS	before / onrequest	after / on response	terminate	parameters	filters	decorators	conventions	ordering	content-negotiation	automatic error handling	dependency injection	schema based model binding	automatic model binding	schema based validation	applying validation	response cache	key-value cache	grouping/dependencies	username + password	Social media accounts	Azure AD	roles/groups/...	jwt	policies	conventions	automatic	levels/environment	schema creation	documentation api	broadcasting (webscoket, sse)	events	scheduling	
ASP.NET Core	.NET	1	1	1	1	1	1	2	1	2	1	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	52
NestJS	Node	1	1	1	1	1	1	1	2	4	1	2	2	2	1	1	1	2	1	1	1	1	1	1	1	1	1	1	1	2	1	2	2	1	1	1	1	1	1	1	1	1	1	1	64
ServiceStack	.NET	1	1	1	1	1	1	4	4	4	3	4	2	1	1	1	1	4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	1	1	68
Laravel	PHP	1	1	1	1	1	1	1	1	4	2	1	1	1	2	1	1	1	1	1	1	1	1	4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	69
FastAPI	Python	1	1	1	1	1	1	1	2	4	1	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	76
Spring	JVM	1	1	1	1	1	2	1	3	3	2	3	2	2	1	2	2	2	2	2	2	2	2	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	1	82
Quarkus	JVM	1	1	1	1	1	1	4	4	4	1	2	3	2	1	1	1	1	1	4	1	1	1	4	1	1	1	1	1	2	2	2	2	2	4	3	2	1	1	1	1	2	2	2	83
Symfony	PHP	1	1	1	1	1	1	1	1	4	2	2	2	1	1	2	2	2	2	2	2	2	2	4	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	83
CakePHP	PHP	1	1	1	1	1	1	2	2	4	2	2	2	2	1	1	1	1	1	1	1	1	1	4	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	84
Dropwizard	JVM	1	1	1	1	1	3	1	4	4	2	2	2	2	1	2	1	1	1	4	1	1	1	4	2	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	85
Django REST Framework	Python	1	1	1	1	1	2	1	3	4	2	2	2	2	1	2	1	1	1	4	1	1	1	4	1	3	1	1	1	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	86
Django	Python	2	1	1	1	1	1	2	2	3	2	2	2	2	1	2	1	1	1	3	1	1	1	4	1	2	4	4	4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	87
Akka HTTP	JVM	1	1	1	1	1	4	4	4	4	1	4	2	2	1	2	1	1	1	1	1	1	1	4	1	2	4	4	4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	88
Play framework	JVM	1	1	1	1	1	1	3	4	4	1	2	2	2	1	2	1	1	1	4	1	1	1	4	1	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	88
Express	Node	1	1	1	1	1	2	2	2	4	1	2	2	2	1	1	1	1	1	2	1	1	1	4	1	2	4	4	4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	90
Flask	Python	1	1	1	1	1	2	2	2	4	1	2	2	2	1	1	1	1	1	1	1	1	1	3	1	2	4	4	4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	91
Micronaut	JVM	1	1	1	1	1	4	4	4	4	1	2	2	2	1	1	1	4	1	1	1	1	1	4	1	1	4	4	4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	92
restify	Node	1	1	1	1	1	2	4	4	4	2	2	2	2	1	1	1	2	1	1	1	1	1	4	1	1	4	4	4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	93
hapi	Node	1	1	1	1	4	2	2	3	4	2	2	2	2	1	1	1	1	1	1	2	4	1	1	1	1	4	4	4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	96
Vert.x	JVM	1	1	1	1	1	4	4	4	4	2	4	2	2	1	1	1	1	1	1	1	1	1	4	1	1	4	4	4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	96
Sails.js	Node	1	1	1	1	1	2	2	3	3	2	2	2	2	1	1	1	1	1	1	1	1	1	4	1	1	4	4	4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	97
koa	Node	2	2	2	2	1	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	4	1	1	4	4	4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	100
Slim	PHP	1	1	1	1	1	1	1	1	4	2	2	2	2	1	1	1	1	1	1	1	1	1	4	1	1	4	4	4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	100
CodeIgniter	PHP	1	1	1	1	1	1	1	4	4	3	2	2	2	1	2	1	1	1	4	1	1	1	4	1	3	4	4	4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	100
Feathers.js	Node	4	2	2	4	4	3	4	4	4	1	2	2	2	1	2	1	2	1	1	1	1	1	4	1	2	4	4	4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	100
AIOHTTP	Python	1	1	1	1	4	4	1	4	4	1	2	2	2	1	2	1	1	1	4	1	1	1	4	1	2	4	4	4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	103
Falcon	Python	1	1	1	1	4	4	4	4	3	1	2	2	2	1	2	1	1	1	4	1	1	1	4	1	3	4	4	4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	104
Phalcon	PHP	1	1	1	1	1	4	4	4	4	2	2	2	2	1	2	1	1	1	4	1	1	1	4	1	2	4	4	4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	109
Tornado	Python	1	1	1	1	1	3	2	4	4	2	4	2	2	1	3	3	3	3	3	3	3	3	3	3	4	4	4	4	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	123

Appendix 6. Frontend technology evaluation results

		Angular	Blazor	React	Svelte	Vue	Vuetify	Quasar
Routing	Path	1	1	2	2	1	1	1
	Parameters	1	1	2	2	1	1	1
	Meta	1	4	2	2	1	1	1
	Hooks / Guards	1	1	2	2	1	1	1
	URL generation	2	2	2	2	1	1	1
	Nesting	1	4	2	2	1	1	1
	Placeholders	1	4	2	3	1	1	1
	Wildcards	1	1	2	2	1	1	1
	Redirect	1	4	2	2	1	1	1
Components	Reusable components	1	1	1	1	1	1	1
	State	1	1	1	1	1	1	1
	Computed state	2	1	1	1	1	1	1
	Change detection	2	1	1	2	1	1	1
	Created / mounted hook	1	1	1	1	1	1	1
	Destroyed / unmounted hook	1	1	1	1	1	1	1
	Parent-child communication	1	1	1	1	1	1	1
	Child-parent communication	1	1	1	1	1	1	1
	Descendant communication	2	1	1	1	1	1	1
	Child reference	1	1	1	1	1	1	1
	Dynamic component selection	2	2	1	1	1	1	1
Rendering	Interpolation	1	1	1	1	1	1	1
	Raw HTML	1	1	1	1	1	1	1
	Conditionals	1	1	1	1	1	1	1
	Loops	1	1	1	1	1	1	1
	Value binding	1	1	1	1	1	1	1
	Two-way model binding	1	1	2	1	1	1	1
	Transclusion	1	1	1	1	1	1	1
	Multi transclusion	1	1	1	1	1	1	1
	Input validation	1	1	2	2	2	1	1
	Pipes	1	2	2	2	1	1	1
	Directives	1	4	4	1	1	1	1
	Attributes	1	1	1	1	1	1	1
	Class	1	1	1	1	1	1	1
	Events	1	1	1	1	1	1	1
	Event modifiers	3	4	4	1	1	1	1
	Scoped styles	1	1	2	1	1	1	1
State management	State management	1	1	2	1	1	1	1
	Modules	1	1	2	2	1	1	1
i18n	Key-value	1	1	2	2	1	1	1
	Interpolation (parameters)	1	1	2	2	1	1	1
	Pluralization	1	2	2	2	1	1	1
	Datetime	1	1	2	2	1	1	1
UI / Utils	Material Design / Bootstrap	2	2	2	2	2	1	1
	Touch gestures	2	3	2	2	2	1	1
	Session / local Storage	1	2	1	1	1	1	1
	Meta	2	2	2	1	2	2	1
TOTAL		56	73	74	66	50	47	46